Benjamin Bara, BSc

Design and Implementation of IIoT Demonstrators using Hardware Security Extensions

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Dr.techn. Rainer Matischek (Infineon Technologies Austria AG)

Graz, May 2019

Affidavit

Acknowledgement

At first, I want to thank the Institute of Technical Informatics (ITI) at the Graz University of Technology for supervising this master thesis. I am very grateful for the marvelous support during the execution, especially from Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger. Also Thomas Ulz, who supported me with incredible input and feedback.

Next, I want to thank Infineon Technologies Austria AG for allowing me to execute this thesis in their buildings and under their advisory. Primarily, I want to thank my advisor Dipl.-Ing. Dr.techn. Rainer Matischek for the continuous support and the great discussions that really brought me forward during the creation of this thesis. Additionally, the members of the “Cooperative Research and Exploration” department, who supported me with their knowledge and their helpfulness. The team events and lunch breaks were a welcome change to everyday life.

I also want to thank my colleagues of the “Digital Security Solutions – Contactless Innovations” department, who welcomed me warm and supported me continuously. Remarkably Andreas Wallner, who backed me with his knowledge and experience in this field. Also, a big thank you to the heads of both departments, Gerald Holweg and Holger Bock, who managed that my employment at Infineon Technologies Austria AG was as convenient and uncomplicated as possible.

Finally, I want to thank:
all the new friends I made during my time in Graz who supported me wholeheartedly;
my hometown friends who let me feel like I never left home;
my flatmates who always made sure that my work-life balance is on point;
my girlfriend, Sophie, who encouraged me in my most difficult moments;
and last but not least, my brother Jonathan and my parents Waltraud and Lounes, who always believe in me and support me amazingly wherever they can.

Thank you all for this great experience!

Benjamin Bara

A part of this project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692480 (IoSense). This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Saxony, Spain, Austria, Belgium, Slovakia.

A part of the work has been performed in the project Power Semiconductor and Electronics Manufacturing 4.0 (SemI40), under grant agreement No 692466. The project is co-funded by grants from Austria, Germany, Italy, France, Portugal and - Electronic Component Systems for European Leadership Joint Undertaking (ECSEL JU).

Abstract

Currently, manufacturers are confronted with the term "Industry 4.0", an umbrella term for various emerging innovations in the industrial field. One of these upcoming trends is the Industrial Internet of Things (IIoT), an interconnection of countless devices that share data and information within a company. The IIoT paves the way for multiple new promising use cases such as predictive maintenance. However, the trends also show security problems that were previously hidden "behind closed doors", for example the use of communication protocols without integrated security mechanisms. This thesis discusses some of these upcoming problems for existing networks and additionally gives an overview of recent ubiquitous hardware attack vectors, where some are even exploitable remotely and therefore become serious attack vectors due to the increased reachability of devices.

One approach to mitigate such attacks is the use of hardware security extensions. These specially designed and evaluated devices are usually connected to a generic and therefore potentially unsecure hardware and then are used for security-critical tasks such as the storage of secret keys. In this thesis, different types of hardware security extensions are discussed and evaluated for two typical industrial use cases: The first use case is partly carried out in the course of the ECSEL JU project "IoSense" [1] and shows how the configuration of a production equipment is updated securely by Near Field Communication (NFC). The reception and validation is done by the chosen hardware security extension. The second use case is partly carried out in the course of the ECSEL JU project "SemI4.0" [2] and demonstrates an approach to get rid of "isolated production networks" within the network infrastructure of a company. Part of the thesis is to design and implement the translation of a legacy communication protocol (SECS/GEM) without security measures to state-of-the-art protocols (OPC UA and MQTT) that include security measures by design. Additionally, the hardware security extension is integrated into the OPC UA implementation, where it is used as cryptographic coprocessor and as secured storage for secret keys.

The two use cases require a USB and an ISO 14443 interface for the communication. Therefore, a hardware security token with a running Java Card OS and Global Platform (GP) support is used. This thesis includes the design and parts of the software implementation of the used security token and the corresponding host library. The host library is required to integrate a connected hardware security extension into the system of a host.

Abstract (German)

Zur Zeit beschäftigt der Begriff “Industrie 4.0” diverse Industrien und Hersteller. Dabei handelt es sich um einen Sammelbegriff für eine Vielzahl an aufkommenden Innovationen im industriellen Umfeld. Einer dieser neuen Trends ist das Internet der Dinge, in dem unzählige Geräte miteinander verbunden sind, um Daten und Informationen auszutauschen. Dieses Netz an Geräten bietet auch im industriellen Umfeld eine Vielzahl an Möglichkeiten. Mit diesen Innovationen kommen aber auch neue Sicherheitslücken und Angriffsvektoren zum Vorschein. Ein Beispiel dafür ist der Einsatz von “ungesicherten” Kommunikationsprotokollen die bisher nur in abgeschotteten Netzwerken verwendet wurden.

Im Rahmen dieser Arbeit werden einige Probleme, die bei der Integrierung solcher abgeschotteten Netzwerke in das Hauptfirmennetzwerk auftreten können, vorgestellt. Zusätzlich werden einige Hardware Sicherheitslücken, die teilweise auch über das Netzwerk ausgenutzt werden können und somit, aufgrund der erhöhten Konnektivität, zu ernsthaften Problemen führen können, präsentiert.

Eine Möglichkeit, um das Potential solcher Angriffsvektoren zu mildern, ist der Einsatz von Hardware Security Erweiterungen. Diese Erweiterungen sind speziell für sicherheitskritische Aufgaben, wie das sichere Lagern von geheimen Schlüssel, entwickelt und evaluiert. Da bei handelsüblichen Geräten typischerweise mehr Wert auf Performance als auf Sicherheit gelegt wird, gelten diese als nicht vertrauenswürdig, weshalb eine solche Erweiterung zusätzlichen Schutz bieten kann.

In dieser Arbeit werden unterschiedliche Typen von Hardware Security Erweiterungen vorgestellt und auf deren Anwendbarkeit für zwei typische Industrie-Anwendungen geprüft. Die erste Anwendung, die teilweise im Zuge des ECSEL JU Projekts “IoSense” [1] durchgeführt wurde, zeigt eine Möglichkeit für einen gesicherten Konfigurationsprozess einer Produktionsmaschine mittels NFC. Der Empfang und die Validierung der geschützten Konfiguration wird dabei von der Hardware Security Erweiterung übernommen. Die zweite Anwendung, die teilweise im Zuge des ECSEL JU Projekts “SemI4.0” [2] durchgeführt wurde, zeigt eine Möglichkeit zur Integrierung eines abgeschotten Produktionsnetzwerks, das auf dem Legacy-Protokoll “SECS/GEM” basiert, in das Firmennetzwerk, das bereits OPC UA als Kommunikationsprotokoll verwendet. Dabei wurde eine Schnittstelle implementiert, die Anfragen und zugehörige Antworten zwischen den Protokollen übersetzt. Zusätzlich wird ein zweites Protokoll, das im Rahmen der Entwicklungen rund um Industrie

4.0 erschienen ist, für eine dritte Netzwerkzone, das externe Netzwerk, verwendet. Es handelt sich dabei um MQTT, das ebenso wie OPC UA bereits integrierte Sicherheitsmöglichkeiten bietet. Zusätzlich dazu wurde im Rahmen dieser Arbeit der bestehende, software-basierte Crypto-Stack von OPC UA durch die Hardware Security Erweiterung ersetzt, wodurch die kryptographischen Operationen und die geheimen Schlüssel in die geschützte Hardware-Umgebung verlegt werden.

Für die beiden Anwendungsfälle ist sowohl eine USB als auch eine ISO 14443 Schnittstelle mit zugehöriger Protokoll-Implementierung nötig. Aus diesem Grund wird ein Hardware Security Token, basierend auf einem Java Card OS mit GP Support, verwendet. In dieser Arbeit wird das Design und Teile der Implementierung des verwendeten Security Tokens und der zugehörigen Host Library vorgestellt. Die Library stellt die Software Schnittstelle zwischen Host und Security Token dar und wird benötigt um den Security Token in das Host System einzubinden.

Table of Contents

List of Figures

List of Tables

List of Code Examples

List of Acronyms

AE	Authenticated Encryption
AES	Advanced Encryption Standard
AGV	Automated Guided Vehicle
APDU	Application Protocol Data Unit
API	Application Programming Interface
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
ATR	Answer to Reset
BLE	Bluetooth Low Energy
BSI	Bundesamt für Sicherheit in der Informationstechnik
CA	Certificate Authority
CAN	Controller Area Network
CCID	Chip Card Interface Device
CB	Contact-based
CC	Common Criteria
CBC	Cipher Block Chaining
cm	centimetre
CPPS	Cyber-Physical Production Systems
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRL	Certificate Revocation List
CSPRNG	Cryptographically Secure PRNG

CSR	Certificate Signing Request
DIN	Deutsches Institut für Normung
DMZ	Demilitarized Zone
DuA	Device under Attack
EAL	Evaluation Assurance Level
E&M	Encrypt-and-MAC
ECC	Elliptic Curve Cryptography
EK	Endorsement Key
ERP	Enterprise-Resource-Planning
EtM	Encrypt-then-MAC
FIDO	Fast IDentity Onine
FIFO	First-In-First-Out
FIPS	Federal Information Processing Standards
GC	Garbage Collector
GEM	Generic Equipment Model
GND	Ground/Earth
GP	Global Platform
GPIO	General Purpose I/O
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HMAC	Keyed-Hash Message Authentication Code
HSMS	High-Speed SECS Message Services
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICC	Integrated Circuit Card
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IIC	Industrial Internet Consortium

IIRA	Industrial Internet Reference Architecture
IoE	Internet of Everything
IoP	Internet of People
IoT	Internet of Things
IIoT	Industrial IoT
IP	Internet Protocol
IR	Intermediate Representation
ISO	International Organization for Standardization
ITI	Institute of Technical Informatics
IV	Initialization Vector
JC	Java Card
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JSON	JavaScript Object Notation
KDF	Key Derivation Function
kHz	Kilohertz
LDAP	Lightweight Directory Access Protocol
LLC	Last-Level Cache
LRU	Least Recently Used
m	metre
M2M	Machine-to-Machine
MAC	Message Authentication Code
MCU	Microcontroller Unit
Mbps	Megabit per second
MES	Manufacturing Execution System
MQTT	Message Queue Telemetry Transport
ms	milliseconds
MSB	Most Significant Bit
MtE	MAC-then-Encrypt

NFC	Near Field Communication
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
NVM	Non-Volatile Memory
OAEP	Optimal Asymmetric Encryption Padding
OASIS	Org. f. t. Advancement of Structured Information Standards
OSI	Open Systems Interconnection
OS	Operating System
OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
PCB	Printed Circuit Board
PC/SC	Personal Computer/Smart Card
PKCS	Public-Key Cryptography Standards
PKI	Public-Key Infrastructure
PID	Product ID
PLC	Programmable Logic Controller
PPS	Protocol and Parameter Selection
PRF	Pseudo-Random Function
PRNG	Pseudo-Random Generator
RAII	Resource Acquisition Is Initialization
RAM	Random-Access Memory
RAMI 4.0	Reference Architecture Model Industrie 4.0
RF	Radio Frequency
RFC	Request for Comment
RFID	Radio-Frequency Identification
RNG	Random Number Generator
RSA	Rivest-Shamir-Adleman Crypto Scheme
RTM	Root of Trust for Measuring
RTR	Root of Trust for Reporting

RTS	Root of Trust for Storage
s	seconds
SCP	Secure Channel Protocol
SC	Smart Card
SCA	Side-Channel Attack
SCADA	Supervisory Control and Data Acquisition
SCOS	Smart Card Operating System
SE	Secured Element
SEMI	Semiconductor Equipment and Materials International
SECS/GEM	SEMI Equipment Communication Standard/GEM
SHA	Secure Hash Algorithm
SM	Secure Messaging
SMEs	Small and Medium-sized Enterprises
SML	SEMI Markup Language
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface
SRK	Storage Root Key
SSH	Secure Shell
SW	Status Word
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
TRNG	True Random Number Generator
TTL	Transistor-Transistor Logic
TTP	Trusted Third Party
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol

List of Acronyms

us	microseconds
USB	Universal Serial Bus
USS	Ultrasonic Sensor
V	Volt
VCC	Supply Voltage
VID	Vendor ID
WSL	Windows Subsystem for Linux
XML	Extensible Markup Language

1. Introduction

In the latest years, the amount of interconnected electronic devices increased exponentially and the forecast keeps this rise [3]. The rapid increase brings tough new challenges to the IT world. Two of these challenges are the unrestrained need for more performance and the increasing miniaturization. Since Moore's Law pushes the electronic parts towards their physical limits, other “hacks” have to be considered to keep the increase alive. These often lead to tremendous vulnerabilities, beginning at the hardware design level.

With the emerging Internet of Things (IoT), the trend goes to multiple interconnected, embedded systems, so-called “smart devices”, that should support the users with their different daily tasks. Since the typical use cases often involve personal data and the devices often communicate over the Internet, confidentiality and other security attributes have to be considered. However, the actual tasks of these systems are usually quite simple (e.g. performing measurements), therefore the performance of the chosen hardware components might not be sufficient for additional cryptographic operations. This further leads to negligence of security measures, specifically in the industrial environment, where real-time constraints might have to be satisfied in addition.

The same “performance before security” anti-pattern can be seen with communication protocols and techniques. The common transport layer protocols, such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), do not provide any security options. Many built-upon protocols also do not target security measures or leave them optional (e.g. HTTP). This also applies to industrial standard protocols such as Modbus/TCP, as [4] points out.

Fovino [5] describes that the usual way to justify the use of unsecured communication is to seal the affected network off from the publicity. With a complete seal-off, Supervisory Control and Data Acquisition (SCADA) functionality becomes unavailable, meaning workers have to be local to the network to maintain the devices in it. A way to enable SCADA without “openness” is by using complex firewall rules that blacklist any network traffic except the required one. As the IoT brings an unavoidable increase of network participants, this becomes an unmanageable effort. Additionally, the increased push to inter-connectivity is contrary to this approach.

Motivation

As Let's Encrypt [6] points out, the number of web pages with HTTPS support increased from 27% in October 2013 to 76% in October 2018. Still, recent discoveries show that encryption is often not enough as there might exist attack vectors for the generic hardware or the cryptographic implementations in use.

For example, [7, 8, 9] show that unprivileged applications can get access to arbitrary memory locations by exploiting hardware performance hacks. Furthermore, [10, 11, 12, 13] show that, under certain circumstances, it is even possible to perform such attacks remotely.

An example for software attacks can be found in [14], where it is shown how secret keys can be inferred from side-channel measurements (e.g. the power consumption). Additionally, with the rising progress and awareness of machine learning and classification techniques, it becomes trivial to perform such an attack even on "protected" software implementations (e.g. in [15]).

As it is shown in the attack "KRACK" [16], it is strongly suggested to encrypt sensitive data even if it is just transmitted inside of a "private" sealed-off network. The authors of this paper showed, that some implementations of the common encryption of WiFi networks, WPA2, had a flaw that can be exploited to decrypt the communication between two WiFi devices. This leads to serious problems, as WiFi networks are often reachable from "outside" their targeted area.

These simple attacks and ubiquitous vulnerabilities show that an increased focus on security measures is urgently required in established and upcoming technologies.

Goals

This thesis seeks to examine capabilities of hardware security extensions in the field of IIoT. For this reason, a security stick based on a Secured Element (SE) with a USB and a NFC interface should be developed. This security stick is then used in two different industrial scenarios: the configuration of production equipment by NFC and the integration of an unsecured production network into the main network by translating a legacy industrial protocol into secured state-of-the-art protocols. The security stick is used in both use cases as crypto coprocessor. Additionally, security-critical data, such as secret keys, is stored in the secured on-chip memory of the SE.

Outline

Chapter 2, 2.3 and 2.4 contain background information about the used technologies. At first, the terms "Industry 4.0" and "Internet of Things" are explained. Then, two state-of-the-art M2M communication protocols (OPC UA and MQTT) are described and compared to a traditional one (SECS/GEM). Further, the cryptographic operations used in the latest OPC UA security policy are explained. Additionally, it is shown why it might not be sufficient to simply "use" cryptography, as there might be other vulnerabilities in software or even in hardware. For this reason, special hardware security modules are discussed.

Chapter 3 gives information about the design and the architecture of the thesis' practical part. At first, the overall architecture of the demonstrators is described. In general, a host system (e.g. a production machine) uses the above mentioned security stick to get additional security protection. Then, the requirements for the security stick are elaborated. After that, the design of the host library is presented. The host library enables a host to communicate with the connected security stick. This library is further used by the host applications, which outsources security-relevant tasks to the hardware security extension. Therefore, the last part of this chapter contains information about the requirements and the design of the demonstrators' applications.

Chapter 4 gives details about the implementational part of this thesis. The implementation chapter is structured similar to the design chapter. In the first section, general implementation designs are presented. Then, the implementation of the security stick is introduced. The last sections deal with the implementation of the host library and the two demonstrators.

Chapter 5 contains information about the results and future work. Additionally, security threats of the presented system are elaborated and discussed.

Chapter 6 gives a short conclusion and future work possibilities.

2. State of the Art and Related Work

This chapter gives information about the current state of the art, related background given by norms and specifications, and related implementation work.

2.1. Industry 4.0

As described by Bassi [17], the term “Industry 4.0” is derived from the “fourth industrial revolution”. In comparison to the previous three, the fourth revolution does not rely on a technical break-through discovery. In fact, this industrial revolution was even named before an actual revolution happened. The term was introduced by the german federal government in 2011 and later defined more properly, mainly by academic research and the industry. A mentionable outcome of the definition process is the Reference Architecture Model Industrie 4.0 (RAMI 4.0), designed by the “Platform Industrie 4.0” and normed by DIN in SPEC 91345. A similar term, which is more common in english-speaking regions, is “Industrial Internet”. Simultaneously, the Industrial Internet Consortium (IIC) is developing the Industrial Internet Reference Architecture (IIRA), with a special view to applying IoT to different industrial areas [18]. However, most of the negotiated targets of Industry 4.0 are towards a “smart factory”, which should increase the operational effectiveness of manufacturers. Further, new business models, services and products should increase the economic value of a factory [19].

For now, different fundamental principles have been evolved:

Smart Factory

“Smart factory” is an umbrella term for several technologies that are combined to make a factory more efficient in different ways. The smart factory is context-aware and assists people and machines during their work by gathering information from their virtual and physical environment. [19, 20]

Cyber-Physical Systems

Cyber-Physical Systems (CPS) are networks of electronic devices that are connected by the Internet or any other form of connection. Basically, CPS bring a fusion between the virtual and the physical world. CPS used in industrial production facilities are also called Cyber-Physical Production Systems (CPPS). [19]

Technical Assistance

Automated Guided Vehicles (AGVs) are full-automatic machines that contribute to the factory life by helping with weightlifting, transportation and other use cases. With increasing interconnection and data collection, the robots become more intelligent. This allows even more automation and a higher efficiency. [19, 20]

Flexibility

One emerging key factor is the increasing flexibility. A smart factory should be able to produce low volumes of highly flexible products. One of the discussed technologies is “additive manufacturing”, e.g. 3D printing. With additive manufacturing, material is added instead of mechanically removed from a solid block. Initially started as method for “rapid prototyping”, 3D printers are used more and more in real production, allowing to provide even small amounts of complex, high quality components at reasonable costs. Also, the factory can become more flexible as the interconnection allows to change the usually centralized approach of a factory into a more distributed one. [17, 19, 20]

Traceability & Product Identification

Product identification and traceability play a huge role in a smart factory. At the start of the production, every component has an unique ID assigned that is kept during the whole production process. This allows to get instant information about the product and its components. The information then can be used for new technologies such as predictive maintenance. One way to identify products is the use of Radio-Frequency Identification (RFID) tags: Every product is equipped with an unique tag that can be read by a RFID reader. [17, 19]

Internet of Everything

Another key factor of the Industry 4.0 is the extensive use of the Internet and the emerging Internet of Things (IoT), even on embedded devices. Further, as stated by Bassi [17], the combination of the IoT and the Internet of People (IoP) results in the IoE. In this scenario, the Internet is not just used as tool to connect devices but as a source of information ("big data"). Further information on the industrial field of IoT can be found in Section 2.1.1.

Communication

As Bassi [17] states, the connections in the different industrial fields are currently divided in several different protocols and field busses that are not compatible to each other. For this reason, RAMI 4.0 comes up with an universal solution: OPC UA. This protocol is adaptable to any use case by supporting several basic communication concepts like publish-subscribe and the common client-server architecture. Three possible communication forms can occur: human-to-human, human-to-machine and machine-to-machine. Information on Machine-to-Machine (M2M) communication can be found in Section 2.1.2.

Security

Due to the increasing "cyberfication", former "physical-only systems" become also vulnerable to cyber threats. As Bicaku et al. [21] point out, this does not simply affect the processes being controlled but also the people who depend on them. Therefore, well-established security standards have to be kept in mind during the design phase of systems. For example, the emerging communication protocols come, in contrast to the legacy ones, with security measurements [18]. Further, systems need to be flexible and easily updatable to counteract possible security breaches and apply new upcoming best practices. Since already existing CPPS might not be able to handle these requirements, additional solutions have to be considered. These include hardware security measurements, where certain security-relevant operations are done on external hardware. Further information on hardware security measurements can be found in Section 2.2.2.

2.1.1. Industrial Internet of Things

Kevin Ashton was the first who phrased the "Internet of Things" as system of linked devices in 1999. The "things" of the IoT are usually called "nodes". Sisinni et al. [22] describe the Industrial IoT (IIoT) as a subset of the IoT that covers the domains around M2M communication, industrial communication and automation methods. As opposed to the consumer IoT, the IIoT is machine- instead of human-oriented. Therefore, also the communication of the IIoT is typically machine-to-machine- and not human-to-machine-based. Sisinni et al. [22], Al-Fuqaha et al. [23] and Bloom et al. [24] divide the IIoT architecture in the perception, the network, and the application layer. Figure 2.1.1 shows the three layers beside the pyramid of automation.

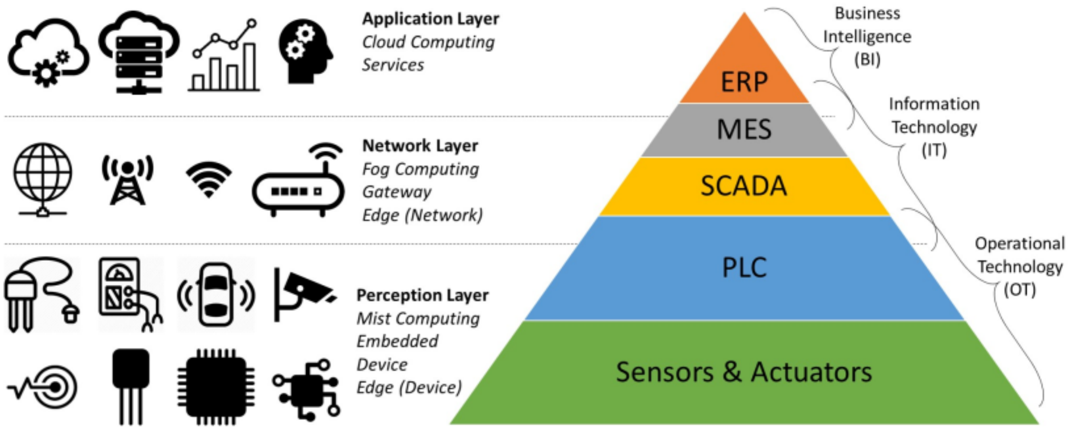


Figure 2.1.1.: Illustration of the hierarchical layers of the IIoT and the relationship to the pyramid of automation. Starting from the bottom, the first layer are the sensors & actuators, which interact with the physical world. Next, the Programmable Logic Controller (PLC) layer, where the data is processed. In Supervisory Control and Data Acquisition (SCADA) layer, the data is monitored and the underlying processes are controlled. Finally, the Manufacturing Execution System (MES) and Enterprise-Resource-Planning (ERP) are used to plan, report and control the whole automation process. Illustration taken from [24].

Perception/Embedded Layer

This is the bottom layer of the architecture, where all the "things" of the Internet of Things are found. Typical communication technologies on this layer are NFC, Bluetooth Low Energy (BLE), WiFi, Ethernet and several wired serial connections and field busses, depending on the application domain, e.g. the CAN bus.

Network/Gateway Layer

This layer is some kind of "middleware" between the underlying perception layer and the overlying application layer. Information is filtered and aggregated before transmitted further. The nodes of this layer are also referred to as "gateways".

Application/Cloud Layer

The business logic and intelligence of the system is found in this layer. Data is stored and processed in the cloud. Usually, clouds are distinguished in public and private ones. Private clouds have a much higher administrative cost but are said to come with a relief in terms of security.

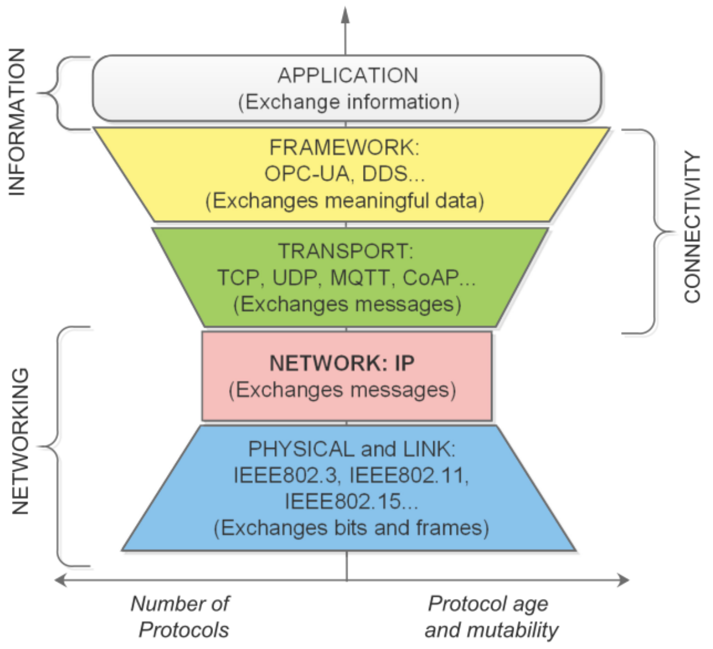


Figure 2.1.2.: Illustration of the IIoT communication protocol stack. The bottom layer is the physical layer, which is used for the physical connection and signals between the participants. In the same layer, but "above" the physical layer, is the link layer. This layer is used to exchange frames with the help of the underlying layer. The network layer is usually represented by the Internet, e.g. IPv4 or IPv6. The upper layers are required to provide syntactic and semantic interoperability. While the transport layer is in charge of exchanging variable length messages, the framework layer exists to transfer structured data on a higher abstraction level, e.g. by having states and events. Illustration taken from [22].

2.1.2. Machine-to-Machine Communication

The current architecture of the Internet (for example the seven layer OSI model), is suited not well for the distributed web of nodes. Figure 2.1.2 shows the typical communication protocol stack of IIoT. Lower levels of the communication stack have to attack scalability and flexibility requirements, while the upper layer must not just be aware of information but also the according semantic rules to correctly interpret data. Bloom et al. [24] mention that applications typically have a low throughput and therefore the focus should be set to the limited hardware capabilities, security and privacy. Additionally, Sisinni et al. [22] conclude that the IIoT is currently not ready to control applications at the field level when bounded reaction-time has to be fulfilled. Instead, the field is currently focusing on supervision, optimization and big data gathering and analysis. Finally, as Bicaku et al. [21] show, different new messaging protocols have been developed with the emerging Industry 4.0.

2.2. Hardware Security

This section contains information about common and recent hardware attack vectors, and how they can be mitigated by hardware countermeasures which are implemented in hardware security extensions.

2.2.1. Hardware Attack Vectors

Skorobogatov [25] gives an overview of hardware attacks. They are divided into two different categories: invasive and noninvasive attacks. Invasive attacks, such as microprobing or reverse engineering, usually require direct access to the chip surface. This is achieved by destroying the package of the chip. Noninvasive attacks do not require physical alteration of the Device under Attack (DuA). Noninvasive attacks are considered as more dangerous, as the attack might not be noticed. In this section, hardware attack vectors and resulting noninvasive attacks are described.

2.2.1.1. Side-Channel Attacks

Side-Channel Attacks (SCAs) are based on exploitations of weaknesses in the system's hardware implementation. Even when cryptographic algorithms are implemented in software, the underlying hardware might leak side-channel information about the implementation. This allows the attacker to bypass theoretically secure cryptographic algorithms. Side-channel attacks are divided in passive and active ones. Commonly exploited passive side-channels include power consumption, electromagnetic emissions and timing. They are exploited by performing measurements and subsequently analyse the observations. Active ones are done by exploiting "ingoing" side-channels, e.g. the supply voltage and are also called "fault injection attacks" [26, 27].

Passive SCAs are further divided in simple and differential attacks [27]. Simple side-channel attacks require information about the concrete implementation. When performing a simple attack, a small number of traces (sets of measurement samples) of the side-channel is mapped to the actual implementation. Based on the traces, the attacker tries to replicate the control flow of the execution. However, differential attacks do not require any knowledge about the concrete implementation, knowing the algorithm is sufficient. This attack form exploits the correlation between the processed data and the side-channel measurements. Therefore, an estimated model of the emitted side-channel signals during the operation under attack is required. A simple but efficient model is the hamming distance of the processed data, before and after the operation.

Power Analysis Attacks

In this passive SCA, the power consumption of the DuA is exploited. The power consumption is divided in two parts: a static and a dynamic one. Static power consumption arises from circuits that intentionally draw power. On the other side, dynamic power consumption occurs as soon as an on-die capacity is charged or discharged. As the voltage of such a capacity represents a 1 or 0 status of a digital system, the dynamic power consumption indicates a state change inside the processor. With this information, the actual attack can be started [27].

CPU Cache Attacks

CPU Cache Attacks belong to the passive SCAs. They are a special form of timing attacks and can be exploited without external measurements. Usually, when a process accesses a value stored in memory, the contents of the memory location are loaded into the cache too. This is done to shorten the access path to the value for the next usage. The general approach is to sort memory addresses by the usage time. If the cache is full, the Least Recently Used (LRU) entry is replaced by the new one. The cache is usually organized in tags, indices and offsets. As Yarom and Falkner [28] state, the actual cache entry is usually pre-decided, since the respective tag and index is mapped from the memory address of the accessed value (either physical or virtual, depending on the cache structure). Cache attacks are software-based and use the access time difference between a cache hit (the value was recently used and therefore loaded from the cache) and a cache miss (the value was not used recently and therefore not in the cache). A processor has different cache levels that provide different attacking scenarios.

Flush+Reload Flush+Reload attacks are used to gather information about memory accesses on shared libraries [28]. They target the Last-Level Cache (LLC), which means that the attack application does not need to run on the same execution core as the attacked application. First, the shared library under attack is also loaded into the attacker's address space. To reduce the memory footprint, the OS shares identical memory pages between processes. Additionally, security mechanisms (like copy-on-write or read-only) are used to protect changing the shared pages for other processes. By using the processors' "clflush" instruction, the cache is flushed [7]. After the flush, the attacker can access the memory lines under attack again while measuring time. Then, the measured time is classified in either cache hit or cache miss by previously defining a threshold. When observing a cache hit, the attacker learns that the victim has accessed the respective memory line in the meantime.

Spectre Modern processors use branch prediction and speculative code execution to get a performance benefit. This is done by "guessing" a branch, either conditional or indirect, and executing it ahead. For example, if the future control flow depends on an uncached memory value, the processor fetches it instead of going into idle mode ("out-of-order" execution). As soon as the branching point is evaluated, the pre-execution is either used or discarded. This allows the processor to execute the code in an incorrect way, since if the result is not used, it is reverted anyway. However, as Kocher et al. [7] point out, this behavior is vulnerable, since the pre-execution has access to the whole address space of the victim's process and leaves measurable traces in the cache. By tricking the processor into executing an erroneous prediction which depends on an arbitrary memory value of the victim's process' address space, an attacker can obtain this memory value. This is done by previously flushing the cache and subsequently performing a flush+reload attack. By multiplying the data with 4096, data accesses are scattered with a distance of 4 kB. As one memory page is usually 4 kB large, the scattering allows to map one byte to an according memory page. This kind of performance "hacks" can also be found in ARM microprocessors that are used in current smart phone generations.

2.2.1.2. Missing TRNG

As RFC 4086 [29] states, random numbers are required for various cryptographic purposes. This includes private or public key generation, initialization vectors, padding and nonces. The use of pseudo-random processes to generate secret quantities can result in pseudo-security. Two examples are the wrong initialization of OpenSSL's PRNG on Android, resulting to compromised Bitcoin wallets [30], and the reuse of nonces by Sony, resulting in a leakage of their private ECC key used to sign software [31].

2.2.1.3. Countermeasures

There are multiple possibilities to avoid a mapping between side-channel information and the processed data:

Tamper Resistance

Tamper resistance protects a component (in this case the SE) from unwanted tampering (e.g. getting access to the internal state). From the outside, it is impossible to appreciate a chip's level of tamper resistance. Additionally, also the data sheets usually do not contain detailed information about the measures. For consumers, it should still be possible to gain assurance of the implemented security. For example, for devices that include cryptographic, federal government agencies are required to be validated by FIPS PUB 140-2 [32] or Common Criteria (CC) [33], also normed by ISO in ISO 15408. FIPS PUB 140-2 states the following four security levels:

- Security Level 1: lowest level of security, basic requirements
- Security Level 2: require tamper evident coatings or seals
- Security Level 3: attempting to prevent intruder vom gaining access to security parameters
- Security Level 4: envelope of protection around the cryptographic module

During the CC evaluation process, a device is submitted to an evaluation laboratory, that has been accredited by the relevant national certification scheme. In Germany, this is the Bundesamt für Sicherheit in der Informationstechnik (BSI). Along with the device, a set of evaluation deliverables is required. These deliverables depend on the targeted Evaluation Assurance Level (EAL). CC defines the following EALs:

- EAL1: “An evaluation at this level should provide evidence that the device functions in a manner consistent with its documentation”
- EAL2: additional to EAL1, developer testing, a vulnerability analysis and independent testing based on the device specification are done.
- EAL3: additional to EAL2, more complete testing coverage of the security functionality and mechanisms that provide some confidence that the device will not be tampered with during development, are needed.
- EAL4: additional to EAL3, more design description, implementation representation of the device and improved mechanisms that provide confidence that the device will not be tampered, are required.
- EAL5: additional to EAL4, semi-formal design description, a more structured architecture and again improved mechanisms are required.
- EAL6: additional to EAL5, a more comprehensive analysis, a structured representation of the implementation, more architectural structure, more comprehensive independent vulnerability analysis and improved configuration management and development environment controls are required.
- EAL7: additional to EAL6, analysis with formal representations and formal correspondence, and comprehensive testing is required.

Additionally, it is possible to get a +, e.g. EAL6+. This means, that the product is certified as EAL6 but some higher level features are included too. However, if a product achieves a certain EAL, this applies just to the tested version. If something is changed, the product loses it certification [34, 35].

Hiding

The attacks rely on a good Signal-to-Noise Ratio (SNR) of the side-channel measurements. Therefore, a possible countermeasure is to introduce noise to the side-channel. The author of [36] proposes to add noise generators that draw random amounts of power or keep the power consumption constant during the execution. Another option is to use balanced logic styles that use a form of dual-rail pre-charged logic, where each logic gate has two outputs and operates in two phases: reset and evaluate. First, the gates are reset to known values (e.g. `LOW` on both outputs), then one of the outputs transits (e.g. the first for a `HIGH` and the second for a `LOW` state). Finally, the two outputs are evaluated. In both cases, one output changes and therefore it is not possible to infer the internal state as the outputs cannot be distinguished from the outside [26, 27].

Masking

This countermeasure tries to remove the correlation between input data and side-channel emissions. This can be done on a per-word scheme, where the input is masked with a random bit vector before entering a cryptographic hardware block. After the crypto operation, the output must be unmasked. With this mitigation technique, the side-channel emission can be measured but it does not correlate with the actual data and is therefore unusable [27].

Different Cache Architecture

One mitigation is the Newcache architecture, described in [37]. With this architecture, the evicted cache line does not depend on the address or is random. Additionally, hiding access times to prevent the classification between cache hit and miss prevents this sort of attacks [28].

2.2.2. Hardware Security Extensions

In this section, three classic hardware security extensions are presented. As described by Lee [37], the general purpose HW-SW security solution is to use software protection mechanisms and protect these in hardware. Hardware security extensions usually implement the described countermeasures of Subsection 2.2.1.3.

2.2.2.1. Trusted Platform Modules

Trusted Platform Modules (TPMs) are standardized in ISO 11889, which is separated in four parts: Part 1 [38] is about the architecture, part 2 [39] about definitions which are used to communicate with a TPM, part 3 [40] shows the possible commands of a TPM and finally, part 4 [41] contains C code that describes the algorithms and methods. However, the original design comes from a consortium called Trusted Computing Group (TCG). The current version of the specification is 2.0 and is also published by the TCG. In comparison to the previous version 1.2, AES-128 became mandatory, while AES-256 is still optional. Additionally, SHA2-256 and HMAC-SHA2-256 became mandatory.

ISO 11889-1 [38] describes that TPMs are designed to provide three roots of trust: storage, measurement and reporting. This is done by implementing hardware protection mechanisms. In general, TPMs are designed as passive components which reply to received commands. The Root of Trust for Storage (RTS) provides commands for creating, managing and protecting cryptographic keys and other data. The Root of Trust for Measuring (RTM) is intended to reflect what software is running on a platform. The TPM is used to start a "chain of measurements". Before a piece of software is started, it is measured and added to the chain of measurements. When the measurements are started with the power-on of a system, the chain contains the measurements of each running software at any time. Modifications on one software piece leads to a measurable alteration of this chain. Together with the root of trust for storage, it is possible to restrict access on data (and keys) to software with specific measurements. This provides separated memory for different concurrently running applications and additionally protects tampered applications from accessing data. The Root of Trust for Reporting (RTR) allows external entities to prove that a certain measurement or data exists in a TPM. A TPM has an Endorsement Key (EK) that is essentially a RSA key pair and identifies the chip. The EK is used to prove if a key exists in a TPM.

A TPM is usually implemented as a physically independent unit and is then connected to the host system. However, it is also possible that a TPM implementation is running on the host processor, when the processor is in a special execution mode [38].

2.2.2.2. Smart Cards

Smart Cards (SCs) are embedded computers in the form factor of a credit card [42]. The integrated Microcontroller Unit (MCU) allows the SC to execute cryptographic algorithms, which can be either implemented in software or on a dedicated hardware. These MCUs usually implement Java Card and have Global Platform support to manage the content of the card. More advanced ones additionally include cryptographic coprocessors for common symmetric and asymmetric crypto operations.

Usually, smart cards are distinguished between contact-based and contactless cards. Contact-based cards have several electrical contacts to communicate with the reader. Contactless cards have an internal antenna to communicate by an electromagnetic field, created by the reader [26].

SCs are similar to TPMs, but they are user-bound instead of hardware-bound.

2.2.2.3. Hardware Security Tokens

Another form of hardware security are Hardware Security Tokens, or sometimes only “Hardware Tokens”. These embedded systems attack the increasing concerns regarding identity theft and are therefore used to authenticate an identity. The common use case is to connect them to a computer (usually by USB) and let a certain piece of software communicate with it to prove the identity. The tokens are usually based on a MCU that can be found in a common Smart Cards, but with an additional USB interface and therefore also another form factor [43].

2.3. Background

This chapter gives background information about the topics treated in this thesis, backed by literature, norms and specifications.

2.3.1. Machine-to-Machine Protocols

In this section, three communication protocols are described. The first is a legacy protocol invented for semiconductor manufacturers. The other two are state-of-the-art and designed for any purpose by involving the previous described communication paradigms.

2.3.1.1. SECS/GEM

SECS/GEM was designed in the 1980s by Semiconductor Equipment and Materials International (SEMI). It is a proprietary protocol stack, developed for semiconductor manufacturers. A SECS/GEM communication has exactly two participants: the host and the according equipment.

SECS/GEM messages contain two numbers to specify the type of the message: the stream number and the function number. The stream numbers mark if the message concerns the equipment (1 or 2), the material (3 or 4) or an alarm (5). While odd function numbers mark requests, the subsequent even numbers mark the corresponding response. Additionally, a message contains the message length (4 bytes), additional header bytes, such as the receiver identification, type of message and sequence number (summed up to 10 bytes) and the payload (up to 8 megabytes). The specification defines the respective payload for each stream and function number. The syntax format is SEMI Markup Language (SML), where elements are noted in angle brackets. One remarkable attribute of SECS-II is the awareness of the data type that is sent at the begin of each variable. The Generic Equipment Model (GEM) specification additionally allows alarms and events. Both are asynchronous messages from the equipment to the host but they have to be enabled explicitly from the host [44, 45].

Underlying Protocol Stack

The protocol stack is divided in three layers:

- Application Layer: defined in SEMI E30 GEM [46].
- Message Layer: defined in SEMI E5 SECS-II [47], uses SML to define the message format.
- Transport Layer: The first transport layer protocol was defined for RS-232 connections and specified in SEMI E4 SECS-I [48]. The communication by TCP/IP is specified in SEMI E37 HSMS [49]. High-Speed SECS Message Services (HSMS) is connection-oriented. Its messages are encapsulated in stream number zero and concern connection establishment, testing and tear-down [44].

Security

Security measures are not considered in SECS/GEM. The communication with the HSMS sub-protocol relies only on TCP/IP and therefore does not provide any optional measures either [44].

2.3.1.2. MQTT

Message Queue Telemetry Transport (MQTT) is a lightweight publish-subscribe messaging protocol with focus on limited bandwidth networks. It is specified by OASIS as open standard and is currently available in version 3.1.1 [50]. OASIS is a non-profit consortium that promotes industry consensus and produces worldwide standards for different technological areas, such as the IoT or security [51].

There are two entities in MQTT: clients and brokers. A MQTT broker is used to manage clients and created topics. A MQTT client connects to a broker, receives a list of topics and subscribes or publishes to one. Topics are organized in hierarchies separated by slashes, for example *working-pc1/sensors/temperature*.

Underlying Protocol Stack

The common underlying protocol stack is TCP/IP. However, any other network protocol that provides ordered, lossless and bidirectional connections can be used, e.g. WebSockets [50].

Message Structure

Messages are packed in MQTT control packets, as described in the specification [50]. These are structured in a fixed header, a variable header, and the payload. The fixed header is always sent to the receiver. The four most significant bits of the first byte contain the packet type. The other four bits are used as flag for the respective type. The second and the following bytes are used for the length of the remaining packet.

Security

By default, MQTT does not provide any mechanisms for confidentiality and authentication. For this reason, TLS can be used, as it provides the required properties for the underlying protocol stack and gives the desired security properties. Additionally, the CONNECT message contains fields for an user-password combination. Due to that, it is possible to integrate an existing authorization system, e.g. LDAP, into the MQTT server. For lightweight implementations, cryptographic algorithms can be used directly on the payload instead of using the whole TLS stack [50]. Nastase [52] describes further details and comparisons with other protocols.

2.3.1.3. OPC UA

OPC UA is a system architecture, designed for the interaction between sensors, actuators, control systems, Manufacturing Execution Systems (MES) and Enterprise Resource Planning (ERP) systems. It is standardized in IEC 62541. The infrastructure model is defined as follows [53]:

- The information model to represent structure, behaviour and semantics.
- The message model to interact between applications.
- The communication model to transfer the data between end-points.
- The conformance model to provide interoperability between systems.

Underlying Protocol Stack

The underlying protocol stack of OPC UA is exchangeable. Currently, there are three different possibilities:

- Transmission Control Protocol (TCP)
- HyperText Transfer Protocol Secure (HTTPS)
- WebSockets

An illustration can be seen in Figure 2.3.1. The TCP stack is the most lightweight and is recommended for embedded systems.

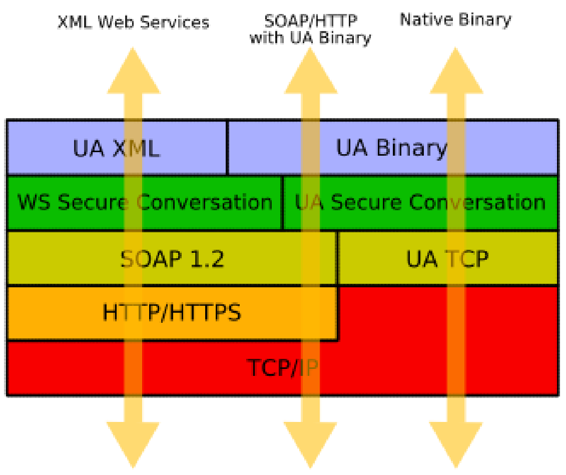


Figure 2.3.1.: Protocol stack of OPC UA. Illustration taken from [54].

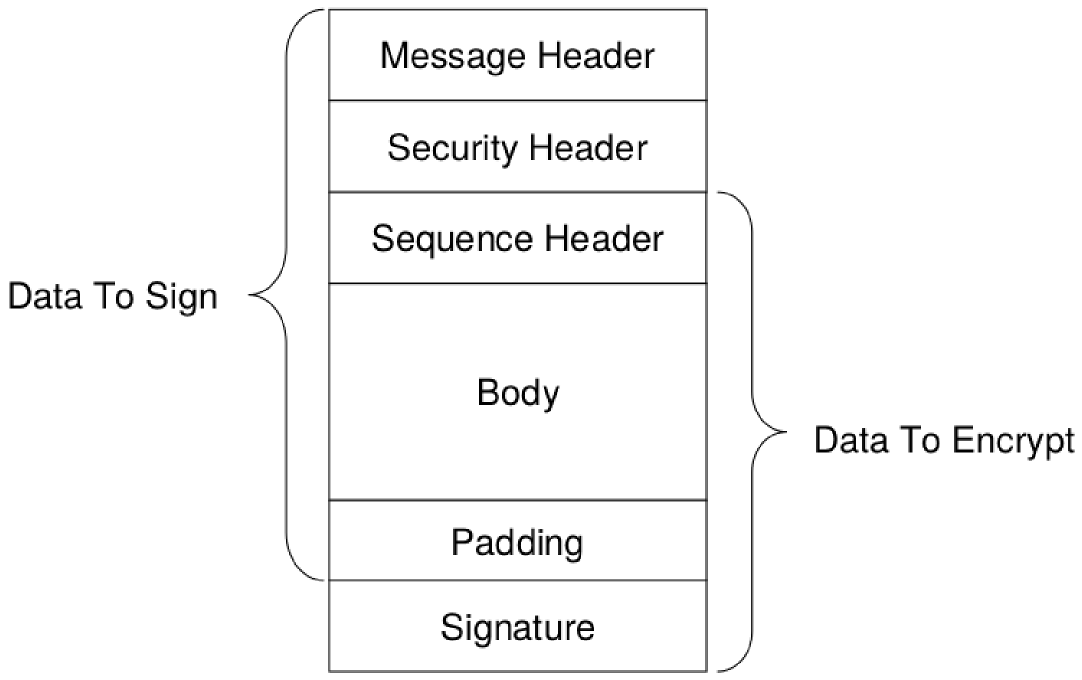


Figure 2.3.2.: Message structure of OPC UA. Illustration taken from [55].

Message Structure

OPC UA messages have three different headers, a body (with padding) and a signature:

• Message Header:	This header contains a message type, which can be either MSG (normal message), OPN (OpenSecureChannel) or CLO (CloseSecureChannel), and the size of the message.
• Asymmetric Security Header:	This header is used during the open secure channel procedure (OPN type). It contains the intended security policy and the sender's certificate.
• Symmetric Security Header:	This header is used with the MSG type. It just contains the token identifier that is used to identify the communication partner.
• Sequence Header:	This header contains a sequence number to keep track of the correct order of the messages, and a request ID if the request is split up in more than one request or response message.
• Padding:	The padding ensures that the message length is a multiple of the encryption block size.
• Signature:	The signature is used to verify the authenticity and integrity of the message.

The message structure can be seen in Figure 2.3.2.

Additionally, three different data encodings are defined in the specification:

- Extensible Markup Language (XML)
- JavaScript Object Notation (JSON)
- UA Binary (native binary)

The native binary encoding is the most lightweight and is recommended for embedded systems.

Security

The security part of OPC UA is specified in the second part of the specification [56]. As written there, established best practices, like the Special Publication 800-131A [57] published by NIST, were minded during the design stage of the protocol. OPC UA addresses security objectives at different levels. On the communication layer, "secure channels" should provide confidentiality, integrity and app authentication. One layer above, on the application layer, sessions take care of user authorization and authentication. These security features bring confidentiality, authenticity, and integrity during the communication.

Before a secure channel is established, both parties have to negotiate a common security policy. The security policy defines which security mechanisms should be used to protect further communication. These mechanisms also include the cryptographic algorithms and key sizes. Since the computing power of computers is continuously increasing, these will change over time. Therefore, it is important to regularly review the implemented security policies and compare them with the recommendations of the OPC Foundation [58].

Additionally, the Bundesamt für Sicherheit in der Informationstechnik (BSI) carried out a study on the security measures of OPC UA [59]. At first, the specification of the version 1.02 was analysed with focus on systematic errors. This a threat analysis and a detailed analysis of the specification. Additionally, the reference implementation of the OPC Foundation was tested with static and dynamic code analysis and fuzzing methods. The results show that OPC UA, in contrast to many other industrial protocols, is able to provide a high level of security.

Security Policies

A security policy contains the following security-relevant specifications:

- symmetric signature algorithm
- symmetric encryption algorithm
- asymmetric signature algorithm
- asymmetric encryption algorithm
- key derivation algorithm
- certificate signature algorithm

Additionally, a security policy specifies the key length of the signature algorithm, the minimum and maximum length of the asymmetric key and the length of the random nonce.

Currently, the following security policies are supported:

- None
- Aes128-Sha256-RsaOaep
- Basic256Sha256
- Aes256-Sha256-RsaPss

Also “None” is a valid option, even though it is not recommended. Currently, Aes256-Sha256-RsaPss is the most secure security policy available in the OPC UA portfolio. The specified cryptographic algorithms of this security policy can be seen in Figure 2.3.1.

Type	**Aes256-Sha256-RsaPss**
Symmetric Encryption	AES256-CBC
Symmetric Signature	HMAC-SHA2-256
Asymmetric Encryption	RSA-OAEP-SHA2-256
Asymmetric Signature	RSA-PSS-SHA2-256
Key Derivation Function	P-SHA2-256
Certificate Signature	RSA-PKCS15-SHA2-256

Table 2.3.1.: Table of the “Aes256-Sha256-RsaPss” OPC UA security policy.

2.3.2. Cryptographic Operations

In this section, the cryptographic operations used in the security policies of OPC UA are described.

2.3.2.1. Asymmetric Cryptography

Asymmetric cryptography, also known as “public-key cryptography”, requires a key pair: the private and the corresponding public key. While the public key must be known by other parties, the private key needs to be kept secret under any circumstances. Today, there are two common families of asymmetric algorithms: Rivest-Shamir-Adleman Crypto Scheme (RSA), which is based on the integer factorization problem and Elliptic Curve Cryptography (ECC), which is based on the discrete logarithm problem. For now, ECC algorithms are not included in the OPC UA security policies. Since textbook RSA has several weaknesses, an additional padding must be used. One of the weaknesses is determination, which means that each input, used with the same key, produces the exact same output. This circumstance can be used by an attacker to recognize known ciphertexts. For this reason, probabilistic schemes are introduced [60].

The usual process of encryption is to craft a ciphertext with the public key of the receiver. The other party then can decrypt the ciphertext with its own private key. However, a signature is generated with the own private key. The signature can then be verified with the corresponding public key of the signee [60].

2.3.2.2. Certificates

One problem with asymmetric cryptography is that public keys are, by default, not authenticated [60]. Therefore, an attacker could start a person-in-the-middle attack and replace a public key with its own. In this scenario, the verifier would not be able to check if the public key really belongs to the correct person. This is one of the challenges that has to be considered when using public key cryptography. An approach that tackles this problem is to exclusively use authenticated channels for communication, but this might not always be possible. Instead, the common approach is to use certificates. Certificates use signatures to provide authentication. However, if a person uses its own key ("self-signed" certificate), the same problem occurs again. Therefore, a mutually Trusted Third Party (TTP) is introduced, which is referred to as Certificate Authority (CA).

Certificate Authority & Certificate Signing Request

The tasks of a CA are distinguished in two use cases: either an user provides an own key pair or the CA is in charge of generating one. The first case again has the authentication problem, therefore it is crucial that a Certificate Signing Request (CSR) is transmitted over an authenticated channel. The second use case also requires a secured channel, since the private key is transmitted too. Another crucial point is the transmission of the public key of the CA, which again requires authentication. Without a CA, an user requires an authenticated channel whenever a new communication partner is introduced. With a CA, the authenticated channel is just required during the setup time, the further authentication is handled by signature verification ("chain of trust").

This entire CA system and its mechanics is referred to as Public-Key Infrastructure (PKI). In practice, certificates have a defined structure and are normed, e.g. X.509 certificates, which are published in RFC 5280 [61]. X.509 is a widely used standard, e.g. in TLS.

2.3.2.3. Symmetric Cryptography

In difference to public-key cryptography, symmetric cryptography uses the same key for the cryptographic operation and its inverse, e.g. encryption and decryption. A known problem with symmetric cryptography is the key distribution problem that occurs when the participants are not aware of the secret key a-priori. Since the communication channel is always assumed to be untrusted and unsecure, the secret key cannot be sent over it. Therefore, the problem of sending a confidential message can be reduced to transmitting and storing the secret key confidentially. The common solution is to use asymmetric cryptography to sent or negotiate a symmetric key.

Symmetric encryption can be done with either stream or block ciphers. Stream ciphers encrypt the input bitwise, while block ciphers have a defined block size that indicates the number of bits that can be encrypted at once.

2.3.2.4. Authenticated Encryption

A secure channel is a connection between two parties which provides confidentiality and authentication. Usually, this is done by combining an encryption scheme with a MAC, but there also exist specially designed crypto schemes. There are three possibilities to combine encryption and MAC algorithms:

- Encrypt-then-MAC (EtM)
- MAC-then-Encrypt (MtE)
- Encrypt-and-MAC (E&M)

The first approach has two advantages: it is theoretically secure and is more efficient since messages can be discarded without decrypting it if the authentication fails. The second approach can have weaknesses depending on the implementation (e.g. padding oracle attacks). The last approach is more efficient since both operations can be done in parallel. However, since the MAC is directly created from the plaintext, it could leak information [62].

The common chosen approach is the first one but in some use cases authentication is more important than confidentiality (being able to modify a message vs. knowing a message) and therefore the second approach is used.

Padding Oracle Attacks

With the MtE approach, the verification algorithm decrypts the ciphertext, inserts the decrypted message into the MAC function and verifies the output against the provided signature. After the decryption, the padding has to be removed to get the actual message. Now, if the padding check errors out before the MAC process is done, the attacker is able to distinguish between correct and incorrect padding based on the needed time. This allows the attacker to craft messages and calculate bytes of the message by measuring time [63, 64, 65].

2.3.2.5. Random Number Generation

Random numbers play a major role in cryptography. There are three types of Random Number Generators (RNGs): True Random Number Generators (TRNGs), Pseudo-Random Generators (PRNGs) and Cryptographically Secure PRNGs (CSPRNGs). TRNGs create outputs that cannot be reproduced.

P-SHA2-256

The “P-SHA2-256” algorithm, which is mentioned as Key Derivation Function (KDF) in the OPC UA security policies, is actually the Pseudo-Random Function (PRF) from the TLS 1.2 specification, published in [66]. This PRF is defined recursively and allows to generate pseudo-random sequences of arbitrary length. For the construction, it uses the HMAC algorithm. The involved hashing function should provide the statistical properties and the involved secret should provide the unpredictability (under the assumption that the secret is confidential).

2.3.3. Java Card

Nowadays, MCUs of SCs commonly implement a Java virtual machine that interprets Java Card, a subset of Java. Java Card is designed and maintained by Oracle [67]. An illustration of the JC architecture can be seen in Figure 2.3.3. However, Java Card is designed as multi-application environment and therefore it is possible to store several applications on one system. The Java Card Runtime Environment (JCRE) is responsible for the execution of the selected application. Applications make use of the Java Card Classic APIs that provides smart card-specific functionalities. These APIs are provided by the Java Card Virtual Machine (JCVM) that implements them either in native code or in Java Card. Usually, the implementation in native code leads to a better performance and is therefore used for resource-intensive operations. The cryptographic operations are usually implemented in hardware or an external crypto coprocessor. The APIs to these are also provided in native code. The lowest layer is the Smart Card Operating System (SCOS), which is also implemented in native code [68].

2.3.3.1. Global Platform

Global Platform (GP) is an organization that has been established to provide a global infrastructure for smart card implementation across multiple industries [69]. Initially, the specification was developed by Visa International and then donated to the GP consortium. The GP specification for smart cards targets multi-application environments including application portability between different card-specific implementations.

In general, it is always possible to install an application, except the card is in the “locked” or “terminated” state. Before an applet can be installed, it has to be loaded to the persistent memory of the card. The executable load file (“CAP” file format in Java Card) is loaded securely to the card, meaning the communication is encrypted and additionally the integrity is checked.

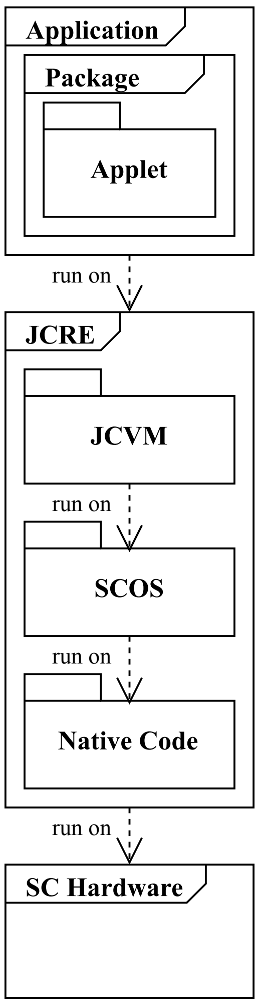


Figure 2.3.3.: Illustration of the Java Card architecture.

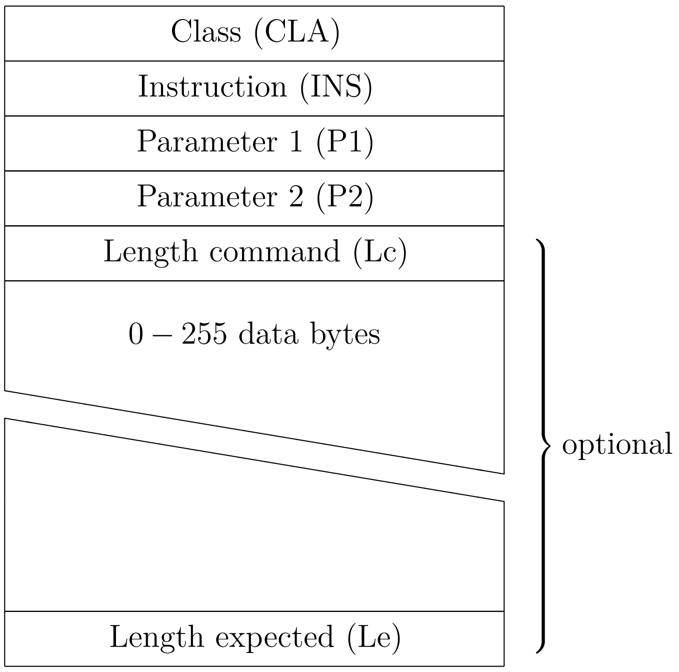


Figure 2.3.4.: Illustration of a command APDU (on byte level).

2.3.4. Smart Card Communication

The ISO 7816 norm contains specifications on chip cards. This specification is split into 14 parts. In the following sections the communication between Integrated Circuit Cards (ICCs) and reader/hosts is explained.

2.3.4.1. Application Protocol Data Unit Messages

The communication with a smart card, as described in ISO 7816 part 3 [70] and part 4 [71], is based on request-response. The smart card receives the command APDU and answers with a response APDU.

Figure 2.3.4 shows the structure of a command APDU. The data per packet can just be 255 bytes long as this is the maximum length specification (Lc has a size of one byte). Even though the Lc byte is named “length of command”, it specifies only the amount of data bytes [71].

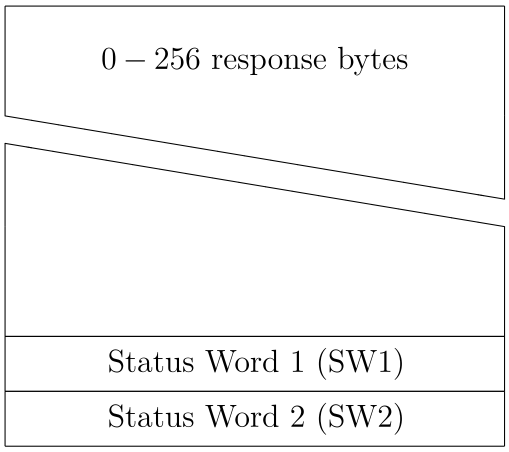


Figure 2.3.5.: Illustration of an APDU response packet (on byte level).

There are four different cases of APDU commands:

- Case 1: Header, containing CLA, INS, P1 and P2 byte.
- Case 2: Header and expected response length byte.
- Case 3: Header, length of data and data.
- Case 4: Header, length of data, data and expected response length.

The class byte indicates the class of the command. The MSB distinguishes between proprietary and inter-industry class. If it is not set, the inter-industry class is active and therefore only the specified instructions can be used. The class byte also indicates if secure messaging is active. Secure Messaging protects all request-response pairs with confidentiality and authentication [71].

Figure 2.3.5 shows an APDU response packet. The response has a possible length of 256 bytes.

2.3.4.2. Contact-based Communication

A chip card has five electrical contacts: two are used for power supply (Vcc/Vpp), one ground, a clock, a reset and an I/O pin. This is defined in the second part of the ISO 7816 norm. The clock varies between 1 MHz and 5 MHz, and is typically set to 3.57 MHz [72]. The other two pins are used for communication.

2.3.4.3. Contactless Communication

NFC is an international standard for contactless communication. It is an amendment to the existing contactless smart card systems but still is compatible to them [73]. NFC is presented in ISO 18092 and compliant to ISO 14443 and Sony's FeLiCa, as well as its own communication method.

Data is exchanged with two inductively coupled coils, generating a magnetic field with a frequency of 13.56 MHz [73]. For the transfer, the field is modulated. One device acts as the initiator, where the other device operates in target mode. In passive mode, only the initiator produces the carrier field. The target, which is introduced into this field, uses it to draw energy and communicates with the initiator by load modulating the field. The initiator communicates by directly modulating the field.

One important property of contactless communication is the possible range between the reader and the card. Usually, the card types are distinguished between proximity (up to 10 cm) and vicinity (up to 1 m). The contactless proximity objects are specified in ISO 14443. Analogous to CB, part 3 [74] and part 4 [75] specify the communication.

2.3.4.4. USB Communication

As described in the USB Chip Card Interface Device (CCID) device class "smart cards" specification [76], the communication is separated in two parts. First, the communication between Integrated Circuit Card (ICC) and CCID, then between CCID and USB host. A CCID device can have multiple slots and might have no, one or more than one chip cards inserted. Anyway, it first communicates with the host and informs it about the capabilities and requirements. After that, the CCID looks for present cards and informs the host about them. Usually the connection between CCID and ICC is based on a contact-based connection standard such as the communication according to ISO 7816. Then, the host is able to communicate with the connected ICCs. A special case are so called "dongles" that already implement the CCID interface on the ICC. This allows a communication between host and ICC without additional CCID device.

2.4. Related Work

In this section, related projects and publications are discussed. The research includes the following emerged topics based on the previously discovered state of the art.

2.4.1. Security Challenges

Sadeghi et al. [77] state that the security concepts of classical IT systems can not be mapped to the IIoT easily. As example, IIoT systems usually have high-availability as fundamental requirement and therefore systems cannot be simply disabled (usual tradeoff between security and availability). Additionally, design and configuration data (intellectual property) must be protected. However, they also state that common security architectures such as software-based isolation and virtualization, is too complex for low-end systems. Instead, they propose different solutions based on hardware-enforced isolation of security-critical code and data from other software on the same platform. Additionally, they conclude that the existing security solutions are inappropriate and further research needs to be done, such as developing novel isolation primitives which are resilient to run-time attacks.

Bou-Harb [78] describes that integrating legacy networks will not just introduce opportunities but also pave the way for exploiters. Additionally, the author states that SCADA systems aim to deliver sensitive control information to the CPPS, which is critical for the reliability of these. As example, an attacker could exploit weaknesses in a communication protocol to alter information and therefore threat the whole facility. However, the author does not provide a concrete solution but concludes that many approaches are too theoretical. These include machine learning approaches to identify outliers or invalid data.

For certain communication methods, a full key-generation might not be applicable or possible. One example is MQTT, where the communication is based on an entity-in-the-middle, the broker. The security concept of MQTT does not take care of end-to-end communication between publisher and subscriber. Instead, the secure channel is negotiated with the broker (usual TLS client-server approach). To still get the desired secured communication between the actual end-points, other solutions have to be considered. One possible way is to perform an a-priori key distribution. Nonetheless, the customer, respectively the end-user, should be in full control of the keys. Therefore, the keys could be distributed “off-channel”. One of the published solutions is [79], where the keys are distributed by NFC. To achieve this, the new keys are generated in the back end, which is located in a secured and trusted environment. There, the new keys are secured by AE, using the old keys. This can be done by combining symmetric encryption methods with symmetric signature methods. In this case, AES256-CBC and HMAC-SHA2-256 are used. The initial keys come from the equipment vendor and can be exchanged as soon as the equipment is received.

The equipment is a tamper-resistant Secured Element (SE) with NFC interface. The SE is also used for cryptographic operations.

Ulz et al. [80] describe a way to securely update and configure IoT devices in a personal as well as an industrial environment by using NFC as communication channel. They propose three different entities: a configuration back end, a configuration device and the IoT device which should be updated. The usual approach is to connect the configuration device (e.g. a smart phone) with the IoT device (by NFC), fetch the identification of the IoT device (by NFC), get the new configuration for the respective device from the configuration backend (by 4G or WiFi), and finally transmit the received configuration to the IoT device (by NFC). The authors also use a SE as tamper-resistant hardware as NFC receiver of the IoT device. However, this project is enhanced in the first demonstrator of this thesis. Instead of the "bridged" compound of the SE and a generic MCU, the proposed security stick with native USB interface is used.

2.4.2. Network Architecture

Weyer et al. [81] provide the architecture of a modular, multi-vendor production system. Their proposed initiative "SmartFactory ^{KL}" can be seen recognized as the first European vendor-independent factory and therefore was in the lead to define standards and concepts regarding the interoperability of systems and integration into a factory. They mention three communication standards in combination with Industry 4.0: OPC UA, RFID and web server technology. Each production machine is equipped with a RFID tag and a reader. These are used to sense the local neighborhood for adjacent modules. This information is then forwarded by OPC UA to a main server. The ultimate goal of their work is to use exchangeable modules within the production line and enable an automatic detection and integration of these ("plug and produce"). They conclude that OPC UA is the major part of the interoperability model. The authors also mention Loskyll et al. [82], who describe a way to store the production process and configuration parameters of a product on a RFID tag which is then connected to the product itself.

Jiang et al. [83] state that a complete industrial network consists of at least four parts: the corporation network, the Demilitarized Zone (DMZ), the process network and the control network. However, the interconnection of these networks also increases the potential malicious threats. They consider just confidentiality to protect the sensitive data, while neglecting integrity and authentication due to the energy and real-time constraints.

2.4.3. Avoiding Isolated Systems

As described in [22], moving away from isolated systems to open, interconnected systems is one of the big challenges tackled by Industry 4.0. The author of [5] presents an overview of cyber threats and vulnerabilities that affect the new trend. One example is “Stuxnet”, a computer worm reportedly designed to take over the control of a field network.

The de facto industrial protocol standard for field busses is Modbus. The devices are connected by either serial buses or TCP/IP-based communication channels. It operates according to a classic master/slave principle, where only the master can start an interaction. Usually, the request-response message paradigm is used. As the author of [5] points out, typical problems during the migration to newer protocols are a lack of authentication between the active components. Since the control network has always been a closed environment, there was no need to integrate authentication mechanisms among the different elements. Also, the separation between process network and control network was very weakly designed. Additionally, everything that messes with productivity or availability is not popular. This includes security updates, which might require a reboot, or anti virus software that might interrupt the running processes. Finally, there might also exist attack vectors in the communication protocol, even by design. Modbus, for example, has several vulnerabilities, e.g. the lack of confidentiality and integrity of messages. Additionally, the entities are not authenticated and the protocol does not incorporate any anti-repudiation or anti-replay mechanisms.

The author of [5] suggests to avoid direct connections from the process network and the Internet. Additionally, a controlled access from the office intranet to the control network should be possible. Since the office network traditionally has access to the Internet, and therefore the office PCs could be used as a bridge by attackers, additional authentication mechanisms should be enforced. Additionally, secured mechanisms should be established for remote control and maintenance.

2.4.3.1. Heterogeneous Data Representation

Currently, nearly each protocol uses its own data representation. Therefore, the data somehow needs to be converted between the different representations.

Eliasson et al. [84] compare XML-based technologies and state that XML in general lead to a high communication overhead due to the verbose nature of it. They propose a differential binary delta-encoding which reduces the overhead significantly. However, OPC UA also provides a binary encoding which already reduces the overhead and is suggested for networks including embedded devices.

2.4.3.2. IoT Gateways

As IoT gateways are one of the open challenges, several approaches came up the recent years. However, most of them do not focus on security or keep security measures open for future work.

Guoqiang et al. [85] propose a smart IoT gateway to integrate multiple heterogeneous perception networks into the network layer of a company. The gateway provides a “pluggable architecture”, allowing to insert multiple different hardware modules (PCI-E cards) to support different communication protocols such as RFID and WiFi, depending on the requirements. Additionally, they use a flexible but self-developed protocol to translate data in a uniform format. The integrated CPU communicates with the user cards by Ethernet, USB or UART. The upstream communication is either by Ethernet, 3G or RS485. However, they do not use a client-server architecture but a browser-server, based on HTTP requests, and do not use any security policies (future work).

Nuratch [86] shows an IIoT gateway which is able to communicate with Modbus/TCP devices. The gateway is implemented on a MCU with connected WiFi module which enables an upstream connection to the network layer, using the WebSocket protocol.

Li-Hong et al. [87] describe the IoT Gateway “high efficiency, high reliability, real-time response, low power consumption, anti-interference ability and good versatility” protocol converter. The protocol conversion happens in two modules: the protocol conversion layer, which is connected to the network layer, and the protocol adaption layer, which is connected to the perception layer. The protocol conversion control layer has two tasks: packaging the data and converting signals and control instructions between the adjacent layers. Then, the protocol adaption layer defines a standard interface to the perception layer, allowing to use the previous created uniform data and signaling.

Yacchirema et al. [88] provide a gateway for pervasive smart environments. Their key component is called “Gateway for Technical and Syntactic Interoperability” (GW-TSI), which transforms, processes, stores and delivers data. Their target is to integrate wireless sensors and actuators that use communication protocols based on different wireless communication protocols such as MQTT over WiFi or over BLE. However, the GW-TSI converts the incoming data to a uniform data format, namely JSON, to enable the use of heterogeneous formats.

Datta and Bonnet [89] describe a smart gateway compliant to the standard recommendations of oneM2M. They base their gateway on RFC 6690 (Constrained RESTful Environments (CoRE) Link Format) and RFC 8438 (Sensor Measurement Lists (SenML)). With these proposed standards, they are able to implement the gateway as RESTful web services, where the structure of the data is specified by SenML.

OPC UA as Wrapper for Legacy Protocols

Beside of different other approaches, more and more projects include the upcoming OPC UA protocol as main communication protocol of their network layer. OPC UA comes with an own data representation and allows different encodings, like JSON and binary.

Toc and Korodi [90] show an approach where the Modbus protocol is wrapped by an OPC UA server. With their approach, the wrapper has to interact with the local automation units by implementing a Modbus client (master) interface to take-over and process data. From the Modbus client, the data is then filtered, processed, structured and finally injected into the nodeset of the wrapper's OPC UA server.

Nsiah et al. [91] provide an open-source retrofit kit which enables Small and Medium-sized Enterprises (SMEs) to evaluate the capabilities of Industry 4.0. Therefore, the framework should be easy integrateable, customizeable and extensible. To allow an easy integration into existing production facilities, the authors use field bus sniffers that bypass the actual PLCs and therefore prevent actual modifications of the running systems. Then, they provide a component that automatically maps the data coming from the field busses to high level data models (OPC UA nodes). Sensor values are represented as OPC UA variable nodes which contain additional semantic information such as data type and unit.

Pessoa et al. [92] describe a slightly different approach to the integration process. They virtualize existing industrial legacy networks (based on OPC) and simulate the integration before performing it on the real networks. Then, the networks are connected to a cloud-based IoT platform, by either using OPC UA or MQTT as communication protocol. Data communicated by MQTT is encoded in the JSON format.

Capa et al. [93] propose a rapid prototype that is used for PLC-to-cloud communication. For this purpose, a Raspberry Pi is used as gateway. The gateway works as OPC UA client on the perception layer and as MQTT publisher on the network layer. However, they also state that it is not feasible to transmit the whole data to the cloud but pre-filter and pre-process on the edge/gateway layer.

Cho and Jeong [94] provide an OPC UA-based gateway for legacy communication protocols such as Modbus. Additionally, they integrate MQTT as upstream protocol to the cloud. However, the focus of the authors lies on using ARM processors instead of x86 processors and how this results in cost reduction, based on factors such as acquisition cost and energy consumption.

Garcia et al. [95] present a low-cost platform for collecting data from the plant's floor processes and transmitting commands to control devices. This is done by integrating a Modbus/TCP-based industrial network into an OPC UA server. To integrate the field data, the authors use a XML-based configuration file. The configuration specifies how the data looks like (data mappings), which field devices are accessible and where the field devices and the according data can be found (address space).

OPC UA as Replacement for Legacy Devices

Veichtlbauer et al. [96] describe how OPC UA can be used to integrate field busses into an OPC UA network. Their approach is to directly implement the OPC UA server on the legacy machines, as a replacement for the legacy communication protocol. They conclude that this approach is feasible but OPC UA has a big footprint which is a disadvantage on the field/perception layer, especially on legacy hardware. However, the use of OPC UA becomes feasible on newer platforms as they are usually more powerful.

2.4.4. M2M Protocols including Hardware Security Extensions

As securing upcoming M2M protocols is an ongoing and open challenge, just one paper about OPC UA including hardware security extension could be found: The authors of [97] designed and implemented OPC UA based on a TPM. At first, the possibilities that could be provided by a TPM, are evaluated. They include remote attestation, random number generation, key generation (and storage) and the usage of the TPM as crypto coprocessor. Remote attestation provides the possibility to give assurance about the trustworthiness of a system. This is done by the RTM of the TPM, which checks the integrity of the platform as soon as the system starts. TPMs usually provide a TRNG, which is based on internal sources of entropy and can be used for different purposes, e.g. key generation. These keys can either be stored in the internal non-volatile memory of the TPM, or an internal Storage Root Key (SRK) can be used to encrypt the keys, allowing to store them on external memory too. Finally, while TPM 1.2 uses encryption algorithms just internally, TPM 2.0 provides a symmetric encryption interface for external usage. They conclude that the asymmetric key material can be generated and stored on the TPM. Additionally, for weak devices, the TPM can serve as crypto coprocessor. Finally, their proposed integration of remote attestation increases the trustworthiness of nodes. However, the TPM is considered as a bottleneck regarding performance.

3. Design

This chapter describes the design phase and the architectural concepts of this thesis.

3.1. System Architecture

In this section, the architecture of the overall system is described. For the architecture, three different entities have to be defined. The first is the security stick, which is used as external security back end. The second is the host library, which is the counter-part of the security stick on the host system. The third is the host itself, which runs the host application as the frontend of the system. A compound of hosts with the security stick in use is named “secured network”. As soon as the host application requires a security-relevant operation, it sends a command to the security back end which processes the request and then responds the results. An illustration of the architecture can be seen in Figure 3.1.1.

The system comprises a separate hardware extension including a SE and the corresponding required communication interfaces. Since in this work a USB-interface is used and the hardware-extension is shaped similarly to a USB-Stick, this hardware-extension is further denoted as "security stick".

3.1.1. Security Stick

The security back end is a separated hardware extension including a SE and the corresponding required communication interfaces. As the security back end should be connected by USB and also the shape of the hardware extension is similar to a USB stick, the hardware extension is further referred to as “security stick”. However, cryptographic and storage operations are implemented here. As there might be several different use cases for the security stick, it must provide high flexibility in terms of programming.

3.1.2. Host Library

This entity implements the glue between the security stick and the host. It is located on the host system and is used for the communication with the security stick, which also includes the marshalling and unmarshalling of the messages. Additionally, the host library provides an API for the functionality of the security stick. As the available commands depend solely on the security back end, a possibility to inform the library about the supported APDUs is required.

3.1.3. Host and Host Application

The hosts are the front ends of the system and run the main applications. The main application uses the equipped security stick and the implemented host library to get access to the secured operations. Beside of that, it implements the required functionality of the respective host. This might be interacting with a connected peripheral or displaying information on a connected display.

Hosts are typically, but not necessarily, running an OS on their hardware.

3.1.3.1. Host Platforms

Since one of the goals of this thesis is high portability, the required platform-dependent code should be abstracted and encapsulated in a way that allows an easy extension to new platforms. Platforms are described by the running OS (if one exists) and the underlying hardware architecture or the respective instruction set. Since OSs usually abstract the hardware and should perform equally and independent of the underlying hardware, it can be assumed that programs running on the x86 version of an OS will also run on the ARM version of it and vice versa. To efficiently manage the different supported architectures, an additional build management tool should be used.

In the following subsections, the used OSs and architectures are enumerated and briefly described.

Microsoft Windows 10

As reference machine, my main developing and working machine is used: A Lenovo Thinkpad T450s, with Microsoft Windows 10 as OS.

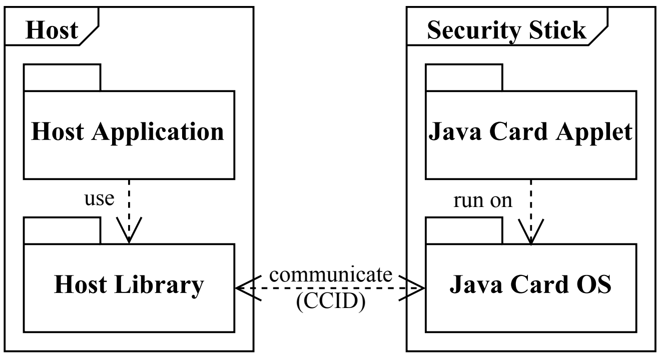


Figure 3.1.1.: This is the general architecture of the system, including the host system and a USB-connected security stick. Security-relevant tasks are shifted into the Java Card applet, which runs on a tamper-resistant SE inside of a JCRE. The shift is done by using the USB CCID protocol, which uses APDUs as message format. The host library implements the PC/SC interface to communicate with the security stick.

GNU/Linux

In this thesis, the Raspberry Pi 3 Model B [98] is used as reference hardware architecture for the arm64 architecture. The running OS is “Raspbian 9.4” (based on Debian Stretch).

For faster development, Windows Subsystem for Linux (WSL) with Ubuntu 18.04 is additionally used on the before-mentioned Windows 10 machine. One disadvantage of WSL is, that for now, the subsystem is not able to access the connected USB devices of the host.

3.2. Security Stick

This section gives information about the design of the security stick.

3.2.1. Requirements

This subsection discusses the requirements and how they should be achieved. The following requirements are considered during the design stage:

3.2.1.1. Tamper-Resistant Platform

The first requirement is a tamper-resistant platform for cryptographic operations and storage. As presented in Section 2.2.1, several attack vectors have to be considered when designing a system that should provide certain security attributes such as confidentiality. Therefore, tamper resistance should be provided by using one of the presented hardware solutions described in Section 2.2.2.

3.2.1.2. Cryptographic Operations

For the first use case, AES256-CBC and HMAC-SHA2-256 are required. For the second use case, an interface to the operations of Table 2.3.1 is required. As elaborated in Section 2.2.2.1, TPMs must implement the HMAC-SHA2-256, but the AES256-CBC is just optional. Also, “P-SHA2-256” is neither specified by TPM nor by Java Card. Anyway, it can be implemented by combining two HMAC-SHA2-256 instances, and therefore has to be composed in the respective application.

3.2.1.3. True Random Number Generator

As described in Subsection 2.2.1.2, random numbers are an essential component of cryptographic schemes. Since the security stick should be useable on multiple platforms, and especially small embedded systems that do not provide any own sources of entropy, an on-board TRNG is required.

3.2.1.4. Flexibility

Since the security stick should be useable for more than one use case, a flexible platform is required. An advantage of smart card security controllers with a running Java Card OS is that they can run multiple applications where each is specially designed for exactly one use case. In comparison, TPMs provide a specified API that is static. Indeed, the API is designed in a generic and multi-purpose way, but some implementations are combinations of more than one implemented crypto primitive. With the current TPM design, the intermediate values between the building blocks have to leave the secured and trusted TPM environment and are re-entered for the next step. This might unnecessarily leak information to the untrusted host environment.

3.2.1.5. Communication Interfaces

For the two use cases, the security stick must be able to communicate based on USB and ISO 14443. Also, an incoming ISO 14443 communication should interrupt an communication ongoing via an existing USB connection. This is useful, since the USB connection is usually static and the NFC communication is just required on events, such as a configuration update in the first use case of the security stick. Both communication types are untypical for TPMs, which are usually connected by a bus,

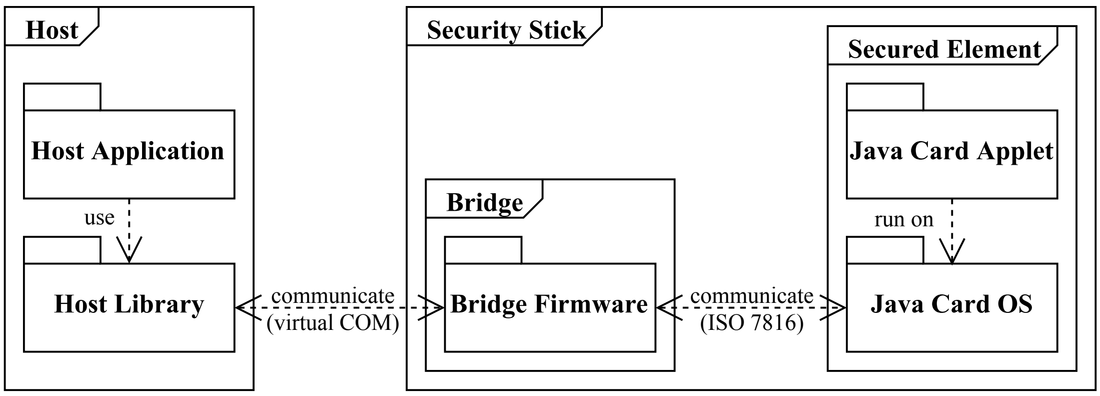


Figure 3.2.1.: On the left, the host system with a running host application which uses the host library to communicate with the connected security stick PCB by virtual COM. The security stick consists of the generic microcontroller (Bridge) and the SE, which communicate by ISO 7816. The Bridge Firmware forwards messages from one side to the other. The SE runs a Java Card Applet which runs in the JCRE, provided by the Java Card OS.

such as SPI, and do not support any type of NFC communication. However, several hardware security tokens support both interfaces.

These requirements lead to the decision, that either a smart card security controller with integrated USB interface or a hardware security token should be used as the SE of the security stick. Also, the security controller should provide a high security standard and therefore should come with a CC EAL 6 or higher certification.

3.2.2. Initial Development Platform

The design of the first prototype is based on off-the-shelf products. A smart card security controller from Infineon Technologies is used as SE. The chosen SE is certified as CC EAL 6+. Since smart cards usually do not have a USB port, the controller does not provide a native USB interface. To still be able to use USB, an additional “bridge” is necessary. The bridge must be able to communicate with the host by USB and with the security controller by ISO 7816. For this reason, a generic microcontroller with native USB support is used. An ISO 7816 interface is not necessary, as the communication can be emulated with a common UART interface. The chosen generic microcontroller is also a product of Infineon Technologies. An illustration can be seen in Figure 3.2.1.

Instead of using the described CCID protocol from Subsection 2.3.4.4, a simpler variant is used for this prototype: a virtual COM port. This is an emulation of the common serial communication but uses the USB interface on the host side.

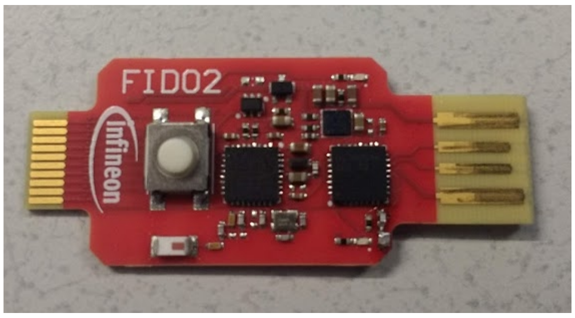


Figure 3.2.2.: Picture of the used security stick. Although "FIDO" can be seen on this and further photos, the hardware extension is used with a different, customized firmware.

The driver is part of the used OS and the device can be used without any further configuration.

After the virtual COM port is enumerated by the host OS, the host application is able to transmit commands to it. An additional byte dedicates the receiver: either the bridge or the security controller. This allows additional control mechanisms from the host, like an ISO reset of the SE. However, the commands are already sent in the APDU format.

The chosen security controller runs the common OS for smart cards: Java Card with GP, as described in Subsection 2.2.2.2. This setup allows to load Java Card applets to the security processor and run them there.

Then, the first idea was to place both chips on one "stick" with a USB type A connector but instead, another approach was investigated.

3.2.3. Finally used Security Stick Hardware

After implementing the first prototype, the concept got evaluated and finally refactored. The security controller of the ultimately used security stick is also from Infineon Technologies and CC EAL 6+ certified, but from a different security processor family with a different processor architecture. An advantage of the new security controller is that it additionally provides a native USB interface. Also, no own Printed Circuit Board (PCB) design must be done, as the security controller is on board of an Infineon Technologies USB hardware security token, which was designed for a different use case, namely a Fast IDentity Onine (FIDO) development kit [99]. A picture of the security stick can be seen in Figure 3.2.2.

However, the FIDO-specific firmware running on the hardware security token must be replaced by a Java Card OS. In comparison to the security controller of the first prototype, there is no Java Card OS available for this security processor family. Therefore, an existing proprietary implementation of a Java Card OS has to be ported to this processor architecture. As the available Java Card OS exists just for smart card security processors, also a USB interface implementation should be integrated into the ported OS. This is done by implementing the CCID driver as dongle, which works as CCID and ICC at the same time, as described in Subsection 2.3.4.4.

3.3. Host Library

In this section, the structure of the host library is described in more detail. The following subsections cover the architecture of the library.

3.3.1. Requirements

The host library is used as glue between host and security stick. In this subsection, the requirements on the host library are elaborated.

3.3.1.1. Host Interoperability

The host library should be working on Linux-based OSs as well as on Microsoft Windows. Additionally, the library should be designed in an easily extendable way for new supported platforms. To provide a high level of host portability, this code part should be written in the C programming language, as this is still the common low level language for any given platform. Additionally, since not every OS contains a fully-implemented C standard library, the use of the C library should be abstracted. This enables an easier integration of different implementations.

3.3.1.2. Security Stick Interoperability

Another requirement is to support different communication protocols on the stick-side. This can be a virtual COM communication as well as a CCID implementation. For this purpose, the communication part should be separated in an own layer. This layer can be easily exchanged depending on the communication that should be used. Additionally, it should be possible to build a shared library of the communication part of the library. This allows to easily exchange the used communication implementation even after compile-time.

3.3.1.3. Security Stick Detection

The host library should be able to detect connections and disconnections of security sticks. This allows the host library to inform the application about the adverse removal of the security stick. Also, especially useful for the first use case, it allows the application to check if a RF field has interrupted the connection and a new configuration can be fetched from the security stick after reconnection. This should be possible in a synchronous as well as an asynchronous way. The synchronous function should block until a reconnect happens. The asynchronous function should return a boolean value that indicates if a reconnect happened since the last check. However, the asynchronous functionality requires help of the underlying PC/SC implementation and the OS. Therefore, this should be an optional feature that can be activated during compile-time.

3.3.1.4. Flexibility

The possible command set differs depending on the selected Java Card application. The application layer of the host library should be easily adjustable to the Java Card application running in the background. Therefore, it should be possible to provide the supported commands as JSON file and let the host library generate the API of the respective application dynamically.

3.3.1.5. Easy Integration

As the host library should then be integrated into other projects, the output should be a shared/dynamic library that can be loaded from other projects. Additionally, the functionality of the shared library should be composed together into one include file ("amalgamation"). Therefore, all include files of the shared library's public functions should be merged together in one file.

3.3.1.6. APDU Handling

The host library inevitably has to handle APDUs that require additional data, like a payload or a message. Required input can be specified in the JSON file, as well as in the final internal representation. The host library should not be able to transmit an APDU with a missing required field. Instead, an error should be returned to the caller to inform about the misbehavior.

3.3.1.7. Static Memory Usage

As the maximum required memory is limited by the maximum sizes of the APDUs and the command execution is synchronous, meaning there cannot be two APDUs sent or received at the same time, the same buffer can be used for every APDU, independently of command or response. Dynamic memory is only required if the JSON load function is used. Therefore, if the host platform does not support a heap or another possibility to allocate memory, static commands must be used instead of the JSON file approach.

3.3.1.8. External and Internal Buffer

The application layer should provide two possibilities for each command with return value. Either the buffer is provided by the caller or the buffer is internally allocated and returned. If an own buffer is provided, the length of the response should be validated against the size of the buffer. The functions that use an internal buffer should be labeled with a `_Alloc` suffix.

3.3.2. Software Architecture

The software is layered in four parts that are described in this subsection. An illustration of the layer can be seen in Figure 3.3.1.

3.3.2.1. Hardware/Host Abstraction Layer

The Hardware Abstraction Layer (HAL) is the layer directly between hardware and lower-layer software libraries. It implements basic functionalities that are usually handled by the underlying OS, like file handling. However, if no OS is in use, this layer must operate directly on the hardware. An illustration can be seen in Figure 3.3.2.

3.3.2.2. Communication Layer

The communication layer abstracts the connection to the security stick. Since there might be different ways to communicate with the stick, the API of this layer has to be designed to be usable by different communication protocols. It builds upon the HAL layer, since the OS handles the device management. For this thesis, a USB implementation is sufficient. In case of extending the host library to a new platform without OS, the USB interface has to be implemented on hardware-level as part of the HAL. An illustration can be seen in Figure 3.3.3.

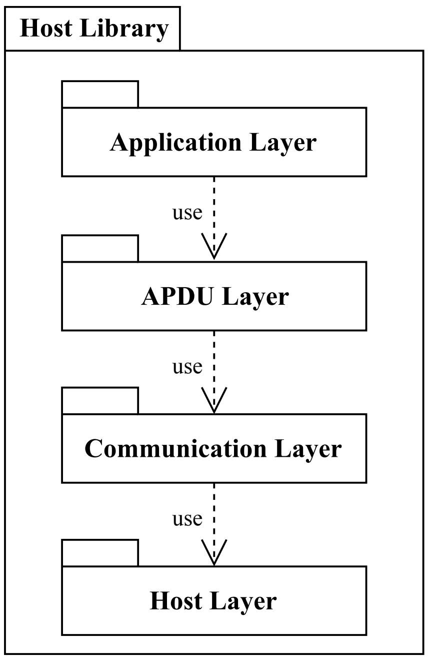


Figure 3.3.1.: The top layer is the application layer, which is directly used by the respective application. The application layer uses the APDU and the communication layer to communicate with the security stick. These use the host abstraction layer, which enables a better portability to other platforms.

3.3.2.3. APDU Layer

The APDU layer contains utility functions to create and fetch APDU commands. Commands are represented by internal "APDU objects" that store the content of the respective APDU. Additionally, it is possible to load a JSON file that contains all available commands. The fetched object can be altered and is then forwarded to the underlying communication layer. There, the object is checked for correctness before it is transmitted.

3.3.2.4. Application Layer

The application layer is the highest abstraction layer. It abstracts the Java Card application with all supported APDUs. This part is rigid and must be created for every used Java Card applet, as the commands of the applet are mapped to host library functions. Furthermore, one "dynamic" application is created. This application creates its APDUs from a JSON file. The APDUs then can be found by searching for the respective name. By analyzing the fields, it decides if parameters are still not set and the user is informed with an error if an incomplete APDU should be sent.

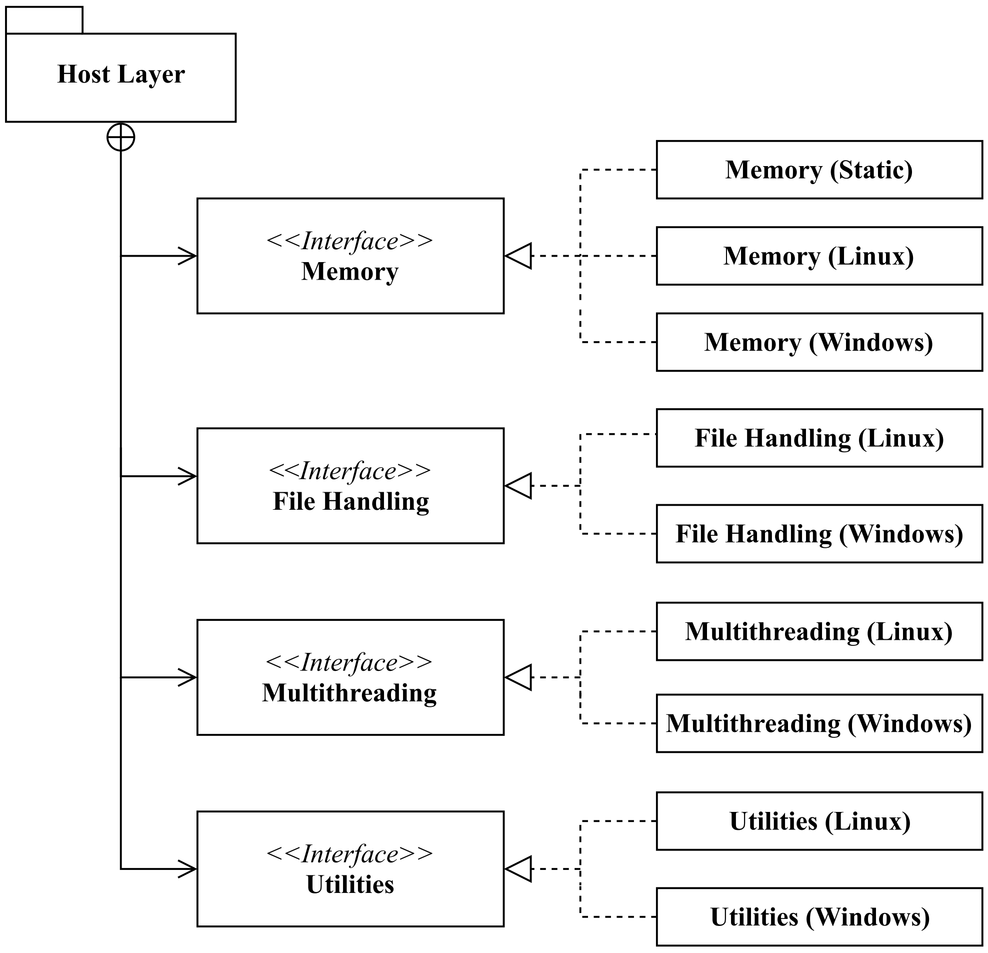


Figure 3.3.2.: The host layer consists of four sublayers which are designed as interfaces: the memory interface, file handling interface, multithreading interface and the utilities.

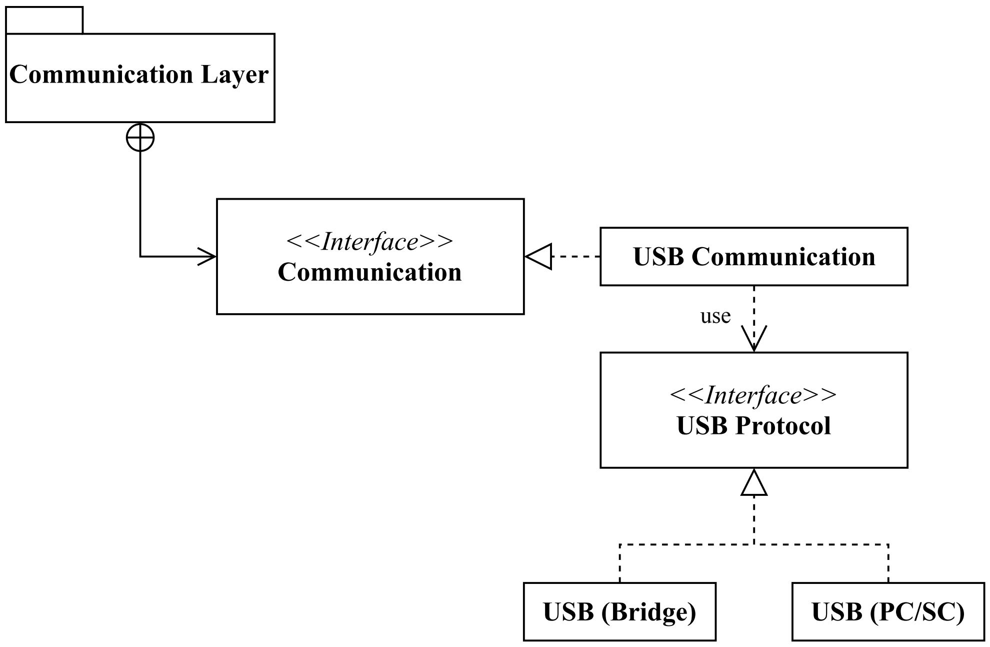


Figure 3.3.3.: The communication layer consists of the communication interface. The USB implementation is one concrete implementation of this interface. It uses the USB protocol interface, which abstracts a concrete USB protocol implementation.

3.4. Use Case I: Secured Configuration

This section describes the design of the first use case: the configuration of industrial fab equipment. This use case is just one part of the Trustworthy Systems ("TrustworSys") demonstrator [100] of the IoSense project [1], which is carried out as a cooperative work between the Institute of Technical Informatics (ITI) and Infineon Technologies Austria AG. The primary focus of this demonstrator is the flexible and secured re-configuration of future smart sensors via NFC - envisioned for the whole lifecycle, during production steps, to customization and re-configuration by the end-users. Beside implementing the required actions in a Java Card applet, the security stick should be provided to the cooperation partners for further tasks.

3.4.1. Use Case Design

The design, structure and processes of this use case were already given. As it was carried out together with Thomas Ulz (ITI), his related work, further described in Chapter 2.4, was considered during the design phase.

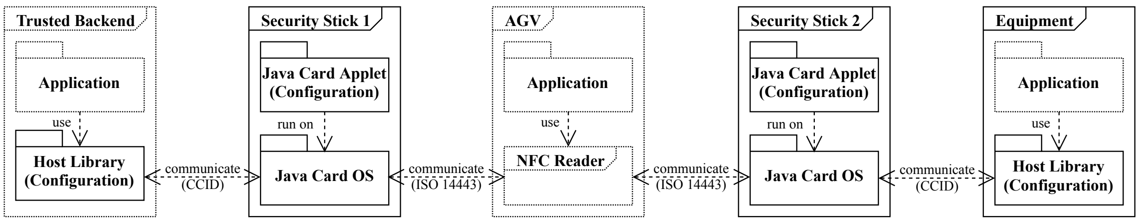


Figure 3.4.1.: Illustration of the configuration use case architecture.

3.4.2. Additional Requirements

As production systems often have real-time conditions to fulfill and these are usually incompatible with synchronous blocks as well as spontaneous asynchronous code execution (such as event callbacks), the host library provides a non-blocking function that checks if the security stick was disconnected and reconnected since the last check. In comparison to an asynchronous callback function, this enables predictability.

Additionally, it should be possible to update the keys in the same way as described in [79]. This means, new secret keys can be enforced by loading them to the security stick by NFC. To provide confidentiality and authenticity, the new secret keys also have to be protected by authenticated encryption with the old secret keys in use.

3.4.3. Physical Architecture

The security stick is permanently connected to the equipment by USB. An AGV can interrupt this connection by activating the RF field of its NFC reader. However, this thesis does not cover the equipment application and the AGV application, as it part is developed by the cooperation partners. An illustration of the interacting components can be seen in Figure 3.4.1.

3.4.4. Process Architecture

The configuration is previously generated by a trusted backend which stores it on the untrusted AGV or smartphone. Since the transport is seen as untrusted and unsecure channel, the configuration is protected by authenticated encryption. This protection already happens on a security stick, which is connected to the trusted backend. The security stick additionally stores the protected message, as the transporter fetches the protected message from there. This is done by interrupting the connection between the security stick and the trusted backend. From there, the AGV drives to the production equipment and interrupts its connection with the USB-connected security stick. This allows to store the protected message on the security stick, where it is, after the message is verified, stored in decrypted form. Finally, after the NFC reader

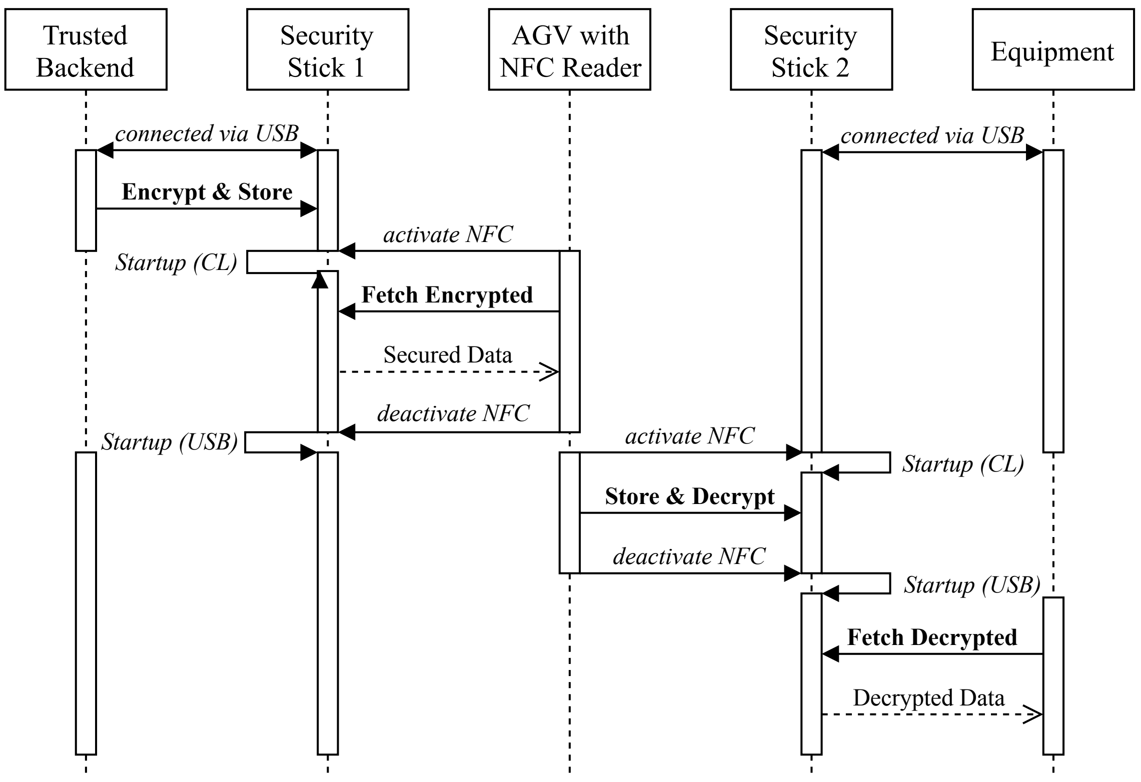


Figure 3.4.2.: Sequence diagram of the configuration use case.

removes its field, the security stick reconnects to the production equipment by USB. An illustration of the process can be seen in Figure 3.4.2.

3.4.5. Commands

This subsection contains the required operations that have to be implemented on the Java Card applet, running on the security stick.

Change Keys

This command is used to change the secret keys that are used for the authenticated encryption. The initial key is set to the zero key. New keys are then protected by authenticated encryption, using the active key. This protection enables an untrusted and unsecured communication channel, like a smartphone, to change them in a secured way. Table B.1 shows the structure of this APDU.

Encrypt Data

This command is used to encrypt and authenticate the incoming data. This should be done in the three steps: init, update and final. Additionally, it is possible to store the encrypted data on-chip. This allows the trusted backend to generate the payload directly on the security stick. Table B.2 shows the structure of this APDU.

Store Data

This command is used to decrypt and verify a secured configuration. Additionally, the decrypted configuration is stored in the secured non-volatile memory of the security stick. Like the previous command, it is designed in the init-update-final format. However, the configuration then can only be read if the signature could be verified. Table B.3 shows the structure of this APDU.

Fetch Data

This command is used to fetch the configuration from the security stick. Since the “Encrypt” command is able to store the output encrypted and the “Store” command stores it decrypted, it must be decided which data should be fetched. If no valid data is available, an error is returned. Table B.4 shows the structure of this APDU.

3.5. Use Case II: Secured Network Protocol Translation

This use case seeks to integrate an isolated production network, which uses a legacy communication protocol without security measures, into the company network. Also this use case is part of a project, namely the SemI40 [2] project.

There exist two different approaches to this challenge: replacing the legacy protocol [96] or wrapping the legacy protocol into a state-of-the-art protocol. The wrapping approach can additionally be split up in two different methods, regarding the interface to the existing network: a passive and an active one. The active one replaces just a part of the existing network by implementing the counterpart to the respective machine protocol entity [90]. The passive one sniffs in the existing network and parses out the relevant information without modifying the network [91]. However, as the SECS/GEM protocol is a proprietary one, the message structure is not available for the open use. Additionally, with this approach the company network is not able to actively interact with the entities and therefore relies on the activity of the network. For these reasons, the partially replacement approach is investigated, where requests and responses are translated between the different protocols.

3.5.1. Use Case Design

The isolated network should be integrated in the OPC UA network, as this is the upcoming state of the art. As the main network is stronger connected with other participants, the attack surface is increased. Therefore, beside of using the latest suggested security policy, hardware security extensions should be integrated.

Beside of that, an additional translator should provide the gateway to the cloud layer of the company's network. This layer uses the MQTT protocol for communication, as it is designed for limited bandwidth networks, which can occur when communicating over large geographical distances.

3.5.2. Additional Requirements

Beside of the general use case functionality, additional one is required for demonstrational purposes:

Different Variables

The demonstrator should differentiable between confidential and public variables. While confidential variables should be kept inside the isolated and the main network, the public variables should be exposed to the cloud layer too. Additionally, it should be possible to have at least one value that comes from a sensor. This allows a person standing in front of the device to physically modify the value of the variable. Therefore, a Ultrasonic Sensor (USS) should be used that provides the private value *range*. The public value, *fillStatus* should be emulated by involving the current time.

Additionally, the demonstrator should differentiate between SECS/GEM status variables (requested synchronously) and SECS/GEM alarms (pushed asynchronously), since these two are the basic communication possibilities used in SECS/GEM.

Graphical User Interface

To visualize the information gathered from the data sources, the operator machines should have a display. Additionally, a Graphical User Interface (GUI) should be developed, which visualizes the current values of the system.

Demonstrator Mode

Beside of the secured operation, an additional "unsecured demonstrator mode" should be implemented. This mode is activated as soon as the security stick is disconnected from a device. Instead of simply disallowing further communication with the device, the display should signal the detachment of the security stick. However, the entity should still allow an unsecured communication. As soon as the security stick is re-attached to the device, it should switch back to the secured, normal mode.

Potential Attacker

Additionally, a potential attacker should be implemented. This entity is primarily for demonstration purposes, in order to verify and particularly visualize the difference between secured and unsecured communication mode during the demonstration. The Potential Attacker, or also called "Packet Sniffer", should be able to sniff the network traffic. The used OPC UA protocol should be analyzed and the transmitted value parsed out. This is, however, only possible if the unsecure demonstrator mode is active. Otherwise, the encryption would protect the values. The sniffing procedure corresponds to the passive wrapping approach, however the values are visualized instead of forwarded.

3.5.3. Physical Architecture

This subsection describes the physical architecture of the demonstrator. It is divided in the network structure and the entities.

3.5.3.1. Network

To demonstrate the protocol translation in a descriptive way, three different networks should be developed. The separation between these networks should be done by using "gateways", as described in Section 2.1.1. The structure of the three networks can be seen in Figure 3.5.1.

Legacy Network

This is the sealed-off network. Part of the network should be a production machine ("equipment") and the gateway. For designing a more realistic demonstration scenario, the selection of this exemplary legacy protocol has been directly derived from a real semiconductor production environment, where SECS/GEM is still typically in use.

Internal/Main Network

The state-of-the-art production machine and the company worker are located in this network. The network security policy requires the use of the security stick and OPC UA as communication protocol.

External Network

The external staff is located in this network and therefore it is connected to the cloud. The used communication protocol is MQTT.

3.5.3.2. Entities

At least six entities are required for a meaningful representation of the required actions. The six entities and the additional Packet Sniffer can be seen in Figure 3.5.1.

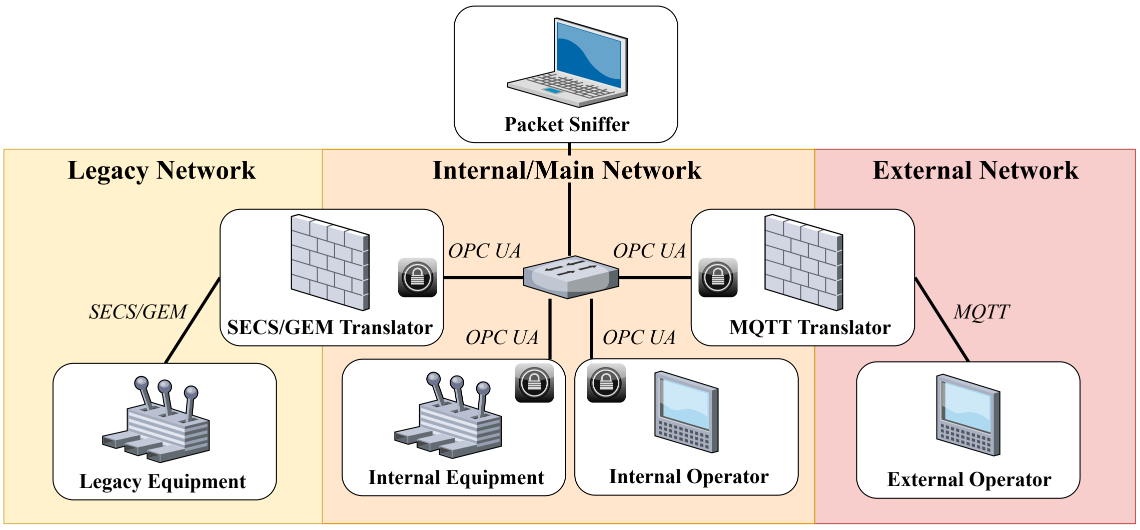


Figure 3.5.1.: Illustration of the demonstrator's network. The padlock denotes that the host is connected to a security stick.

Legacy Equipment

The Legacy Equipment should have two managed variables: a private and a public one. The private should be fetched from the connected sensor, the public should be an internally emulated fill state with three possible values: "Refilled", "Refill" and "Critical". This entity represents the SECS/GEM equipment and is part of the Legacy Network.

SECS/GEM Translator

The SECS/GEM Translator is part of the Legacy Network and the Main Network. It represents a SECS/GEM host in the Legacy Network and an OPC UA server in the Internal Network. The main task of this entity is to react to synchronous requests from the Main Network and forward synchronous responses and asynchronous alarms to the Main Network. The translator requires a security stick as it is connected to the Main Network. Additionally, a USB NIC is required to participate in both networks.

Internal Equipment

The Internal Equipment is part of the Main Network. It represents an OPC UA server and provides the same variables as the Legacy Equipment. Additionally, it is uses a security stick to communicate with the other entities of the Main Network. This entity corresponds to a OPC UA server on the field layer and therefore demonstrates the replacement of legacy machines.

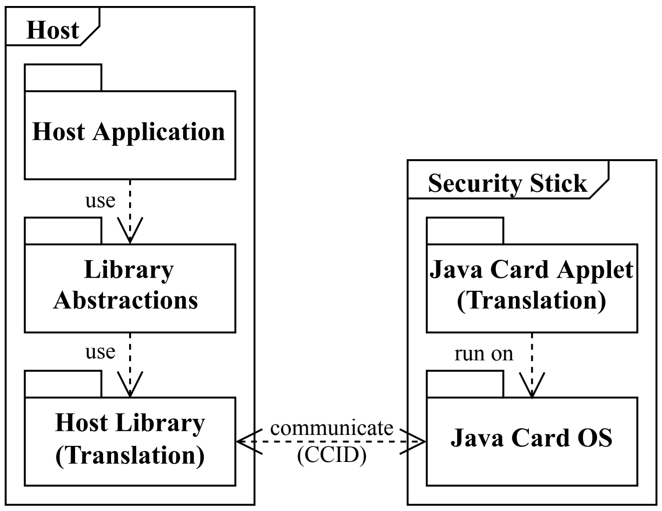


Figure 3.5.2.: Illustration of the architecture of a host with connected security stick. The external libraries, which are abstracted to provide the applications an implementation-independent interface, have to be adjusted to use the provided application layer of the host library instead of the software-based crypto stack.

Internal Operator

The Internal Operator is part of the Main Network. It represents a OPC UA client and additionally has a connected display that shows the values of the fetched variables. Also the Internal Operator requires a security stick.

MQTT Translator

The MQTT Translator is the gateway between the Main Network and the External Network. It represents an OPC UA client in the Main Network and a MQTT publisher in the External Network. Due to the participation in the Main Network, beside the USB NIC, a security stick is required too. Additionally, the public values should be published to the External Network by using two topics.

External Operator

The External Operator is part of the External Network, for example of an external company providing remote maintenance services. It represents a MQTT subscriber and additionally makes use of a connected display to show the received values. The interfaces have been defined in a way that this part could be replaced by another demonstrator network.

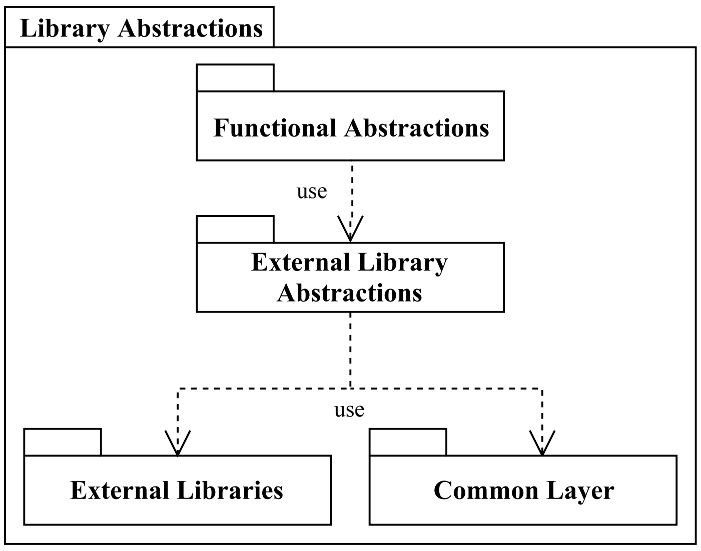


Figure 3.5.3.: Illustration of the library abstractions.

3.5.4. Software Architecture

The host applications are the main programs running on the entities. The functionality of these is already briefly described in the previous subsection. For easier reuse and better configurability, a common code base, which abstracts the required actions of each supported protocol entity, should be developed. This common code base wraps the external libraries, allowing to easily exchange them. The design of software architecture on a host can be seen in Figure 3.5.2.

3.5.4.1. Library Abstractions

The applications directly use the customized functional abstractions, which provide reusable and library-independent functionality, as they depend on the library abstraction interfaces instead of the concrete implementations/libraries. The bottom layer is composed of the actual implementations and a common layer, which is used to provide semantical interoperability of the data. An illustration can be seen in Figure 3.5.3.

Functional and Library Abstractions

As mentioned, the functional layer provides an interface for use case specific functionality. This interface is implemented by concrete classes that use the interfaces of the external library layer. This layer abstracts the functionality that is required from the underlying external library. However, it is designed implementation-independent. An illustration of the interfaces and classes can be seen in Figure 3.5.4.

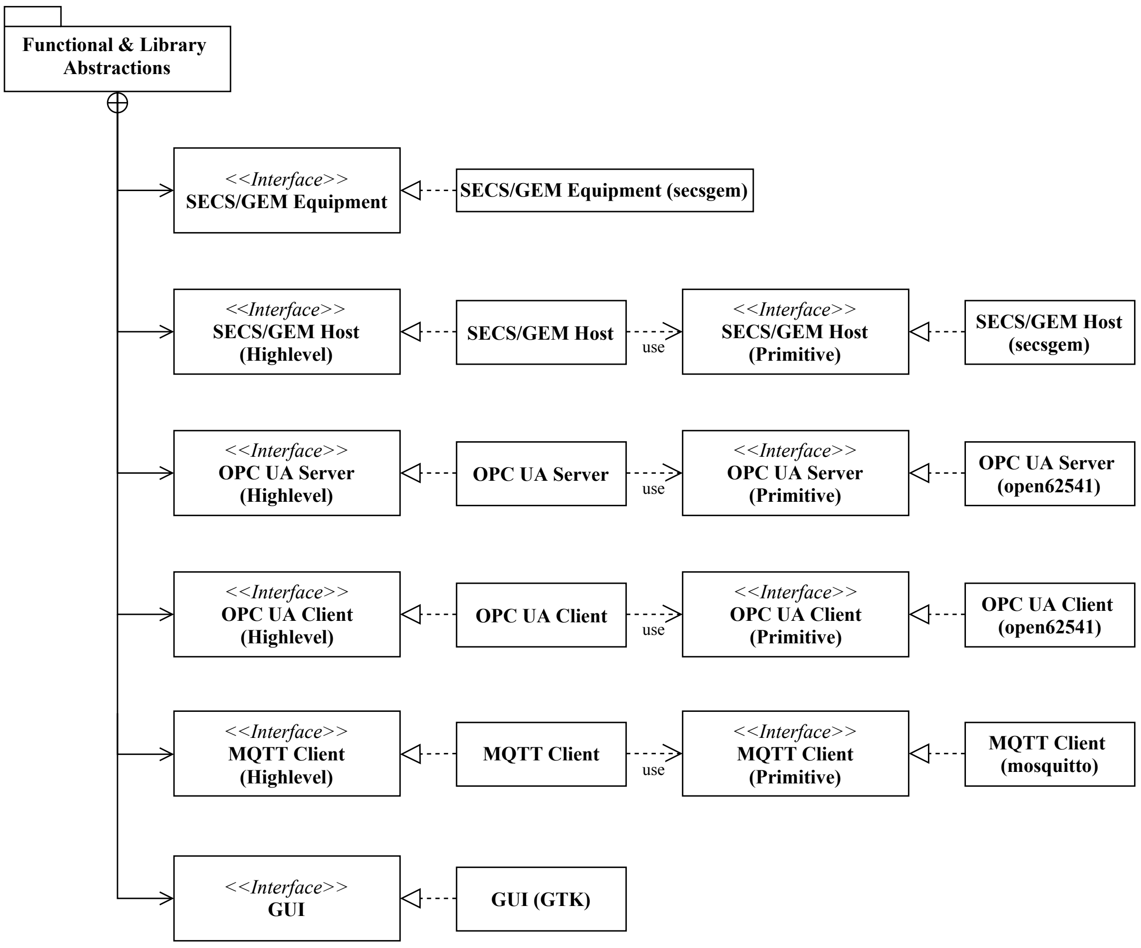

Functional & Library Abstractions
<<Interface>> SECS/GEM Equipment
SECS/GEM Equipment (secsgem)
<<Interface>> SECS/GEM Host (Highlevel)
SECS/GEM Host
use
<<Interface>> SECS/GEM Host (Primitive)
SECS/GEM Host (secsgem)
<<Interface>> OPC UA Server (Highlevel)
OPC UA Server
use
<<Interface>> OPC UA Server (Primitive)
OPC UA Server (open62541)
<<Interface>> OPC UA Client (Highlevel)
OPC UA Client
use
<<Interface>> OPC UA Client (Primitive)
OPC UA Client (open62541)
<<Interface>> MQTT Client (Highlevel)
MQTT Client
use
<<Interface>> MQTT Client (Primitive)
MQTT Client (mosquitto)
<<Interface>> GUI
GUI (GTK)

Figure 3.5.4.: Illustration of the functional and library abstractions with the concrete implementations.

External Libraries

This layer contains the concrete external libraries that are used in this project.

Common Layer

The common layer is used for the common data representation, enabling semantic interoperability. However, the syntactic interoperability is derived from the semantic representation and implemented in the respective external library abstraction layer.

3.5.5. Process Architecture

There are several processes going on inside of this demonstrator. However, to focus on the key components of this work, the security and the translation processes are described in particular.

3.5.5.1. Security

This part contains the security processes of this demonstrator.

Certificate Generation with Security Stick

Since the OPC UA security policy requires X.509 certificates, and the security stick contains the asymmetric key pair used in such a certificate, it should be possible to generate a CSR with the public RSA key of the security stick. Therefore, the host first generates a new asymmetric key pair on the security stick. Then, the host can fetch the generated public key and insert it into the certificate request information. This information, together with the signature identifier, is then signed by the security stick, using the previously generated private part of the asymmetric key. The resulting CSR is then transmitted to a CA, where it is signed to get the final certificate. The final certificate is transmitted back to the host and stored there. An illustration of the process can be seen in Figure 3.5.5.

OPC UA with Security Stick

When asking a OPC UA server for its supported endpoints, the server responds with the supported security policies and its certificate. The public key of this certificate is extracted and inserted into the security stick. Then, the certificate is verified with the previously inserted key. If the signature is valid, the public key of the OPC UA server is inserted into the security stick and the OPN request is crafted. Subsequently, the request is encrypted and signed by the security stick and then sent to the according

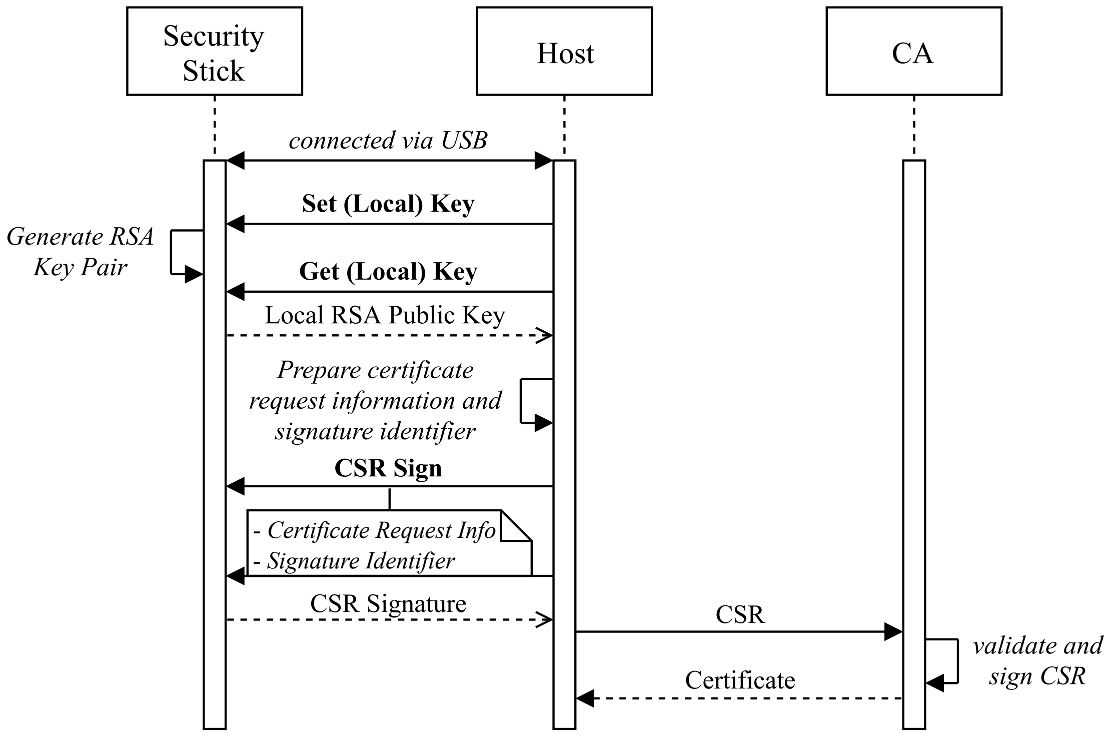


Figure 3.5.5.: Sequence diagram of the certificate generation.

endpoint of the OPC UA server. The OPC UA server, which is also connected to a security stick, does a similar approach. However, instead of encrypting and signing, the OPC UA server verifies and then decrypts the message. Then the OPC UA server crafts a new response and again encrypts and signs it with the security stick. When the OPC UA client receives the OPN response, it is verified and decrypted. Then, both parties are aware of both nonces and therefore are able to derive the symmetric secrets, which are used for the further communication. An illustration of the process can be seen in Figure 3.5.6.

Packet Sniffer

This should be achieved by poisoning the ARP cache of the Internal Equipment and the Internal Operator. The poisoning lets the Packet Sniffer impersonate the Internal Operator and therefore the Internal Equipment will send the messages to the wrong receiver. The OPC UA messages should be intercepted, displayed and then forwarded to the intended receiver. Further, the attacker tries to parse the message to get to the communicated information. An illustration of the process can be seen in Figure 3.5.7.

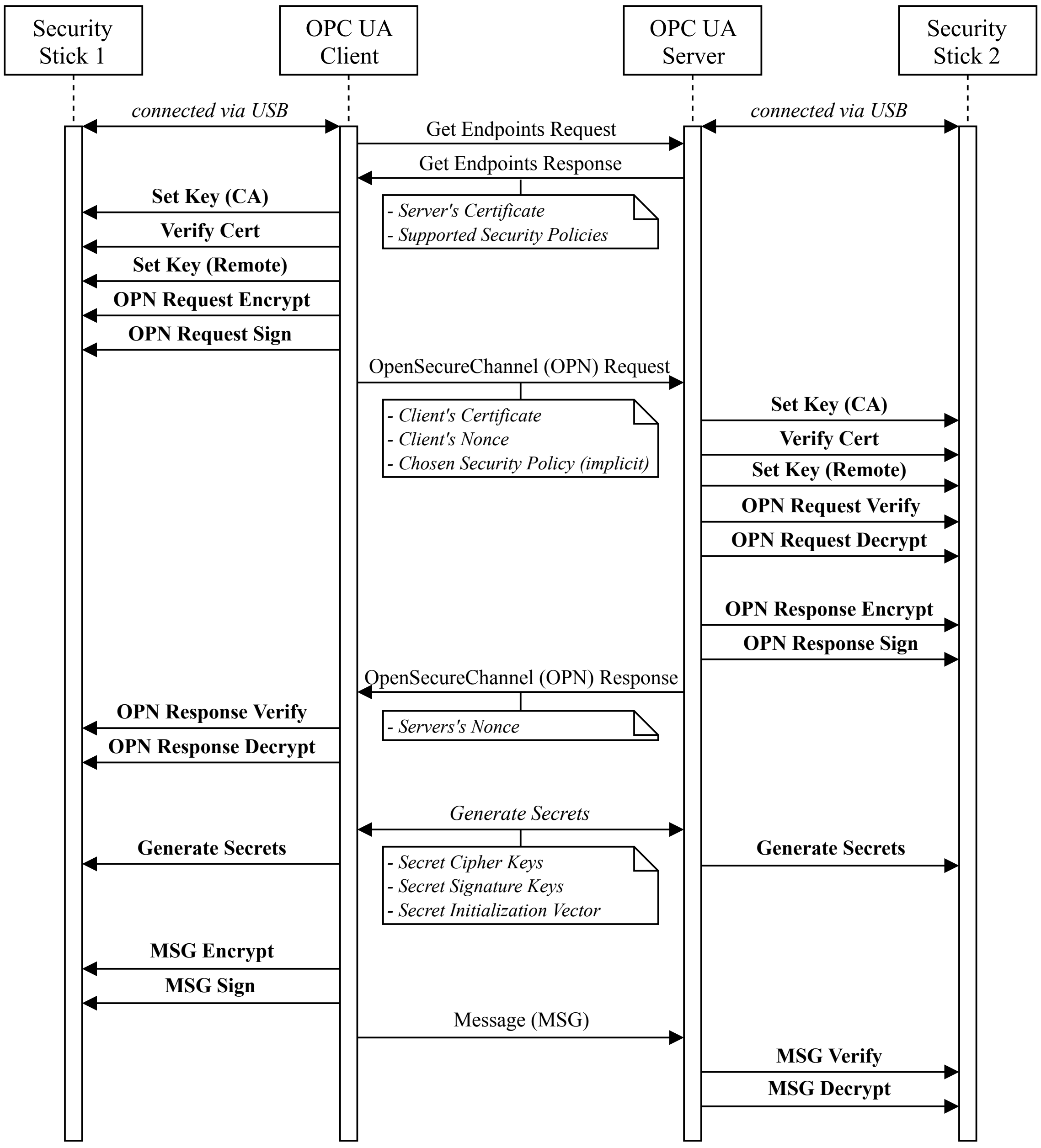


Figure 3.5.6.: The OPC UA connection establishment including the additional communication with the security sticks. However, previous to the Get Endpoint Request, both entities have generated a certificate, as shown in Figure 3.5.5. This certificate is then used in the “Get Endpoints Response” (server) or in the OPN Request (client). The remote side verifies the CA that has signed the certificate and then verifies the certificate by using the public key of the CA and the given signature of the certificate. The used approach can also be applied to self-signed certificates. The according APDU responses are neglected for better readability.

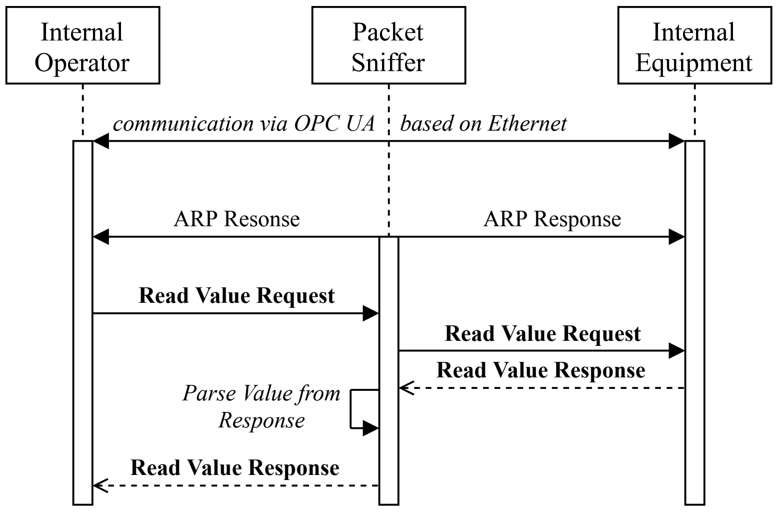


Figure 3.5.7.: Sequence diagram of the Packet Sniffer.

3.5.5.2. Translation

This part contains the translation processes of the demonstrator.

SECS/GEM Translator

The SECS/GEM Translator mirrors the status variables of a SECS/GEM equipment into the its OPC UA variable nodeset. OPC UA clients can further fetch the values of the variables. The OPC UA requests are translated to SECS/GEM requests and forwarded to the equipment. The response again is translated to OPC UA, by using the common layer. An illustration of the process can be seen in Figure 3.5.8.

MQTT Translator

The MQTT Translator has local OPC UA variable representations that are regularly updated, and publishes the value as soon as it has changed.

3.5.6. Commands

This subsection describes the commands of the second Java Card applet. The secret keys are distinguished between local and remote keys. The local keys are the own keys and the remote keys are the keys of the connection partner. There is no distinction between secret keys of the client and the server.

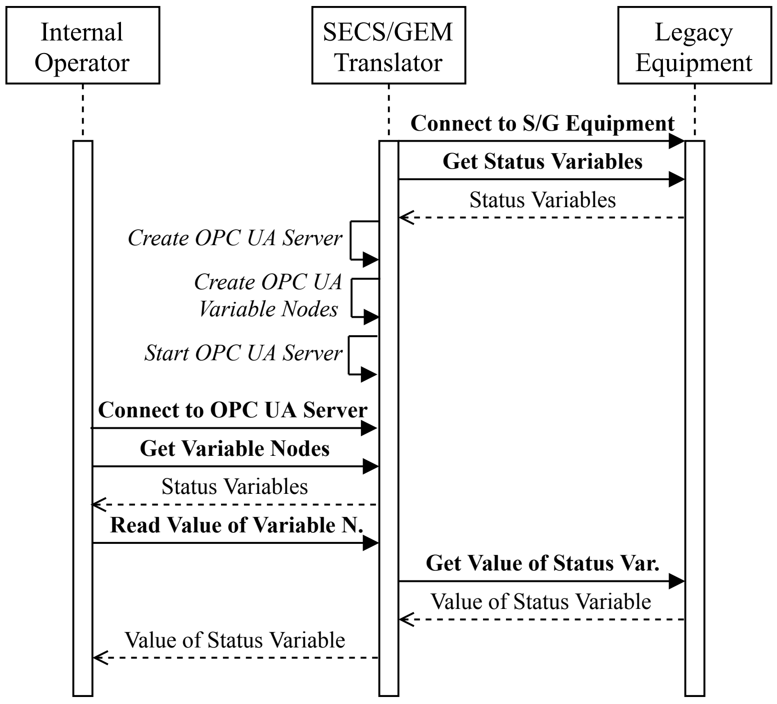


Figure 3.5.8.: Sequence diagram of the SECS/GEM Translator.

3.5.6.1. First Approach

The first approach was to map the cryptographic operations of the security policy from Table 2.3.1 to according APDUs. During the installation, the applet should generate its own 2048-bit RSA key pair. Further, the applet should be able to handle 16 remote identities, represented by an unique ID and the corresponding public RSA key. The ID management should be done by the host application, therefore the host library must provide an interface for it. This approach resulted to the following instructions:

Set Key

This command is used to inform the Java Card applet about the keys that should be used. While the local RSA key pair is generated on-chip and is therefore stored there anyway, the remote public RSA key is received during the connection establishment. Since RSA keys consist of two parts, the exponent and the modulus, two different cases need to be considered. In addition, the local and remote symmetric keys, which are generated during the connection build-up, have to be mapped to the respective identity and stored accordingly. Additionally, the applet should be able to generate a new own asymmetric key pair. This is required to fulfill company security policies that enforce a change of the secret keys after a certain time interval.

Get Local Public Key

This command should return the local public RSA key. This is required to be able to generate a corresponding certificate which is further used by the OPC UA protocol.

Encrypt & Decrypt

The encryption and the decryption process have their own respective command. The P1 and P2 arguments are used to specify the type (asymmetric or symmetric), the current state (init, update or final) and the respective remote identifier. However, the application layer of the host library is in charge of handling the IDs. If a key is not initialized, an error should be returned. Additionally, the generated secret keys of the symmetric cipher have to be set previous to an operation. Also, the generated IV has to be transmitted during the initial step.

Sign & Verify

The signing and verification should be handled similar to the encryption & decryption. They also require the three additional specifier. To differ between message and signature, the data of the final verification step includes only the signature.

Random Number Generation

This command is used to get bytes from the SE's TRNG. The amount of needed random bytes is required as argument.

HMAC for P-SHA2-256

Instead of doing the whole KDF algorithm on the SE, just the cryptographic operations, namely the HMAC, is done on the secured hardware.

CSR Sign and Cert Verify

Finally, each host should be able to generate a CSR based on the RSA key pair on its security stick. The first two parts of the CSR, the certificate request information and the signature algorithm identifier, are generated on the host. The third and last part, the actual signature, is done by the SE. The OPC UA security policy prescribes the signature algorithm for the CSR and is set on all available security policies (except "None") to RSA-PKCS15-SHA2-256.

3.5.6.2. Final Enhanced Design

The final design does not expose the generated secret keys or any other intermediate values to the host. Instead, the OpenSecureChannel request message is generated on the host but is, before the actual encryption, modified by the Java Card applet. The given random nonce of the message is replaced by an internally generated random that is not exposed to the host. Further, the remote nonce is extracted and replaced from the OPN response message before the decrypted message is returned to the host. The two specially designed encrypt and decrypt commands allow the SE to generate the secret keys and IVs without any additional intermediate steps on the host, as both required parts are already stored on the SE. This approach results in the following instructions:

Set RSA Keys

This command is basically the same as the set key command of the first approach but just the asymmetric public key of a remote connection partner can be set and the own asymmetric key pair can be generated. Table B.5 shows the structure of this APDU.

Get Local RSA Key

This command stays exactly the same as the respective command of the first approach. Table B.6 shows the structure of this APDU.

OPN Encryption & Decryption

This is one of the new commands. The OpenSecureChannel messages are the only asymmetrically encrypted messages. As can be seen in Figure 3.5.6, this message contains the respective certificate and nonce. Instead of simply encrypting and decrypting the message, the nonce is inserted before encryption and extracted after decryption. These modifications require the exact position of the respective nonce inside the message. Since the position differs between request and response, the applet has to be informed about the type of the current message. Table B.7 shows the structure of these APDUs.

OPN Signature & Verification

This is the second new command. Similar to the OPN encryption/decryption, the nonce has to be replaced to have the signature of the actual, modified message. The nonce can be found in the last 255 bytes, therefore the actual replacement happens in the final step, which must contain each of the last 255 bytes. Since the final step of the verification includes just the 256 byte long signature and there might be multiple update steps, an additional identifier that signals the last 256 bytes of the message has to be introduced. This command type is referred to as “Final Update”. Table B.8 shows the structure of these APDUs.

Key Generation

After the OPN messages are exchanged, the SE got the local nonce from the encryption and the remote nonce from the decryption. Therefore, no additional data beside the identity is required to derive the symmetric keys and the IVs. Table B.9 shows the structure of this APDU.

MSG Encryption & Decryption

These commands are used to symmetrically encrypt and decrypt the MSG messages after the secrets were generated. They are structured like the respective commands of the first approach. The only difference is that the final command does not require the IV as part of the initial step. Table B.10 shows the structure of these APDUs.

MSG Signature & Verification

These commands are used to sign and verify the MSG messages. They are structured like the respective commands of the first approach. Table B.11 shows the structure of these APDUs.

Random Number Generation

This command is the same as designed in the first approach. Table B.12 shows the structure of this APDU.

CSR Sign & Certificate Verify

These commands are used to sign a CSR and verify a certificate. Table B.13 shows the structure of these APDUs.

4. Implementation

This chapter describes the implementation process. Due to the interaction with proprietary code and tools, some implementation details are described in less detail or are neglected completely.

4.1. Implementation Design

In general, the project is implemented in the C programming language. However, some functionality is just available in different programming languages and therefore used instead of re-implementing it. Nevertheless, this section gives further information about the implementation design decisions.

4.1.1. Build Environment

The build environment is the collection of tools that support the developer translating human-readable code into executable machine code. This subsection contains information about the used build environment. An illustration of the host library development process can be seen in Figure 4.1.1.

CMake

CMake [101] is an open-source cross-platform build tool that allows to organize projects and helps to build and test them. One of the advantages is that it enables build processes without focusing on compilers or platforms. Also, CMake builds are usually done "out-of-source", meaning all generated files are stored in an own folder, keeping the source code folders clean. Additionally, it organizes the internal and external dependencies and manages the build order accordingly.

The configuration is written into *CMakeLists.txt* files which are usually distributed in all folders that contain code files. The main configuration is located in the root folder and contains general information about the project such as the used programming language, the compiler flags and the main targets. The targets are either libraries (`add_library`) or binaries (`add_executable`). Further, it is possible to modify the

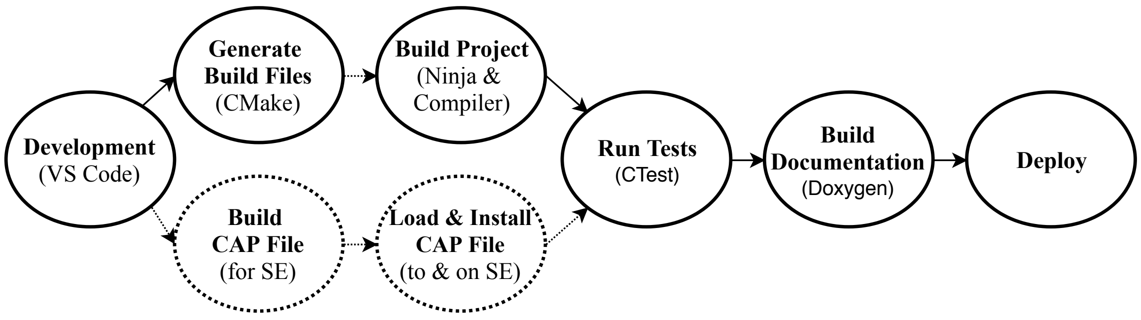


Figure 4.1.1.: Build process of the host library. After development, the build is done by using CMake, Ninja and the specified compiler. In parallel, the CAP file is generated which then is loaded and installed on the SE. After that, the tests can be executed, as they involve the host library and the Java Card application. Finally, the documentation is build from the source code with doxygen and then the final library is ready for deployment.

build process by specifying additional options, such as the communication type (bridge or PC/SC) to use.

Ninja

Ninja [102] is a build system (in CMake language: "generator"), with special focus on speed. It is designed to be generated by a higher-level build system (CMake in this case), therefore the generated rules for the build process are human-readable but not convenient to write by hand. In this thesis, it is used as a replacement for the classic UNIX Makefiles, since the builds are always executed in parallel and especially incremental builds are faster than with UNIX Makefiles.

4.1.1.1. Compiler

Compiler are the programs that actually convert human-readable code to machine-executable code. Usually, they are split up in a front end and a back end. The front end converts the programming code (e.g. C or C++) into an Intermediate Representation (IR), which is a virtual instruction set. Then, the back end converts the virtual IR into a real instruction set such as x86 or ARM. This separation allows an easy extension to new programming languages (front end) or new architectures (back end).

GCC

The GCC C compiler [103] is the most used open-source C compiler world wide. The first version was released in 1987 by the Free Software Foundation, Inc. Today, it has a wide variety of supported languages and target platforms.

Clang/LLVM

The LLVM project was started in 2000 at the University of Illinois [104]. One of the key advantages of LLVM is the clean architecture in comparison to the historically grown GCC compiler. Additionally, it was not designed to be a stand-alone compiler but a "base" that can be easily used as library for other purposes such as code optimizing or analyzing.

The last mentioned property is the main reason for additionally using this compiler. The high variety of tools which are based on the LLVM back end allow easy and fast code optimization, analysis and testing.

MSVC

MSVC (Microsoft Visual C++) is Microsoft's C/C++ compiler which targets Microsoft Windows machines [105]. Since Visual Studio 2017, it is also possible to cross compile for targets running a Linux-based OS.

Additionally, CMake allows to specify a "toolchain" file which contains the relevant information to compile for a different platform, like the compiler to use and where the required libraries can be found.

4.1.2. Coding Style

In this subsection, the general coding style of the thesis is described. Three different programming languages are used: C, Python and Java Card.

4.1.2.1. C Development

Most of the code base is developed in the C programming language. Since C is considered as low-level language, some convenient features, which are well-known from higher programming languages, need to be either dispensed or have to be "emulated" with available language constructs.

Return Value

Since the C programming language lacks exception functionality, every implemented function has the same return value: `Result_t`. Internally, this type is defined as an enum that represents different possible error types, e.g. `SUCCESS` or `FAILURE_USB_SEND`. If the function calls another function, the return value is implicitly checked for an error. As soon as an error occurs, it is forwarded to the caller of the respective function. Therefore, actual values have to be returned by reference. As the C programming language does not provide any functionality to distinguish between pointers and references, the names of references have a `_ref` suffix.

Logging

For logging purposes, an own function:

```
Utils_PrintOutput(LogLevel_t level, char *format, ...)
```

is used. On GNU/Linux systems, this uses the printf function or writes the output to a given file. Additionally, it takes the log level (error, info, debug, trace) of the provided string format. This allows an additional granularity of logging, which can be helpful for debugging purposes. The active log level is provided as a preprocessor define or can be specified as CMake option. This allows to change the log level at compile time.

Additionally, every function starts and ends with a function-like preprocessor macro:

```
INIT_FUNC("TestFunction")
...
END_FUNC
```

This enables easier debugging and tracing of the program's control flow. The `INIT_FUNC` function introduces an internal `Result_t` variable. Additionally, it prints the function name, which is provided as parameter, on the trace log level. The `END_FUNC` function introduces a new label "END" and returns the internal result variable.

Resource Management

A called function is in charge of freeing internally allocated resources in case of an error. Since a function is able to allocate multiple different resources such as files or memory, and the program can fail in different states, it has to somehow keep track of the currently allocated resources that need to be freed. Since no Garbage Collector (GC) or Resource Acquisition Is Initialization (RAII) functionality is available in the C standard, this is internally done by setting labels before the respective freeing. Labels are distinguished between normal teardown and erroneous teardown. An example can be seen in Listing C.5.

Classes

Since the C programming language offers no support for classes, members, and methods, the classes are emulated by structs, where the members are struct fields and the methods are function pointers that use a reference to the own struct as additional input parameter.

Interfaces

Since the C programming language does not support interfaces or any other types of abstraction, a different approach has to be pursued: It is done by declaring types as pointer to structs in header files. The structs are defined in the source files, and are therefore only known and valid for exactly this implementation. An example implementation of a file type can be seen in Listing C.1.

Abstraction – Implementation

Since the implementations of this thesis should be useable on different platforms and be agnostic towards the communication with the security stick, it is important that the interface (header) is designed without implementation-dependent code. This is achieved by using interfaces, as described in the previous paragraph, and allows to easily exchange implementations that depend on external software. In addition, cleaner interfaces are designed which implicitly result in cleaner code. By using CMake as build management tool, implementations can be easily exchanged during compile time.

Data Types

The implementation uses the data type definitions from *stdint.h*. These types include the actual size as part of the type name, e.g. *uint32_t*, which represents a 32-bit long unsigned integer. Additionally, also bytes and C strings are stored in the unsigned format. This increases the portability for systems with different architectures, as e.g. the size of integers is usually depending on the architecture. Also, the signedness of *char*, which is usually taken to represent one byte, is not specified in the C standard and is therefore represented as *uint8_t*.

C++ Compatibility

As C++ supports function overloading, the compiler additionally "mangles" the function names to have an unique identifier for each overloaded function. As this is not done by C compilers, each header file of this project is declared as "C" header file. This is done to inform a C++ linker that no mangling is done and the function name can be found without it. An example can be seen in Listing C.2. Additionally, when compiling with Clang or GCC, the *Wc++-compat* flag is enabled, which causes a warning to be thrown if incompatible code is detected during compilation. Also, the support for exceptions is enabled by compiling with the *fexceptions* flag. This flag allows C code to interoperate with exception handlers written in C++.

Compiler Flags

Clang and GCC support a various number of compiler flags that help to enforce additional restrictions by throwing warnings during the compilation process. This includes the *Wall*, *Wextra* and *Wpedantic* flags that enable the highest possible warning level. Some of the other added compiler flags are:

- *Wformat* checks the format argument of functions like printf for correct and expected data types.
- *Wformat-security* prohibits format-string attacks on format arguments.
- *Wcast-qual* prohibits removal of type qualifier (e.g. const) during casting.
- *Wuninitialized* prohibits the use of uninitialized variables.

Documentation

Doxygen [106] is a tool that can be used to generate documentation from annotated source code files. The source and header files should be commented in a way that allows Doxygen to generate a useable and understandable documentation of the code. Also, the different implementations and their differences should be seen there. Therefore, while the documentation of the prototype is kept generic, the documentation of the actual function contains implementation-dependent information such as the possible return values. An example can be seen in Listing C.3.

4.1.2.2. Java Card Development

In this subsection, general information about the used Java Card coding style is given.

Missing Garbage Collector

In general, a Java Card applet has two different phases that must be implemented: first the installation phase and then the phase after the applet is selected. One important difference to the common Java runtime is that the GC is usually disabled as it requires too much resources, especially time. This results in the aggravating circumstance that memory, which is allocated after the installation phase, is never freed. Therefore, the `process` function, which is called when an APDU command is received, is not allowed to contain a *new* keyword or any other function that internally allocates memory.

Signed Data Types

Java Card applets typically use just two native data types: byte and short. The integer data type is usually not implemented. Additionally, Java as well as Java Card does not allow to specify a signedness for its data types. However, byte and short are signed data types, which has to be considered when performing arithmetic operations or bitshifting.

Memory Location

The SE has two possible memory locations: a fast, transient RAM and a slow, persistent NVM. Depending on the use case, the correct memory location must be chosen. The arrays that are allocated with the array-specifier *[]* are written to the NVM by default. For temporary arrays, the `JCSystem.makeTransientByteArray` function must be used. This function allows to specify when the array should be cleared: either on deselection or on reset.

Exceptions

If a runtime error occurs, an according *ISOException* is thrown. This leads to an abrupt stop of the execution and a response of the respective SW. Additionally, the applet should be implemented in a way that an occurring error could not lead to a malicious internal state.

Representative Member

As an APDU command contains the fields INS, P1 and P2, which are used to control the execution, the applet contains *static final byte* variables that represent the possible control byte values. As example, Listing C.4 shows the three possible P1 values that represent the respective cryptographic state. This improves the clarity as reads and writes happen with a representative name instead of the actual value.

256 Data Bytes

One problem occurs when using RSA-2048 keys: the encrypted data is 256 bytes long but the APDU command just offers space for 255. Therefore, the 256th byte of the message is written to P2. This allows to transmit the message in just one instead of two messages.

Init – Update – Final

The Java Card API of the cryptographic operations is designed in three steps: init, update and final. The initial step usually loads the corresponding key into the crypto instance, sets the mode (e.g. encryption/decryption or sign/verify) and resets the internal state of the instance. The update step is used to update the internal state of the respective instance with the incoming data. However, encryption and decryption updates might already return data. Then, the final step indicates that the last data is inserted and the operation can be completed. For better compatibility, also the implemented instructions are designed in this way.

4.2. Security Stick

This section contains information about the firmware implementation running on the security stick, where an existing proprietary Java Card OS variant has been used as firmware basis, which has been enhanced in the course of this thesis.

4.2.1. First Prototype

In this subsection, the firmware implementation of the first security stick prototype is described. To provide a common interface for the security stick, the Vendor ID (VID) and Product ID (PID) of the USB generic microcontroller ("bridge") is inserted as udev rule in the host system:

```
ACTION=="add", SUBSYSTEM=="tty", ATTRS{idVendor}=="058b",
ATTRS{idProduct}=="0058", SYMLINK+="security_stick"
```

The rule generates a new symbolic link in the device filesystem of the host as soon as the respective USB device is connected and enumerated.

4.2.1.1. Bridge Firmware

The firmware is written with DAVE [107], a software development tool from Infineon Technologies. Beside the Eclipse-based IDE, DAVE is additionally shipped with low-level drivers for the on-chip modules of the generic microcontroller. These can be easily configured with a graphical plugin inside the IDE. This tool is ideal for rapid prototyping as it is required for a first prototype.

After a reset, the bridge initializes its internal structures and modules. The UART interface of the generic microcontroller is enabled and configured to transmit frames with 8 data bits, 2 stop bits and an even parity bit. Additionally, a timer is used to provide a clock signal of 5 MHz to the SE. Then, a static buffer is initialized that stores incoming bytes, as we know the maximum lengths. Finally, the low-level driver for the virtual COM USB driver is initialized. The implementation of this driver is provided by DAVE.

After the initialization, the UART interface is enabled and the SE is reset. This is done by an additional GPIO, which is connected to the ISO 7816 reset pin of the SE. The SE responds with an ATR that signals that the reset is done. Then, the protocol and parameter selection (PPS) is done. If the PPS is accepted, the baud rate is changed from 13400 to 312500 baud and the T=1 protocol is used for further communication. Next, the information field size is adjusted and the application is selected (layer 4 reached).

At this point, the USB driver is enabled, which creates the device symlink on the connected host. After that, the endless receive loop of the bridge is started: a command is received from the USB interface and then forwarded to the SE. Vice versa, the response is received from the SE and forwarded to the USB host.

4.2.2. Java Card OS

As base of the security stick for the final implementation, a proprietary Java Card OS implementation of Infineon Technologies is used. The target platform of the existing Java Card OS does not come with native USB support, therefore the implementation had to be ported to a different platform.

4.2.2.1. Startup Phase

Java Card reacts to synchronous requests and therefore serves usually just one connection interface at a time. Consequently, one of the initial tasks is to find out which interface should be used. The SE has different possibilities to do so. One of these possibilities is to find out how it is powered. If the power is sourced from the contactless interface, it is sure that the active interface has to be the contactless one. Otherwise it is powered by a contact-based interface and therefore the interfaces have to be checked for an initialization from the other side.

For the first use case, it is important that a contactless connection can interrupt a contact-based one. This is necessary because the secured hardware extension is connected to the equipment by USB all the time. When an AGV wants to transmit a new configuration, the USB connection must be interrupted, since only one interface can be active. This is realized by activating the field detection interrupt, which causes a reset as soon as the SE detects a RF field. After the reset, the startup phase is done again and will take the contactless interface as active one. If the RF field is removed, another reset will happen and the USB interface will be active again. This process is illustrated in Figure 4.2.1.

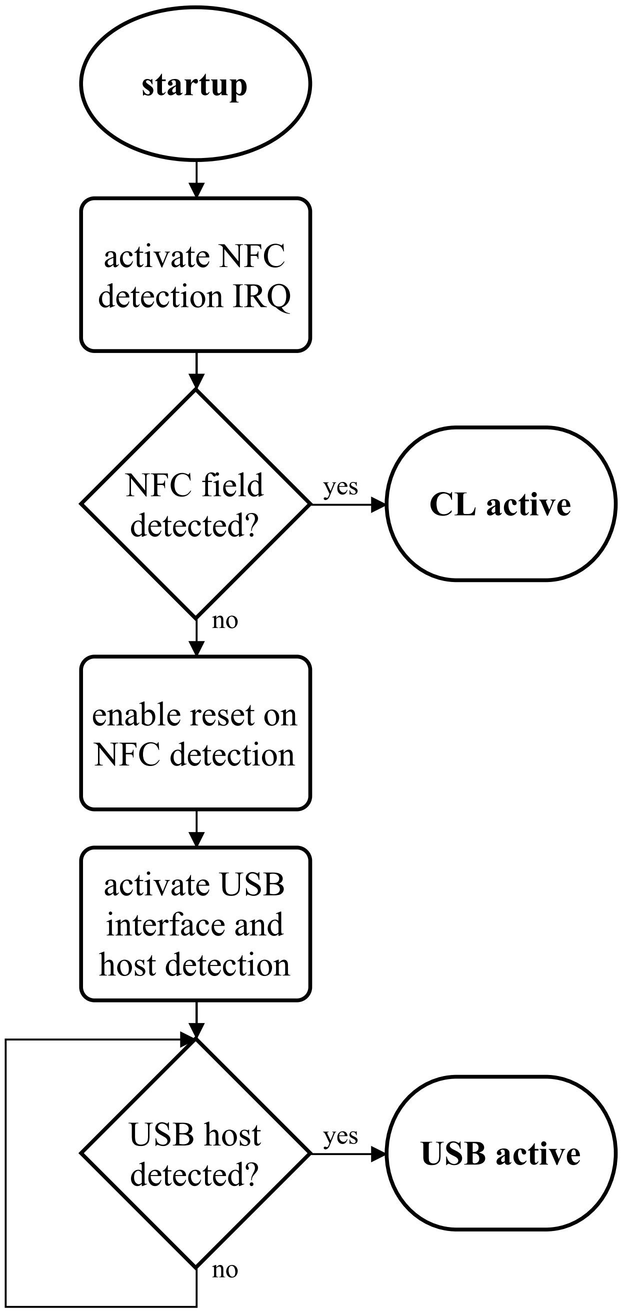

startup
activate NFC detection IRQ
NFC field detected?
yes
CL active
no
enable reset on NFC detection
activate USB interface and host detection
USB host detected?
yes
USB active
no

Figure 4.2.1.: High-level flowchart of the startup process, with regard to the active interface detection.

4.2.2.2. USB Interface

The USB interface was not supported by the Java Card OS used as basis for this project, and therefore had to be implemented for this thesis. The USB protocol communicates by setting states on the anti-parallel bus signals. This allows to signal different events and states of the USB protocol. In USB, a digital “high” is represented by a voltage greater than 2.8 V. To get a digital “low”, the voltage must be below 0.3 V. To get a differential “high”, the D+ signal line has to be pulled to digital high, and the D- signal line to digital low. To get a differential “low”, the lines have to be pulled vice-versa [108].

A USB reset is detected by holding both signal lines, D+ and D- to low. To identify the device as full speed device (12 Mbps), a pull-up resistor has to be connected to the D+ I/O during connection. Low speed devices (1.5 Mbps) are identified by a pull-up resistor on the D- I/O. Configuring the chip-internal pull-up resistors to the signal lines can be done by firmware. Additionally, the CCID specification [76] is implemented, which enables the communication with a PC/SC implementation of a connected host.

4.3. Host Library

This section contains information about the implementation of the host library.

4.3.1. Project Structure

A good folder structure helps keeping a good overview and is crucial when working with different platforms, because files need to be “replaced” by other files and it is hard to keep track if this happens in an unorganized manner.

As CMake can be used to build “out-of-source”, different *build* folders are used, where each represents one build configuration. Within this project, such a configuration consists of an used toolchain.

The *cmake* folder contains the supported toolchain files. The toolchain files give information about the target platform and the respective compiler and libraries that should be used. An example can be seen in Listing C.6.

The *doc* folder contains the Doxygen configuration file. The Doxygen file contains configurations which are used during the generation process, e.g. the language of the source code and the output format, like HTML. After a doxygen run, the generated files are also placed in this folder.

The *include* folder is empty. During the building process, the public interface of the host library is amalgamated and placed into this folder. This amalgamation process merges all relevant header files into one. The generated header file can then be used by developers as single include file for this library. Also the *lib* folder is empty at the beginning. After the build process, the compiled host library is placed there.

The *misc* folder contains two tools used by the build process: the amalgamation tool [109] and the ninja build tool for Windows. Additionally, it contains the *pcsc* folder with the *libccid_Info.plist* file that includes the security stick as supported CCID device.

The *src* folder contains all the source and header files that are needed to compile the host library. It is divided in four different subfolders, each representing one layer as described in Subsection 3.3.2.

Finally, the *test* folder contains the test-relevant files. More information about this folder can be found in Subsection 4.3.3.5.

An illustration of the folder structure can be seen in Figure 4.3.1.

4.3.2. External Libraries

In this subsection, the used external libraries are presented. Implementations for PC/SC, CCID and JSON handling are required.

4.3.2.1. winscard/libpcsclite

The WinSCard [111] header provides the API of the PC/SC implementation of Microsoft. Additionally, there exists an open-source version for UNIX systems, called libpcsclite [112], providing the same interface as Microsoft's WinSCard. The interface includes functions to fetch all connected devices, connect to one, transmit commands and disconnect again. Additionally, the API provides functionality to get status changes of connected smart cards and readers.

4.3.2.2. libccid

The libccid [113] project contains the source code for a generic USB CCID driver. The implementation comes with a *libccid_Info.plist* file that contains all supported smart card readers. The USB vendor and product ID of the security stick had to be added to get the driver working accordingly. The actual communication is done by making use of the abstracted PC/SC interface, which uses this driver internally.

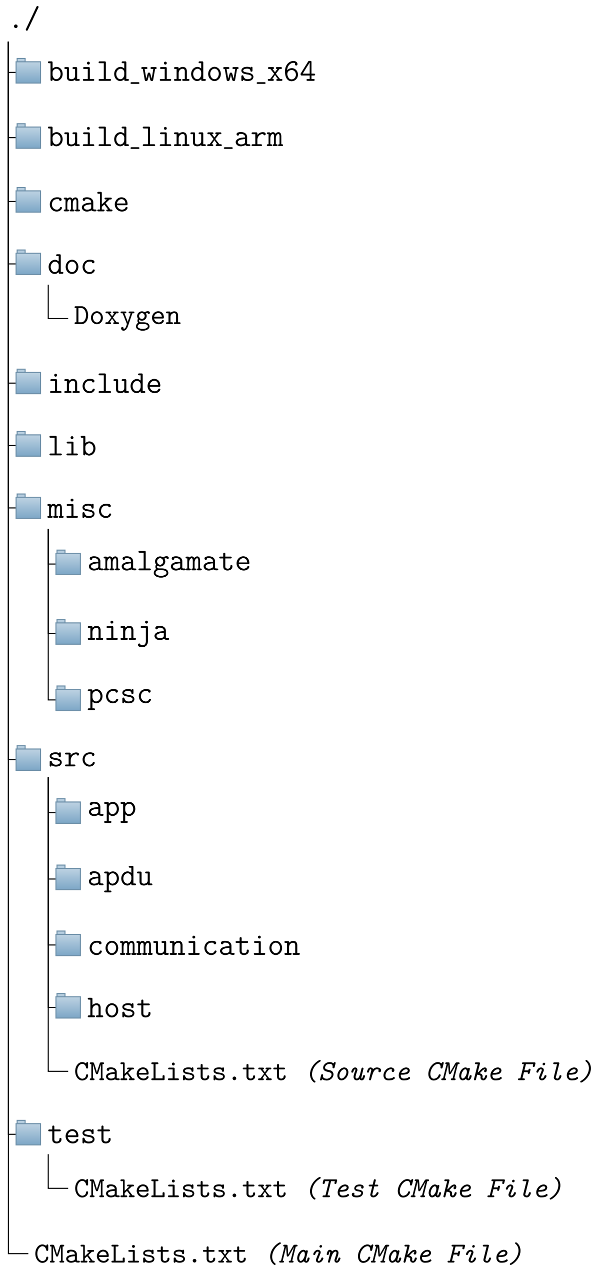


Figure 4.3.1.: Folder structure of the host library project. CMake Files in the source subfolders are neglected for readability reasons. Also the remaining build target folders are neglected. Style taken from [110].

4.3.2.3. JSMN

JSMN [114] is a minimalistic JSON parser written in C. It is designed for easy integration into embedded projects, by e.g. not depending on any libraries. The project is licensed under the MIT license and consists of one source and one header file. The API is composed of one function to initialize a JSON parser and another to parse a JSON string. The parsing process results in an array of JSON tokens, where one contains the start index, end index and the type of the token (like *string* or *primitive*).

4.3.3. Development

This subsection contains information about the development of the host library.

4.3.3.1. Hardware/Host Abstraction Layer

This layer includes all abstractions of the underlying OS and C standard library. As embedded systems usually come without dynamic memory support, this functionality is implemented in the hardware abstraction layer. Additionally, file handling, which is used for the dynamic command JSON file, is required if the dynamic application should be built. However, the multithreading implementation is just required if the CMake option `THREADING_ACTIVE` is set. This functionality is required to check connection events of the security stick in the background. The modules of this layer are implemented for two OSs: Linux and Microsoft Windows. Therefore, the interfaces are located in the *src/host* folder, while the implementations are inside of either the *src/host/linux* or the *src/host/windows* folder.

Memory

The *host_memory.h* file contains the interface for the dynamic memory management. The dynamic memory management is used to encapsulate additional checks and variable handling. Beside the interface, there exist three implementations: two for the mentioned OSs and one platform-independent (based a global byte array). The interface definition can be seen in Figure A.1.

`Mem_Alloc` is used to allocate dynamic memory. The first argument is a reference to the buffer, the second argument is used for the required memory size. This function checks if the incoming buffer reference is valid. Additionally, the buffer should point to NULL as this indicates that it is currently not in use. If the allocation is not successful, an according error is returned.

`Mem_Free` is used to free the allocated dynamic memory. Beside checking if the buffer is initialized (not NULL), it also sets the buffer back to NULL after the actual free process.

`Mem_Realloc` is used to extend (or shrink) the allocated dynamic memory. Additionally, it behaves like a `Mem_Alloc` if the buffer points to NULL and like a `Mem_Free` if the size is zero. The last one is a debug function that returns the amount of currently allocated memory.

File Handling

This interface is used to handle JSON files and therefore is just required if the dynamic application should be built. However, for Linux and Windows, the functions simply wrap the file methods of the C standard library, namely the *stdio.h* library. The interface definition can be seen in Figure A.2.

Multithreading

The multithreading implementation offers a function to start a thread that takes a pointer to the function that is executed and a pointer to the according arguments. On Linux, this requires the *pthreads* interface. On Windows, this functionality is covered by the Windows API (*windows.h*). The interface definition can be seen in Figure A.3.

Utilities

This interface is used to provide *helper functions* that are required by the other layers. These include:

- Conversion of a hex string (e.g. “AA-BB-CC”) to a byte array and vice versa
- Sleep for a given amount of milliseconds
- Get a current timestamp, e.g. since startup, in milliseconds
- String operations: get length, string contains and string equals
- Set and copy memory blocks
- Debug output function

The functionality of the *utils.h* also mainly wraps functionality that is provided by the C standard library. The interface definition can be seen in Figure A.4.

4.3.3.2. Communication Layer

The implementation of the USB interface is done in *src/communication/communication_usb.c*.

The module has five functions which can be seen in Figure A.5. The two transmit functions use the underlying communication implementation to transmit a command to the security stick and receive the response from it. The event-based functions expose the internal USB functionality to the upper layers.

USB Protocol

Since both communication variants, bridge and PC/SC, are using the USB interface, a common API is specified in *src/communication/usb.h.* For the USB bridge implementation on Linux, the host communicates with the security stick by using the symbolic link of the device, which is created by the udev rule as described in Subsection 4.2.1. This is handled differently in Microsoft Windows and other OSs. As the connection is represented by the device “file”, also the communication is file-based. Therefore, to communicate, first the device is opened, then the bytes are either written or read, and finally the file has to be closed again.

Since the libpcsclite Linux library is a replica of the Microsoft Windows implementation, the same PC/SC layer can be used for both OS. Internally, the implementation communicates with the PC/SC daemon that manages the connected CCID instances. The functions `SCardEstablishContext` and `SCardConnect` are used to “open” a connection with the security stick. Then, the `SCardTransmit` function is used to send a command and receive the response from the security stick. The function `SCardGetStatusChange` blocks until a status change happens on the given CCID instance.

4.3.3.3. APDU Layer

This layer contains the APDU command collection and provides an interface to interact with them. The APDU command collection is an array of APDU objects. The APDU class can be seen in Listing C.7. It contains the fields as specified in ISO 7816 and some additional information fields, such as the intended receiver (used in bridge mode) and the name of the command. Also, the command length *Lc* is not 8 bit but 32 bit long. This is done to allow longer data that is internally split up in more than one command. Since APDUs usually require additional data that is not known at compile time, such as the payload, the info field is used to keep track of the still missing fields.

The API consists of five functions: `APDU_GetAPDUbyName`, `APDU_CreateDataFromArray`, `APDU_SetAdditionalData`, `APDU_SetData`, and

APDU_FreeAPDU. The first function retrieves an APDU object with a given name from the collection. The second function moves an array to the heap. This is required to keep the data valid outside the current scope. The third function is used to set required fields (P1, P2, Lc, data, Le) and handle the info field correctly. The fourth function sets the data field and the length field of an APDU object. The last function frees the allocated memory of an APDU object.

4.3.3.4. Application Layer

The application layer encapsulates the functionality of a respective APDU collection and provides an useful interface that can be easily integrated into external applications. Three applications are created: one for each demonstrator and one that is JSON-based. The applications of the demonstrators are based on an internal APDU collection. These contain the name and the fields that are known during compile time. Then, the structure of the application functions is similar:

- The correct APDU command object is fetched by using APDU_GetAPDUbyName
- The missing information is set accordingly
- If known, it is checked if the buffer provides enough space for the response
- If the data is too long for one command, it is split up accordingly
- The APDU is transmitted and the response appended to the buffer
- Internally allocated data is freed at the end

In general, the application functions correspond to the according applet running on the security stick.

JSON-based Command Set

Instead of using an internal array that is known during compile time, a JSON file can be loaded during the startup. The APDU commands have the exact same representation in the JSON file:

```
"hash_digest":
{
    "target": 0,
    "CLA": "80",
    "INS": "51",
    "P1": "",
    "P2": "00",
    "data": "",
    "Le": "00"
}
```

The values can be either given as ASCII byte (denoted by the quotes) or integer value. Additionally, empty values are handled as missing and therefore set in the info field. The *Lc* is never explicitly set but calculated from the actual data. Also, the data must be given as hexadecimal string with dashes as delimiter (e.g. “00-AB-CD”).

4.3.3.5. Testing

The tests are implemented in the *test* folder. Three different test types are implemented: unit tests, functional tests and symbolic executed tests. For organizing tests, CTest is used. It is an application from the CMake stack and enables running tests and reporting results.

Test Design

For testing, an own “test library” is developed. This library is composed of different pre-processor macros that enable control structures. The start of a test is indicated by a `TEST_START`. During the start process, the amount of currently allocated memory and the current timestamp is saved. The end of a test is indicated by a `TEST_END`. Internally, again the current allocated memory and timestamp is fetched to compare with the values from the start. Additionally, `SUCCESS` is returned as an error would lead to an early exit with an error as return value. Then, several `ASSERT` functions are implemented. These functions allow to expect a variable to have a certain value. If the actual value does not match the expected one, an error is returned.

Unit Tests

These tests belong to the white-box tests. Typical examples are boundary-checking the value of the arguments or checking expected errors. An example can be seen in Listing C.8.

Functional Tests

These tests are black-box end-to-end tests that involve the complete system including the security stick. An example can be seen in Listing C.9.

Symbolic Execution

Symbolic execution is the process where program inputs are replaced with symbolic values and concrete program operations are replaced with symbolic manipulations. This allows to reach a higher code coverage and is, in comparison to the previous two processes, an automated approach. In this thesis, KLEE is used to execute the tests. An example of a test file that uses KLEE can be seen in Listing C.10.

4.4. Use Case I: Secured Configuration

This section contains information about the development of the first demonstrator. However, part of this thesis is just the security back end, consisting of the security stick and an according Java Card applet. The design of this applet is described in Section 3.4.

4.4.1. Java Card Applet

As explained in Subsection 4.1.2, it must be distinguished between the installation phase and the process phase. Additionally, as the applet uses two different buffers (encrypted and unencrypted), a helper class is implemented that represents such a buffer.

4.4.1.1. Storage Helper

The storage helper is the representative class for a buffer. It contains the following members:

- The raw buffer (byte array)
- The current fill size (short)
- A boolean that indicates if the content of the buffer is valid
- A send index (short)

Additionally, methods that allow the following functionality are implemented:

- Insert a byte array into the buffer
- Set the buffer valid or invalid
- Get the validity status of the buffer
- Send the buffer content as response

In general, the buffer is set to invalid. If the correct signature is received, it is set to valid. However, as soon as the buffer is modified in any way, it is set invalid again. A buffer must be valid to send it.

4.4.1.2. Constructor

During the installation, two storage helper objects are instantiated: one for the decrypted and one for the encrypted buffer. Additionally, the following members are instantiated:

- A TRNG instance
- A HMAC-SHA2-256 signature instance with a respective key
- An AES-256-CBC-PKCS5 cipher instance with a respective key
- A transient buffer

The keys are initially set to the zero key.

4.4.1.3. Process

This function is executed if an APDU is received. Depending on the INS byte, the respective functionality is executed. If the INS byte is unknown, an exception is thrown (incorrect INS).

Set Key

The keys are received encrypted and authenticated, therefore the payload has always 16 bytes IV + 32 bytes encrypted data + 16 bytes padding + 32 bytes signature. If the length is not correct, an exception is thrown (data invalid). Then, the signature is verified and the actual key is decrypted. Subsequently, depending on P1, it is decided if the AES or the HMAC key is set. Another P1 leads to an exception (incorrect P1/P2).

Store

The store command has the three states init, update and final. During the init state, the cipher, signature and decrypted buffer instance is initialized, which clears the according internal state. Additionally it is possible that the init step already contains encrypted data. In this case the HMAC instance is updated with the cipher text and the decrypted plaintext is appended to the decrypted buffer. However, the buffer is kept invalid.

Then, any number of update commands can be received. An update step includes an update of the HMAC instance with the cipher text and then a concatenation of the decrypted bytes to the decrypted buffer.

The last bytes of the message can be sent as part of the final step. However, the last 32 byte of the data must contain the signature of the message. If the received APDU command has a smaller size, a wrong length exception is thrown. Then, the remaining bytes previous to the signature are inserted into the HMAC instance and then the signature is verified with the internal state of the HMAC instance. If the data could not be verified, an invalid data exception is thrown. Otherwise, the final decrypted cipher text is appended to the decrypted buffer and the buffer is set to valid.

Encrypt

The encrypt command also has the three states init, update and final. The initial step first uses the TRNG to create a random IV for the encryption. As the IV is previous to the actual encrypted data, the length of the incoming data is not allowed to exceed 256 - 16 (size of the IV) bytes. Then, the IV and the data is encrypted with the freshly initialized AES instance. The cipher text is additionally added into the freshly initialized HMAC instance. If the P2 byte is set accordingly, the data is stored inside the encrypted buffer too. Then the cipher text is returned to the host.

The update step just updates both instances and additionally adds the cipher text to the encrypted buffer, if wanted.

Finally, the final step is not allowed to exceed 256 bytes - 16 bytes padding - 32 bytes signature. Then, the plaintext is encrypted, the cipher text inserted into the response buffer, the response buffer inserted into the signature instance and then the final signature is appended to the response buffer. Again, if specified the response buffer is appended to the encrypted buffer and finally the buffer is set to valid.

Fetch

The fetch command can be used to fetch the arbitrary data from either the decrypted or the encrypted buffer. If the chosen buffer is valid, the send process starts. As the maximum size of responded bytes is 256, the storage helper uses the internal index to keep track of the current position within the array. Therefore, the last byte of the response is used to inform the host if there is still data to fetch. If the response is shorter than 256 bytes, the host implicitly knows that there is no more data.

4.5. Use Case II: Secured Network Protocol Translation

This section contains information about the second use case: translation.

Device	/dev/eth0	/dev/eth1
External Operator	192.168.0.2/24	—
MQTT Translator	192.168.0.1/24	192.168.1.1/24
Internal Operator	192.168.1.2/24	—
Internal Equipment	192.168.1.3/24	—
Packet Sniffer	192.168.1.200/24	—
SECS/GEM Translator	192.168.1.100/24	192.168.2.1/24
Legacy Equipment	192.168.2.2/24	—

Table 4.5.1.: Table with IP addresses for each entity. Subnet 192.168.0.0/24 represents the External Network. Subnet 192.168.1.0/24 represents the Main Network. Subnet 192.168.2.0/24 represent the Legacy Network. The translators need two NICs as they interact between two different subnets.

4.5.1. Project Structure

The *bin* folder contains the final binaries and the Python scripts. The *build* folder is used for the out-of-the-source build artifacts. The *cert* folder contains the certificate of the CA and the generated local X.509 certificate that is based on the connected security stick. The *config* folder contains the configuration for the MQTT broker, e.g. which TLS version should be used. The *ext* folder contains the external libraries `mosquitto`, `open62541` and `secsgem`. The *misc* folder contains the `screen` configuration that will be described further in Subsection 4.5.3. Additionally, it contains the autostart units that will be further described in Subsection 4.5.10. The *src* folder contains the code base for the host applications. The *wrapper* folder contains additional initialization that must be done previous to the host applications, such as exporting the DISPLAY variable to direct the graphical output to the correct display output. The root *CMakeLists.txt* is used to organize the whole project with external libraries and internal code base. The *sync_raspi.sh* file is described in Subsection 4.5.3.

4.5.2. Network Setup

The demo is built up like described in Section 3.5. Three sub networks are created: External Network, Main Network and Legacy Network. The address allocation of each device can be seen in Table 4.5.1. The translators additionally serve as firewalls between the respective networks. The IP packet forwarding is disabled within the kernel, the only possibility to reach a neighboring network is by using the translator applications.

A firewall should provide a strict separation between the networks, allowing just known hosts to send messages to the translators (especially to the external one). This is done by the Linux user space application “iptables”.

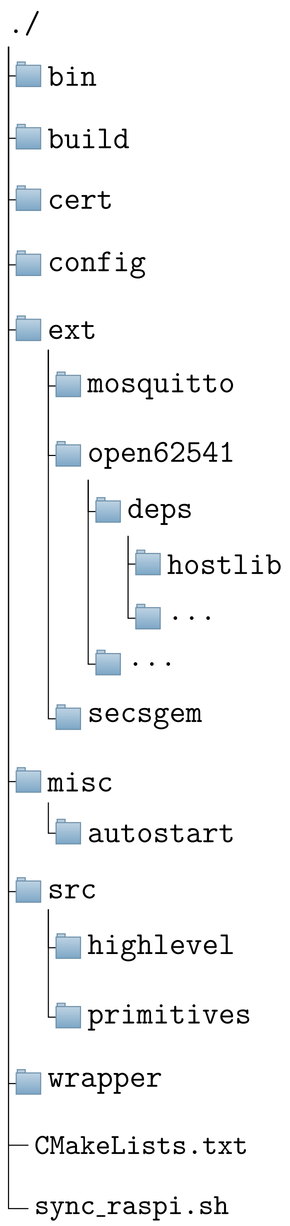


Figure 4.5.1.: Folder structure of the translation demonstrator. CMake Files in the source subfolders are neglected for readability reasons. Also the build target folders are neglected. Style taken from [110].

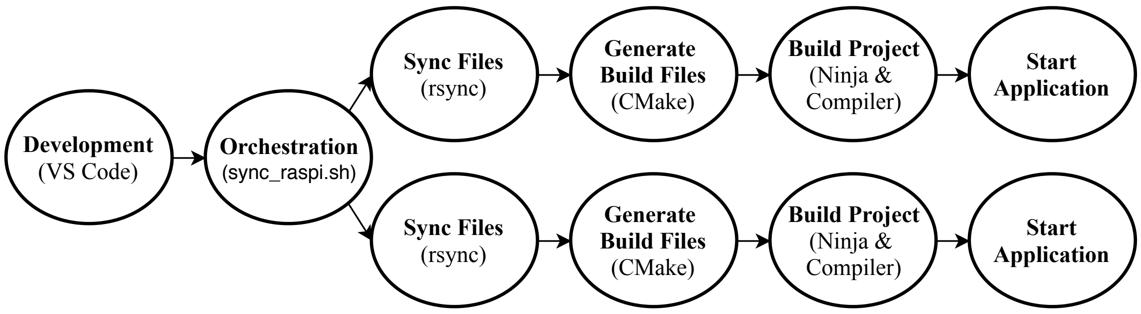


Figure 4.5.2.: After the development, the orchestration script is used to sync the code base to all entities. Then, the build process is done and finally the applications are automatically (re-)started.

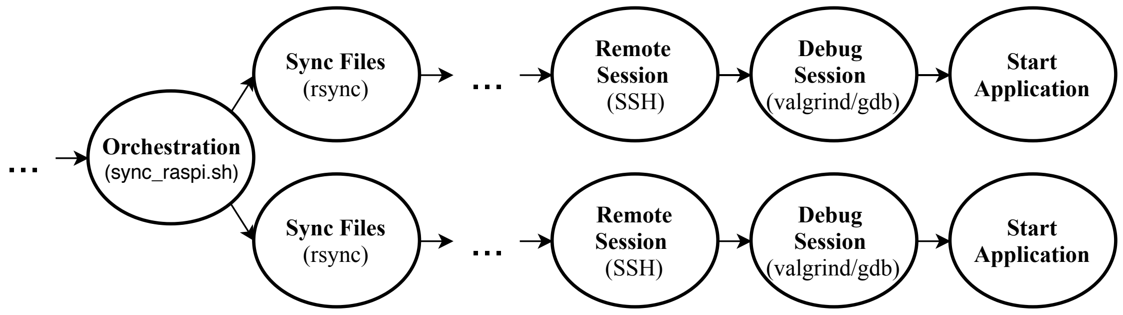


Figure 4.5.3.: Similar to Figure 4.5.2, but additionally the CMake debug build is activated. Then, after the applications are built, which is neglected on this illustration, remote sessions to the entities are established. Additionally, the applications are started in a debug session (either valgrind or the gdb debugger). This allows memory leak detection and step by step debugging.

4.5.3. Synchronization

This subsection describes the process of synchronizing the files between all participating devices. During development, my working machine was part of the Main Network and was used to distribute the project structure to all entities. For this purpose, a shell script (*sync_raspi.sh*) was developed. An illustration can be seen in Figure 4.5.2 for the release process and in Figure 4.5.3 for the debug process. The script uses the following technologies and applications:

rsync

The tool rsync [115] is an user space program to synchronize files across networked computers. The algorithm is based on delta encoding and considers modification times and size of files to minimize network usage. Rsync is used to synchronize the whole code base to all entities.

SSH

Secure Shell (SSH) is a network protocol that allows to securely connect to remote systems. Usually a shell is started after the connection. This allows to execute commands remotely through the SSH channel. During development, SSH was used to build, execute and debug the host applications on the remote systems. Additionally, it was used to get a “tunnel” to the hosts of the External and the Legacy Network.

GNU screen

GNU screen [116] is a window manager that multiplexes a physical terminal between several processes. This allows to maintain several SSH connections in one window. GNU screen supports the use of a config file. This config file can contain tabs and commands that should be executed at startup. During development, GNU screen was used to connect to all entities by SSH and keep the connections located in one window. Additionally, a debug configuration was created that automatically starts the respective host application in a valgrind or gdb session on the remote host.

4.5.4. Java Card Applet

This subsection contains the implementation of the applet designed in Subsection 3.5.6.

4.5.4.1. Constructor

During the installation, a local 2048-bit RSA key pair is generated. Then, storage for 16 identities is reserved. For each identity, the following members are allocated:

- Local and remote AES-256 key
- Local and remote HMAC-SHA2-256 key
- Local and remote IV for the symmetric cipher
- Local and remote nonce for the symmetric key generation
- Remote RSA-2048 public key

As the host library encapsulates the according init-update-final to one atomic operation, it is not necessary to have more than one instance for one specific cryptographic operation. The other members are:

- A TRNG instance
- A RSA-SHA2-256-PKCS1 signature instance (CSR)
- A HMAC-SHA2-256 signature instance (KDF)
- Transient buffers for intermediates (e.g. KDF)

4.5.4.2. Process

The following part describes the process that happens when an APDU command is received. Depending on the INS byte, the corresponding functionality is executed. An additional remote identity must be specified for most of the instructions. This is done as part of the P1 argument.

Set Key

If P1 indicates that the own RSA key pair should be set, a new key pair is generated internally. Otherwise, P1 indicates the identity and if the exponent or the modulus should be set. If the modulus is chosen, the length must be 255 and additionally the P2 byte must be used as last byte.

Get Key

The Get Key command enables the host to fetch the local public key. Depending on the P1 byte, either the exponent or the modulus is sent. Other keys cannot be fetched and result in a wrong P1/P2 exception.

MSG Encrypt & Decrypt

These instructions represent the common symmetric cipher operations, done in the common three steps: init, update and final. In the first step, the AES instance is initialized with the respective mode (encryption or decryption), the respective identity key and the respective IV. The second step simply updates the cipher instance. The final step additionally appends or removes the padding. Each step generates output, therefore each call results in a response.

MSG Sign & Verify

The signature process is analogous to the cipher process but just the final step generates an output.

Generate Nonce

The Generate Nonce command also requires an identity. Then, the local nonce of the respective identity is filled with random bytes.

Generate Key

This command represents the P-SHA2-256 algorithm. At this point, the respective local nonce was generated randomly and the respective remote nonce was extracted during the decryption of the OPN message. Both nonces are used to generate the IVs and keys for the corresponding identity.

Generate Random Number

This function simply returns the requested amount of random bytes.

OPN Sign

Again, the sign process of an OPN message has the three steps: init, update and final. The first two steps do the casual init and update process. However, the final step additionally inserts the local nonce as the host is not aware of it. Subsubsection 2.3.1.3 shows the structure of the message. The last bytes contain the padding. The value of the last byte also indicates the amount of padding bytes. These bytes are stripped off. Then, the requested lifetime (4 bytes) is stripped off. This is just part of the request (OPC UA client), the response (OPC UA server) does not contain this field. The next field is the local nonce, therefore the next 32 byte are replaced by the internally stored ones.

OPN Encrypt

In the encryption case, the local nonce is part of the first bytes. As Figure 2.3.2 shows, the message and security header are not part of the data to encrypt. The sequence header and the body are static, therefore a constant offset can be used. As the request and the response slightly differ, the local nonce of the request (OPC UA client) starts at index 57 and the local nonce of the response (OPC UA server) starts at 64. The next 32 bytes are replaced by the according local nonce. The update and the final step are the casual steps.

OPN Decrypt

Analogous to the encryption process, the decryption process extracts the remote nonce from the decrypted bytes, stores it accordingly, and then replaces the according bytes of the response with random bytes. Therefore, the host is not able to learn the remote nonce.

OPN Verify

For the verification, the remote nonce has to be inserted back to the correct position, since otherwise the signature cannot be verified with the previous inserted random bytes. The other steps behave like expected.

CSR Sign & Cert Verify

These functions again map to the respective method of the respective crypto instance.

4.5.5. External Libraries

Instead of implementing the communication protocols SECS/GEM, OPC UA and MQTT by myself, open-source third-party implementations are integrated into the project. In general, the own implementations are implemented in the C programming language. However, some libraries are only available in Python and therefore had to be integrated differently into the C code base. Also, since the build environment of this project is based on CMake, libraries that also use CMake are preferred.

4.5.5.1. secsgem

The secsgem [45] library is an open-source implementation of the SECS/GEM protocol with underlying HSMS connection layer. It is the only open-source implementation of the proprietary SECS/GEM protocol, and therefore had to be used in this work. Additionally, as it is implemented in pure Python language, the implementation is platform-independent, as long as a Python interpreter is available.

4.5.5.2. open62541

The open62541 [117] project is an open-source implementation of the OPC UA communication protocol written in the C language. Additionally, it provides native support for different architectures beside GNU/Linux, e.g. Microsoft Windows and FreeRTOS. The project offers a library where both, the server and the client, can be included separately. Also, the good structure allows an easy integration of the security stick, since the security policy of OPC UA must be handled by the security stick. This project uses mbedtls [118], the TLS implementation of ARM, as internal crypto library. Additionally, it is also based on CMake.

4.5.5.3. mosquitto

Mosquitto [119] is an open-source implementation of the MQTT communication protocol, offered by Eclipse and written in C. Similar to open62541, it provides an easy-to-use API and an easy integration possibility as library. This project uses OpenSSL [120] as internal crypto library. Additionally, it is also based on CMake.

4.5.5.4. GTK 3

GTK [121] is an open-source GUI toolkit and part of the GNU project. It allows to easily build GUIs in the C programming language and therefore is used for the visualization of the Internal and the External Operator. Additionally, GTK is also available on several other OSs, e.g. Microsoft Windows.

4.5.5.5. scapy

Scapy [122] is a Python-based packet manipulation library. It is intended to be cross-platform and therefore runs on several platforms, including GNU/Linux and Microsoft Windows. The Packet Sniffer implementation is based on this library, as it provides the necessary tools for ARP poisoning.

4.5.5.6. TkInter

TkInter [123] is also an open-source GUI toolkit but part of the Python environment. As the first GUI toolkit developed for Python, it is the most commonly used one and is therefore part of the default Python packages and also implemented on other architectures, e.g. Microsoft Windows. Since the Packet Sniffer is based on scapy, the gathered data is visualized with TkInter.

4.5.5.7. wiringPi

WiringPi [124] is a library to control the GPIOs of a Raspberry Pi in a convenient way. An advantage of the library is that it is installed per default on Raspberry Pis. Additionally, it is ported to several programming languages, including C and Python. In this project, wiringPi is used to interact with the USS.

4.5.6. Library Adaptions

This subsection contains information about the adaptions of the external libraries.

4.5.6.1. open62541

At first, the host library has to be integrated into the project structure of open62541. Then, the code base of the open62541 project has to be adjusted to make use of the functionality provided by the application layer of the host library.

Structural Integration

At first, the host library is integrated as dependency of the open62541 library. This is done by adding the git repository of the host library as submodule inside of the *deps* folder. Additionally, the host library is included into the CMake structure. This is done by adding the *deps/hostlib* folder as subfolder of the main CMake file of the open62541 project. Additionally, the *sectrans* library, which can be generated with the help of the CMake file that is stored in the previously included subfolder, is added as dependency of the *open62541* CMake target. Further, the include folder of the host library is added to the CMake targets by adding it to the *include_directories*. Since the open62541 project has the security policy *Aes256Aes256-Sha256-RsaPss* not implemented yet, a new source file *plugins/ua_securitypolicy_aes256sha256rsapss.c* is added beside the existing *plugins/ua_securitypolicy_basic256sha256.c* file too.

Functional Integration

At first, the new security policy is implemented in *plugins/ua_securitypolicy_aes256sha256rsapss.c*. As base file, the existing security policy is taken. This file provides one function for every required crypto operation. These functions internally call the underlying mbedtls library, which must be replaced with calls to the security stick. Listing C.11 shows an example of the initial state and Listing C.12 shows the according replacement.

The same process is done for the cryptographic operations.

Then, the new security policy must be added to the available security policies, which is done in the *plugins/ua_securitypolicies.h* file. After that, the default configurations for a server and a client have to be adjusted, which is changed in *plugins/ua_config_default.c*. Both use either the new security policy (if the security stick is connected) or no security policy. This is done to fulfill the requirement of the demonstrator mode, as described in Subsection 3.5.2.

4.5.7. Additional Hardware

This subsection contains information about the additional hardware that is used in this project.

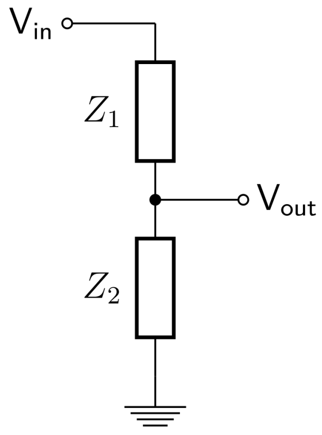


Figure 4.5.4.: Illustration of a voltage divider. V_{in} is connected to the echo pin of the USS, V_{out} is connected to a GPIO of the Raspberry Pi, Z1 is set to 10 kΩ and Z2 to 15 kΩ. Taken from [125].

4.5.7.1. Ultrasonic Sensor

The Legacy and the Internal Equipment both require a connected USS to gather the value for the private “range” variable. Therefore, an additional development kit is used. This kit offers connection pins to the GPIOs and a small breadboard, where the USS is placed. The USS in use (“HC-SR04”) has four pins: VCC, GND, a trigger and an echo pin. The VCC and GND can be directly connected to the respective pins of the Raspberry Pi. One problematic part is that the USS is designed for TTL level but the operating voltage of the Raspberry Pi is just 3.3 V instead of 5 V. However, the trigger pin also reacts to the 3.3 V and therefore can be directly connected to a GPIO of the Raspberry Pi. Consequently, as soon as the trigger pin is set to `HIGH` for 10 us, the USS sends out an 8 cycle burst of ultrasound at 40 kHz. In contrast, the input GPIO does not tolerate 5 V. Hence, a voltage divider has to be interposed between Raspberry Pi and USS. The voltage ratio from 5 to 3.3 (0.66) is nearly the same as the resistor ratio from 15 to 10 (0.67), therefore a 1.5 kΩ and a 1 kΩ resistor are used. An illustration of a voltage divider can be seen in Figure 4.5.4. As the USS becomes inaccurate on longer distances, the maximum distance is capped at 200 cm.

The sent out signal is reflected on a near target and then measured by the USS. The USS sets its echo pin to `HIGH` as soon as the signal left and back to `LOW` as soon as a reflected signal is measured. To get the measured distance, the `HIGH` time is measured and multiplied with the velocity of sound (340 m/s) divided by 2 (outgoing and incoming length). However, if no signal is measured in around 23 ms (around 400 cm), the USS automatically sets back to `LOW`.

The Legacy and the Internal Equipment both use the same circuit and wiring. GPIO 5 is used to trigger the USS and GPIO 6 is used to catch the echo signal from the

USS. The final setup can be seen in Figure 5.1.2 and in Figure 5.1.3.

4.5.8. Library Abstractions

The *primitive* and *highlevel* subfolder of the *src* folder contain the abstractions and implementations written in C. In general, the *primitive* layer encapsulates the functionality of a specific library. Then, the *highlevel* layer uses this implementation-independent interface to implement the functionality that is required by the use cases. This approach offers general-purpose interfaces that wrap the implementation details (*primitive* layer) and further an implementation-independent interface that is tailored to the required functionality for this demonstrator (*highlevel* layer). The functionality of the Python implementations are directly implemented into the Python files inside the bin folder.

4.5.8.1. Common Layer

Since the host applications finally mixes different implementations with different representations, the used data should be abstracted in a generic way. The common layer contains definitions that are used in all implementations. One noteworthy aspect of OPC UA is that the strings are not defined as zero-terminated character arrays ("C strings") but as struct that consists of a length field and a non-terminated byte array. This interferes with different implementations that rely on C strings and therefore a common string type and respective conversion methods are implemented. Additionally, common data types for server variables and variable lists are implemented. This eases the transfer of variables from one protocol to another.

Every function of the primitives abstraction layer makes use of these defined data types. This allows simpler and more structured host applications, since the implementation details are encapsulated in this layer. Additionally, the *src/config.h* C header file contains necessary information about the different entities, such as IP addresses, ports, paths to certificates and constant strings.

4.5.8.2. SECS/GEM Equipment

The SECS/GEM equipment functionality is encapsulated in the SecsGemEquipment class inside of the *bin/secsgem_equipment.py* file. Internally, it uses the secsgem Python library and therefore is implemented in Python. The constructor already creates the *range* status variable. Beyond that, three asynchronous alarms, which represent the *fillStatus*, are created. Each alarm includes two events: a set alarm and a clear alarm event. Additionally, a classification needs to be chosen for each alarm. The most accurate classification for the *fillStatus* is *equipment status warning*.

Beside that, other classifications, such as *personal safety* or *irrecoverable error*, exists. The class also requires a handler that is called if the value of a status variable is requested. As soon as the “range” is requested, the value is fetched from the USS. This is done as described in Subsection 4.5.7. Finally, the class gets methods that allow the application to modify the *fillStatus*. Internally, the member sets or clears an alarm. The equipment is the passive component and therefore waits until a host connects to it.

4.5.8.3. SECS/GEM Host

The SECS/GEM host functionality is split into two parts: the SECS/GEM host implementation, which uses the secsgem Python library and extends the functionality with required members and methods, and the API wrapper, which integrates and abstracts the SECS/GEM host functionality into the “C world” via a defined interface.

Functionality

Similar to the SECS/GEM equipment, the functionality is encapsulated in the SecsGemHost class inside of the *gem_host.py* file. The constructor already enables the specified alarms for the *fillStatus* variable. This is necessary since otherwise the secsgem library would discard an incoming alarm. Additionally, the handler for incoming alarms is implemented. The handler forwards incoming alarms to the “C world”. Additionally, the SECS/GEM host functionality must be able to fetch status variables and their actual values. This is done by two methods that internally call the *list status variables* to get all added status variables of the connected equipment and the *request status variable* to get the current value of a given variable. The host is the active component and therefore must connect to one or more waiting equipment machines.

Interface

As the SECS/GEM host functionality is implemented in Python and the chosen programming language for the host applications is C, an additional wrapper is implemented which allows the host applications to access the SECS/GEM functionality. This is done by using three UNIX First-In-First-Outs (FIFOs) pipes for the communication between C wrapper and Python implementation. The first FIFO *fifo_opcua2secsgem* is used to transmit requests from the C world to the SECS/GEM host. The second FIFO *fifo_secsgem2opcua* is used to transmit the response from the SECS/GEM host to the C world. Finally, the FIFO *fifo_alarm* is used to forward asynchronous alarms coming from the SECS/GEM equipment to the host application.

The interface is kept generic so that it can easily be replaced with a native solution. For simplicity reasons, functions such as `Semi_SECSGEM_Host_enableAlarm` are kept as a stub, since they are already done in the Python implementation by default. However, `Semi_SECSGEM_Host_getStatusVariables` and `Semi_SECSGEM_Host_getValueOfStatusVariable` are implemented by interacting with the FIFOs. The chosen encoding is JSON, since the host library already uses a JSON library and the messages are smaller than messages encoded with XML. Additionally, JSON objects and arrays can be easily converted to Python objects and the message design is easily extendable. The *get variables* command and its response look like this:

```
// Request Example:
{
    "cmd": "get_variables"
}
// Response Example:
[
    {
        "key": "range",
        "name": "range",
        "unit": "cm"
    }
]
```

The *get value* command and its response look like this:

```
// Request Example:
{
    "cmd": "get_value",
    "arg": "range"
}
// Response Example:
{
    "value": 200
    "type": "int"
}
```

Additionally, the function to start up a SECS/GEM host takes a callback that is executed as soon as something is received on the alarm FIFO.

4.5.8.4. OPC UA Server

As previously described, the OPC UA server is split into a primitive and a highlevel part. The primitive part wraps implementation details about starting and stopping a server and managing its local variables. Most of the calls are targeted to the underlying open62541 library. However, the highlevel layer wraps these functions to three actual use cases that are required by this demonstrator.

Primitive

The primitive layer of the OPC UA server is implemented in the files *src/primitive/opcua_server.h* and *src/primitive/opcua_server_open62541.c*. Internally, a OPC UA server is represented by its configuration, which is implemented in the implementation-independent header file. The config contains a list with its variables and the internal implementation-dependent config. The configuration of the primitive layer can be seen in Listing C.13.

The first function is the function to create such a config, depending on the underlying OPC UA implementation, in this case *open62541*. The internal configuration for this implementation contains the configuration of the open62541 server, the actual open62541 server "object", the running state of the server, a function pointer that is executed if the value of a variable should be read, and a context, which provides additional information to the read handler. The `Semi_OPCUA_Server_createConfig` function is used to instantiate such a configuration. The actual representation can be seen in Listing C.14.

The second function, the `Semi_OPCUA_Server_start` function, wraps the start function of the open62541 implementation and results in an endless loop that can be interrupted by setting the *running* boolean to false. This is done with the stop function.

The third function is used to add an OPC UA variable to the server. This function prepares a node, containing the access level (e.g. "read only"), the display and the browse name, the node ID, the node ID of the parent node, the data type and the node type. Additionally, a "data source" can be specified. If the variable is specified as data source variable, an additional function is called where the value of the variable is set. In this case, a couple of configurations can already be preset by the implementation: the access level is set to read only, therefore no OPC UA user can change the values. The display and the browse name are set to the same string. The parent node ID is always set to NS0, which represents the "root" of an OPC UA server's nodeset.

The fourth function is used to to change the value of a variable that does not have a data source.

Highlevel

The highlevel functions cover demonstrator-specific functionality on a high abstraction level. The first function `Semi_OPCUA_Server_Create_HL` covers the creation of an

OPC UA server. This includes the initial check if the security stick is connected. However, in this demonstrator, the OPC UA server always offers two endpoints: a secure (*Aes256-sha256-RsaPss*) and an unsecure (*None*). Additionally, the certificate is fetched from the file system and forwarded to the implementation.

The second function `Semi_OPCUA_addVariable_stringKey_HL` allows to add a variable with a string as node ID.

4.5.8.5. OPC UA Client

The OPC UA client also has a config struct, similar to the OPC UA server. This config represents a connection to an OPC UA server. Additionally, three functions are required: The first function is used to connect to a OPC UA server. This function asks the OPC UA server internally which endpoints it has available. Then, depending if "secured mode" is active, it connects to the secure or unsecure one.

The second function is used to get the variables of an OPC UA server, similar to the SECS/GEM host. The third function, also similar to the SECS/GEM host, can be used to fetch the current value of a variable.

4.5.8.6. MQTT Client

Again, an own config type is used that represents the connection to a MQTT broker. Since this is the same for publisher and subscriber, and the functionality only include publishing and subscribing, these are aggregated in a common *mqtt_client.h* and a *mqtt_client_mosquitto.c*. Again, the subscribe function lets the caller define a callback function which is executed as soon as a message is published on a given topic.

4.5.8.7. Graphical User Interface

Since GUI toolkits usually follow their own design principle, the functionality of GTK could not be abstracted implementation-independently. In general, the primitive layer contains functions to:

- Create a window
- Set the window color
- Create a label
- Modify a label (font, color)
- Create an image
- Create a virtual and a horizontal box
- Add elements to boxes and containers
- Add a timer that executes something synchronously
- Start the GUI

Starting the GUI results in an endless loop that handles GUI events. To be able to still communicate, a timer job is added to the GUI handler.

Since the same GUI should be used for the Internal and the External Operator, the highlevel aggregates the underlying functionality and provides a function to create it. Only the title and the labels can be changed by the caller. Additionally, the labels can be proposed to the outside. Another highlevel function, `Semi_GUI_setLabelToVariableValue_HL` can then set the according label to the value of a variable.

4.5.8.8. CSR Generation

The CSR generator is an application that can be used to generate a CSR based on the asymmetric key pair of the security stick. To achieve that, parts of a generated valid CSR are stored hard-coded and then modified. The base CSR was generated by OpenSSL and then analyzed with an online ASN.1 parser. This approach is static but simpler than using an ASN.1 generator to craft the structure of the CSR. The only thing that the user has to provide is the common name of the entity. For security reasons, just valid/known names are allowed.

At first, the host library is used to check if a security stick is connected. Then, the public key is fetched from the security stick. After that, the CSR is crafted with the public key and the common name. Subsequently, the CSR is sent to the security stick to sign it. The signature is then appended to the CSR and the final string is stored in the local file system. After that, the *cert* folder is checked for a CA certificate and key. If found, the CSR is “converted” to a certificate. This is done via OpenSSL in software, as a local CA is in use for this thesis. Subsection 6.2 contains information about switching to a remote one.

4.5.9. Host Application Development

This subsection contains information about the host applications running on the seven different Raspberry Pis, representing one of the defined network entities.

4.5.9.1. Legacy Equipment

The host application of the Legacy Equipment is implemented in the same file that also contains the functional abstraction, namely *bin/secsgem_equipment.py*. Additionally, the handling of the USS is implemented here. The function *get_range* is called as soon as the SECS/GEM host requests the value of the status variable *range*. The main loop of this host application manages the value of the *fillStatus* variable. A counter is incremented every second and, depending on the current counter value, the status value is either `Refilled`, `Refill` or `Critical`. A triggered alarm indicates a state switch to the connected SECS/GEM host.

4.5.9.2. SECS/GEM Translator

The host application is used to translate between the SECS/GEM protocol on the legacy side and the OPC UA protocol on the internal side. This means, it represents a SECS/GEM host and an OPC UA server that interact together.

The host application starts by creating the FIFOs and then starts up the SECS/GEM host. Subsequently, the OPC UA server is started by using the provided highlevel abstraction. Then, the status variables of the connected SECS/GEM equipment are fetched by using the provided interface. Further, the collected variables are transferred to the OPC UA world by adding them to the nodeset of the OPC UA server. All SECS/GEM status variables are added as data source variables. The OPC UA node ID is set to the SECS/GEM key and the OPC UA browse and display name is set to the SECS/GEM name, which allows an easy mapping. Since the variables are data sources, OPC UA value requests can be directly forwarded to the SECS/GEM equipment. Additionally, an OPC UA variable is added which represents the value of the *fillStatus*. Since the alarms are usually used in a different way, but in general just the translation should be shown, this is the easiest way to follow the current value and value changes of the *fillStatus*. Further information about this topic can be found in Subsection 6.2. However, there is no possibility to get the name of the variable, which is represented by the alarms, dynamically. Therefore, this information is stored inside the common config file. Finally, the SECS/GEM alarm handler is set up to change the value of the created *fillStatus* variable to the message that is contained inside of the alarm.

4.5.9.3. Internal Equipment

The Internal Equipment represents an OPC UA server. Additionally, it is connected to a USS and therefore has to implement the functionality to get the actual distance for the *range* variable. At first, the host application creates a configuration for the OPC UA server. Then, it adds the two variables *range* and *fillStatus* to the nodeset of the OPC UA server. Both are added as data source variables. Therefore, if a request is received, either the *range* is fetched from the USS or the current *fillStatus* value is calculated from the current time. This function is provided as callback for the data source variables. Finally, the OPC UA server is started.

4.5.9.4. Internal Operator

The Internal Operator represents an OPC UA client. Additionally, it is connected to a display and therefore has to implement a GUI. This is done by calling the highlevel function to generate the default window. Additionally, two label references are provided to the call, which are then used to update the actual values.

Then, a thread fetches the values in the background. This thread first creates variable containers for the four values: *range* and *fillStatus* of Internal and Legacy Equipment. Then, an endless loop is started. The loop first checks if a security stick is connected to the device. Additionally, it maintains a state showing if the OPC UA client configuration is currently connected or not. This leads to three different possibilities, regarding connected security stick ("secured mode") and connected OPC UA server:

1. Connected in secured mode
2. Connected in unsecured mode
3. Not connected at all

If number 1 is the case and the security stick is still connected, everything is as expected. If number 1 is the case and the security stick is not connected, the OPC UA client has to reconnect to the respective OPC UA server but this time with the unsecured mode (done internally). This represents the demo case: someone disconnects the security stick from the Internal Operator.

If number 2 is the case and the security stick is still not connected, everything is as expected, since the unsecured demonstrator case is active. If number 2 is the case and the security stick is connected now, the previous connection is disconnected and connected again in secured mode (done internally).

If number 3 is the case, it is the first time (or a network error occurred), so we connect. The connection functionality checks internally if a security stick is connected and therefore uses the respective endpoint of the OPC UA server.

If the connection was successful, the values are gathered from the respective OPC UA server. Otherwise the values are set to “-”, indicating that no connection could be established. After that, a sleep time is added and then the loop starts from its beginning.

However, parallel to that, a label update job is added to the GUI handler. This job uses the respective variables and updates the according labels. Since the job runs in a different thread than the fetch loop, a mutex is used to protect the variables from race conditions. The job also checks if the unsecured or the secured mode is active and updates the GUI accordingly.

4.5.9.5. MQTT Translator

The MQTT Translator is the bridge between External and Main Network. Therefore, it represents an OPC UA client and a MQTT publisher. Additionally, it runs a MQTT broker.

The application is an OPC UA client, which is additionally able to publish on a MQTT broker. First, the application connects to the MQTT broker. Then, it connects to both OPC UA servers of the network. Subsequently, it creates the containers for the public variables. The information if a variable is either public or private cannot be found in the network. Therefore it is taken from the common config file.

Then, the event loop starts. If not connected (e.g. connection lost), the application connects to both OPC UA servers. After that, the values of the variables are fetched. If the value has changed since the last time, the new value is published to the respective MQTT topic. After a defined sleep time, the loop starts from its beginning.

4.5.9.6. External Operator

The External Operator is implemented as MQTT client. Since it is connected to a display, it is structured similar as the Internal Operator.

First, the MQTT client connects to the MQTT broker, which runs on the MQTT Translator. Then, it subscribes to the two default topics which are known from the common config file. As handler, the two variables are updated to the new value. Similar to the Internal Operator, a label update job is added that shows the current values of the variables.

4.5.9.7. Packet Sniffer

The Packet Sniffer is a standalone application. It consists of three parts: Address Resolution Protocol (ARP) spoofing, parsing traffic and visualization of results. However, the network manipulation is done with scapy, which allows to easily craft network packages. The GUI is designed with TkInter.

Address Resolution Protocol

In general, the Address Resolution Protocol (ARP) is used to discover the Media Access Control (MAC) address of a device of which only the IP address is known. This is crucial, since the MAC address of a device is required to target Ethernet frames to it. Essentially, an ARP packet contains four information fields: type of the packet (request or response), the sender's hardware and protocol address, and the target's hardware and protocol address. As soon as a device wants to send data to another device of which only the IP address is known, it sends out such an ARP request, which contains the target IP address, to the broadcast address (ff:ff:ff:ff:ff:ff). Frames that are sent to the broadcast address are received by all participants within the same network. Normally, the respective host answers with an ARP response, containing its own MAC address as sender's hardware address so that the requester can add it to its own ARP cache. However, the protocol has an additional feature that ARP responses are also allowed without previous request. This is normally used during the startup of a device, to inform the other network participants that a new host is online.

ARP Spoofing

ARP spoofing (also ARP poisoning) is a technique where the ARP cache of a host is "poisoned". This means that a wrong entry is inserted into the ARP cache of the victim, allowing to impersonate an IP address and therefore receiving all data targeted to another host, e.g. the gateway. At first, the MAC address of the Internal Operator and the Internal Equipment is gathered by sending an ARP request with the respective IP address to the MAC broadcast address. If online, both respond with their respective hardware address. Then, new ARP responses are crafted letting the Internal Equipment think that the Packet Sniffer is the Internal Operator and the Internal Operator that the Packet Sniffer is the Internal Equipment. Figure 4.5.5 shows the state of the affected participants. Then, IP forwarding is activated on the Packet Sniffer. On Linux systems, this can be done by writing the value 1 into */proc/sys/net/ipv4/ip_forward*. This is required so that the packet is forwarded to the intended receiver. Finally, all the network traffic between the two affected entities is sniffed and analyzed. One of the reasons for using this approach was to implement a realistic "person-in-the-middle" attacker scenario, which typically is used in reality by potential attackers.

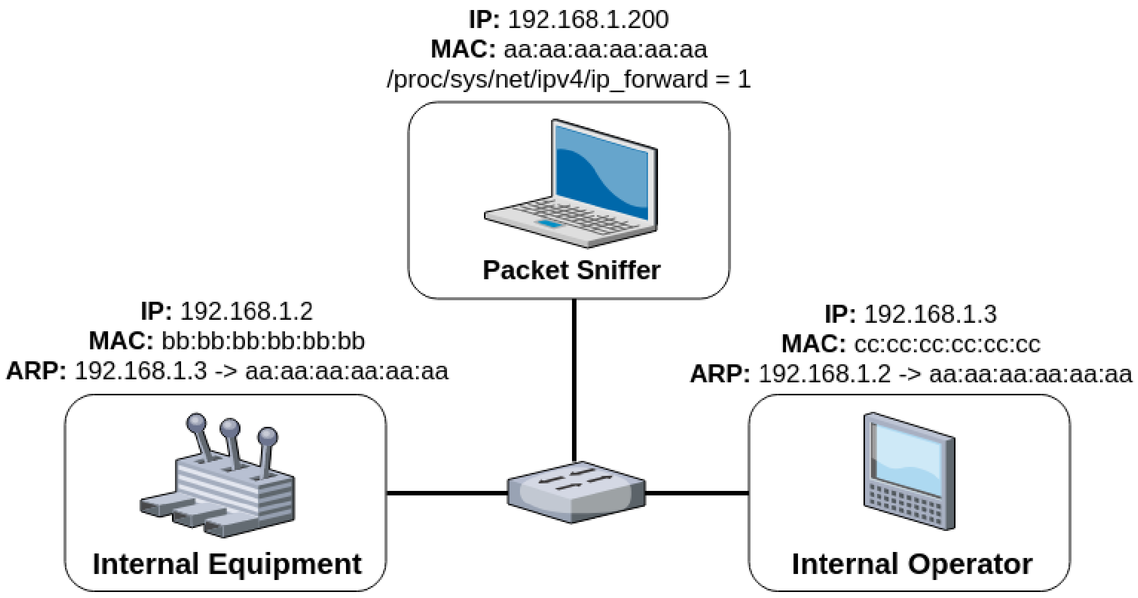

IP: 192.168.1.200
MAC: aa:aa:aa:aa:aa:aa
/proc/sys/net/ipv4/ip_forward = 1
Packet Sniffer
IP: 192.168.1.2
MAC: bb:bb:bb:bb:bb:bb
ARP: 192.168.1.3 -> aa:aa:aa:aa:aa:aa
Internal Equipment
IP: 192.168.1.3
MAC: cc:cc:cc:cc:cc:cc
ARP: 192.168.1.2 -> aa:aa:aa:aa:aa:aa
Internal Operator

Figure 4.5.5.: Illustration of the network after poisoning ARP caches of the Internal Equipment and the Internal Operator. The Internal Equipment and the Internal Operator both think they communicate with each other but in fact both communicate with the Packet Sniffer (person-in-the-middle).

Traffic Parsing

At first, the network traffic is filtered by the IP address of the Internal Operator. The sniff function of scapy additionally requires a “PRN”, which is specified as “function to apply to each packet”. Since the OPC UA MSG messages have a static format, the data type of the transmitted variable, which is part of the encrypted payload, always has the same offset (123). This must indicate either a string (*fillStatus*) or a 32 bit integer (*range*). The next four bytes indicate a four byte number: either the length of the string or the integer itself. In this demonstrator, both possible integers have a value below 256 and therefore, the three most significant bytes have to be zero. If this is not the case, the message is classified as encrypted and cannot be parsed. Otherwise, either the *range* or the value of the *fillStatus* can be parsed.

Visualization

The GUI is designed with TkInter. the main frame contains two text fields: the top one for bytes and the bottom one for the bytes converted to ASCII. All non-printable characters are replaced with dots. Additionally, “tags” are defined which highlight tagged text in yellow. For demonstration purposes, the text fields are updated only after a defined time span and do not contain the full packet (more than 128 bytes). If the parse process detects valid values, the according bytes and characters are highlighted yellow. Figure 5.1.4 shows the Packet Sniffer with encrypted payload, Figure 5.1.6 shows unsecured data with highlighted values.

4.5.10. Autostart Host Applications

This subsection describes how the host applications are automatically started at system startup. Debian comes with systemd as system and service manager. Beside of that, systemd is also used as init system and therefore controls the startup and initialization process. The tasks are organized in "units", and groups of units are organized in "targets". As example, the graphical.target contains all units that are necessary to start a device with GUI. Additionally, dependencies can be created, e.g. since some host applications come with a GUI, the display-manager.service must be started before the applications can be executed. The user space application to manage systemd is called "systemctl". An example of such a service file can be seen in Listing C.15.

5. Results and Evaluation

In this chapter, the results and evaluation of the work is described.

5.1. Results

This section contains the final outcome of this thesis and pictures of the entities and networks.

Security Stick

The hardware of the security stick used for the final implementation can be seen in Figure 3.2.2. The used hardware is designed in a small form factor and provides a USB type A connector and a tamper-resistant SE. Beside of a CCID dongle implementation, which enables communication via the USB interface, layer 3 and 4 of ISO 14443 are implemented, which enable contactless communication with the security stick. On the software side, a Java Card OS with GP support is implemented. This allows to maintain multiple Java Card applets on one security stick, each targeting one specific use case. The Java Card API provides access to common cryptographic operations like asymmetric RSA encryption with different paddings. Additionally, the startup phase is designed in a way that prioritizes contactless communication over USB communication. This behavior is required for the first use case, where an AGV must be able to interrupt the persistent USB connection to transmit data.

Host Library

The host library is a cross-platform library which enables communication with a USB-connected security stick. The layered structure allows to exchange implementations at compile time, for example the communication with the security stick can be done either by PC/SC or by virtual serial port. Additionally, the application layer allows to abstract the functionality of a Java Card application running on the connected security stick. This requires to inform the host library about the supported APDU

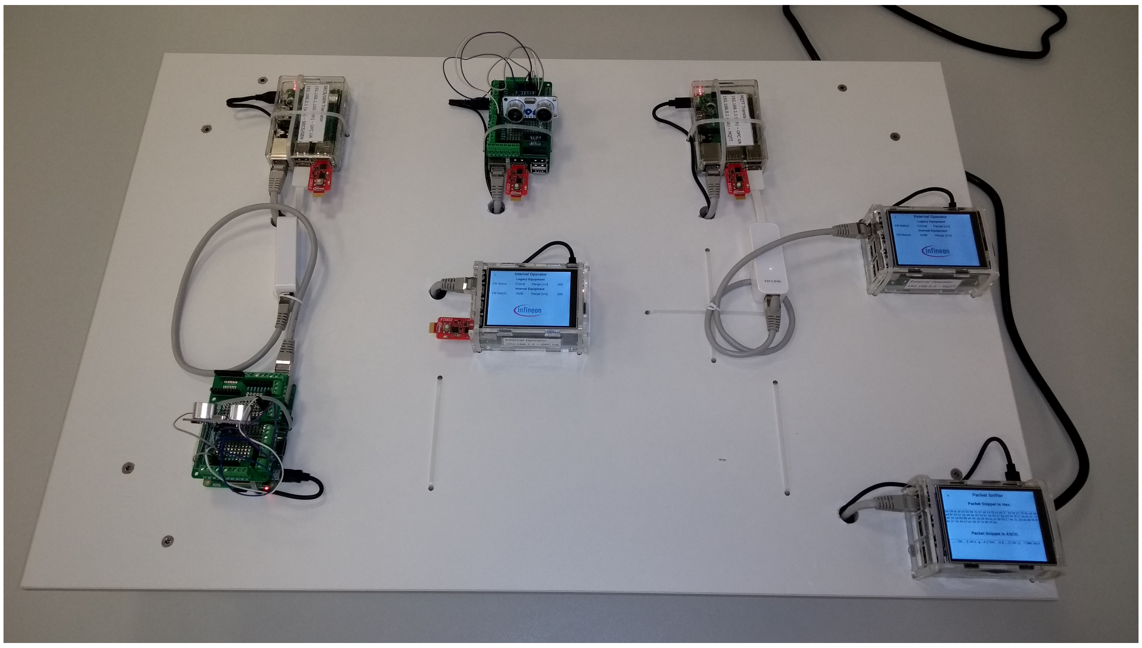


Figure 5.1.1.: Picture of the final translation demonstrator. Starting from top, left to right: SECS/GEM Translator, Internal Equipment, MQTT Translator; middle, left to right: Internal Operator, External Operator; bottom left corner: Legacy Equipment; bottom right corner: Packet Sniffer.

commands of the Java Card applet, which can also be done by providing a JSON file. The host library allows to conveniently use the security stick, hiding any complexity from the actual user.

Use Case I: Secured Configuration

This demonstrator, developed together with the project partner ITI, seeks to examine the capabilities of using the security stick as NFC receiver, secured storage and crypto coprocessor. The data, in this case configurations for production machines, is secured with authenticated encryption (AES-256-CBC and HMAC-SHA-256). This should allow only entites that are aware of the respective secret keys to create, modify and read the protected payload.

Use Case II: Secured Network Protocol Translation

The final demonstrator can be seen in Figure 5.1.1. The outcome is an implementation that integrates an unsecured legacy communication protocol network into the main network of a company. This is done by defining a gateway node, the SECS/GEM Translator, that translates status requests coming from the secured network to the unsecured network and vice versa. The Legacy Network consists of two entities: the Legacy Equipment and the above mentioned SECS/GEM Translator. Both entities can be seen in Figure 5.1.2. The Legacy Equipment represents a “data source” and therefore provides two different variables: the *range*, which is gathered from the connected USS and implemented as a SECS/GEM status variable, and the *fillState*, which depends on the current time and is implemented as SECS/GEM alarms. While the *fillState* is handled as “public variable”, which is also exposed to the External Network, the *range* is kept inside of the Main Network.

This implementation shows one approach to translate SECS/GEM alarms into the OPC UA world. In this case, three alarms are used to represent the three states of one variable. There are different approaches, such as using just one alarm and changing the alarm message accordingly. However, this is not practicable as alarms are typically used in a static manner, meaning one alarm represents exactly one state and is not changed dynamically during the run time.

As described, the SECS/GEM Translator is the interface to the Main Network. Additionally, the Main Network consists of the Internal Operator and the Internal Equipment. A picture of these two can be seen in Figure 5.1.3. The Internal Equipment provides the same variables as the Legacy Equipment. However, they are directly implemented as OPC UA variable nodes. The Internal Operator connects to the SECS/GEM Translator and the Internal Equipment and fetches the current values of the respective OPC UA variable nodes. As the Internal Operator is connected to a display, these values are visualized on the GUI.

For demonstration purposes, a Packet Sniffer is implemented. The Packet Sniffer is able to intercept the communication between Internal Equipment and Internal Operator by using ARP spoofing and further tries to parse the intercepted OPC UA messages. A picture of it can be seen in Figure 5.1.4. During the normal mode, the messages are encrypted by either RSA-OAEP (OPN type) or by AES-CBC (MSG type). However, just for demonstrational purposes, an “unsecured demo mode” is implemented too. This mode is entered as soon as the security stick of the Internal Operator is disconnected from the device. Figure 5.1.5 shows the Internal Operator without a connected security stick. Figure 5.1.6 shows the Packet Sniffer that can parse the unencrypted communication that results from the unsecured demo mode. The string value `Refilled` can be read with the respective ASCII encoding.

Finally, also the MQTT Translator is part of the Main Network and is used as interface to the External Network. The MQTT Translator fetches the public variables (the two *fillStates*) and publishes changes to the respective MQTT topic. The External Operator visualizes the new *fillStates* as soon as a change is received on the respective subscribed topic. The External Network can be seen in Figure 5.1.7.

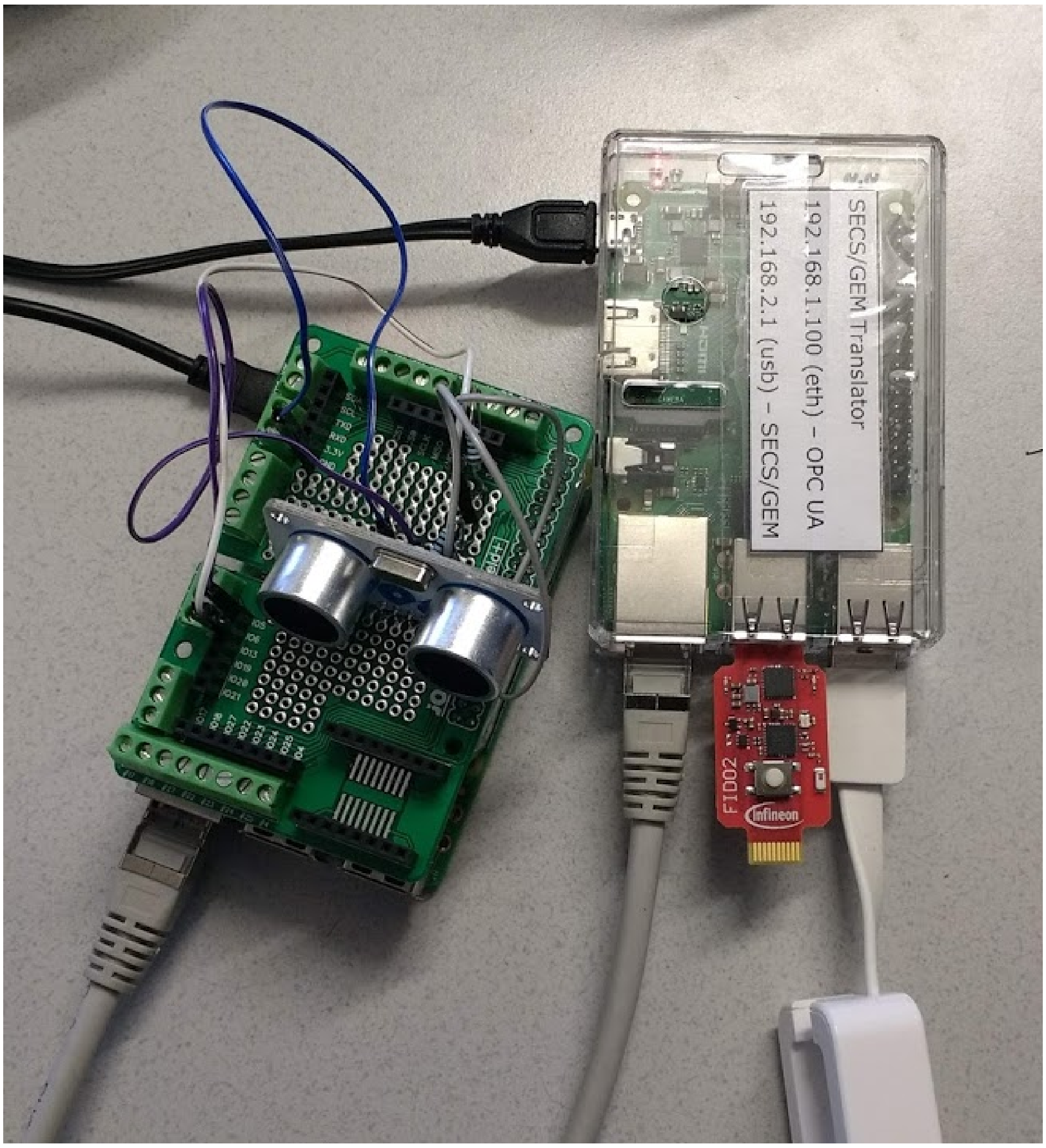


Figure 5.1.2.: Picture of the legacy network. From left to right: Legacy Equipment, SECS/GEM Translator. The Legacy Equipment has a Ultrasonic Sensor (USS) connected. The SECS/GEM Translator is equipped with a security stick and is connected via a USB NIC to the Legacy Equipment. The communication between these two is based on the SECS/GEM protocol. The integrated NIC connects the SECS/GEM Translator with the Main Network.

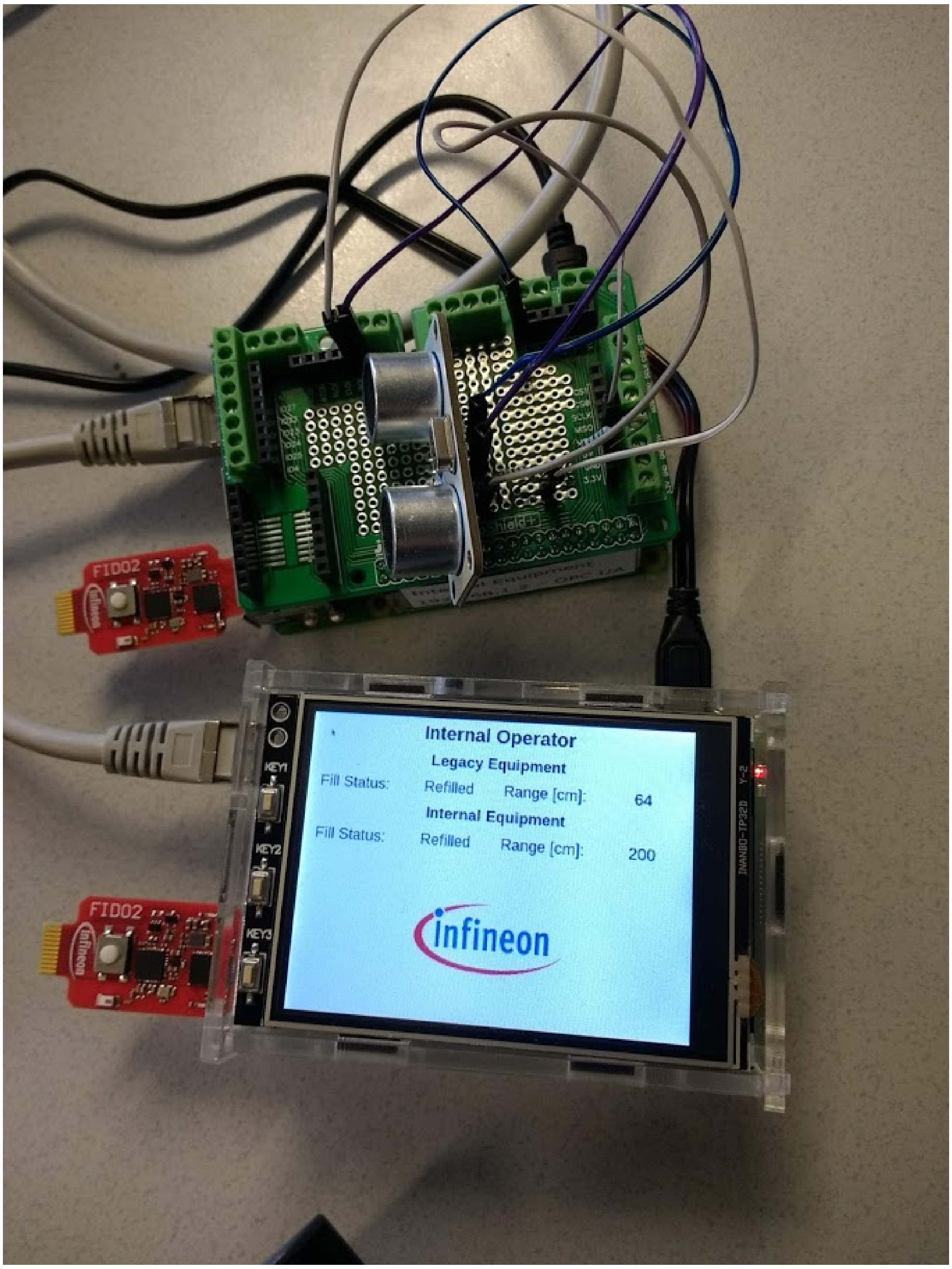
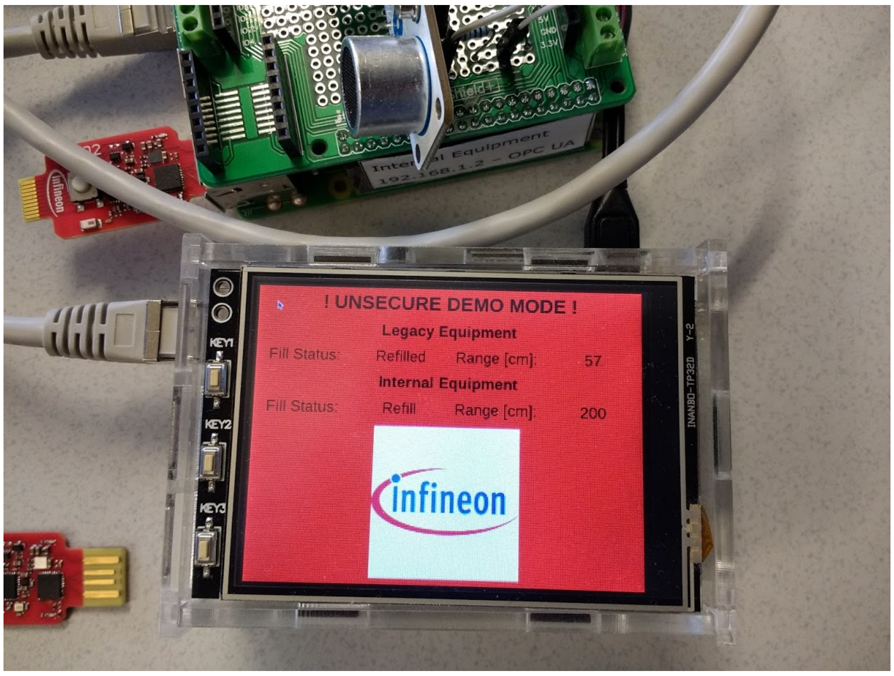


Figure 5.1.3.: Picture of the Internal Network.

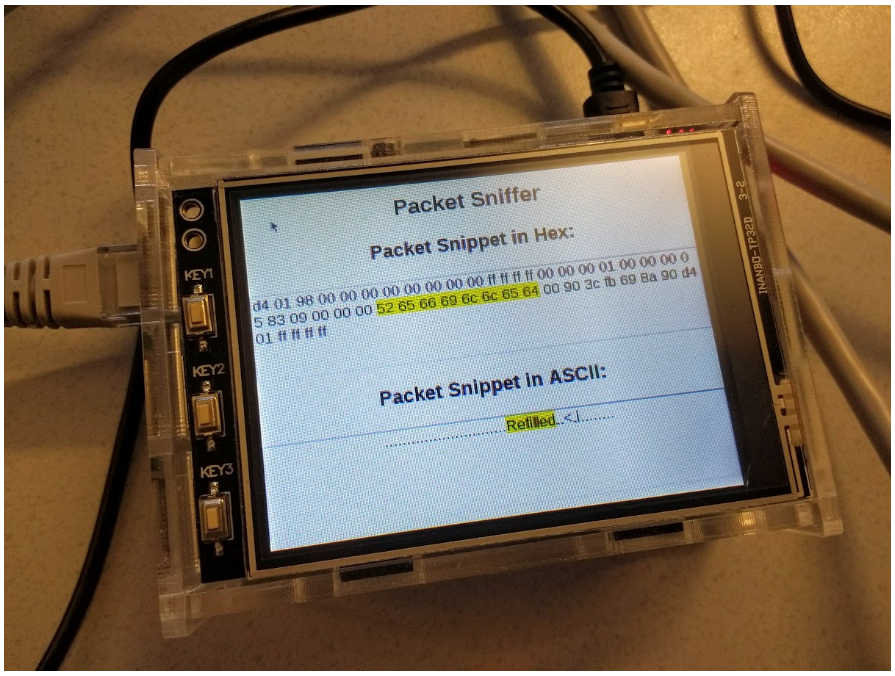
```
Packet Sniffer

Packet Snippet in Hex:

ad c2 b0 07 a5 09 1c e0 82 f9 9c c9 df 88 7a 74 77 b6 28 47 2c d0
09 98 ff 3f 3f bd 0e 60 f2 75 62 b7 b9 51 50 64 b0 fa c3 0c 8e e4 4
e e5 7f 73 6a 21 c9 12 d3 22 dc 37 a6 7d 32 8c 24 74 c7 4f 25 78
48 30 b5 99 09 50 c9 d6 63 d2 82 f8 4e d2 1c c9 cb 87 ee bd b8 2
8 ff 43 7a 5a ca b0

Packet Snippet in ASCII:

...............ztw.(G,....??..`.ub..QPd......N..sj!...".7.}2.$t.O%xH0...P..c...
N........(.CzZ..
```

Figure 5.1.4.: Picture of the Packet Sniffer.



```
! UNSECURE DEMO MODE !
Legacy Equipment
Fill Status:   Refilled   Range [cm]:   57
Internal Equipment
Fill Status:   Refill     Range [cm]:   200
```

Figure 5.1.5.: Picture of the Internal Operator during the unsecured demo mode.



```
Packet Sniffer
Packet Snippet in Hex:
d4 01 98 00 00 00 00 00 00 00 00 ff ff ff ff 00 00 00 01 00 00 00 0
5 83 09 00 00 00 52 65 66 69 6c 6c 65 64 00 90 3c fb 69 8a 90 d4
01 ff ff ff ff
Packet Snippet in ASCII:
..........................Refilled..<.|........
```

Figure 5.1.6.: Picture of the Packet Sniffer during the unsecured demo mode.

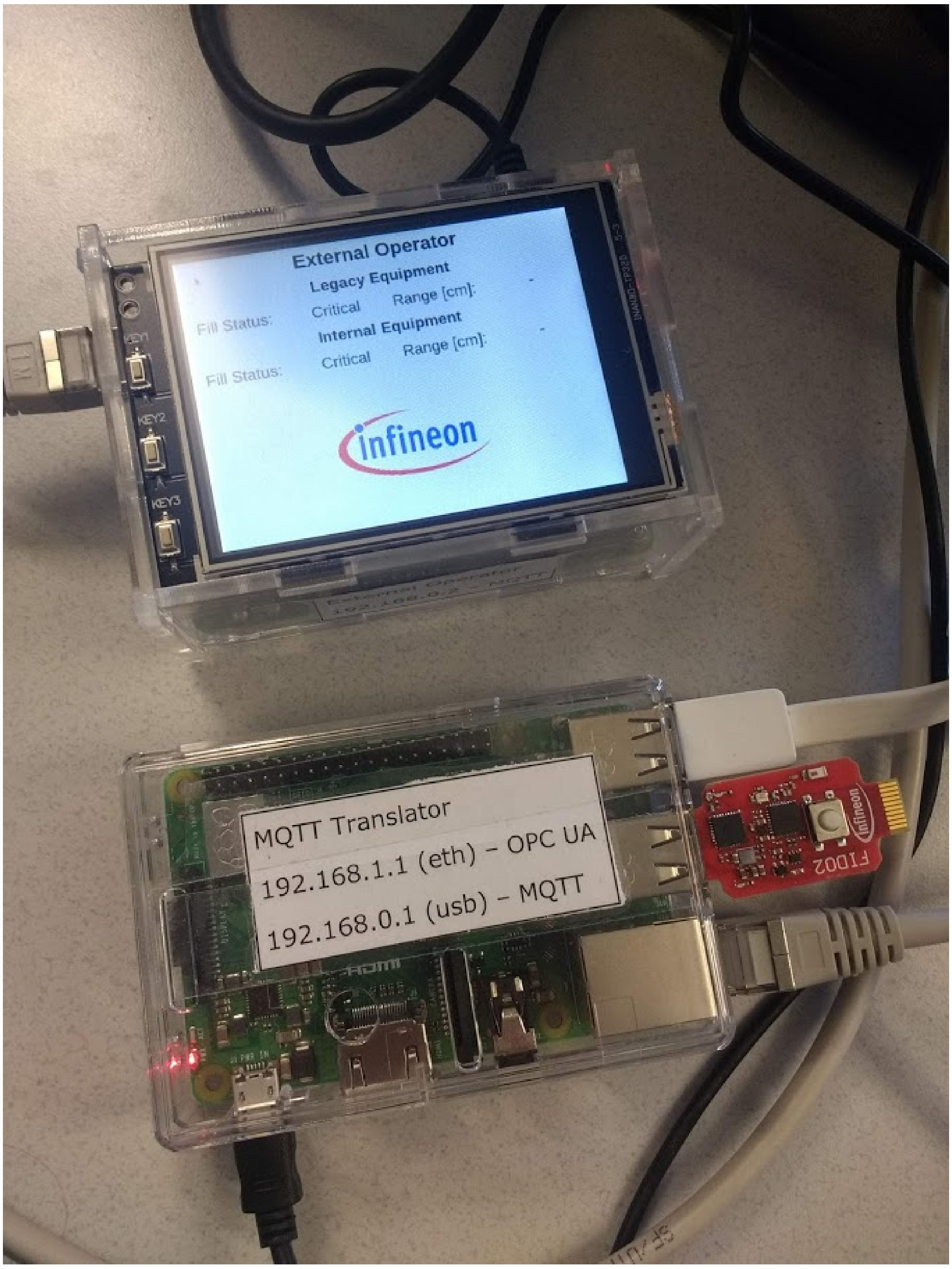


Figure 5.1.7.: Picture of the External Network.

5.2. Evaluation

This section contains the evaluation part of the thesis. However, as the functionality is already presented in the previous results section, this section focuses on the evaluation of the integrated security concepts.

5.2.1. Security Concepts

This section describes the security concepts of the two use cases. In general, the used secrets are stored on the SE of the security stick instead of the host's RAM. Additionally, the cryptographic operations are executed on the SE of the security stick and are therefore resistant against known side-channel attacks. This strongly decreases an attacker's possibilities to learn the secret keys of a host and further steal the host's identity.

Use Case I: Secured Configuration

In this use case, the hardware security extension works as secured storage and crypto coprocessor. Since the SE has an secured and tamper-resistant memory on chip, it is possible to decrypt the messages when they are received and store them in plain form. After the encrypted data, the authentication code is received. The Encrypt-then-MAC (EtM) approach is used, which means the MAC is generated from the encrypted data. During the *final* step, the last encrypted message bytes are verified before they are decrypted. This implicitly mitigates the traditional padding oracle attacks that exploit timing differences between the error cases “padding wrong” and “signature wrong”. The MtE approach would not make sense, as it is vulnerable even with hardware security extensions in use, as the signature verification takes about half of the time of the *final* step and therefore is indeed measurable. The E&M is considered as weaker than the other two and the performance advantage due to parallelization cannot be used as there is just one core. Details on the different approaches are explained in Subsection 2.3.2.4.

For additional protection, two different keys are used for the encryption and the authentication process. As encryption algorithm, AES is used in CBC mode with a key size of 256 bit. For the authentication, a HMAC is used with SHA2-256 as hash function. This is state-of-the-art and both algorithms are e.g. approved for US government use by NIST SP800-131A [57]. Additionally, the cryptographic operations are done in the SE, which offers protection against known side-channel attacks that attack the cryptographic implementations directly.

The according Java Card applet allows to exchange the secret keys used for both algorithms. This process is again protected by authenticated encryption and therefore is just possible if the current secret keys are known. However, it is not possible to fetch one of the secret keys from the SE by any command.

Use Case II: Secured Network Protocol Translation

As required, the Internal Network is additionally secured with a hardware security extension. Each participant of the Internal Network is equipped with the security stick. All crypto operations are implemented and executed on the SE.

The Java Card applet does not provide any possibility to set or get the local private RSA key. This restriction makes it hard to steal or tamper with the RSA key of the SE. However, one command allows to generate a new RSA key pair that might be needed if the company requires to change the keys after a defined time span.

The first message from an OPC UA client to an OPC UA server is the “Get Endpoint Request”. Since the OPC UA server has to respond with all supported security policies and its certificate, every entity has the possibility to generate a X.509 certificate for the own RSA key pair. The certificate is internally signed with the private RSA key and is issued by the authority of the respective company where the security stick is in use. However, for now, the received client certificate is verified just locally with a copy of the CA certificate. This is also further discussed in Subsection 5.2.2.

The next message from an OPC UA client to an OPC UA server is the “Open Secure Channel Request”. This message contains the certificate and the nonce of the client, both signed and encrypted. In the further sequence, the nonces of the server and the client are used as part of the KDF input. If both nonces are known, it is trivial to create the symmetric keys for further communication between those two parties. As the hosts are potentially untrusted, this is problematic and leads to the requirement that at least one nonce must not be known by a host. Therefore, some message modifications are done on the SE before the message is encrypted or after it is decrypted. At first, the own nonce is replaced with true random bytes before the message is encrypted with the remote public key. The host is not able to get its own nonce as the remote private key must be known to decrypt the message. However, the generated nonce is stored on the SE as it is required later. On the remote side, the encrypted message is sent to the SE to decrypt it. The SE decrypts it, extracts the nonce, stores it and finally overwrites it with random bytes that are not related to the actual nonce. Therefore, the host also does not have access to the remote nonce. Without knowing the nonces, the host is not able to create the secrets that are required for the further symmetric cryptographic operations.

5.2.2. Security Threats and Countermeasures

This section discusses some security threats and according countermeasures.

Removable Hardware

One disadvantage of USB hardware security tokens is that they can be easily removed from a connected host. In general, the physical access to industrial workplaces is restricted to working staff only. Still, in security-relevant areas even the staff might be untrusted. Therefore, the application is notified about a removal as soon as it happens. As a removal might be required as in the first presented use case, an additional timer could be used to still protect against a not permitted disconnect. In the second use case, this behavior is demonstrated by immediately switching to the "unsecured demo mode", implemented just for demonstrational purposes.

However, in a productive environment, the SE would be soldered onto the hardware. Still, this might not always be possible, as some legacy equipment might be unmodifiable. In this case, the security stick must be used and a removal would lead to an internal alarm. The system administrator, or the CA, would react by placing the public key of the security stick to the revocation list or by enforcing a key change. These approaches would block any communication with a stolen security stick.

Use Case I: Secured Configuration

The most problematic part of this use case is the key lifecycle, especially the key generation, key distribution and the key replacement.

Initial Keys

Since the SE moves through various stations until it lands at the customer's place, and not all intermediate entities might be trusted, the customer should be able to change the secret keys of the security stick.

Another approach would be to prohibit the exchange of the secret keys. In this case, the final keys would be known by the vendor, which might not be trusted. It might also be able to just allow to exchange keys one time, but since security policies of companies usually enforce a secret exchange interval, this approach also might not be suitable.

Confidentiality Breach

If an attacker learns both secret keys, the secret keys of the security stick can be changed. This attack disables the security stick, as the new, malicious secret keys cannot be changed without knowing them [79].

Use Case II: Secured Network Protocol Translation

This subsection describes the security threats of the second use case.

Local Certificate Authority

For now, each entity has the CA's certificate stored in its local file system. The certificate contains the CA's public key and can therefore be used to verify the certificates that are issued by this CA. However, the CA's CRL is static and therefore the entities cannot be updated if a security breach happens. This could be circumvented by using a remote CA, as described in Subsubsection 6.2.

Untrusted Hosts

The current systems do not use any additional security measures that attest the genuineness of any application running on the system. This means if an intruder could access and modify the file system, the host applications could be replaced unnoticed by malicious ones. Additionally, currently it is not prevented that another running application accesses the security stick. This could be circumvented by using a technique such as Trusted Execution Environment (TEE) or Remote Attestation, as described in Subsection 6.2.

6. Conclusion and Future Work

This chapter contains the conclusion and the future work of the thesis.

6.1. Conclusion

At first, this thesis examines that a hardware security token suits well for security-related IIoT use cases. This is due to the multiple supported interfaces and the increased flexibility regarding the application layer. The advantages of such a multi-interface device is shown in the first demonstrator, where the used security stick communicates by NFC and USB. The second demonstrator shows how a SECS/GEM-based legacy industrial network can be integrated into the main network of a company. This increases the inter-connectivity inside of a company and enables connection-based technologies, such as SCADA, by using the OPC UA protocol. Then, it is shown how the OPC UA-based communication can be protected with hardware security extensions. In comparison to the TPM-based protection as shown in [97], the increased flexibility of the Java Card-based security stick is used to provide an applet specially customized for the OPC UA security requirements. This applet is integrated into an OPC UA implementation as replacement for the software-based crypto stack, and allows to outsource the used secrets, and even the intermediates, to the hardware security extension, without giving the host/user the possibility to reach them trivially. Finally, a library is implemented for the communication between host system and hardware security token. This library is designed in a portable, customizable and generic way, which enables an easy exchange of components such as the host system, communication technology or “underlying” Java Card application.

6.2. Future Work

In this section, ideas to extend the practical part of this thesis are provided.

Remote Attestation

One additional security feature is remote attestation. Remote attestation allows to verify the current state of a host remotely and is usually part of a TEE. This should help that the system of the remote party is not tampered and works like expected. Birnstill et al. [97] describe an approach to integrate remote attestation into the OPC UA protocol by using a TPM. However, there exist different approaches that do not require a TPM.

Remote Certificate Authority

In the current implementation, the keys are signed by a “local” Certificate Authority (CA). This signature is verified by the open62541 implementation. Additionally, a local trust list and revoke list can be provided to the implementation. However, instead of the local verification process, a trusted remote CA could be used that verifies the certificate against its trust and revoke list. That would provide one central trusted point that maintains the authenticity of the entities.

Dynamic Variable Forwarding

In the current implementation, the implementation uses all (SECS/GEM to OPC UA) or just a static subset (OPC UA to MQTT) of the variables of the respective variable space. This could be changed by implementing configurations, which are stored in JSON files. Additionally, the access level could be added to the OPC UA variables. One possible approach would be using two different nodes inside of NS0: one public and one private. The MQTT Translator would further publish just changes of nodes belonging to the public node.

Multiple SECS/GEM Equipment Machines

In the current implementation, the SECS/GEM host layer is implemented in a static way. This does not support more than one connected SECS/GEM equipment. By implementing multiple SECS/GEM hosts with additional FIFOs and appending the respective SECS/GEM equipment to the OPC UA node ID, it should be possible to extend it to support further machines. However, also the sniffing approach of Nsiah et al. [91], already presented in Chapter 2.4, would provide a solution to this problem.

OPC UA Publish-Subscribe

For now, the asynchronous SECS/GEM alarms are treated synchronously, as the OPC UA standard did not provide true asynchronous messaging possibilities. The "monitored items" provide an asynchronous interface, however they are implemented to poll internally. The current OPC UA version (1.04) supports native publish-subscribe on different transport layers. One of the proposed layers is MQTT, so the conversion from OPC UA to MQTT could be done implicitly by the OPC UA implementation. However, this was not specified in the used OPC UA version and is still not implemented in the open62541 project up to today, therefore it was not used in this thesis.

Platform Independence

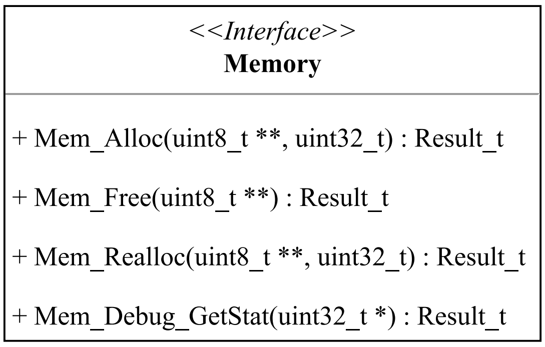
The external libraries in use are in general designed to be platform-independent. Of course, the Python implementations require a working Python interpreter and therefore just run on Microsoft Windows, Apple macOS or GNU/Linux. However, there exist approaches such as CPython [126], which compile Python code to C code to allow better compatibility, also on embedded devices. Additionally, also some design decisions of the host applications, such as the use of UNIX FIFOs, restrict the supported platforms for simplicity reasons. Therefore, the libraries and the implementations both could be replaced by more cross-platform friendly approaches.

Secure Messaging & Secure Communication Protocol

"Secure Messaging" is defined in the ISO 7816-4 [71] and tackles the secured communication between host system and smart card. Additionally, the "Secure Channel Protocol (SCP)" is a standard for secure communication, defined by GP. SCP02 is based on symmetric and SCP10 on asymmetric cryptography. An occurring problem with SM and SCP is, that both require to store secrets on the host system. This might be problematic, since in the setup of this thesis, the host system is not trusted. Therefore, for now, neither SCP nor SM is in use. Lu and Ali [43] show an alternative approach to provide secured communication. In this paper, the token also contains the application and provides it by enumerating as mass storage. Additionally, the code is regularly verified by requiring the software to calculate hashes of code blocks and provide them to the token that compares them with stored digests.

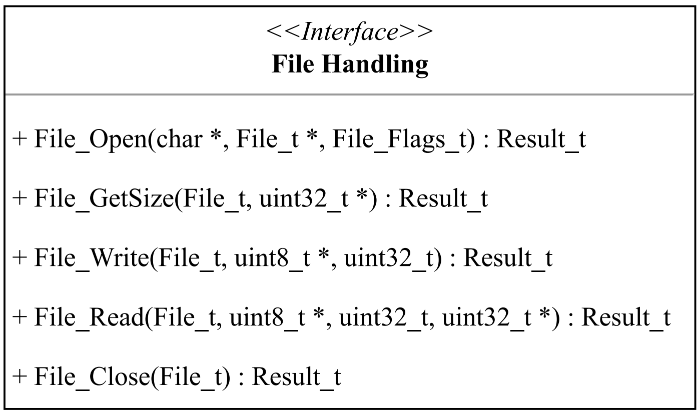
Appendices

A. Illustrations of Interface and Class Specifications



```
<<Interface>>
Memory

+ Mem_Alloc(uint8_t **, uint32_t) : Result_t
+ Mem_Free(uint8_t **) : Result_t
+ Mem_Realloc(uint8_t **, uint32_t) : Result_t
+ Mem_Debug_GetStat(uint32_t *) : Result_t
```

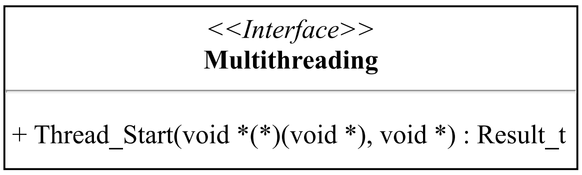
Figure A.1.: Interface definition of the Memory Layer.



<<Interface>>
File Handling

+ File_Open(char *, File_t *, File_Flags_t) : Result_t
+ File_GetSize(File_t, uint32_t *) : Result_t
+ File_Write(File_t, uint8_t *, uint32_t) : Result_t
+ File_Read(File_t, uint8_t *, uint32_t, uint32_t *) : Result_t
+ File_Close(File_t) : Result_t

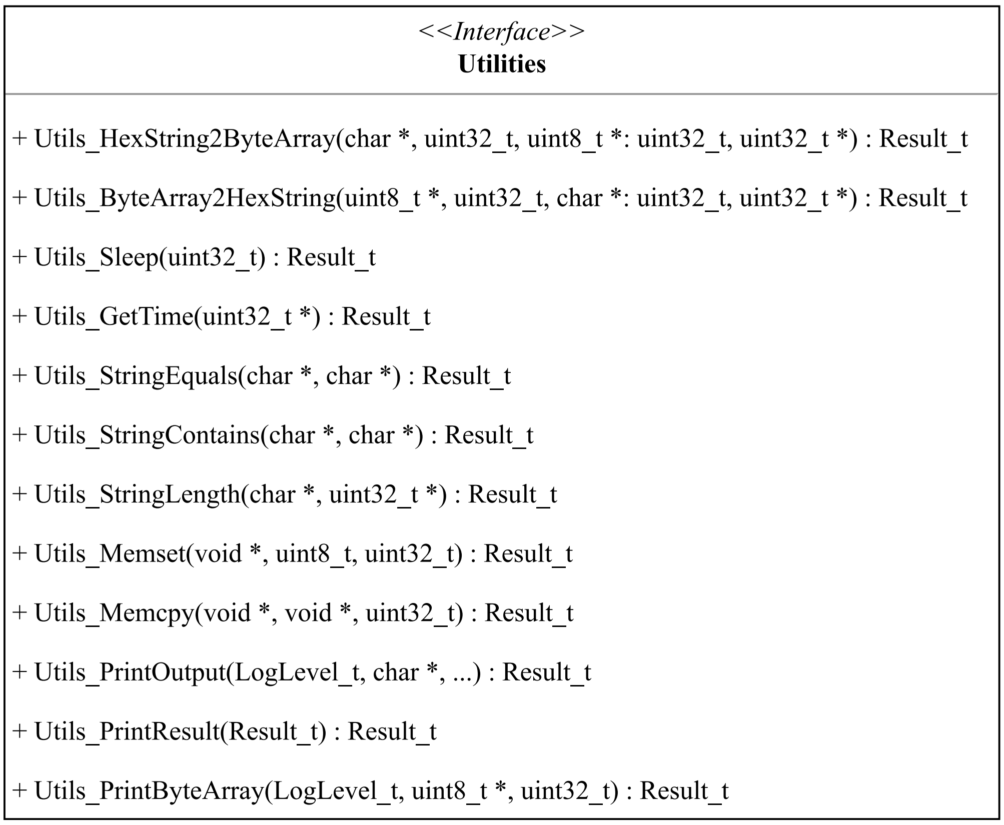
Figure A.2.: Interface definition of the File Handling Layer.



<<Interface>>
Multithreading

+ Thread_Start(void *(*)(void *), void *) : Result_t

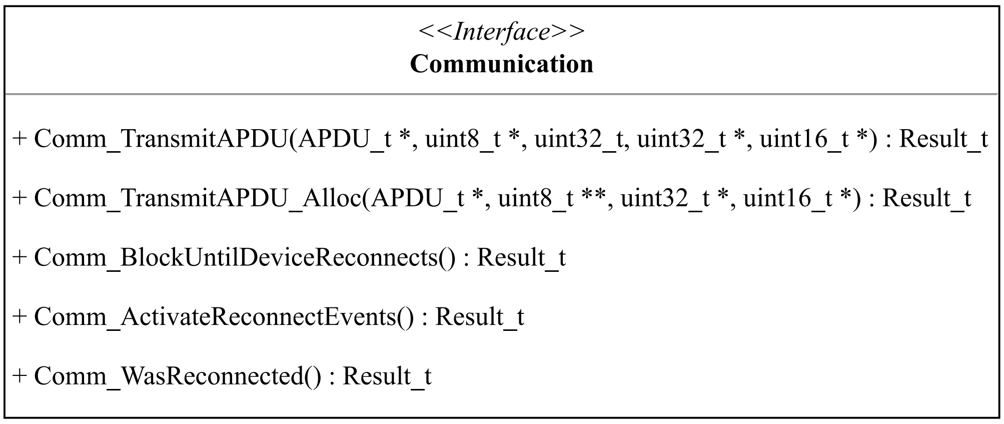
Figure A.3.: Interface definition of the Multithreading Layer.



<<Interface>>
Utilities

+ Utils_HexString2ByteArray(char *, uint32_t, uint8_t *: uint32_t, uint32_t *) : Result_t
+ Utils_ByteArray2HexString(uint8_t *, uint32_t, char *: uint32_t, uint32_t *) : Result_t
+ Utils_Sleep(uint32_t) : Result_t
+ Utils_GetTime(uint32_t *) : Result_t
+ Utils_StringEquals(char *, char *) : Result_t
+ Utils_StringContains(char *, char *) : Result_t
+ Utils_StringLength(char *, uint32_t *) : Result_t
+ Utils_Memset(void *, uint8_t, uint32_t) : Result_t
+ Utils_Memcpy(void *, void *, uint32_t) : Result_t
+ Utils_PrintOutput(LogLevel_t, char *, ...) : Result_t
+ Utils_PrintResult(Result_t) : Result_t
+ Utils_PrintByteArray(LogLevel_t, uint8_t *, uint32_t) : Result_t

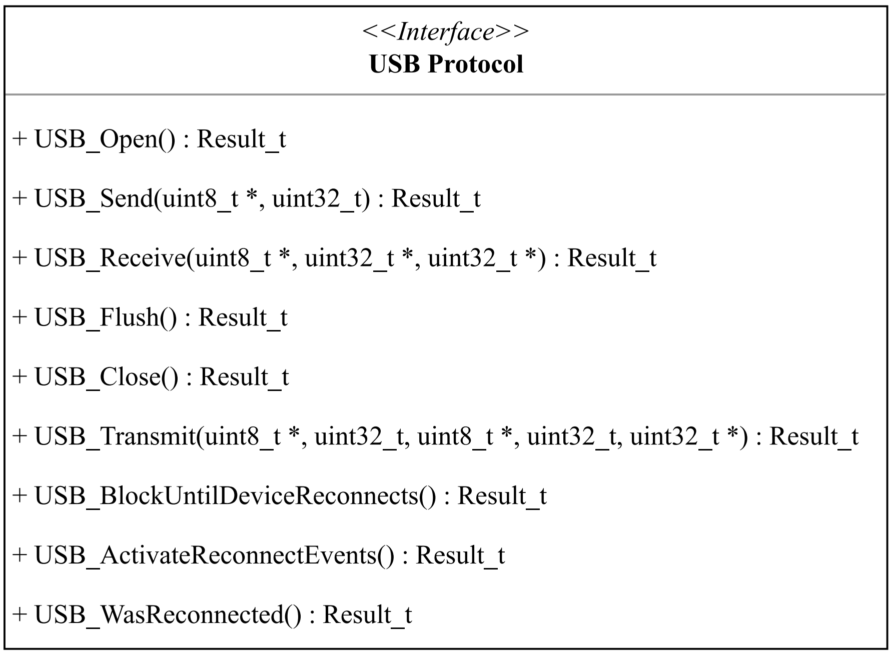
Figure A.4.: Interface definition of the Ultilities Layer.



<<Interface>>
Communication

+ Comm_TransmitAPDU(APDU_t *, uint8_t *, uint32_t, uint32_t *, uint16_t *) : Result_t
+ Comm_TransmitAPDU_Alloc(APDU_t *, uint8_t **, uint32_t *, uint16_t *) : Result_t
+ Comm_BlockUntilDeviceReconnects() : Result_t
+ Comm_ActivateReconnectEvents() : Result_t
+ Comm_WasReconnected() : Result_t

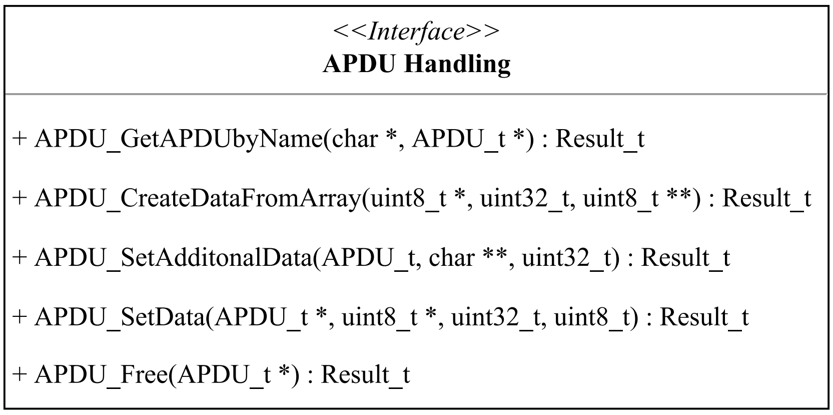
Figure A.5.: Interface definition of the Communication Layer.



<<Interface>>
USB Protocol

+ USB_Open() : Result_t
+ USB_Send(uint8_t *, uint32_t) : Result_t
+ USB_Receive(uint8_t *, uint32_t *, uint32_t *) : Result_t
+ USB_Flush() : Result_t
+ USB_Close() : Result_t
+ USB_Transmit(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *) : Result_t
+ USB_BlockUntilDeviceReconnects() : Result_t
+ USB_ActivateReconnectEvents() : Result_t
+ USB_WasReconnected() : Result_t

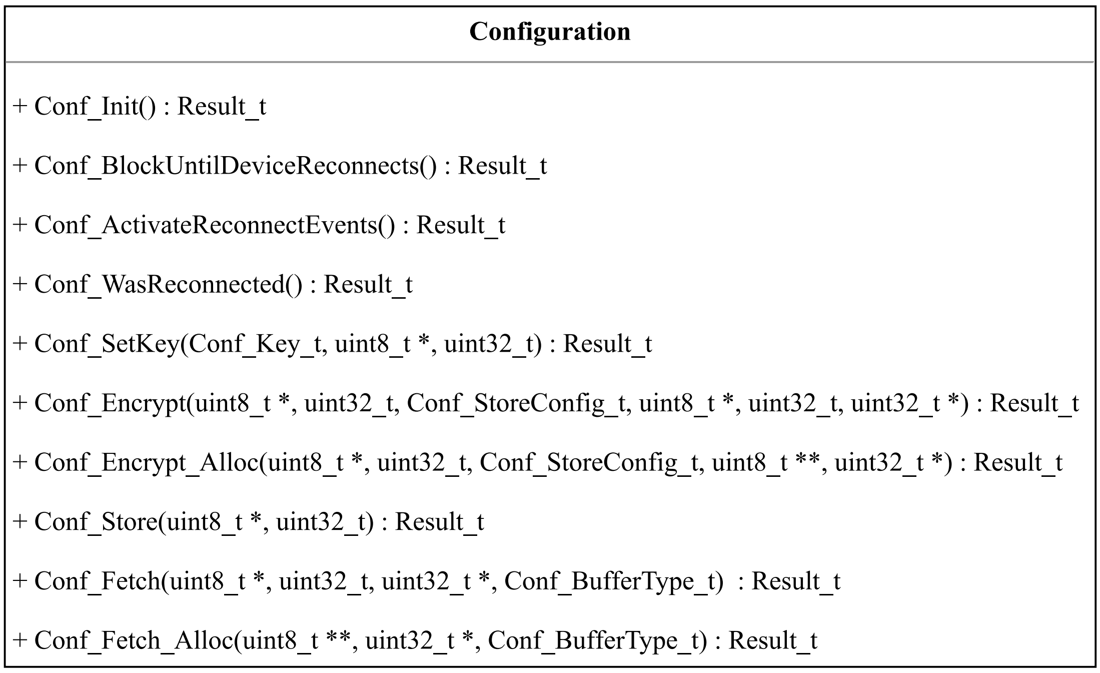
Figure A.6.: Interface definition of the USB Protocol.



<<Interface>>
APDU Handling

```
+ APDU_GetAPDUbyName(char *, APDU_t *) : Result_t
+ APDU_CreateDataFromArray(uint8_t *, uint32_t, uint8_t **) : Result_t
+ APDU_SetAdditonalData(APDU_t, char **, uint32_t) : Result_t
+ APDU_SetData(APDU_t *, uint8_t *, uint32_t, uint8_t) : Result_t
+ APDU_Free(APDU_t *) : Result_t
```

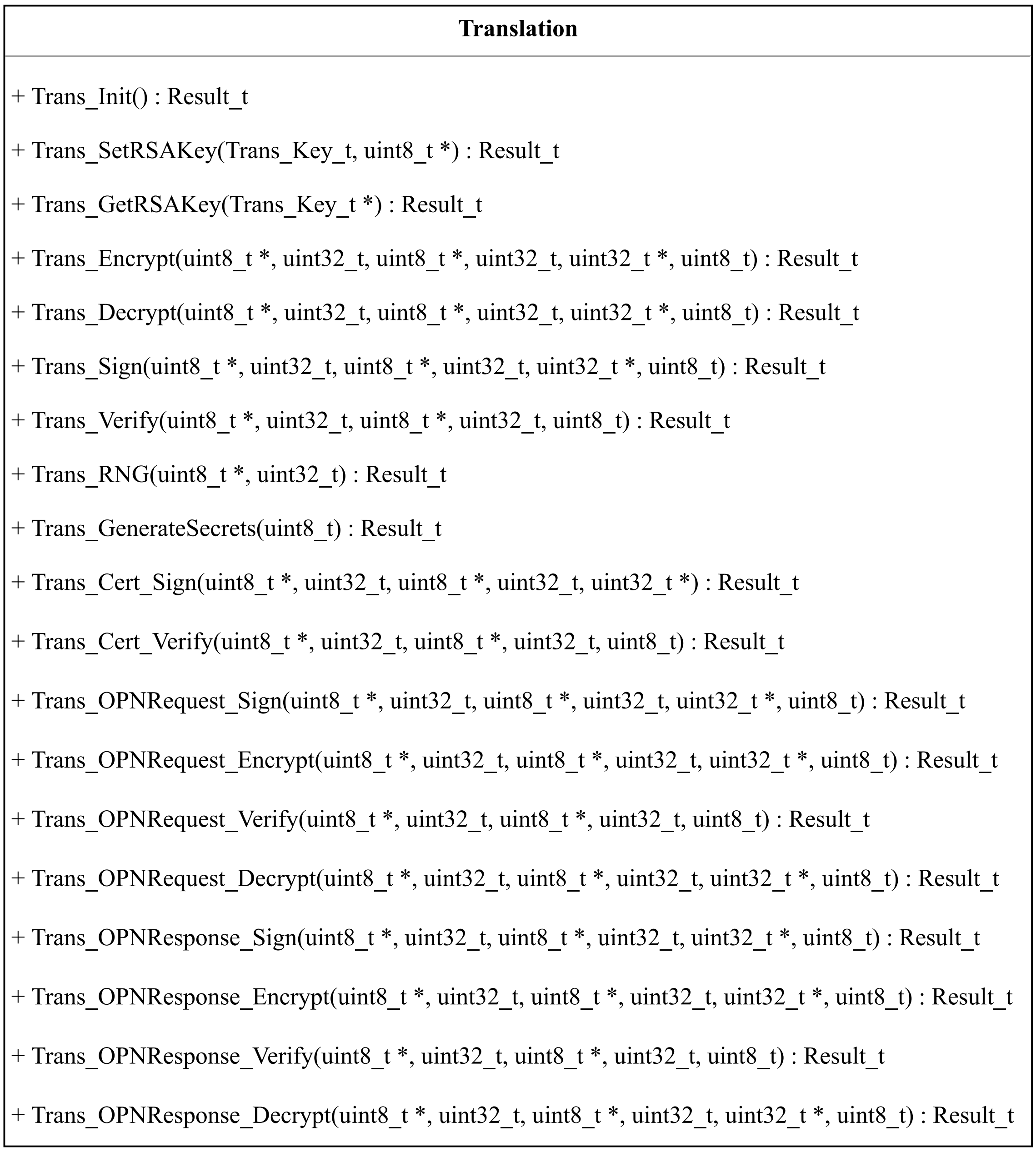
Figure A.7.: Interface definition of the APDU Layer.



Configuration

```
+ Conf_Init() : Result_t
+ Conf_BlockUntilDeviceReconnects() : Result_t
+ Conf_ActivateReconnectEvents() : Result_t
+ Conf_WasReconnected() : Result_t
+ Conf_SetKey(Conf_Key_t, uint8_t *, uint32_t) : Result_t
+ Conf_Encrypt(uint8_t *, uint32_t, Conf_StoreConfig_t, uint8_t *, uint32_t, uint32_t *) : Result_t
+ Conf_Encrypt_Alloc(uint8_t *, uint32_t, Conf_StoreConfig_t, uint8_t **, uint32_t *) : Result_t
+ Conf_Store(uint8_t *, uint32_t) : Result_t
+ Conf_Fetch(uint8_t *, uint32_t, uint32_t *, Conf_BufferType_t)  : Result_t
+ Conf_Fetch_Alloc(uint8_t **, uint32_t *, Conf_BufferType_t) : Result_t
```

Figure A.8.: Class description of the Configuration's Application Layer.



Translation

```
+ Trans_Init() : Result_t
+ Trans_SetRSAKey(Trans_Key_t, uint8_t *) : Result_t
+ Trans_GetRSAKey(Trans_Key_t *) : Result_t
+ Trans_Encrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_Decrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_Sign(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_Verify(uint8_t *, uint32_t, uint8_t *, uint32_t, uint8_t) : Result_t
+ Trans_RNG(uint8_t *, uint32_t) : Result_t
+ Trans_GenerateSecrets(uint8_t) : Result_t
+ Trans_Cert_Sign(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *) : Result_t
+ Trans_Cert_Verify(uint8_t *, uint32_t, uint8_t *, uint32_t, uint8_t) : Result_t
+ Trans_OPNRequest_Sign(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_OPNRequest_Encrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_OPNRequest_Verify(uint8_t *, uint32_t, uint8_t *, uint32_t, uint8_t) : Result_t
+ Trans_OPNRequest_Decrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_OPNResponse_Sign(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_OPNResponse_Encrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
+ Trans_OPNResponse_Verify(uint8_t *, uint32_t, uint8_t *, uint32_t, uint8_t) : Result_t
+ Trans_OPNResponse_Decrypt(uint8_t *, uint32_t, uint8_t *, uint32_t, uint32_t *, uint8_t) : Result_t
```

Figure A.9.: Class description of the Translation's Application Layer.

B. Tables of APDU Commands

This chapter contains the tables with all implemented APDUs. However, the CLA byte is neglected as it is always set to zero. All values are listed in the hexadecimal notation. len(data) represents the actual length of the data. The index of the byte arrays start at zero.

Type	**INS**	**P1**	**P2**	**Lc**	**Data**
AES-CBC	11	00	—	40	data
HMAC-SHA2	11	01	—	40	data

Table B.1.: Table with values for the "Set Key" APDU. The data is the key, protected by authenticated encryption with the previous secret keys.

Type	INS	P1	P2	Lc	Data
Init	13	00	00	*len*(data)	data
Update	13	01	00	*len*(data)	data
Final	13	02	00	*len*(data)	data
Init & Store	13	00	01	*len*(data)	data
Update & Store	13	01	01	*len*(data)	data
Final & Store	13	02	01	*len*(data)	data

Table B.2.: Table with values for the "Encrypt" APDU. The data bytes are filled with the data to encrypt.

Type	INS	P1	P2	Lc	Data
Init	12	00	—	*len*(data)	data
Update	12	01	—	*len*(data)	data
Final	12	02	—	*len*(data)	data

Table B.3.: Table with values for the "Store" APDU. The data bytes are filled with encrypted & authenticated data which should be stored.

Type	INS	P1	P2	Lc	Data
Decrypted Data	14	—	00	—	—
Encrypted Data	14	—	01	—	—

Table B.4.: Table with values for the "Fetch" APDU.

Type	INS	P1	P2	Lc	Data
Local RSA Key	21	00	—	—	—
Remote RSA Exponent	21	1+ID	exp[255]	*len*(exp)	exp[0:254]
Local RSA Key	21	20	—	—	—
Remote RSA Modulus	21	4+ID	mod[255]	*len*(mod)	mod[0:254]

Table B.5.: Table with values for the "Set Key" APDU. The four most significant bits of the P1 byte are used for the key type, the other 4 bits are the ID of the remote side. P2 is used to transmit the 256th bit of the respective data. The other 255 bytes are sent in the data field.

Type	INS	P1	P2	Lc	Data
Local RSA Exponent	27	00	—	—	—
Local RSA Modulus	27	20	—	—	—

Table B.6.: Table with values for the "Get Key" APDU.

Type	INS	P1	P2	Lc	Data
Encrypt Request Init	31	0+ID	—	*len*(data)	data
Encrypt Response Init	31	8+ID	—	*len*(data)	data
Encrypt Request Update	31	1+ID	—	*len*(data)	data
Encrypt Response Update	31	9+ID	—	*len*(data)	data
Encrypt Request Final	31	2+ID	—	*len*(data)	data
Encrypt Response Final	31	A+ID	—	*len*(data)	data
Decrypt Request Init	33	0+ID	data[255]	FF	data[0:254]
Decrypt Response Init	33	8+ID	data[255]	FF	data[0:254]
Decrypt Request Update	33	1+ID	data[255]	FF	data[0:254]
Decrypt Response Update	33	9+ID	data[255]	FF	data[0:254]
Decrypt Request Final	33	2+ID	data[255]	FF	data[0:254]
Decrypt Response Final	33	A+ID	data[255]	FF	data[0:254]

Table B.7.: Table with values for the "Encrypt OPN" and "Decrypt OPN" APDU. The four least significant bits of P1 are the identifier of the remote connection partner. The data bytes are filled with the data to encrypt or to decrypt.

Type	INS	P1	P2	Lc	Data
Sign Request Init	30	0+ID	—	*len*(data)	data
Sign Response Init	30	8+ID	—	*len*(data)	data
Sign Request Update	30	1+ID	—	*len*(data)	data
Sign Response Update	30	9+ID	—	*len*(data)	data
Sign Request Final	30	2+ID	—	*len*(data)	data
Sign Response Final	30	A+ID	—	*len*(data)	data
Verify Request Init	32	0+ID	—	*len*(data)	data
Verify Response Init	32	8+ID	—	*len*(data)	data
Verify Request Update	32	1+ID	—	*len*(data)	data
Verify Response Update	32	9+ID	—	*len*(data)	data
Verify Request Final Update	32	3+ID	—	*len*(data)	data
Verify Response Final Update	32	B+ID	—	*len*(data)	data
Verify Request Final	32	2+ID	sign[255]	FF	sign[0:254]
Verify Response Final	32	A+ID	sign[255]	FF	sign[0:254]

Table B.8.: Table with values for the "Sign OPN" and "Verify OPN" APDU. The four least significant bits of P1 are the identifier of the remote connection partner. The message is referred to as data, while the signature is referred to as "sign".

Type	INS	P1	P2	Lc	Data
Generate Secrets	29	0+ID	—	—	—

Table B.9.: Table with values for the "Generate Key" APDU. The four most significant bits of the P1 byte are used for the key type, the other 4 bits are the ID of the remote side.

Type	**INS**	**P1**	**P2**	**Lc**	**Data**
Encrypt Init	23	0+ID	—	*len*(data)	data
Encrypt Update	23	1+ID	—	*len*(data)	data
Encrypt Final	23	2+ID	—	*len*(data)	data
Decrypt Init	24	0+ID	—	*len*(data)	data
Decrypt Update	24	1+ID	—	*len*(data)	data
Decrypt Final	24	2+ID	—	*len*(data)	data

Table B.10.: Table with values for the “Encrypt MSG” and “Decrypt MSG” APDU. The four least significant bits of P1 are the identifier of the remote connection partner. The data bytes are filled with the data to encrypt or to decrypt.

Type	**INS**	**P1**	**P2**	**Lc**	**Data**
Sign Init	25	0+ID	—	*len*(data)	data
Sign Update	25	1+ID	—	*len*(data)	data
Sign Final	25	2+ID	—	*len*(data)	data
Verify Init	26	0+ID	—	*len*(data)	data
Verify Update	26	1+ID	—	*len*(data)	data
Verify Final	26	2+ID	—	*len*(data)	data

Table B.11.: Table with values for the “Sign MSG” and “Verify MSG” APDU. The four least significant bits of P1 are the identifier of the remote connection partner. The data can be either the message, or, in the final step of the verification, the signature.

Type	**INS**	**P1**	**P2**	**Lc**	**Data**
Generate Random Bytes	22	amount of bytes	—	—	—

Table B.12.: Table with values for the “Generate Random Bytes” APDU. P1 contains the amount of requested bytes.

Type	**INS**	**P1**	**P2**	**Lc**	**Data**
CSR Sign Init	50	0	—	*len*(data)	data
CSR Sign Update	50	1	—	*len*(data)	data
CSR Sign Final	50	2	—	*len*(data)	data
Cert Verify Init	51	0+ID	—	*len*(data)	data
Verify Cert Verify Update	51	1+ID	—	*len*(data)	data
Verify Cert Verify Final	51	2+ID	sign[255]	FF	sign[0:254]

Table B.13.: Table with values for the “CSR Sign” and “Cert Verify” APDU. The four least significant bits of P1 are the identifier of the remote connection partner. The message is referred to as data, while the signature is referred to as “sign”.

C. Listings of Code Examples

This chapter contains the listings with code examples.

```
/** src/host/filehandling.h **/
typedef uint8_t File_Flags_t;
#define FILE_FLAGS_READ_ONLY 1

struct File_Impl;
typedef struct File_Impl *File_t;

// function to open a file
Result_t File_Open(File_t *file_ref, const char *path, const File_Flags_t
    flags);

// function to read from a file
Result_t File_Read(File_t file, char **content_ref, uint32_t *length_ref);

/** src/host/linux/filehandling_linux.c **/
struct File_Impl
{
    FILE *file;
    File_Flags_t flags;
};

/** file_example.c **/
void testFunction(void)
{
    File_t file;
    File_Open(&file, "test.txt", FILE_FLAGS_READ_ONLY);

    char *content;
    uint32_t length;
    File_Read(file, &content, &length);
    printf("File Content (Length \%u): \%s\n", length, content);
}
```

Listing C.1: Code example of a generic interface.

```
#ifdef __cplusplus
extern "C" {
#endif

// Function Declarations

#ifdef __cplusplus
}
#endif
```

Listing C.2: Code example of extern “C” usage.

```
/**
 * @brief Function to get the size of an opened file.
 *
 * <b>Interface</b>
 *
 * @param[in]  file_handle    File handle to identify the file.
 * @param[out] size_ref       Reference where the file size should be
 *                            stored.
 * @retval     #SUCCESS       if successful.
 */
Result_t File_GetSize(File_t file_handle, uint32_t *size_ref);

/**
 * <b>Implementation (Linux)</b>
 *
 * @retval #FAILURE_NULLPTR  if NULL would be dereferenced.
 */
Result_t File_GetSize(File_t file_handle, uint32_t *size_ref) {
    // ...
}
```

Listing C.3: Code example of a doxygen-conform documentation.

```
static final byte P1_SECCONF_INIT = 0x00;
static final byte P1_SECCONF_UPDATE = 0x01;
static final byte P1_SECCONF_FINAL = 0x02;

// ...

if (apduP1 == P1_SECCONF_INIT) {
    // ...
}
else if (apduP1 == P1_SECCONF_UPDATE) {
    // ...
}
else if (apduP1 == P1_SECCONF_FINAL) {
    // ...
}
else {
    ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
}
```

Listing C.4: Code example of representative member.

```
INIT_FUNC("Test")
uint8_t *internal_buffer = NULL;
CHECK_CALL(Mem_Alloc(&internal_buffer, 1024))
...
// if an error occurs here, jump to TEARDOWN1
...
CHECK_CALL2(Mem_Alloc(external_buffer, 1024), TEARDOWN1)
...
// if an error occurs here, jump to ERROR1
...
NEW_LABEL(TEARDOWN1)
Mem_Free(&internal_buffer);

END_FUNC

NEW_LABEL(ERROR1)
Mem_Free(external_buffer);
JUMP_TO_LABEL(TEARDOWN1)
```

Listing C.5: During normal code execution, this guarantees that the internal buffer is freed and the provided buffer pointer is still correctly set for the caller. The CHECK_CALL macro checks the returned Result_t and if it is not SUCCESS, it jumps to the END label which is introduced by the END_FUNC macro.

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_C_COMPILER "arm-linux-gnueabihf-gcc")
set(CMAKE_PREFIX_PATH "/usr/arm-linux-gnueabihf")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

set(LIB_PCSCLITE "/usr/lib/arm-linux-gnueabihf/libpcsclite.so.1.0.0")
```

Listing C.6: Code example of a cross compilation toolchain in CMake.

```
typedef enum {
  APDU_TARGET_SECCHIP = 0x00,
  APDU_TARGET_BRIDGE = 0x01
} APDU_Receiver_t;

typedef struct {
  APDU_Receiver_t target;
  uint8_t info;
  char *name;
  uint8_t CLA;
  uint8_t INS;
  uint8_t P1;
  uint8_t P2;
  uint32_t Lc;
  uint8_t *data;
  uint8_t Le;
} APDU_t;
```

Listing C.7: Representation of the APDU command class.

```
#include "testing.h"
#if defined(HOST_OS_LINUX)
#include "../src/host/linux/filehandler_linux.c"
#elif defined(HOST_OS_WINDOWS)
#include "../src/host/windows/filehandler_windows.c"
#endif

TEST_START(file_path_is_null) {
  File_t handle;
  TEST_ASSERT_RESULT(File_Open(NULL, &handle, FILE_FLAGS_WR),
                     FAILURE_NULLPTR,
                     "Expected null pointer error!");
}
TEST_END

int main(int argc, char *argv[]) {
  INIT_TESTS;
  RUN_TEST(file_path_is_null);
  END_TESTS;
}
```

Listing C.8: Code example of an unit test.

```
TEST_START(encrypted_store) {
    // check if the memset works
    uint8_t data[128];
    TEST_ASSERT_SUCCESS(Utils_Memset(data, 0xAB, sizeof(data)),
                        "Memset failed!");

    // check if the encryption works
    uint8_t *encrypted_data; uint32_t encrypted_data_length;
    TEST_ASSERT_SUCCESS(
        Conf_Encrypt_Alloc(data, sizeof(data),
                           SECCONF_STORE_ENCRYPTED_DATA,
                           &encrypted_data, &encrypted_data_length),
        "Encryption failed!");

    // check if the encryption storage works
    uint8_t *stored_encrypted_data; uint32_t stored_encrypted_data_length;
    TEST_ASSERT_SUCCESS(
        Conf_Fetch_Alloc(&stored_encrypted_data,
                         &stored_encrypted_data_length,
                         SECCONF_BUFFERTYPE_ENCRYPTED),
        "Fetch encrypted failed!");

    // compare the data
    TEST_ASSERT_SUCCESS(
        Test_ByteArrayEquals(encrypted_data, encrypted_data_length,
                             stored_encrypted_data,
                             stored_encrypted_data_length),
        "Fetched data is not equal!");
  }
  TEST_END
```

Listing C.9: Code example of a functional test.

```c
// test/klee_secconf_encrypt.c
#include <klee/klee.h>
// include all necessary C files

int main(int argc, char *argv[])
{
    // input
    uint8_t data[128];
    uint32_t data_length = sizeof(data);
    klee_make_symbolic(&data, sizeof(data), "data");

    // output
    uint8_t *ciphertext;
    uint32_t ciphertext_length;

    // call and exit with return value which contains error code
    return Conf_Encrypt_Alloc(data, data_length, &ciphertext,
                              &ciphertext_length);
}
```

Listing C.10: Code example of a KLEE test.

```c
static UA_StatusCode
asym_verify_sp_basic256sha256(const UA_SecurityPolicy *securityPolicy,
                              Basic256Sha256_ChannelContext *cc,
                              const UA_ByteString *message,
                              const UA_ByteString *signature) {
// ...
mbedtls_sha256(message->data, message->length, hash, 0);
mbedtls_rsa_set_padding(rsaContext, MBEDTLS_RSA_PKCS_V15,
                        MBEDTLS_MD_SHA256);
int mbedErr = mbedtls_pk_sign(&pc->localPrivateKey,
                              MBEDTLS_MD_SHA256, hash,
                              UA_SHA256_LENGTH, signature->data,
                              &sigLen, mbedtls_ctr_drbg_random,
                              &pc->drbgContext);
// ...
}
```

Listing C.11: Code example of calling the mbedTLS crypto stack.

```
// ...
result = Trans_OPNRequest_Sign(message->data,
                               (uint32_t)message->length,
                               signature->data,
                               (uint32_t)signature->length,
                               (uint32_t*)&(signature->length),
                               cc->identity);
// ...
```

Listing C.12: Code example of calling the implemented crypto stack.

```
struct _Semi_OPCUA_Server_Internal;
typedef struct _SemI_OPCUA_Server_Internal *Semi_OPCUA_Server_Internal;

typedef struct Semi_OPCUA_Server_Config {
  Semi_VariableList variable_list;
  Semi_OPCUA_Server_Internal internal;
} Semi_OPCUA_Server_Config;
```

Listing C.13: Code example of an abstract configuration.

```
struct _Semi_OPCUA_Server_Internal {
  UA_ServerConfig *config;
  UA_Server *server;
  UA_Boolean running;
  Semi_Result (*readValueFunc)(void *context, Semi_Variable *var);
  void *read_context;
};
```

Listing C.14: Code example of a concrete configuration.

```
[Unit]
Description=Host Application of External Operator
After=display-manager.service

[Service]
Type=idle
User=pi
ExecStart=/home/pi/SemI/wrapper/external_operator.sh

[Install]
WantedBy=graphical.target
```

Listing C.15: Code example of a systemd service unit.

Bibliography

[1] *IoSense*. IoSense Consortium. URL: `http://www.iosense.eu/` (visited on 09/13/2018).

[2] *SemI40*. SemI40 Consortium. URL: `http://www.semi40.eu/` (visited on 09/13/2018).

[3] *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025*. Statista. URL: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/` (visited on 08/28/2018).

[4] N. Cai, J. Wang, and X. Yu. "SCADA system security: Complexity, history and new developments." In: *6th IEEE International Conference on Industrial Informatics*. IEEE. 2008, pp. 569–574.

[5] I. N. Fovino. "SCADA System Cyber Security." In: *Secure Smart Embedded Devices, Platforms and Applications*. Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 451–471.

[6] *Percentage of Web Pages Loaded by Firefox Using HTTPS*. Let's Encrypt. URL: `https://letsencrypt.org/stats/#percent-pageloads` (visited on 10/23/2018).

[7] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *40th IEEE Symposium on Security and Privacy (S&P 19)*. 2019.

[8] M. Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[9] M. Seaborn and T. Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges." In: *Black Hat* 15 (2015).

[10] D. Gruss, C. Maurice, and S. Mangard. "Rowhammer. js: A remote software-induced fault attack in javascript." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 300–321.

[11] M. Lipp et al. "Nethammer: Inducing Rowhammer Faults through Network Requests." In: *arXiv preprint arXiv:1805.04956* (2018).

[12] A. Tatar et al. "Throwhammer: Rowhammer Attacks over the Network and Defenses." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, USA: USENIX Association, 2018, pp. 213–226. URL: `https://www.usenix.org/conference/atc18/presentation/tatar`.

[13] M. Schwarz et al. “NetSpectre: Read Arbitrary Memory over Network.” In: *arXiv preprint arXiv:1807.10535* (2018).

[14] P. Kocher, J. Jaffe, and B. Jun. “Differential power analysis.” In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[15] B. Ege, G. Perin, and J. van Woudenberg. “Lowering the Bar: Deep Learning for Side Channel Analysis.” In: *Black Hat USA* 18 (2018).

[16] M. Vanhoef and F. Piessens. “Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2.” In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[17] L. Bassi. “Industry 4.0: Hope, hype or revolution?” In: *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*. IEEE. Sept. 2017, pp. 1–6.

[18] Z. Ma et al. “Security Viewpoint in a Reference Architecture Model for Cyber-Physical Production Systems.” In: *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. IEEE. Apr. 2017, pp. 153–159.

[19] M. Hermann, T. Pentek, and B. Otto. “Design Principles for Industrie 4.0 Scenarios.” In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE. Jan. 2016, pp. 3928–3937.

[20] J. Wan, H. Cai, and K. Zhou. “Industrie 4.0: Enabling technologies.” In: *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things (ICIT)*. IEEE. Jan. 2015, pp. 135–140.

[21] A. Bicaku et al. “Towards trustworthy end-to-end communication in industry 4.0.” In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. IEEE. July 2017, pp. 889–896.

[22] E. Sisinni et al. “Industrial Internet of Things: Challenges, Opportunities, and Directions.” In: *IEEE Transactions on Industrial Informatics* 14.11 (Nov. 2018), pp. 4724–4734.

[23] A. Al-Fuqaha et al. “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications.” In: *IEEE Communications Surveys Tutorials* 17.4 (June 2015), pp. 2347–2376.

[24] G. Bloom et al. “Design patterns for the industrial Internet of Things.” In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. June 2018, pp. 1–10.

[25] S. Skorobogatov. “Physical Attacks and Tamper Resistance.” In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. New York, NY, USA: Springer New York, 2012, pp. 143–173.

[26] P. Schaumont and Z. Chen. “Side-Channel Attacks and Countermeasures for Embedded Microcontrollers.” In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. New York, NY, USA: Springer New York, 2012, pp. 263–282.

[27] K. Mai. “Side Channel Attacks and Countermeasures.” In: *Introduction to Hardware Security and Trust*. Ed. by M. Tehranipoor and C. Wang. New York, NY, USA: Springer New York, 2012, pp. 175–194.

[28] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: Aug. 2014.

[29] *Randomness Requirements for Security*. Best Practice. Internet Engineering Task Force, June 2005. URL: `https://tools.ietf.org/html/rfc4086`.

[30] *Some SecureRandom Thoughts*. Android Developers Blog. URL: `https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html` (visited on 02/17/2019).

[31] *Console Hacking 2010*. Chaos Computer Club e.V. URL: `https://media.ccc.de/v/27c3-4087-en-console_hacking_2010` (visited on 02/17/2019).

[32] *Security Requirements for Cryptographic Modules*. Specification. Gathersburg, MD, USA: NIST, May 2001.

[33] *Common Criteria Portal*. Common Criteria. URL: `https://www.commoncriteriaportal.org` (visited on 01/17/2019).

[34] *Common Criteria for Information Technology Security Evaluation – Part 3: Security assurance components*. Standard. Common Criteria, Apr. 2017.

[35] T. Boswell. “Security Evaluation and Common Criteria.” In: *Secure Smart Embedded Devices, Platforms and Applications*. Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 407–427.

[36] J. Daemen. “Resistance against implementation attacks. A comparative study of the AES proposals.” In: *Second Advanced Encryption Standard Candidate Conference* (1999).

[37] R. B. Lee. “University research in hardware security.” In: *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE. Aug. 2014, pp. 1–27.

[38] *Information technology – Trusted platform module library – Part 1: Architecture*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2015.

[39] *Information technology – Trusted platform module library – Part 2: Structures*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2015.

[40] *Information technology – Trusted platform module library – Part 3: Commands*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2015.

[41] *Information technology – Trusted platform module library – Part 4: Supporting Routines*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2015.

[42] K. Mayes and K. Markantonakis. "An Introduction to Smart Cards and RFIDs." In: *Secure Smart Embedded Devices, Platforms and Applications*. Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 3–25.

[43] H. K. Lu and A. Ali. "Communication Security between a Computer and a Hardware Token." In: *Third International Conference on Systems (icons 2008)*. IEEE. Apr. 2008, pp. 220–225.

[44] *A Guide To Unterstanding GEM - SECS - HSMS*. EdgeIntegration. URL: `http://www.edgeintegration.com/downloads/Guide_to_understanding_SECS.pdf` (visited on 12/30/2018).

[45] *secsgem*. Github. URL: `https://github.com/bparzella/secsgem` (visited on 12/30/2018).

[46] *Specification for the Generic Model for Communications and Control of Manufacturing Equipment (GEM)*. Standard. Milpitas, CA, USA: SEMI, Apr. 2018.

[47] *Specification for SEMI Equipment Communications Standard 2 Message Content (SECS-II)*. Standard. Milpitas, CA, USA: SEMI, Dec. 2017.

[48] *Specification for SEMI Equipment Communications Standard 1 Message Transfer (SECS-I)*. Standard. Milpitas, CA, USA: SEMI, Apr. 2018.

[49] *High-Speed SECS Message Services (HSMS) Generic Services*. Standard. Milpitas, CA, USA: SEMI, Apr. 2013.

[50] *MQTT Version 3.1.1*. OASIS. URL: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html` (visited on 09/07/2018).

[51] *Organization for the Advancement of Structured Information Standards*. OASIS. URL: `https://www.oasis-open.org/` (visited on 01/08/2019).

[52] L. Nastase. "Security in the Internet of Things: A Survey on Application Layer Protocols." In: *2017 21st International Conference on Control Systems and Computer Science (CSCS)*. May 2017, pp. 659–666.

[53] *OPC Unified Architecture – Part 1: Overview and concepts*. Technical Report. International Electrotechnical Commission, Oct. 2016. URL: `https://webstore.iec.ch/publication/25997`.

[54] *OPC UA Protocols*. ascolab GmbH. URL: `https://upload.wikimedia.org/wikipedia/de/e/e4/Uaprotocols.png`.

[55] *OPC Unified Architecture – Part 6: Mappings*. International Standard. International Electrotechnical Commission, Mar. 2015. URL: `https://webstore.iec.ch/publication/21993`.

[56] *OPC UA - Specification Part 2: Security Model*. Specification. Scottsdale, AZ, USA: OPC Foundation, Aug. 2018.

[57] *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Special Publication. National Institute of Standards and Technology, Nov. 2015.

[58] *OPC UA - Security Policies*. OPC Foundation. URL: `https://opcfoundation.org/UA/SecurityPolicy/` (visited on 01/14/2019).

[59] *OPC UA Security Analysis*. Report. Bonn, DE: Federal Office for Information Security, Mar. 2017.

[60] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009.

[61] *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Document. Internet Engineering Task Force, May 2008. URL: `https://tools.ietf.org/html/rfc5280`.

[62] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[63] M. Bellare and C. Namprempre. “Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm.” In: *Journal of Cryptology* 21.4 (Oct. 2008), pp. 469–491.

[64] J. Rizzo and T. Duong. *Practical Padding Oracle Attacks*. Aug. 2010.

[65] S. Vaudenay. “Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS...” In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by L. R. Knudsen. Berlin, DE: Springer Berlin Heidelberg, 2002, pp. 534–545.

[66] *The Transport Layer Security (TLS) Protocol Version 1.2*. Document. Internet Engineering Task Force, Aug. 2008. URL: `https://tools.ietf.org/html/rfc5246`.

[67] *Java Card Technology*. Oracle. URL: `https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html` (visited on 02/10/2019).

[68] R. N. Akram, K. Markantonakis, and K. Mayes. “An Introduction to Java Card Programming.” In: *Secure Smart Embedded Devices, Platforms and Applications*. Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 497–513.

[69] *GlobalPlatform Technology – Card Specification V2.3.1*. Standard. GlobalPlatform, Mar. 2018.

[70] *Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2006.

[71] *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange.* Standard. Geneva, CH: International Organization for Standardization, Apr. 2013.

[72] M. Tunstall. "Smart Card Security." In: *Secure Smart Embedded Devices, Platforms and Applications.* Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 145–177.

[73] G. Madlmayr, C. Kantner, and T. Grechenig. "Near Field Communication." In: *Secure Smart Embedded Devices, Platforms and Applications.* Ed. by K. Markantonakis and K. Mayes. New York, NY, USA: Springer New York, 2014, pp. 351–367.

[74] *Cards and security devices for personal identification – Contactless proximity objects – Part 3: Initialization and anticollision.* Standard. Geneva, CH: International Organization for Standardization, July 2018.

[75] *Cards and security devices for personal identification – Contactless proximity objects – Part 4: Transmission protocol.* Standard. Geneva, CH: International Organization for Standardization, June 2018.

[76] *Smart Card CCID Device Group Specification.* USB Device Working Group (DWG). URL: `http://www.usb.org/developers/docs/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf` (visited on 08/29/2018).

[77] A. Sadeghi, C. Wachsmann, and M. Waidner. "Security and privacy challenges in industrial Internet of Things." In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* June 2015, pp. 1–6.

[78] E. Bou-Harb. "Passive inference of attacks on SCADA communication protocols." In: *2016 IEEE International Conference on Communications (ICC).* May 2016, pp. 1–6.

[79] T. Ulz et al. "Bring your own key for the industrial Internet of Things." In: *2017 IEEE International Conference on Industrial Technology (ICIT).* IEEE. Mar. 2017, pp. 1430–1435.

[80] T. Ulz et al. "Secured and Easy-to-Use NFC-Based Device Configuration for the Internet of Things." In: *IEEE Journal of Radio Frequency Identification* 1.1 (Mar. 2017), pp. 75–84.

[81] S. Weyer et al. "Towards Industry 4.0 - Standardization as the crucial challenge for highly modular, multi-vendor production systems." In: *IFAC-PapersOnLine* 48.3 (2015). 15th IFAC Symposium onInformation Control Problems inManufacturing, pp. 579–584.

[82] M. Loskyll et al. "Context-Based Orchestration for Control of Resource-Efficient Manufacturing Processes." In: *Future Internet* 4 (Aug. 2012).

[83] W. Jiang et al. "Dynamic Security Management for Real-time Embedded Applications in Industrial Networks." In: *Comput. Electr. Eng.* 41.C (Jan. 2015), pp. 86–101.

[84] J. Eliasson et al. “Towards industrial Internet of Things: An efficient and interoperable communication framework.” In: *2015 IEEE International Conference on Industrial Technology (ICIT)*. Mar. 2015, pp. 2198–2204.

[85] S. Guoqiang et al. “Design and Implementation of a Smart IoT Gateway.” In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. Aug. 2013, pp. 720–723.

[86] S. Nuratch. “The IIoT devices to cloud gateway design and implementation based on microcontroller for real-time monitoring and control in automation systems.” In: *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. June 2017, pp. 919–923.

[87] W. Li-Hong, T. Hai-Kun, and Y. G. Hua. “Sensors Access Scheme Design Based on Internet of Things Gateways.” In: *2014 Fifth International Conference on Intelligent Systems Design and Engineering Applications*. June 2014, pp. 901–904.

[88] D. C. Yacchirema, M. Esteve, and C. E. Palau. “Design and implementation of a Gateway for Pervasive Smart Environments.” In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2016, pp. 004454–004459.

[89] S. K. Datta and C. Bonnet. “Smart M2M Gateway Based Architecture for M2M Device and Endpoint Management.” In: *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*. Sept. 2014, pp. 61–68.

[90] S. Toc and A. Korodi. “Modbus-OPC UA Wrapper Using Node-RED and IoT-2040 with Application in the Water Industry.” In: *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)*. Sept. 2018, pp. 000099–000104.

[91] K. A. Nsiah et al. “An open-source toolkit for retrofit industry 4.0 sensing and monitoring applications.” In: *2018 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. May 2018, pp. 1–6.

[92] M. A. O. Pessoa et al. “Industry 4.0, How to Integrate Legacy Devices: A Cloud IoT Approach.” In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2018, pp. 2902–2907.

[93] Birol Çapa et al. “Rapid PLC-to-Cloud Prototype for Smart Industrial Automation.” In: *Proceedings of the 2Nd International Symposium on Computer Science and Intelligent Control*. ISCSIC ’18. Stockholm, Sweden: ACM, 2018, 33:1–33:5.

[94] H. Cho and J. Jeong. “Implementation and Performance Analysis of Power and Cost-Reduced OPC UA Gateway for Industrial IoT Platforms.” In: *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*. Nov. 2018, pp. 1–3.

[95] M. V. García et al. “OPC-UA communications integration using a CPPS architecture.” In: *2016 IEEE Ecuador Technical Chapters Meeting (ETCM)*. Oct. 2016, pp. 1–6.

[96] A. Veichtlbauer, M. Ortmayer, and T. Heistracher. “OPC UA integration for field devices.” In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. July 2017, pp. 419–424.

[97] P. Birnstill et al. “Introducing remote attestation and hardware-based cryptography to OPC UA.” In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. Sept. 2017, pp. 1–8.

[98] *Raspberry Pi 3 Model B*. Raspberry Pi Foundation. URL: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/` (visited on 08/23/2018).

[99] *FIDO multi-factor authentication*. Infineon Technologies AG. URL: `https://www.infineon.com/cms/en/product/promopages/about-fido/#fido2` (visited on 03/02/2019).

[100] *Intermediate Results of the TrustworSys Demonstrator*. YouTube. URL: `https://www.youtube.com/watch?v=aaXXFiZluzs` (visited on 02/28/2019).

[101] *CMake*. Kitware. URL: `http://cmake.org/` (visited on 03/14/2019).

[102] *Ninja*. Ninja-Build. URL: `https://ninja-build.org/` (visited on 09/12/2018).

[103] *GCC, the GNU compiler collection*. Free Software Foundation, Inc. URL: `https://gcc.gnu.org/` (visited on 09/04/2018).

[104] C. Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization.” MA thesis. Urbana, IL, USA: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[105] *Visual C++*. Microsoft. URL: `https://docs.microsoft.com/en-us/cpp/` (visited on 09/04/2018).

[106] *Doxygen*. Dimitri van Heesch. URL: `https://www.stack.nl/~dimitri/doxygen/` (visited on 09/04/2018).

[107] *DAVE*. Infineon Technologies AG. URL: `https://www.infineon.com/cms/en/product/promopages/aim-mc/DAVE3-development-platform.html` (visited on 03/07/2019).

[108] *Universal Serial Bus 2.0 – Specification*. Standard. USB Implementers Forum, Inc., Apr. 2000.

[109] *Amalgamate*. Github. URL: https://github.com/edlund/amalgamate (visited on 09/04/2018).

[110] *Making a directory tree of folders and files*. Stackexchange. URL: https://tex.stackexchange.com/a/328890 (visited on 09/04/2018).

[111] *winscard.h header*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/desktop/api/winscard/ (visited on 08/29/2018).

[112] *PCSC lite project*. Ludovic Rousseau. URL: https://pcsclite.apdu.fr/ (visited on 03/07/2019).

[113] *CCID free software driver*. Ludovic Rousseau. URL: https://ccid.apdu.fr/ (visited on 03/07/2019).

[114] *JSMN*. GitHub. URL: https://github.com/zserge/jsmn (visited on 03/07/2019).

[115] *rsync*. Samba-Team. URL: https://rsync.samba.org/ (visited on 03/24/2019).

[116] *GNU screen*. Free Sofrware Foundation, Inc. URL: https://www.gnu.org/software/screen/ (visited on 03/24/2019).

[117] *open62541*. GitHub. URL: https://github.com/open62541/open62541 (visited on 03/07/2019).

[118] *mbedTLS*. GitHub. URL: https://github.com/ARMmbed/mbedtls (visited on 03/07/2019).

[119] *Eclipse Mosquitto*. Github. URL: https://github.com/eclipse/mosquitto (visited on 09/07/2018).

[120] *OpenSSL*. GitHub. URL: https://github.com/openssl/openssl (visited on 03/07/2019).

[121] *GTK*. GitHub. URL: https://github.com/GNOME/gtk (visited on 03/07/2019).

[122] *scapy*. GitHub. URL: https://github.com/secdev/scapy (visited on 03/07/2019).

[123] *TkInter*. Python Software Foundation. URL: https://wiki.python.org/moin/TkInter (visited on 03/07/2019).

[124] *WiringPi*. Wiring Pi. URL: http://wiringpi.com/ (visited on 03/07/2019).

[125] *A simple voltage divider*. Wikipedia. URL: https://en.wikipedia.org/wiki/Voltage_divider#/media/File:Impedance_voltage_divider.svg (visited on 03/07/2019).

[126] *Embedded Python*. Python Software Foundation. URL: https://wiki.python.org/moin/EmbeddedPython (visited on 03/07/2019).