FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Security concepts for Linux based CPS applicable in safety critical infrastructures

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martin Führer
Matrikelnummer 1225177

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Stefan Seifried, BSc

Wien, 5. September 2019

Martin Führer	Wolfgang Kastner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Security concepts for Linux based CPS applicable in safety critical infrastructures

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Martin Währer
Registration Number 1225177

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dipl.-Ing. Stefan Seifried, BSc

Vienna, 5th September, 2019

Martin Währer	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Martin Würher
Theumermarkt 1/9/8
1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. September 2019

Martin Würher

Kurzfassung

Cyber-physische Systeme (CPS) bestehen aus unterschiedlichsten Komponenten, unter anderem aus Sensoren, Aktoren, Steuerungsgeräten oder HMIs (Mensch-Maschine-Schnittstellen). Üblicherweise sind die einzelnen Komponenten der CPS eng miteinander verbunden. Für übergeordnete Verbindungen (z.B. Verbindungen zwischen Steuerungsgeräten) spielt IP-basierte Kommunikation eine immer wichtiger werdende Rolle. Weiters kann das Betriebssystem Linux in vielen CPS eingesetzt werden, solange die dahinter liegenden Komponenten die Anforderungen erfüllen.

Sind diese CPS-Komponenten Teil einer sicherheitskritischen Infrastruktur (SCCPS), sind weitere Security-Anforderungen gefordert. Daher werden in dieser Arbeit drei Sicherheitsmaßnahmen für Linux basierte Komponenten einer SCCPS vorgestellt. Zuerst wird ein sicherer Startprozess für Linux basierte SCCPS Komponenten präsentiert, der sicherstellt, dass auf den individuellen Komponenten eines SCCPS nur signierte Software ausgeführt werden kann. Des Weiteren wird ein sicherer, IP-basierender Kommunikationsansatz für die eng miteinander verbundenen Komponenten eines SCCPS diskutiert. Abschließend wird ein sicherer Updatemechanismus für die Komponenten eines SCCPS erläutert, der sicherstellt, dass nur signierte Software-Updates installiert werden können und der für ein abgebrochenes Update eine Fallback-Lösung bereitstellt.

Um diese Sicherheitsmaßnahmen analysieren zu können, wird ein Threatmodell erstellt. Dieses umfasst die genannten drei Sicherheitsmaßnahmen und kann daher als Startpunkt für die Sicherheitsanalyse verwendet werden. Weiters wird in einem Proof-of-Concept gezeigt, dass diese Maßnahmen durchführbar sind. Da die vorgestellten Sicherheitsmaßnahmen Linux und IP-basierte Kommunikation voraussetzen, muss Linux auf den entsprechenden SCCPS Komponenten lauffähig sein und IP-basierte Kommunikation zum Einsatz kommen.

Abstract

Cyber-physical systems (CPSs) consist of a heterogeneous set of components like sensors, actuators, control devices or HMIs. Typical for these CPSs are the ubiquitous interconnections between their components. Especially for higher level connections (e.g. connections between control devices), IP-based communication plays an important role. According to current observations IP will become even more important in the future. Additionally, the OS Linux can be used in many CPSs as long as the underlying components fulfill the requirements.

If the CPS components are part of a safety critical infrastructure (SCCPS), additional security requirements may be demanded. Therefore, this thesis proposes three major security measurements for Linux-based components of SCCPSs: First, a trusted boot mechanism for Linux-based SCCPS components ensuring that only signed software is executed on the individual components of a SCCPS. Second, a secure communication approach for IP-based communication enhancing the security for the ubiquitous communication between several components of a SCCPS. And finally, a secure update mechanism ensuring that only signed software updates can be installed on the components of a SCCPS and providing a fallback for aborted update processes.

In order to analyze the approaches regarding the security properties, a threat model is introduced. It covers the three approaches and can be used as a basis for a security analysis. To ensure that the proposed approaches are feasible, a proof of concept is performed. In conclusion, the presented security measures may not be applicable for some (legacy) parts of a SCCPS, whenever the underlying components are not capable of running Linux or do not use IP-based communication.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The digitizing of the world, as well as the huge adoption of Internet of Things (IoT) systems and cloud computing in industrial applications has led to Industry 4.0. This fourth industrial revolution (also known as smart manufacturing) is induced by the trend to interconnect the system components from the management level down to the field level. In many cases, this interconnection approach has already been used in fields that are safety critical (like nuclear power plants, water treatment systems, telecommunication systems, energy distribution or fire alarm systems). In order to describe the Industry 4.0 concepts, the Reference Architecture Model for Industry 4.0 (RAMI 4.0) was introduced. It is a model that consists of three axes (and is shown in figure 2.2): 1. The first axis describes six unique layers that are required to define the assets, their data and functionality. 2. The second axis represents the life cycle and value stream of the assets and the processes. 3. The third axis denotes the hierarchy levels (extended levels of the automation pyramid that is introduced in section 2.1.1) [1].

One feature of RAMI 4.0 is that it covers the complete life cycle of assets and processes that is defined in EN 62890. Thus, it covers everything from the development (type) to the production, maintenance and usage of the products (instances) [1]. According to EN 62890 the product type consists of a product ID, development documents, manufacturing- and test descriptions as well as technical documentation. Each product is a unit of a product type. To identify each individual product, a product serial number can be used. EN 62890 contains a generic life cycle model of a product type and a product instance. It is shown in figure 1.1. The life cycle model of the product type consists of a development phase, a sales phase and an after-sales support phase. After the “after sales support phase” the product (type) becomes obsolete. It is important that the standard service of the product starts when the first product has been delivered and ends when the product becomes obsolete (product is abandoned). This standard service typically

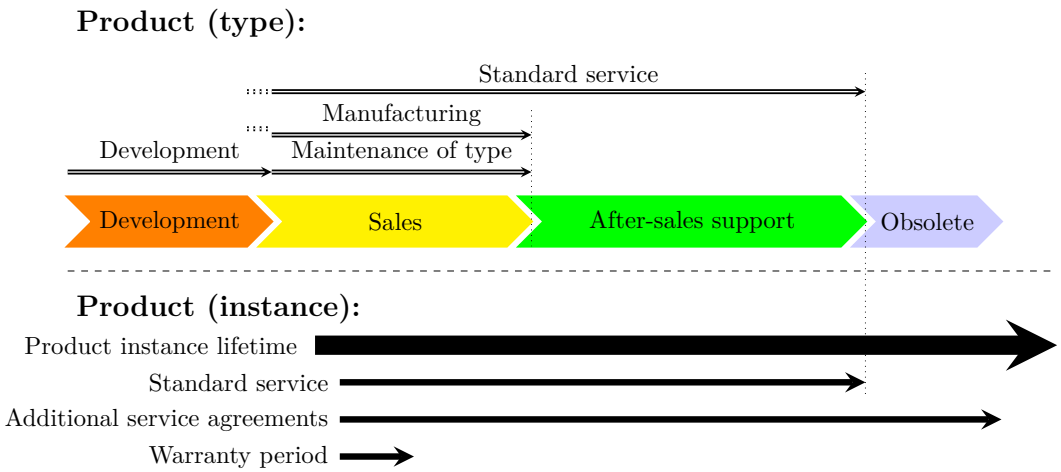


Figure 1.1: Generic life cycle model of a product type and a product instance [EN62890].

includes software/device updates and upgrades, repairs or provision of spare parts. The lifetime of the product instance starts when the product instance has been manufactured and ends with the disassembling or disposal (can be much longer than the life cycle of the product type). However typically, the time in which the product instance is in use is less than the lifetime, as it doesn't take outage time and installation time into account and ends with decommissioning. Furthermore, the warranty period as well as the standard service starts when the customer had received the product, even though, it may not be in operation at this point [EN62890].

In cases in which a system (type) consists of several components (types) and each component has its own life cycle, the system integration can be challenging. Thus, the more components that are involved in an automation system, the more challenging the integration is. Therefore, compatible components are required that satisfy the requirements of the original (abandoned) component. Additionally, due to the life cycle, it may be required to provide a secure software update mechanism for some devices in order to add additional features, ensure compatibility or to fix bugs and errors in the future. Usually, it is required that these components are able to communicate with each other. This requirement is the basis of Industry 4.0 which requires the interconnection of the involved components as well as the communication with cloud services. Thus, Industry 4.0 automation systems can be labelled as CPSs as they consist of a compound of interconnected devices (see section 2.2). Due to the huge set of interconnections between the components, CPSs enable new facilities like remote control or remote software updates, too. Furthermore, if these CPSs are connected to the Internet, the involved devices can communicate with cloud services. It may therefore be desired that all these communications are secured as well. Some components of these automation systems may be part of a safety critical infrastructure (like railway signaling systems, fire-alarm-systems, telecommunication systems, energy distribution or nuclear engineering). These safety

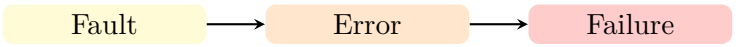


Figure 1.2: Relationship between faults, errors and failures.

critical automation systems are responsible for controlling several safety critical physical processes. If CPSs are part of a safety critical infrastructure, in which a failure or a malfunction leads to severe issues, these systems are called safety critical CPSs (SCCPSs) (see section 2.2.2) [EN62890], [2].

To avoid any misunderstandings in the following chapters, faults, errors and failures are defined as follows (and shown in figure 1.2):

Failure: A failure is "[...] an event that occurs when the delivered service of a system deviates from correct service" [3].

Error: "An error is the part of the system state that may lead to a failure" [3].

Fault: "A fault is the cause of an error" [3].

A SCCPS typically consists of several different device types (including sensors, actuators, human machine interfaces (HMIs), programmable logic controllers (PLCs) or control devices). All of them must be interconnected to control and monitor the physical process. Generally, safety critical automation systems consist of several devices that are scattered across the installation location. Therefore, communication security plays an important role for SCCPSs. In particular, the authentication of the communication partners and the message authentication is absolutely required, too. Additionally, some applications require confidential communication and therefore rely on message encryption. Especially if some SCCPS components communicate with cloud services on the Internet, message encryption can be desirable.

Additionally, as already mentioned, due to the long life cycle of an automation system, there may be devices of a SCCPS (e.g. control devices) that facilitate software updates. Thus, the devices must be engineered in such a manner that the installed software can be updated. In order to prevent malicious software update packages from being installed on these safety critical devices, it may be demanded that only trusted vendors are able to perform a software update. Furthermore, each updateable device must be able to verify the installed software before it is executed. This step is required to make sure that the software on the device has not been manipulated by adversaries [2].

1.2 Problem statement

Some parts of SCCPSs may be installed in environments with many passersby (like public transport stations or agencies) where people potentially have malicious intent. Thus, adversaries may have physical access to the devices or to the network infrastructure that

interconnects the SCCPS components. In order to prevent eavesdropping- or man-in-the-middle (MITM) attacks, authenticated and encrypted communication is desirable. Furthermore, as the trend towards interconnecting the components of safety critical systems is ever increasing, this topic will become even more important in the future. Moreover, the number of SCCPSs with connections to the Internet is also increasing. Thus, it is important to provide a secure communication channel to other devices in the Internet [4], [5], [FR1].

Unfortunately, the built-in security mechanisms of several devices may be insufficient or weak (e.g. weak passwords, revealed keys or weak algorithms). Additionally, some security facilities may be configured improperly. In order to address all these issues for devices that have already been installed in the field, a software update can be applied. Therefore, the components of a SCCPS must particularly be capable of software updates that can be applied in the future. Ideally, all components of a SCCPS receive software updates in time and are supported with software updates during the whole product life-cycle. Otherwise adversaries may be able to utilize known weaknesses in order to perform attacks on the SCCPSs. However, if the software of some SCCPS components can be updated, adversaries may be able to inject malicious software. Thus, it is important to authenticate the software that will be executed on these devices. This authentication can be achieved by a trusted booting mechanism. In order to implement this trusted booting mechanism for each individual SCCPS component, special hardware facilities are required [6], [7], [FR2], [FR3].

A major problem for SCCPS is the heterogeneity of the SCCPS components as a SCCPS typically consists of many different devices and device types. Moreover, many different network technologies can be used to interconnect the SCCPS components (see section 2.1.3). Unfortunately, some SCCPS components have constrained resources and several components and network technologies have different security capabilities. Thus, a system wide basic security level may be hard to achieve (see section 2.1). Therefore, it is challenging to provide a universally applicable secure update-, trusted booting- and secure communication mechanism for all devices in a SCCPS.

Since Linux based systems are becoming more and more attractive also in the automation field, this thesis concentrates on SCCPS with Linux operating systems.

1.3 Aim of the work

In order to increase the security of SCCPSs, three basic concepts for Linux based SCCPS components are introduced:

1. A trusted boot mechanism for Linux based SCCPS components ensures that only signed software is executed on the individual components of a SCCPS. Thus, this mechanism ensures that these devices do not execute unvalidated (potentially malicious) code. A possible way to achieve this is relying on a Read-only Memory (ROM), in which the execution-code is stored read only. However, if a ROM is

used, it is not possible to apply any future software updates, as the content in the ROMs can't be modified. Therefore, and because remote updates are necessary nowadays, an approach of authenticated booting for re-writeable memory is introduced.

2. A secure communication approach for IP based communication enhances the security for the ubiquitous communication between several components of a SCCPS. This approach ensures that the communication between two control devices and the communication between control devices and cloud services as well as other devices can be carried out authenticated and encrypted. This approach is a transparent unicast communication layer which encrypts and authenticates the payload of upper layers. As the proposed secure communication layer operates on top of the Internet Protocol (IP) layer, it is required that the communication is IP based. However, in section 5.2.1 a concept is introduced that enables IP communication for EIA-232 connections. Thus, even if the utilized network technologies are not capable of IP communication, it may be possible to upgrade other network protocols to support IP communication.

3. A secure update mechanism ensures that only signed software updates can be installed on the components of a SCCPS and providing a fallback for aborted update processes. This is required in order to fix security related bugs and increase the feature set of the SCCPS devices in the field. Additionally, the update-process should be resilient against connection-loss and power-loss. Moreover, if an update fails, the device should recover by starting the previous system-software state. Furthermore, the update image must be signed by a trusted entity and optionally partly or fully encrypted.

Furthermore, to analyze the security concepts, a threat model is created (see chapter 3). This threat model is created for an abstract system model that provides all minimal requirements in order to apply the three concepts.

In practice, a SCCPS generally consists of a heterogeneous set of distinct devices with different capabilities. However, if the minimal requirements (see section 3.3.2) are fulfilled, all three concepts can be applied. These requirements may be hard to achieve, particularly for lower level devices like sensors and actuators. But as such devices are not the main target of this approach, these components of a SCCPS are omitted. Nevertheless, if these minimal requirements are met, the concepts can be applied to other device types or lower level devices, too. This uniform device requirements and homogenous security mechanisms make it easier to provide a basic level of security. Furthermore, these uniform security facilities can be configured similarly across multiple devices.

1.4 Methodological approach and structure of the work

This thesis introduces several security related approaches for control devices of a SCCPS. First, a state-of-the-art analysis for common security concepts that suit the requirements

of SCCPS is performed in chapter 2. It is based on literature and Internet research and contains definitions of automation systems as well as CPSs. It introduces cryptographic properties of cryptographic algorithms, cryptographic communication protocols and hardware security schemes. Furthermore, the Linux startup process is introduced, as it is required for the trusted boot approach. Finally, some common software update approaches are introduced.

Chapter 3 contains a short introduction about threat models. It includes a system model for a fictitious SCCPS. This system model is used as the basis for the threat model in section 3.4

In chapter 4, a trusted boot approach, a secure communication concept and a secure update procedure for control devices of a SCCPS are introduced. These rely on the requirements that have been introduced in section 3.3 and mitigate the threats that have been identified in section 3.4. In order to demonstrate that these approaches are feasible, chapter 5 introduces a possible implementation for the trusted boot approach on an NXP i.MX7Dual platform as well as a secure communication layer that relies on Internet Protocol Security (IPSec). Furthermore, some hints for a secure update implementation based on SWUpdate are given in section 5.3.

Finally, chapter 6 concludes this thesis with a short recap of the presented approach and the proof of concept. Furthermore, some hints regarding future work are given.

CHAPTER 2

State-of-the-art

2.1 Automation systems

An automation system is a technical system which completes a specific task without human intervention. As human perception and environment interaction are very complex and hard to be imitated by machines, automation was initially used in controlled environments where the limited capabilities of the automation systems could only handle small, monotonous tasks. Some examples are periodic switching functions, switching hysteresis, sorting plants or robot-based drilling. All of these are straight-forward and monotonous tasks which, if repeated by humans, would result in more frequent errors. Therefore, the main reason for automation is the execution of such repeating tasks with increased accuracy and more reliability than would be achievable by humans. Additionally, automation helps reducing costs and can also be deployed in hazardous environments [8].

Automation trends like sensor fusion use different technologies of the domains artificial intelligence (AI), statistical estimation, pattern recognition and many more to create more sophisticated automation systems. These systems can even be used within the field of transportation and autonomous driving, in which growing demand can be observed [9], [10]. Furthermore, these developments lead to automation systems that can make semi-automated or automated decisions.

In summary, automation systems are used in different areas and domains including automotive, railway, aerospace, process and factory automation as well as the building automation domain [8], [11]. Depending on their respective domain, automation systems themselves as well as their requirements and properties vary greatly. Here are some examples:

Building automation and control systems (BACSs) may consist of many nodes. For huge buildings or building complexes, even more than 10,000 nodes are possible.

Most of the time, only soft real-time requirements are needed, as many BACSs are event-driven and do not need much regular traffic. However, installations that contain video-surveillance require BACSs that are capable of high data traffic. BACSs must be adaptable as buildings in general have a long lifespan, and it is likely that the system will be modified or extended in the future. Over the last few years, fire alarm and intrusion detection systems have become very popular and extend the BACSs domain [8].

Automotive systems typically have fewer nodes than BACSs in large buildings. As automotive systems have a much shorter lifespan than buildings, these systems are not as adaptable as BACSs. However, the data transfer is more time-critical, and transmission errors should be detectable [8]. For autonomous driving, in which light detection and ranging (LIDAR) sensors, radar sensors, video cameras or infrared cameras are used, automotive systems must be capable of transmitting at high data rates [10].

Industrial systems are similar to automotive systems regarding time-criticality and transmission errors. If the automation system relies on sensors that require high data rates (such as cameras), the system should be able to handle high transfer rates. However, large systems typically consist of many nodes. Like BACSs, industrial systems can also have long lifespans [8].

Thus, modern distributed automation systems consist of numerous nodes. These involve sensors, actuators, PLCs, HMIs, information & communication technology (ICT) devices like PCs, workstations or servers. Furthermore, network coupling devices such as switches or routers are used [12], [13], [14], [15], [16].

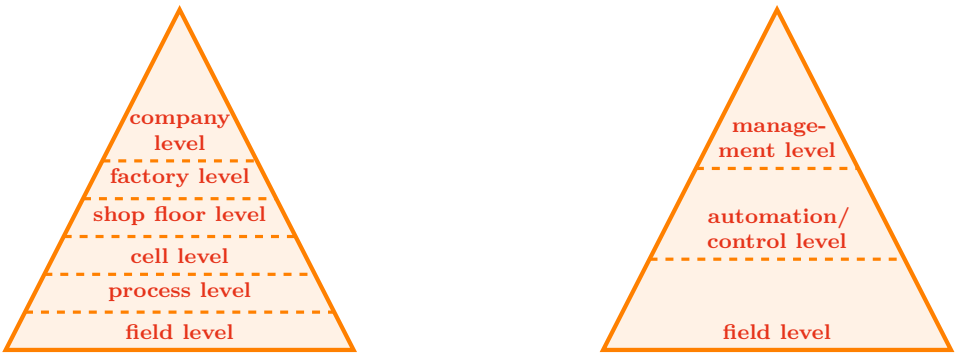
2.1.1 Automation pyramid

Although the diversity in the field of automation systems has led to many different communication variants, the automation pyramid concept has permeated all of them. Initially, the automation pyramid was finely segmented into four to six levels categorizing the means of communication and process automation (see figure 2.1a) [17], [18], [11], [EN62264-1], [19]. This number may vary, but typically the following levels were used: “company level”, “factory level”, “shop floor level”, “cell level”, “process level” and “field level” [11].

Company level is the top level, which is responsible for rough production planning and order handling.

Factory level is below the company level and is responsible for detailed production planning, defines the production processes and performs quality checks.

Shop floor level collects the data from the sensors in the lower levels and monitors the system. Additionally, this level is responsible for data archiving, can control the machines in the plant and react to extraordinary situations.



(a) Early automation pyramid consisting of six levels [11].

(b) Modern automation pyramid consisting of three levels [EN16484-2], [16], [11].

Figure 2.1: Different automation pyramids

Cell level controls each machine in the plant individually.

Process level is the interface between the sensors and actuators in the field level and the controlling devices in the cell level.

Field level consists of sensors and actuators [20].

The form of a pyramid lends itself well to the structure of an automation system: Lower levels like the field level are populated by many nodes, while the number of nodes decreases in higher automation levels. Furthermore, at lower levels, data validity is short, while data storage mechanisms are located at higher automation levels. Due to a possible large number of devices in the field level, many systems store only a fraction of all available sensor and actuator data at the company or management level. Therefore, at higher levels, data-filtering becomes more important [21], [FR4].

Nowadays, due to the advent of ICT, the pyramid has collapsed to three levels: "management level", "automation/control level" and "field level" (see figure 2.1b) [EN16484-2], [16], [17]. The evolution of ICT-systems in modern distributed automation systems has led also to enterprise resource planning (ERP)-systems that mostly form the management level of the three-level pyramid. These ERP-systems are responsible for several different business processes such as asset management, logistics, sales, maintenance, controlling, interfaces with customers. While control devices like PLCs or HMIs are part of the automation/control level and sensors or actuators are part of the field level [16].

Going by current observations, the projected future form of the pyramid would be condensed into only two levels: An automation level and a "field level". This is because many modern systems no longer have strict level boundaries anymore. As ICT technology plays an increasingly important role even at the "field level". Therefore, the boundaries between the levels of the automation pyramid are increasingly blurred. However, this gives rise to new challenges, as typical automation systems have a much longer life cycle than ICT systems [17], [11], [19], [11].

2.1.2 RAMI 4.0

In section 1.1, the Reference Architecture Model for Industry 4.0 (RAMI 4.0) (RAMI 4.0) has already been presented. It was introduced in 2015 to satisfy recent requirements of modern Industry 4.0 systems. Thus, RAMI 4.0 is an enhancement of the classic automation pyramid, as it covers two additional perceptions beside the automation levels of the automation pyramid: The product life cycle and the system architecture [22]. Thus, RAMI 4.0 is a 3D model which consists of three axes (see figure 2.2):

1. The vertical axis is the architecture axis. It consists of six unique layers: "Assets", "Integration", "Communication", "Information", "Functional" and "Business". The "Asset" layer represents the assets in the physical world. The layers above always correspond to an object in the "Asset" layer. Thus, the "Information" layer provides the transition between the physical- and the digital view (e.g. HMIs). The "Communication" layer is responsible for the communication/remote access between the individual components of the system. The "Information" layer handles the data as well as events that are used in the "Functional" layer. The "Functional" layer can be considered the brain of the system. As it defines the functions to monitor and control the assets. Thus, all functions, applications and systems intersect at this layer. Finally, the "Business" layer forms the big picture, as it is the result of the previous layers and defines the organization and business processes [1], [23].

2. The first horizontal axis is the complete life cycle of assets and processes according to [EN62890], which has already been explained in section 1.1 and shown in figure 1.1 [1], [23].

3. The second horizontal axis defines the hierarchy levels: These levels are structurally similar to the levels of the automation pyramid in section 2.1.1 and [EN62264-1], as it covers field devices, control devices, work centers as well as the "Enterprise" which denotes the company level. Furthermore, the pyramid levels have been extended with the "Connected world" level that represents connected industrial plants as well as external companies that are involved in the development/production processes. Additionally, the "Product" level has been added too, as the product is considered in RAMI 4.0 too [24], [22].

2.1.3 Fieldbus

As mentioned in section 2.1, there are very specific requirements needed at the field level depending on the domain of the automation system. Additionally, the number of devices in the automation system has increased over time. Therefore, star-topology (dedicated), point-to-point connections between sensors, actuators, HMIs, PLCs, single loop controllers and control or monitoring devices have become very expensive. Thus, the goals of the fieldbus systems are a reduction of wires and package pins on the

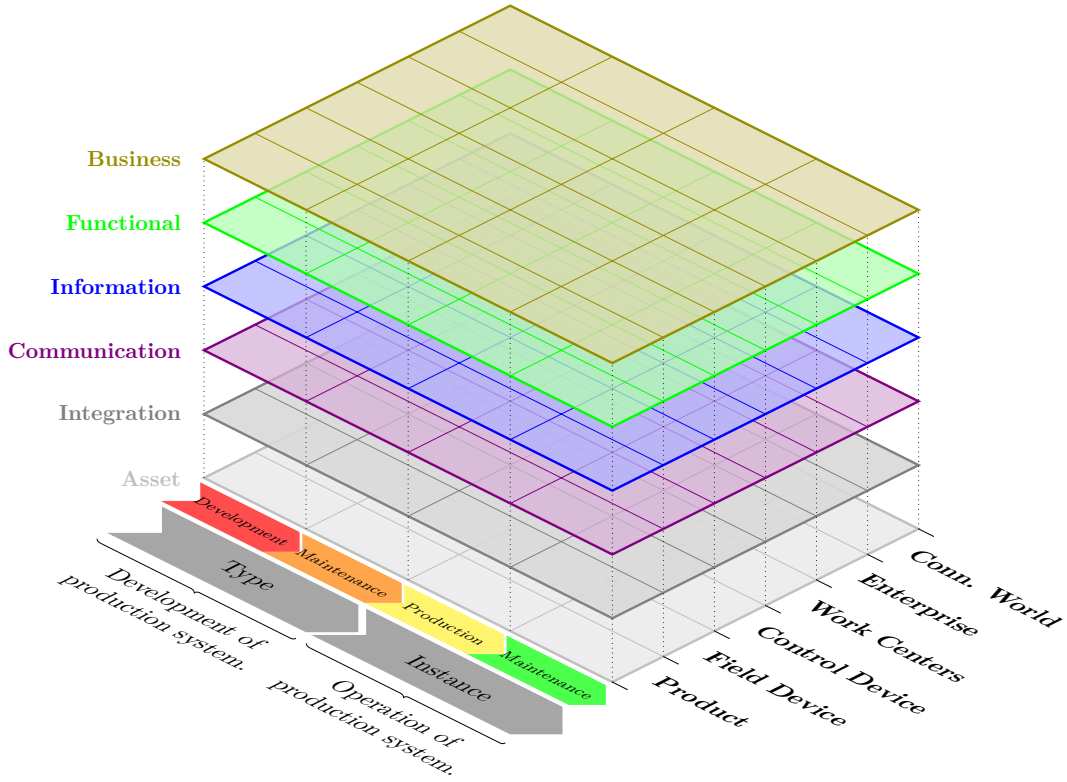


Figure 2.2: The Reference Architecture Model for Industry 4.0 (RAMI 4.0) [1], [25], [26].

integrated circuits (ICs) and the enhanced possibilities for accessing the sensor values and actuator commands. Additionally, the specific requirements of the domain of automation mentioned in section 2.1 must be met [EN61158-1], [11], [8], [27], [19].

Fieldbuses lead to an increased flexibility and modularity of the installation, but also the system configuration, commissioning and maintenance is simplified. These fieldbuses typically contain only the lower layers 1 and 2 as well as the application layer 7 of the Open Systems Interconnection (OSI) model. Typical fieldbus properties are serial transmission of data, support for long distances, master-slave communication models and integrated controllers. These properties were all present in the MIL 1553-bus, which was released in 1970 and can be considered as the first “real” fieldbus. Even though when the first industrial fieldbus systems were introduced in the early 1970s, they only started to emerge en masse in the 1980s. The first fieldbus systems were specially designed for automation domains, which in turn led to many different approaches and protocols being introduced because many companies were developing their own fieldbus systems. This era has been called “the fieldbus war”. Due to the enormous investment that had already been made to create active systems as well as due to national interests, the war persisted even when several companies had decided to create open specifications. So, even national standards competed against other national standards, but in the long run, open systems

finally became accepted and international standardization has become increasingly important. From this point on, different vendors have been able to produce compatible network nodes that has led to more customer choice. The fieldbus war ended with the release of the IEC 61158 norm at December 31^{st}, 2000. This norm was created by Comité Européen de Normalisation Électrotechnique (CENELEC) and includes the specification of several important fieldbus systems such as Process Field Bus (PROFIBUS), Factory Instrumentation Protocol (FIP) or Foundation Fieldbus (FF). In contrast to that of the industrial fieldbus networks, a much less controversial standardization process took place in the fieldbus networking area of building automation (BACSs) [28], [11], [27], [19], [29], [EN61158-1], [11].

Today, there still exist many different fieldbuses that address similar application domains and can technically be used for similar applications. The main differences are the physical transport media that form the individual characteristics of each fieldbus (e.g. in terms of reliability, distance, data rate, latency, etc.) [19].

This diversity also leads to a big challenge as the networking concepts that are used are mutually incompatible in general. Thus, many people tend towards Ethernet as the physical transport medium to achieve a homogenous enterprise network not only for office use but also for automation device interconnection. In the early days of Ethernet, real-time capabilities were missing. Nowadays, there exist several techniques like Process Field Network (PROFINET), Ethernet Powerlink or Ethernet for Control Automation Technology (EtherCAT) that all support real-time traffic. Additionally, some approaches like time-division multiple access (TDMA) exist, in which non-real-time traffic like TCP/IP (TCP/IP) or UDP/IP (UDP/IP), can be used alongside real-time traffic [11]. These changes are also preparations for the upcoming IP-based networks in the field level.

2.1.4 WSN

As already stated in section 2.1, there are many distributed automation systems that contain several thousand nodes each that all need to be interconnected. Several of these connections can already be realized by wireless technologies that replace traditional physical fieldbus systems. Providing more flexibility and lower connection costs, wireless connectivity has been adopted in industrial systems, building automation and e-Health applications to name a few, and its importance is expected to continue to rise in the future [8], [30], [31]. The term wireless sensor networks (WSNs) has been introduced to define the wireless connections between the sensors, actuators and control devices. From a current point of view, wireless connectivity will not completely replace wired connections in the future, but rather that both will coexist. Most wireless protocols use standard technologies as the Institute of Electrical and Electronics Engineers (IEEE) 802.x-standards that include 802.11 (Wi-Fi), 802.15.4 (wireless PAN (WPAN)) and 802.15.1 (Bluetooth) [11].

Many devices in WSNs are low cost, energy-efficient devices, that are interconnected wirelessly and battery-powered. Very often, WSNs are sensing systems that can be

easily updated or extended by additional wireless devices. Particularly in cases in which additional nodes are added to the network, scalability and self-organization features are required [32].

For WSNs, a shift to IP-based communication can be observed: This refers not only to Wi-Fi, which is already capable of IP, it corresponds more so to WPAN-systems with additional IP capabilities as those defined in IPv6 over Low Power WPAN (6LoWPAN) [11]. Additionally, there also exists a Requests for Comments (RFC) for Bluetooth to enable 6LoWPAN (IP) support [RFC7668].

2.1.5 Hybrid networks

In the future, the hybrid approach consisting of wireless field networks, traditional field-buses and modern Industrial Ethernet networks on the field level will play an important role [11]. Additionally, IP is seeing more frequent adoption in networks built on both wireless and wired connectivity. The clear advantage of IP is that the underlying transport medium plays only a secondary role, because it is transparent to the application under the assumption that requirements like real-time capabilities, latency or throughput are met.

Even though wireless connectivity brings many benefits at field level, wired technologies like Ethernet or even fiber remain the first choice at the upper levels of the automation pyramid (backbone networks). This is mainly due to wired connections allowing for higher data rates and not being as prone to electrical interference.

2.2 CPS

"CPSs are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops in which physical processes affect computations and vice versa." [33] According to this definition, a CPS is an embedded system or automation system connected with many devices, capable of sending and receiving data via networks. Because modern automation systems already monitor, coordinate and control the required operations and data in distributed ways, there exist many CPSs today. Also, in the future, more and more objects in the physical environment will gain computing and communication capabilities. This includes many areas such as critical infrastructure control, process control and factory automation, distributed robotics, building and environmental control, assisted living and traffic control systems. These systems form the connection between the cyber-world and the physical world [33], [34], [5], [2].

2.2.1 IoT, M2M and Industry 4.0

As the data exchange and the interconnection between CPSs or even to the Internet is the most important feature of CPS, IoT can be realized with CPS. The main idea with IoT is that many low-cost CPS-nodes (computing devices, sensors, etc.) with small

form factors and even wireless connections are part of an overarching distributed system (system of systems). Furthermore, through the connection to the Internet, data transfer from the nodes to the cloud may also be possible [34], [5].

Therefore, IoT and CPS are the basis of the growing Industry 4.0 initiative. Which is the fourth iteration of the industrial revolution. The first industrial revolution was "mechanization", the second was "mass production", the third was "digitization" and the fourth is the usage of CPS in the industry [34].

When considering IoT and Industry 4.0, machine-to-machine (M2M) communication also must be mentioned, which is exactly the main feature of CPSs: Devices that are highly interconnected communicate with each other. Based on this communication, these devices make automated decisions. Due to the huge interconnection of CPSs, the strict separation of distinct systems, is not feasible anymore [35]. Hence, a holistic security approach is needed.

2.2.2 SCCPS

Safety is defined in [EN61508] as "the absence of unacceptable risk of physical injury or damage to the health of people". Hence, safety-related systems deploy mechanisms that reduce the risk to a particular level or that embed appropriate countermeasures. Furthermore, when a fault induces a fatal error, which leads to physical injury or damage to the health of people, the system can be described as a safety critical system [8]. CPSs that can be safety critical systems are called SCCPSs.

Fatal failures that become critical may be caused by defect hardware, environmental damage, unintended operating errors and much more. However, security threats can also lead to critical failures. Some example domains, in which a failure can lead to severe consequences are aerospace, energy, automotive, railway, BACS and healthcare [2].

A concrete example of a SCCPSs is a fire alarm system, in which malfunctions may lead to health hazards. Furthermore, security flaws may be used, to produce enormous damage or even physical injury. Such a fire alarm system is depicted in figure 2.3 [36]. This example shows typical characteristics of CPSs like the interconnection between all involved devices, the Internet connection as uplink connection to other systems, the actuators and the sensors. These, in the form of field devices that monitor and interact with the physical processes. Furthermore, the fire alarm control panels (FACPs), that collect and visualize the sensor data, perform the actual computation and control the actuators.

This fire alarm system is a simple safety critical BACS arranged in a 2-level architecture (field level and management level). At the field level, manual alarm triggers, sprinklers, alarm lights, sirens and fire detectors can be seen. The field devices are connected via a fieldbus directly with the responsible FACP and the fieldbus forms a physical ring for reliability reasons. The FACPs itself are part of the management level and are

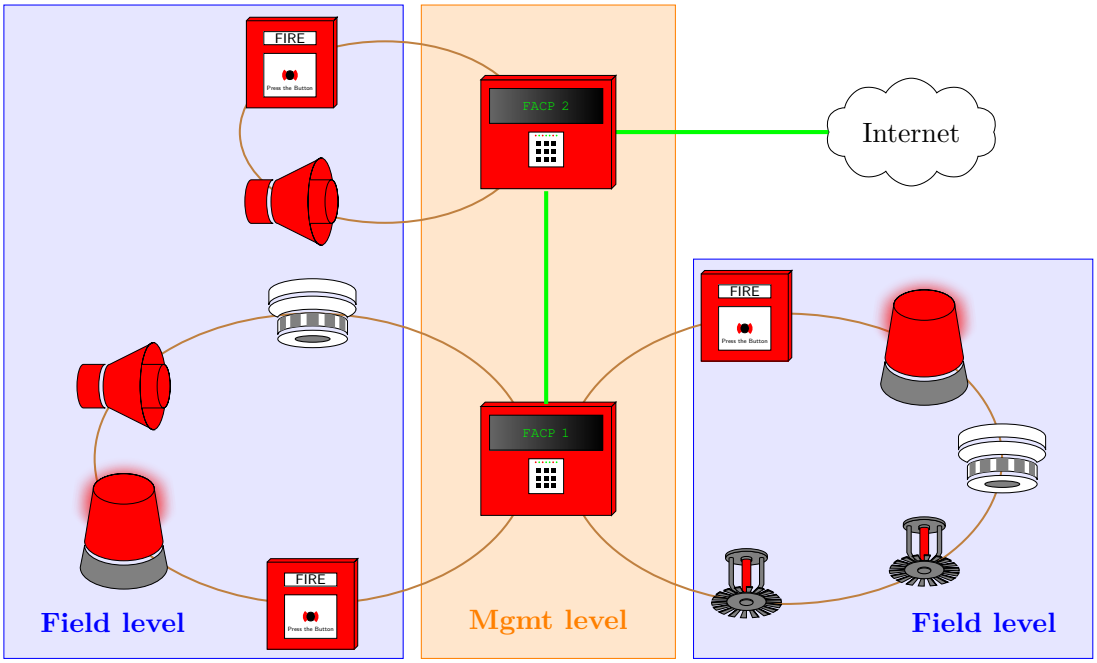


Figure 2.3: Fire alarm system as an example of a SCCPS. Backbone connections are marked with a green line, while fieldbus connections are marked with a brown line.

interconnected by fiber connections. FACP 2 is further connected to the Internet by fiber connections as well. Via this Internet connection, alarms can be sent directly to the fire department or software updates can be pushed to the devices. Additionally, if FACP 2 has Internet sharing capabilities, it is possible for FACP 1 to gain access to the Internet as well.

Due to the possible damage or physical injury of SCCPS, not only functional safety is an important goal of SCCPSs, also device and network security is a substantial task during the design of such systems. As the complexity and the adoption of safety critical systems increase, it is very important to focus on the system security. In particular, because many system developers considered security as a marginal issue for many years [2], [36].

Several security concepts for automation systems rely on the concept of physical separation ("air-gapping"). However, as mentioned in section 2.2.1, for CPSs that operate in several dimensions (electrical, mechanical, pneumatic, hydraulic or kinetic), this is not feasible anymore and thus need to be revisited, because all these operation modes are potentially penetrable. Additionally, new trends like remote system access, ubiquitous connectivity, or cost reduction by reusing of existing networks lead to dependencies that contradict the physical separation principle. Even at field level, many measure and instrumentation application trends to distributed data acquisition arise [35], [36], [11], [37], [8].

Another problem of modern communication techniques is that sufficient security mechanisms are often not supported or not sufficient enough. This may be because these security mechanisms are not mandatory or they lead to some drawbacks at the communication [37], [38], [39]. Therefore, this work covers mechanisms to enhance the security of SCCPSs. In section 2.3, a short summary of common security mechanisms is given.

2.3 Security considerations

A goal of this work is to establish a holistic security concept for SCCPSs. In this section, some basic understanding of existing security measures is conveyed.

2.3.1 Security objectives

Confidentiality, integrity and availability are three important targets in the context of security and are therefore often abbreviated as CIA [38], [37], [39], [40]. In figure 2.4, this CIA-triad is shown. It contains the three primary security goals: [8]

Confidentiality: Confidentiality means, that protected data is kept secret, as long as no authorized entity has successfully authenticated itself. It is important, that authentication is not part of this property [41], [42].

Integrity: Integrity ensures, that information was not altered or modified. If the information is transferred, integrity ensures that the content was not changed by some entities between the sender and the receiver [41], [42].

Availability: Availability ensures, that authenticated entities have data access each time they needed it. This means, that the system needs to be in a functional state. Stopped systems may be highly secure, but they are not functional and therefore no entities are able to request data [8], [40].

Even though some additional objectives are often used:

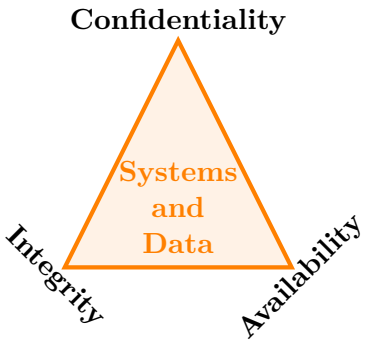


Figure 2.4: CIA-triad [40]

Authentication: Authentication consists of two parts; one is responsible for entity authentication (i.e. to ensure the correct identity of an entity) and the other part is responsible for message authentication. Authentication ensures therefore the identity and origin of communication [43], [42], [8].

Authorization: Authorization is preventing illegitimate entities from doing actions, that require special legitimation (access control) [8], [37].

Non-repudiation: Non-repudiation is a combination of integrity and authentication. It allows an entity to be sure, that another particular entity is responsible for a particular action [44], [42].

If the focus lies on cryptography itself and not at the system, instead of the CIA-triad, the main four objectives of cryptography are used. They are: 1. confidentiality, 2. integrity, 3. authentication and 4. non-repudiation [41], [42].

2.3.2 Cryptographic algorithms

There exist three main categories for cryptographic algorithms: Symmetric cryptography, asymmetric cryptography and hashing algorithms.

2.3.2.1 Hashing algorithms

Integrity can be achieved with the use of hashing algorithms. The purpose of a hashing algorithm is the creation of a unique digital fingerprint of a given input data set. This digital fingerprint is always equal for the same input data (comparability). A secure hash function is a function which generates a fixed size output of an input of arbitrary length and can't be reversed. Hence, it should not be possible, to generate the same hash with two different data sets (collision). Furthermore, it shall be hard to infer a data set of a given hash. Therefore, hashing functions have no inverse function and are often called "one-way" functions [45], [42], [46].

2.3.2.2 Symmetric cryptography algorithms

Symmetric cryptography algorithms provide confidentiality and use one single key for de- and encryption. Therefore, each entity that needs to decrypt the enciphered text must have knowledge of the key. In contrast to hashing algorithms that only generate fingerprints, all data must be restorable. Therefore, symmetric cryptography algorithms are often called "two-way" functions. Hence, it should be hard, to extract the original data or the used key of the enciphered text. Symmetric cryptography is well suited for large data and is in general faster than asymmetric cryptography [40].

2.3.2.3 Asymmetric cryptography algorithms

Asymmetric cryptography algorithms are also known as public key cryptography algorithms. In contrast to symmetric cryptography, asymmetric cryptography uses two keys:

A public key and a private key. Because these keys are mathematically related, both are generated by a single entity and no one can conclude from one of both keys the other corresponding key. While the private key must be secretly stored by the owner, the public key can be given to everyone [45], [40], [42].

When Bob encrypts data for Alice, Bob uses the public key from Alice to encrypt the data and send the enciphered text to her. Finally, Alice uses her own private key to decrypt the enciphered text from Bob. Additionally, both keys can be used in the reversed direction as well (encrypt data with private key and decrypt with public key), as the encryption operation applied with one key can simple be reversed by the other key. Therefore, not only symmetric cryptography, but also asymmetric cryptography provides confidentiality. Additionally, asymmetric cryptography provides non-repudiation under the assumption that the private key is kept private, as the only entity which has knowledge of the private key can perform the required operations [42], [40], [45].

2.3.2.4 DH

One major problem in symmetric cryptography is the transfer of the encryption key. As the symmetric key should in general not be transferred in plain text via public accessible transport media. To overcome this problem, key agreement protocols can be used. One of the first key agreement protocols that is also widely used today is the Diffie-Hellman (DH) key agreement protocol[1] [47]. This key agreement protocol relies on the public and private keys of two communication partners and creates a common secret key based on these keys.

Assume Alice and Bob want to derive a symmetric key that should later be used for symmetric encryption: First, Alice and Bob agree on two integers $p \in \mathbb{P}$[2] and $g \in \mathbb{P}$. In the next step Alice chooses a random integer a and Bob chooses a random integer b[3] as their private key. For the corresponding public keys Alice calculates $A = g^a \mod p$ and Bob calculates $B = g^a \mod p$. As both calculated their public keys on their own, Alice sends A to Bob and Bob sends B to Alice. Once Alice receives B, she calculates with her private key a the equation (2.1):

$$\Big(\underbrace{B}_{=g^b \mod p}\Big)^a \mod p = \underbrace{(g^b \mod p)^a \mod p}_{=(g^b)^a \mod p} = \underbrace{(g^b)^a}_{=g^{a\cdot b}=g^{b\cdot a}} \mod p = K \tag{2.1}$$

And when Bob receives A he calculates with his private key b the equation (2.2):

$$\Big(\underbrace{A}_{=g^a \mod p}\Big)^b \mod p = \underbrace{(g^a \mod p)^b \mod p}_{=(g^a)^b \mod p} = \underbrace{(g^a)^b}_{=g^{b\cdot a}=g^{a\cdot b}} \mod p = K \tag{2.2}$$

Therefore, both Alice and Bob get the exact same result K that can further be used as an encryption key for symmetric encryption. Thus, with this protocol it is possible that

[1] The DH key agreement protocol is also known as DH key exchange. However, no key is exchanged, instead two parties agree on a common key [47].

[2] p should be sufficiently large.

[3] Also, a and b should be sufficiently large.

two parties that are connected via a publicly accessible media are able to agree on a key that is only known by themselves without the need for Pre-shared keys (PSKs) [47], [40], [48].

2.3.2.5 PFS

Some cryptographic protocols support Perfect forward secrecy (PFS). In this operation mode the derived symmetric keys are only valid for a certain lifetime or a session. If the session is closed or the lifetime of the symmetric key is expired, both parties must agree on a new symmetric key (e.g. by doing a new DH). Once the new symmetric key was created, previous symmetric keys can be destroyed. Thus, even if adversaries can get a symmetric key, they are able to decrypt all messages that were encrypted with this particular key. However, if the symmetric keys change sufficiently often, it is hard for adversaries to decrypt other messages. In order to implement PFS correctly, new keys must be generated independently of each previous key. Otherwise, adversaries may be able to derive future/past keys from a captured key. [49], [47]

2.3.3 Hybrid cryptography

As mentioned in section 2.3.2, none of the algorithms presented provides support for all four objectives of cryptography. Thus, these algorithms are in practice combined to obtain the properties of all cryptography objectives. These combinations are called hybrid cryptosystems [42].

2.3.3.1 Digital signatures

The aim of digital signatures is to achieve Integrity, Authentication and Non-Repudiation. This can be achieved by a combination of asymmetric cryptography and hashing: In more detail, the first step is to use the hash function, to generate a hash of the data which should be signed. The second step is the asymmetric encryption of the hash with the private key. The encrypted hash is further called signature and can be distributed along with the data and the corresponding public key. With this signature, everyone who knows the public key corresponding to the private key, which was used to create the signature, can prove (by decrypting the signature with the public key) that the signature was created by the private key. Therefore, the entity which signed the data can be verified (authenticity of the sender) and the signed entity cannot disown the signature by claiming that it was forged (non-repudiation). Hence, if the signature was successfully decrypted, the hash of the data is received. If both, the hash of the distributed data and the decrypted signature are equal, it is guaranteed that the data was not changed by someone (integrity). Thus, a digital signature is similar to a real-world signature, which also doesn't provide confidentiality, but ensures that the signing entity agrees to certain terms and conditions (e.g. for contracts) [45], [42], [50].

2.3.3.2 Hybrid encryption

Many cryptography implementations combine also symmetric and asymmetric encryption algorithms, to achieve the performance of the symmetric encryption and the key exchange advantages of asymmetric cryptography: When Bob encrypts data for Alice using hybrid encryption, Bob creates at first a random symmetric key, which is used for the symmetric encryption algorithm. After Bob encrypted the data with a symmetric encryption algorithm, the symmetric key is then encrypted from Bob with the public key of Alice by an asymmetric encryption algorithm. Once Alice received both, the encrypted key and the encrypted data, she simply uses her private key, to decrypt the symmetric key. Furthermore, with this decrypted symmetric key, she can decrypt the encrypted data also [48].

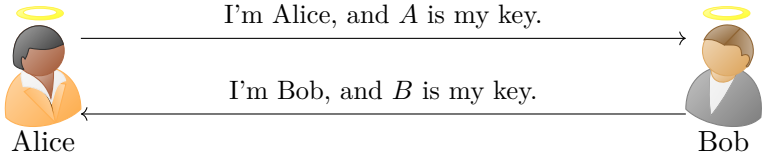
2.3.4 PKI

A huge disadvantage of the presented methods is the vulnerability against MITM attacks. If two entities do not know each other, it may be hard for them, to safely exchange their public keys. In figure 2.5, this problem is sketched. In figure 2.5a, a regular public key exchange mechanism can be seen: Alice sends Bob her public key (A) and Bob responds with his public key (B). This mechanism works well, if both know each other and are certain that the received key comes for sure from Bob or Alice. However, if Alice can't be sure that the received key comes really from Bob and Bob can't be sure that the received key was sent by Alice, a MITM-attack is possible. In figure 2.5b, this case is sketched: Alice and Bob think both that they are exchanging the keys with each other, but Mallory catches the public keys (A and B) of both and acts as the respective opponent. This means that Mallory sends her own public keys (M_1 and M_2) instead of the expected keys (A and B) to Alice and Bob.

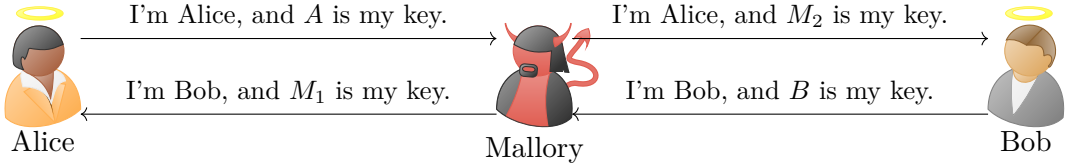
As both of them send only the public key and the public key can be published everywhere, the cryptosystem itself is not broken by this attack, but when Alice encrypts data for Bob, she uses the public key M_1, which she got at the key exchange step before. Because M_1 was from Mallory and not from Bob, Alice encrypts the message directly for Mallory with it. Even though because Bob thinks that Mallory is Alice, Mallory can encrypt the decrypted message received from Alice and encrypt it again with the public key (B) for Bob. In this case, Mallory acts as a transparent MITM-attacker, in which both Alice and Bob think that they are talking to each other encrypted, but Mallory can decrypt all encrypted messages.

2.3.4.1 public key infrastructure components

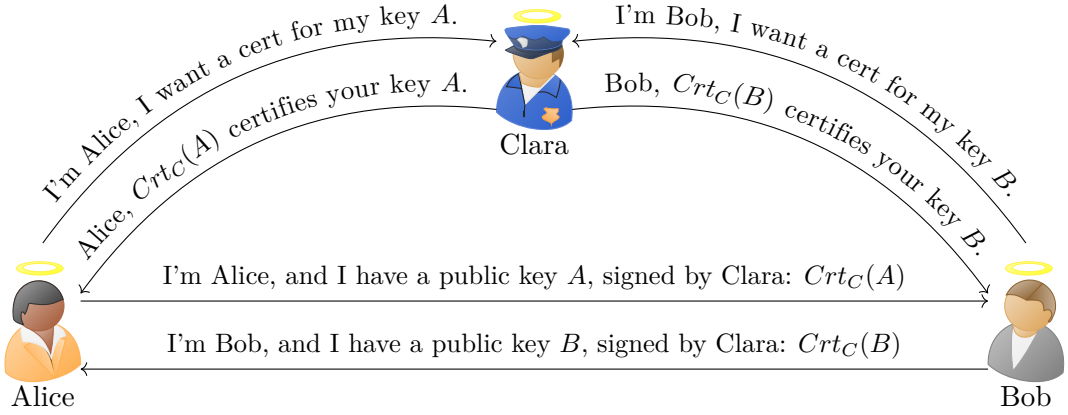
Public key infrastructures (PKIs) can be used, to prevent MITM-attacks. A PKI consists of the following components: A certificate, a certificate authority (CA), a registration authority (RA) and a certificate revocation list (CRL) [51], [48]. The following descriptions mean the simple direct trust model, when trust is stated in the text. It is explained in more detail (even with additional trust modules) in section 2.3.4.2.



(a) Regular public key exchange without an attacker.



(b) Public key exchange with Mallory, who performs a MITM-attack.



(c) Public key exchange with certified keys. Clara acts as a CA/RA and issues the certificates for the public keys.

Figure 2.5: Initial public key exchange example. Alice and Bob try to exchange their public keys. In order to prevent the interception of the messages by Mallory, the certificates from Clara can be used.

CA: A CA is comparable to a passport office, which issues passports [48]. Normally, a passport vouches a person, therefore if the passport office is trusted, each passport, that was issued by this passport office can be trusted too. Consequently, it is enough, to look at a passport, to make sure that a person is really the person, it pretends to be. For CAs, the situation is similar, but instead of passports, they issue certificates. Hence, if the CA is trustworthy, also the issued certificates are trustworthy, and a certificate vouches an entity [52].

RA: One of the tasks that must be accomplished at the passport office is checking the identity of the passport requesting persons. In a PKI, the registration and authentication are typically carried out by a RA. Even though CA and RA are defined as two separate entities, they may be also combined in one entity or one CA can have several RAs [46].

CRL: In the event of a failure in the PKI process (e.g. a leaked private key or an insufficient authentication by the RA), an already issued certificate might be revoked. All revoked certificates are collected and published by a CA. Therefore, before a certificate is trusted, it should be checked, if this certificate is listed in the CRL [40].

In figure 2.5c, a public key exchange with certified keys is sketched: Alice and Bob do not know each other, but both know Clara, which acts as a CA/RA in this example. Both, Alice and Bob got already the public key of Clara via a secure channel and trust the certificates issued by her. Furthermore, Alice and Bob can distinguish between certificates from Clara and certificates issued by other CAs, as Clara uses her private key to generate the certificates for Alice and Bob (see section 2.3.3.1). This is denoted in figure 2.5c by Crt_C, which means that the certificate is issued by Clara and can be checked with the public key C of Clara. Before Alice and Bob start the key exchange process, they ask Clara, to issue a certificate for each of their public keys. When Bob receives the certified public key of Alice, he can be sure that the key is from Alice without knowing Alice before. Therefore, Mallory has no chance to pretend that her keys are the public keys from Alice or Bob, because Clara would not issue certificates for Alice or Bob to Mallorys keys, as Clara verifies the identities of the certificate requesting entities.

2.3.4.2 Trust models

In section 2.3.4.1, the concept of PKIs is introduced and explained with an example in which Alice and Bob trust Clara which acts as a CA/RA. This trust-dependency to the CA leads to another challenge: How can the trust be established in a generic fashion? In large networks like the Internet, where many devices communicate with each other and it is not practicable, that each entity directly exchange their public keys via a secure channel. In order to address this challenge, there exist some several trust models:

Direct trust is the simplest trust model. It comes into play, when two entities can identify each other [45]. All other trust models rely on direct trust [53]. For

example, if Alice gets Bobs public key face to face and therefore trusts his key or Bob and Alice trust Clara in her role as CA (see section 2.3.4.1).

Third entity trust is also a simple trust model, in which a trust path between two entities is established, by a trusted third entity. For example, if Alice and Bob do not know each other but both already direct trust Carol and Carol directly trusts Alice and Bob, the trust path between Alice and Bob can be established via Carol. [45].

Trust lists contain a list of trusted public keys, root certificates or self-signed certificates, which are unconditionally trusted. Trust lists are used, to support certificates of more than one CA. The advantage of this approach is that many CAs can be trusted at the same time and none of the CAs must be related to each other. Modern operating systems (OSs) and Web browsers are already shipped with a preinstalled trust list [53], [52].

Hierarchical trust is a very common and heavily used trust model. It is based on an unconditional trust to a root entity, which itself is related to several sub-entities. As a particular entity trusts the root entity, the trust path to the sub-entities can be established. For example, if a PKI consists of several CAs (a root CA and an arbitrary number of intermediate CAs), hierarchical trust can be used to establish a trust path to all intermediate CAs once the root CA is trusted. Therefore, if certificates are issued by intermediate CAs, they can be backtracked to a root CA [40].

Cross trust is a model, which is used between the PKIs of different organizations. The idea is that two CAs mutually sign their certificates to establish two-way trust of all certificates, belonging to one of both CAs. Thus, both organizations can retain their own root CAs, but can also trust the certificates issued by the other CA [40].

Web of trust is a special case of a trust model, which doesn't necessarily require PKIs or CAs. In contrast to hierarchical trust, it is a decentralized trust model, where the entities trust a public key, if they have directly obtained it from the owner or if enough trusted entities trust the public key. In this model, each user is responsible for certificate exchange und signing [45], [53].

2.3.4.3 public key infrastructure in practice

As mentioned in section 2.3.4, PKIs exist, to authenticate the public key owner [51]. PKIs are widely used in the Internet Protocol Transport Layer Security (TLS). It is part of Hypertext Transfer Protocol Secure (HTTPS) connections that are established between Web browsers and Web servers [53]. As mentioned in section 2.3.4.2, the root of trust is provided by a trust list for root CAs that is shipped by each OS and each Web browser [52]. However, as stated in section 2.3.4.2, not only the root CAs are able to issue valid certificates, there are additional models of trust that can be used. Besides

the trust lists, also hierarchical trust and cross trust can be used. In this case, the PKI is called hybrid PKI, because it uses arbitrary combinations of the trust models [54].

2.4 Cryptographic communication protocols

In section 2.1.3 and section 2.1.4 several different communication protocols are introduced. Not only these communication protocols are different, also the cryptographic protection possibilities are very protocol specific. Some protocols already provide cryptographic functionality out of the box. This is the case for many wireless protocols such as 802.11 (Wi-Fi), 802.15.4 (WPAN) and 802.15.1 (Bluetooth). However, in the domain of BACS the two most common systems KNX and LonWorks also support several security mechanisms. Furthermore, in the field of industrial and process automation PROFIBUS or FF provide amongst others some basic security functions [55]. Other protocols have cryptographic extensions or work on top of already existing communication protocols. Examples for this kind of protocols can be found in the Internet protocol suite (e.g. IPSec, TLS, Datagram TLS (DTLS)).

Although many mentioned protocols already have built-in basic security functionality several extensions and proposals exist to upgrade the security. Because some of them provide only rudimentary security mechanisms (like authentication without encryption) or contain flaws in their security implementations [38], [56]. One example for LonWorks is proposed in [38], where smartcards are integrated in the fieldbus nodes.

If a heterogeneous set of different security mechanisms and protocols is used in a system it is hard to provide a consistent basic security level, due to the different security capabilities each of them provides. Thus, in terms of security it is important to provide a basic level of security which is supported by all used communication protocols. If a system doesn't provide a consistent basic security level, potential attackers may combine the weaknesses of several protocols to achieve a weaker total level of security in the system. In the example sketched in figure 2.6 a system uses Protocol A (with message authentication but no encryption facilities) and Protocol B (with encryption but no message authentication facilities) side by side. Furthermore, a gateway is used which translates passing packets between both protocols. On one side attackers may be able to inject encrypted (e.g. by replaying packets) packets at Protocol B because it doesn't use message authentication. On the other side attackers may be able to decrypt the messages by sniffing on the Protocol A side in which no encryption can be used. Therefore, if the packets pass the gateway the security features of both protocols can simply be bypassed.

2.4.1 Internet protocol suite

Thus, from a security perspective CPS intercommunication would benefit from a communication protocol that are used in the whole CPS: The continuing emergence of Industry 4.0 and IoT led to an increasing adoption of the Internet protocol suite for CPS interconnection. The Internet protocol suite consists of four layers. They are defined

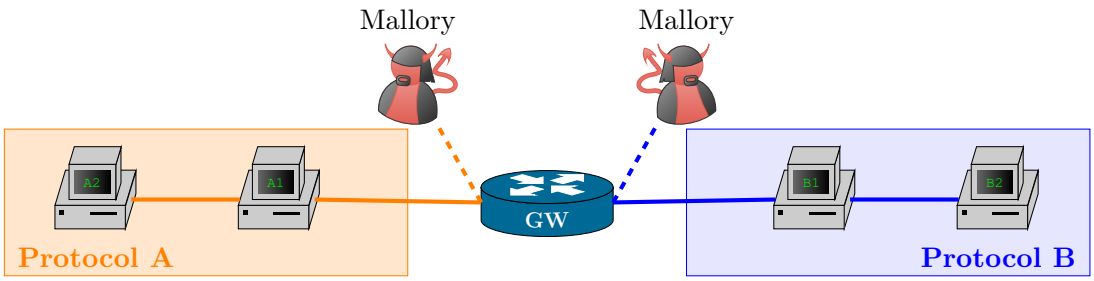


Figure 2.6: A System of different devices that communicate via two different protocols. In order to translate the messages from Protocol A to Protocol B (and vice versa) a Gateway with two attack points is used.

by [RFC1122] as the following: 1. link layer, 2. Internet layer, 3. transport layer and 4. application layer. In figure 2.7 these layers are listed inside the Internet protocol stack.

The OSI model (defined in [ISO7498-1]) is similar and also widespread used as a network communication model but it consists in contrast to the Internet protocol suite of seven layers: 1. the physical layer, 2. the data link layer, 3. the network layer, 4. the transport layer, 5. the session layer, 6. the presentation layer and 7. the application layer . The physical layer (OSI layer 1) is responsible for transmitting the communication bits. The data link layer (OSI layer 2) handles the packet transfer between the communicating endpoints. The network layer (OSI layer 3) is responsible for package routing and host addressing. The transport layer (OSI layer 4) is responsible for data transfer between communicating nodes. All layers above OSI layer 4 are application specific and therefore not considered furthermore [57].

As both models describe different levels of network communication, the OSI layers can be compared to the Internet protocol suite layers [58]:

- The link layer corresponds to the OSI physical layer and the data link layer,
- The Internet layer corresponds to the OSI network layer.
- The transport layer corresponds to the OSI transport layer.
- The application layer corresponds to the OSI session layer, presentation layer and application layer.

Because the Internet protocol suite was defined with the IP-based protocols in mind (e.g. UDP/IP, TCP/IP, IP or Internet Control Message Protocol (ICMP)), it is used henceforth [58], [RFC1122]. Thus, the OSI model is omitted in favor of the Internet protocol suite. Furthermore, due to the huge adoption of the Internet protocol suite several different security approaches exist. Each solution can be applied at different

protocol levels. In the remaining part of this section, some common security approaches based on one of the four Internet protocol suite layers are discussed. However, this list of approaches is not exhaustive.

2.4.2 Security layers related to IP

TLS is a very prominent application of hybrid cryptography combined with digital signatures and a PKI. TLS works between the Transmission Control Protocol (TCP) layer and the application layer of a TCP/IP stack. It encapsulates the application layer payload and applies cryptographic functions to ensure confidentiality, integrity and authenticity. It works on top of a socket connection: Once a secure socket is established between two peers the whole data is protected. TLS is used in HTTPS, which is a secure variant of Hypertext Transfer Protocol (HTTP) [53], [48], [46], [RFC2818].

DTLS is similar to TLS but uses UDP/IP instead of TCP/IP as transport protocol.[4] On one hand, it is very similar to TLS as the development goal was to maximize the code and infrastructure reuse from TLS. On the other hand, it must be designed in a way that the protocol is capable of operation via unreliable links (reordered, replayed or even got completely lost datagrams) [46].

IPSec is strictly speaking no protocol on its own but is more a protocol suite than a single protocol. It contains the components Authentication Header (AH), Encapsulating Security Payload (ESP) and Internet Key Exchange protocol (IKE). As it works parallel to IP, all Internet applications can use secured IPSec connections between two IP peers. During the development of the IP version 4 (IPv4) standard several security aspects were neglected. For IP version 6 (IPv6) attention was payed to security in which the IPSec protocols evolved. Finally, the IPSec protocols were also backported to IPv4. Either AH or ESP can be used at the same time [RFC4301], [46], [47]. ESP can be "used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic flow confidentiality." [RFC2406] whereas AH can be "used to provide connectionless integrity and data origin authentication for IP datagrams (hereafter referred to as just 'authentication'), and to provide protection against replays" [RFC2402].

Kerberos is an authentication and key distribution protocol [46]. It works on the application layer and is used to prove identities in the network, but not integrity or confidentiality (see figure 2.7).

[4] It can also be used also on top of Datagram Congestion Control Protocol (DCCP). DCCP can be seen as User Datagram Protocol (UDP) plus congestion control [46].

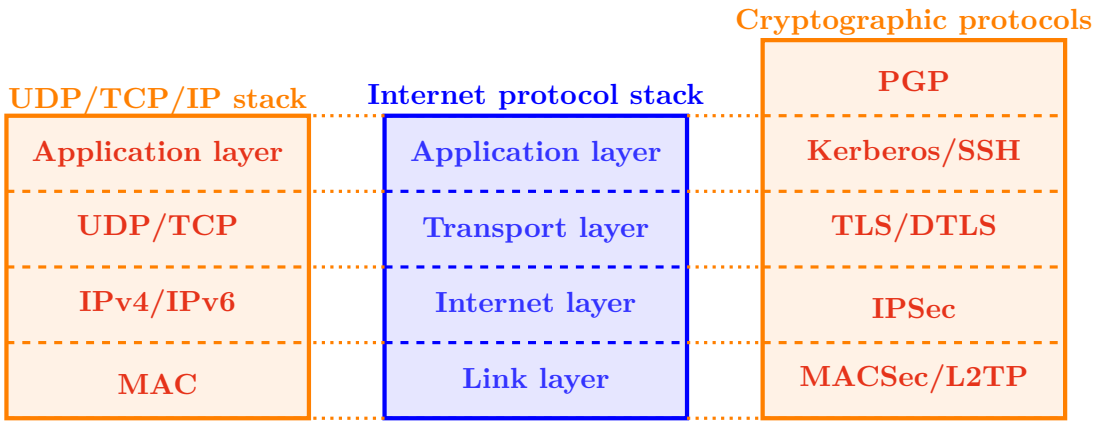


Figure 2.7: Different cryptographic transport protocols for the Internet protocol stack, visualized at which layer they operate [46], [RFC1122].

Secure Shell (SSH) is a protocol which was originally developed as a secure replacement for Telnet[5]. Additionally, it supports tunneling of any applications that can be run remotely via an encrypted tunnel. An example for such an application is File Transfer Protocol (FTP) tunneled over SSH. The SSH File Transfer Protocol (SFTP) is something similar but natively included in SSH. Furthermore, a tunnel variant with Point-to-Point Protocol (PPP) over SSH is possible. This approach is similar to a virtual private network (VPN) [47], [40]. SSH "[...] performs server host authentication, key exchange, encryption, and integrity protection" [RFC4253].

Pretty Good Privacy (PGP) is a protocol suite which is often used to secure emails. In figure 2.7 PGP is above the application layer as it can operate on top of traditional application layer protocols. PGP is an Internet Engineering Task Force (IETF) RFC and is defined in [RFC4880]. PGP provides "confidentiality, key management, authentication, and digital signatures." [RFC4880] In contrast to TLS, PGP uses the web of trust model which is described in section 2.3.4.2 [46], [40].

MAC Security (MACSec) has been published under the name IEEE 802.1AE and provides security extensions for the link layer (Medium Access Control (MAC)). It provides connectionless security with "user data confidentiality, frame data integrity, and data origin authenticity" [IEEE 802].

Layer 2 Tunneling Protocol (L2TP) is an encapsulation protocol which has been created out of Point-to-Point Tunneling Protocol (PPTP) (Microsoft) and Layer 2 Forwarding (L2F) (Cisco). L2TP does not provide authentication or encryption, but it is possible to use IPSec for this purpose on top of L2TP [40], [RFC3193].

[5]Telnet allows to attain access to a local shell via a remote console. However, it has insufficient security mechanisms [40].

Figure 2.7 shows that different cryptographic protocols are working on different layers of the Internet protocol stack. The applications are generally more involved the higher the cryptographic protocol is in the IP stack [46]. For example, if the OS establishes a secure IPSec channel security is transparent to the application.

The drawback of low-level security protocols is their possible inference with already existing network infrastructure [46]. In this case higher layer protocols like TLS or SSH can be used as both work above the UDP/TCP layer. A special case is PGP which works on top of the application layer and can thus be used to enhance the security of unsecure application layer protocols like electronic mails.

Another issue is the protocol overhead for security at higher levels of the IP stack. If the encryption is enabled on lower levels the protocol specific information (like header information) of the upper layers is protected. However, if the security mechanisms are applied on higher levels protocol specific information can easily be extracted. If PGP is used for email protection, the sender and the receiver of the emails are stored in plain text.

2.5 Hardware security

2.5.1 Important Hardware components from a security perspective

In 1971, Intel released the first microprocessor and five years later Intel introduced the first microcontrollers that combine Central Processing Unit (CPU), instruction memory (ROM), data memory (Random Access Memory (RAM)) and input-output-functionality. This has become the basis for the following embedded systems [59]. Even today's microcontrollers and CPUs are related to these first devices available and they consist even nowadays of the same basic components that were used decades ago.

From a security perspective, the data stored on these devices may be sensitive. Therefore, a short introduction into important memory parts of embedded systems is introduced here:

RAM defines in general volatile memory (even though also non-volatile RAM is already available) which has been used to read or write data during runtime [59], [60].

Dynamic RAM (DRAM) holds the data in small capacitors, but with the drawback that the data must be refreshed every few milliseconds. This requires special refresh controller hardware but is widely used in traditional PCs [60].

Static RAM (SRAM) memory chips are less dense and more expensive than DRAM chips but hold the data as long as power is provided and allow higher transfer rates. Microcontrollers mostly use SRAM, because they are also less temperature sensitive and require in general less memory than PCs [60].

ROM defines a group of read only, non-volatile memory which has been used in different forms since the early days of computers, microcontrollers and other embedded devices. Its main application domain is the program memory [59], [60].

Mask ROM (MROM) defines a special case of ROMs, in which the content of the memory is stored by the semiconductor manufacturers. Compared to other non-volatile memory, MROMs can be produced very dense but require large investment costs. Additionally, the data on the memory can't be changed [60].

Programmable ROMs (PROMs) are also special cases of ROMs, but with the difference, that the system developers can program the memory. It is important to mention, that PROMs can be programmed only once [60].

Erasable PROMs (EPROMs) are PROMs that can be erased and reprogrammed. Some EPROMs can be deleted by a very intensive UV light source. This operation can take up to 30 minutes [59].

Electrically EPROM (EEPROM) are more modern EPROMs that can be reprogrammed electronically. The reprogramming step is possible up to hundreds of thousand cycles. Compared to UV-EPROMs, the programming can be accomplished much faster [59].

Flash are EEPROMs of a more modern generation that are widely used and can be programmed very fast [59], [61].

As mentioned in this section, the program is mainly stored in memory of the ROMs group. Thus, not only the communication between the devices needs special security considerations, also the software which is stored in the memory and runs on the device needs special care and should be protected. Also running unintended software on the device should be prohibited. Additionally, it may be required that copying or modifying the application should be prevented. In the last decades several approaches to enhance the security for the program memory were developed: Those, which rely only on the hardware hardening are explained in section 2.5.2. However, there also exist approaches that involve hardware as well as software. These approaches are explained in section 2.5.3.

2.5.2 Hardware hardening approaches

More than forty years ago, the main components of an embedded system (CPU, RAM or ROM) were only available separately. Components that are today available in single System on Chips (SoCs) were soldered together on the boards and could be replaced easily. This was also the time, when ROMs were mostly made from mask-ROMs. At this time, there were no protection methods against ROM chip cloning or even chip replacing. For attackers, it was simple, to replace the ROM chips with other ROMs or EPROMs and execute unintended code [59].

In the late seventies, microcontrollers with internal memory and security protection against unauthorized access to the internal memory contents became popular. Even though the first microcontrollers required separate non-volatile memory chips, the next generation already contained EEPROM storage inside the same plastic package. Getting access to the data on these microcontroller packets required already special equipment with microprobes or special bonding technologies. However, also exploiting a software bug to attain access to the data became feasible [59].

Other security mechanisms against data access were hardware fuses that disabled data access (e.g. disabling the read-back function of the programmers). However, using parts of the main memory to control the data access has been a feasible way too. It has been realized by latching the information at a specific address at power-up or using passwords to grant access to the memory. Additionally, several EEPROM data memory bus encryption methods were implemented as well as a top metal sensor mesh, where all sensors in this mesh were continuously monitored and if an abnormal event was triggered, the EEPROM memory gets deleted. Both approaches were implemented as a protection against micro-probing attacks [59].

2.5.3 Hybrid (software and hardware) hardening approaches

At the beginning of the nineties, a slightly different approach has been taken with software based cryptographic security implementations that rely on hardware supplements.

2.5.3.1 HSM and TPM

One approach for these supplements is called Hardware Security Module (HSM). These modules have often been used in the financial sector. One example is the crypto processor IBM 4758 or its successors (4764, 4765 or 4767) [62].

Like HSMs work Trusted Platform Modules (TPMs). They have been introduced by the organization Trusted Computing Group (TCG) which itself has been founded as an industrial initiative to standardize trusted computing functionality and requirements. TPMs are special hardware security modules that have been designed for mass market devices and are therefore very cost-effective. Because these tamper-evident modules can be connected directly via common embedded interfaces like Inter-Integrated Circuit (I^2C) or Serial Peripheral Interface bus (SPI) they are predestinated for many embedded and IoT platforms [63], [64], [62], [65], [66].

Both, HSMs and TPMs have been developed with the goal of creating a Trusted Platform (TP). Even the term TP has been defined by the TCG as a system with at least the following three features:

Protected Capabilities involve shielded locations that can be accessed by special commands, storage of integrity measurements, key storage, key management, Random Number Generation (RNG) and data sealing [67], [FR5].

Attestation guarantees accuracy of information and provides information about the identity and the state of the TP [67], [68], [FR5].

Integrity Measurement, Logging and Reporting: Integrity Measurement mostly checks integrity of components by digests or hashes. These integrity measurements are stored inside the shielded locations from where they may be logged and attested [67], [68], [FR5].

Even though HSMs and TPMs have similar goals, they are used for different applications. HSMs are typically used in applications that require high performance cryptography or storage for many keys, but also if the module should be upgradeable with new protocols or applications like in financial applications [62]. TPMs in contrast were designed amongst others for devices like Notebooks or PCs. Additionally, TPMs have been a requirement for each Microsoft Windows Vista capable device [69] and are still a requirement in Windows 10 if BitLocker or device encryption is used[6].

2.5.3.2 TEE

However, TPMs or HSMs are not the only approaches that extend hybrid hardening approaches. Another concept is Trusted Execution Environment (TEE) that is also a special hardware feature which support isolated execution environments for parts of the running software. To better distinguish between the isolated and the normal execution environment, the terms TEE and Rich Execution Environment (REE) are used. The term TEE denotes all hard- and software components that are required to serve secure storage and an isolated execution environment. In contrast to TEE-applications, REE-applications are executed outside of the TEE (like classic applications on a processor without TEE). However, there exist defined interfaces in which REE-applications can communicate with the applications running inside a TEE. The goal when writing applications that use TEE facilities is to write REE-applications that require that sensitive operations are run inside the TEE and the corresponding data never gets out of the TEE [70], [71], [72], [73], [FR6].

Closely connected to the TEE specification is another prominent organization: GlobalPlatform (GP) which focuses on the security of smart mobile and connected devices and creates specifications for these devices. Part of this specifications are the TEE specifications that are in general architecture independent of the underlying hardware and provide a generalized set of Application Programming Interfaces (APIs) to implement trusted applications. Some common examples for TEEs are ARM TrustZone, Intel Trusted Execution Technology (TXT)/Software Guard Extensions (SGX) or AMD Secure Processor. There also exist special TEE operating systems like Open Portable - TEE (OP-TEE), Trustonic or SecuriTEE that are typically tiny OSs running parallel to the REE-OS. Due to the small code base and limited communication abilities of TEE-OSs, the attack surface is limited. OP-TEE has been ported for several ARM-platforms

[6] https://www.microsoft.com/en-us/windows/windows-10-specifications

with enabled ARM TrustZone. It is an example which fulfills the GP specifications for TEE [72], [73], [FR6].

The first mobile phones with TEEs appeared more than ten years ago and since several years, almost every sold smartphone and tablet supports TEEs. Possible applications for TEEs are also software based TPMs that can be run on ARM TrustZone devices, even if they do not provide any TPM features. This is possible, because TEEs provide Root of Trust (RoT) facilities and can protect secret keys (RoT is explained in section 2.5.3.3). As all devices with a TPM are in general capable of TP features, this applies also to devices that support a TEE, because TPMs can be run inside the TEE [71], [73], [74].

2.5.3.3 Trusted boot

As mentioned in section 2.5.3.1 and section 2.5.3.2, the adoption of TPMs (in particular on PCs or Notebooks) and TEEs (in particular on mobile devices) is currently quite high. Therefore, TP is currently not only a concept which exists in theory it can already be widely used. The trusted boot concept is the first step on the way to a TP [75]. These mechanisms involve boot integrity checks that verify the boot sequence and try to detect unauthorized modifications of the instructions involved at the boot steps. They can't prevent attacks to the TP, but they should prohibit that the TP executes unintended code during startup [76].

This involves two of the three (see section 2.5.3.1) features of a TP: 1) Measurement and 2) attestation. These features are the important steps during the boot process [75]. In this context, "measurement" means that a hash value for each particular component is computed, while "attestation" denotes that this hash-value is verified against accepted authentic values. If the attestation step succeeds, the component can be trusted [75], [77], [78]. If one of these measurements differs from the accepted values, the component can't be trusted anymore, and the boot process must be canceled [65]. Because the validation of the first steps during the boot processes must happen very early after the system reset, (well before the bootloader or even the basic input/output system (BIOS)/Unified EFI (UEFI) is started) parts of these mechanisms must be implemented in hardware [79]. As it is very hard to detect failures in the components that are invoked early at the boot process, it is essential that these elements are trusted unconditionally. These unconditionally trusted components are therefore also called RoTs and they are key elements of each TP [FR5].

The TCG defines the RoTs for a TP as three components: 1) Root of Trust for Measurement (RTM), 2) Root of Trust for Storage (RTS) and 3) Root of Trust for Reporting (RTR) [FR5].

RTM is a RoT which is the first part in the transitive trust chain. Transitive trust means in this context that a parent element (i.e. the RoT) has the ability, to generate a unique and trustworthy id/measurement (i.e. a hash value) of an underlying element (i.e. software). The goal of this component is the ability of doing

reliable integrity measurements. Therefore, it is called Root of Trust for Measurement. It applies the required integrity measurements on the components and sends the results to the RTS. It is important that it runs on tamper proof hardware. In a particular device, a RTM is typically implemented by a Core Root of Trust for Measurement (CRTM). This CRTM contains the instructions that ensure, that the code of the next booting stage can be trusted. Therefore, it contains the first instructions that are executed during system boot or reset. On a PC, it would first measure the BIOS/UEFI and if it is trustworthy, it would then pass control to it. Therefore, the CRTM is the unconditional RoT [80], [67], [75], [81], [77], [78], [82], [FR5].

RTS and RTR both can be designed generically for many platforms, in contrast to RTMs or CRTMs that are distinct for specific platforms. Therefore, it is very common that the RTS- and RTR-capabilities are combined into a TPM which is itself usable from different platforms. The main task of the RTS component is to store the integrity measurement results of the RTM in a confidential and integrity protected way. Furthermore, the RTR is an entity which reports the integrity measurements to the platform configuration [83], [67], [68], [81].

Even though TPMs or TEEs can be used to provide trusted boot, they are not the only ways to provide this feature. High Availability Boot (HAB) is another example which has been created by NXP Semiconductors and is available on some of their i.MX-Platforms. It provides boot integrity checking without a TEE or a TPM. In the HAB case, the boot ROM acts as a CRTM [65], [FR7]. Even though, HAB doesn't provide all features TPMs or TEEs have, they are enough to provide boot integrity checks.

Furthermore, it must be mentioned that TPMs, TEEs or HAB are not solely used to provide boot integrity checks, they can also be used as a part of the key derivation process which is used to generate the decryption keys for encrypted program memory. This enables encrypted storage or even full disk encryption facilities too. As long as the decryption key remains unique for each device, this form of key derivation prevents off-box attacks, because the TPM, the TEE or the HAB feature of the device is needed, to obtain the decryption key [81], [40], [84], [85].

Thus, not only the execution of unintended code can be prohibited, also the code which is executed at startup and during the whole runtime can be stored encrypted in the program memory (e.g. EEPROM or flash), as it can be automatically decrypted during startup. In order to combine the terms TPM, TEE, HAB and HSM for the task of providing a trusted boot mechanism, the term Trusted Boot Module (TBM) is used in the following sections. Therefore, the term TBM stands as a synonym for hardware modules that can be used to ensure boot integrity as well as well as confidentiality.

Concluding this section, it is easy to see that combined software and hardware security mechanisms are very efficient in prevention of arbitrary code execution. Additionally, full device encryption provides a modern and effective method against code extraction.

Both goals exist since decades (see section 2.5.2) but in contrast to hardware hardening approaches that were used earlier, the hybrid approaches with hardware (in the form of TBMs) and software are currently state of the art. Furthermore, these hardware security mechanisms may be supplemented by additional hardware based and security related mechanisms like instruction set extensions (e.g. the AES New Instructions (AES-NI) that have been proposed by Intel [FR8]) or Hardware Random Number Generators (HRNGs).

2.5.4 2FA

Another domain in which hardware security mechanisms are used is the domain of user authentication. In particular, for safety critical systems it is extremely important to keep the credentials used for authentication on devices or services secret and use passwords that are hard to guess [86].

A very popular mechanism to enhance the security of the authentication procedure is the so called two-factor authentication (2FA) which introduces beside the password an additional second factor. In general, there exist two common approaches of 2FA:

One-time passwords (OTPs) that are either generated by the server or the client. Client generated OTPs contain an identifier that enables the server to distinguish between several different clients. Some examples are the HMAC-Based One-time password (HOTP) algorithm published in [RFC4226] or its extension Time-Based One-time password (TOTP) in [RFC6238]. In contrast, server generated OTPs are transmitted to the client and must be sent back to the server. One example for this type is the OTP-SMS approach, where the OTP is sent to the client via an additional SMS communication channel [87].

Challenge-response approaches are slightly different, as the server creates a challenge which is sent to the client. Once the client receives this challenge, it creates a response by using cryptographic functions and private key(s). This response is than sent back to the server. When the server receives the response, it can distinguish easily, if the response was created by the correct private key(s) [87].

This second factor can be either pure software approaches, OTPs via a second channel or dedicated hardware like Smartcards, USB token or near-field communication (NFC) token. The recommended approach is using hardware-based solutions, that have a built-in security chip for this second factor. As OTPs through separate channels like SMS or pure software solutions have several drawbacks. One advantage of these hardware approaches is that the security chip provides a RoT similar to a TBM, as it contains a fixed public and a private key that are created during the device manufacturing process. Additionally, they can act as crypto processors [62], [88], [87].

A common protocol which implements hardware-based 2FA is the Universal Second Factor (U2F)-protocol. It initially has been developed by Google and has been standardized

by the industry consortium Fast Identity Online (FIDO), which itself has been founded 2012 [86]. FIDOs goal is therefore the design of secure authentication protocols. Thus, U2F is an example of a hardware-based challenge-response approach [87].

2.6 Linux startup

The CPS definition in section 2.2 states that a CPS consists of several interconnected devices. In general, each of them is different and thus runs distinct software. Some devices of a CPS may have only limited computation capabilities as they are driven by simple microcontrollers (like lightweight sensors). Thus, they may not be able to execute full-fledged OSs. These devices are not targeted in this section, as they typically run very device- and application-specific software and therefore, the startup process is very device specific. Even though, many devices are not capable of running a full-fledged OS, this doesn't apply to all such devices. Examples for such devices are the control devices that are introduced in section 2.1. For these devices, the Free and Open Source Software (FOSS) Linux kernel is often used. Due to its huge hardware support, the rich user space toolset and the zero royalty fees, Linux is used in many embedded systems and devices in a CPS [89], [90], [91], [92].

Due to the diversity of all devices that are supported by Linux, some preparation work must be done before the Linux kernel can be executed. This part is explained in section 2.6.1. Once the Linux kernel has been invoked, the kernel starts with its initialization (section 2.6.2). The next step is the init-process, that is started by the kernel. This step is explained in section 2.6.3. Finally, the system services running in the user space (section 2.6.4) are started and the system transitions in the operational state. Some possibilities of the Linux startup are shown in figure 2.8.

2.6.1 Initial booting sequence

The main purpose of the initial booting sequence is to set up the components of the device into an operational state. The kernel relies on this. As only the SRAM and the CPU core are directly operational at system start, the initial booting sequence involves the setup of DRAM controllers, NAND-flash controllers, MMC controllers, USB controllers or network interfaces (if network boot is desired) [93], [94].

Since several different possibilities exist, how the initial booting sequence can be implemented, three of them are explained in more detail here: An execute in place (XIP) bootloader, the multi-phase boot sequence with a boot ROM, and a UEFI multi-phase boot sequence. These three approaches are also visualized in figure 2.8.

2.6.1.1 XIP bootloader

If memory with XIP[7] support is used (like NOR-flash), the program memory can be addressed like the SRAM. Therefore, it is possible to map the whole memory to the address space of the processor. Thus, the CPU can execute the instructions at a specific start address without setting up additional memory controllers directly at startup. As this start address is executed after a system reset it is typically called "reset vector". Usually the instruction at this address is a jump command to the area where the bootloader is located. Once the bootloader is invoked, it initializes the DRAM controller, copies the kernel and its dependencies into the DRAM and starts the kernel [94], [93], [FR9].

Therefore, the XIP approach is very simple. In particular, if the bootloader and the kernel can be stored in memory that supports XIP and requires no initialization.

2.6.1.2 Multi phase boot sequence with boot ROM

If the bootloader and the kernel are in memory that is block accessible (like NAND-flash) or if they are received as byte streams from USB or Ethernet, no XIP functionality is provided in general. Thus, the instructions of the bootloader and the kernel can't be executed directly from the CPU. Furthermore, the source in which these instructions are stored must be initialized before it can be used [94], [93], [FR9] (e.g. setting up the NAND-controller or setting up the network interface). Therefore, introduction of several steps before the kernel can be invoked is required:

Boot ROM: The first step is a static code, that is copied into the SRAM and executed at startup immediately after a system reset. As this static code is device specific, it is commonly stored inside the SoC and typically provided by the SoC manufacturer. This static code usually can load small chunks of data from block memory (like NAND-flash) into the SRAM. These chunks may be selected either by the block address or by a specific filename if the source is structured as a file system. Additionally, the boot ROM may be able to setup the network configuration and receive a byte stream containing the instructions of the bootloader [93], [94]. This boot ROM (including the contained instructions) forms the first phase of the boot process.

Bootloader: The bootloader usually initializes the DRAM. If the kernel is stored on block devices or can be fetched via peripheral interfaces (like network or serial interfaces), these must be initialized too. After all these components have been initialized, the bootloader fetches the kernel image and its dependencies from the previously initialized components and copies the content into the DRAM. The last task of the bootloader is to execute the kernel from the DRAM [94].

As the instructions of the bootloader must be copied into the SRAM before they can be executed, the size of the SRAM also limits the size of the bootloader.

[7] With XIP memory it is possible to execute the instructions stored on the memory directly without copying them into the SRAM.

In order to overcome this limitation, the optional concept of Secondary Program Loaders (SPLs) and Ternary Program Loaders (TPLs) has been introduced: A SPL is utilized if the whole bootloader doesn't fit into the SRAM. Therefore, the SPL usually initializes the DRAM, copies the main bootloader (TPL) into the DRAM and executes it from there. As this bootloader split is a common task, some major bootloaders like U-Boot directly offer the possibilities of creating these two parts. This reduces the effort needed to create two bootloaders (SPL and TPL) [94], [93].

2.6.1.3 UEFI multi-phase boot sequence

Devices that use the UEFI firmware standard (like many x86/x86_64 architectures) use the following boot sequence [93], [95], [96]:

Phase 1 The CPU loads the UEFI boot manager firmware into the SRAM. The UEFI boot manager firmware is loaded either from a NOR-flash directly or the code contained in an Extensible Firmware Interface (EFI) on-chip ROM loads it from a serial flash [93], [95].

Phase 2 The UEFI boot manager firmware is executed. This involves the initialization of the DRAM controller as well as loading the UEFI bootloader from the EFI System Partition (ESP), the data storage (like a Flash) or the network. This phase is similar to the SPL explained in section 2.6.1.2 [93], [95], [97].

Phase 3 The previously loaded UEFI bootloader, which can start the Linux kernel, is executed. This phase is similar to the TPL-bootloader in section 2.6.1.2. Some examples are GRUB 2, ELILO, systemd-boot or gummiboot [93], [95], [96], [97].

2.6.1.4 Trusted boot

To provide trusted boot facilities, the initial booting sequence must be able to verify the integrity of the Linux kernel image. Therefore, TBMs that are introduced in section 2.5.3.3 can be used to achieve this task. Thus, the initial booting sequence must be able to use a TBM to provide the trusted boot feature. Due to the different types of TBMs (e.g. HSM, TPM, HAB or TEE) and the different initial booting sequences, trusted boot is highly device specific. In [98], [88] the trusted booting process on platforms with UEFI/BIOS and TPMs is described and in the application note [FR10] the trusted boot process on i.MX-processors with HAB support is explained.

2.6.2 Starting the Linux kernel

Once the bootloader has finished the initialization tasks, it hands over the control to the kernel. The kernel is responsible for initializing the remaining hardware. It also manages the available system resources. Thus, the kernel receives additional information (referred to as additional kernel dependencies in section 2.6.1) like a description of further system

hardware components (e.g. the device tree) from the bootloader. Additionally, the Linux kernel relies on the root file system, where all programs, configurations and data are stored. The root file system can either be a partition on a mass storage device (like a NAND-flash), a network file system or an initramfs that has been copied from the bootloader into the DRAM. Due to this huge number of possibilities, the bootloader informs the kernel where the root file system is stored or whether an initramfs is used [93].

2.6.3 Starting the init process

If the kernel initialization has been finished, the kernel executes the init process as its first regular process with PID 1 in the root file system. The purpose of this init process is the execution of all further processes the system relies on (like system services). Furthermore, the init process outlives all other processes and handles system shutdown, system restart, system service restart and optionally handles runtime events like recognizing new hardware. Several implementations of the init processes exist. Example implementations are "Busybox init", "System V init" and "systemd". These are explained in more detail in [93], [99]. Additionally, these different init processes implementations are visualized in figure 2.8 too.

2.6.4 Starting the system services

The system services (or system daemons) define the functionality of the device. This is the case for CPSs, in which the system typically starts without human interaction and all required services should be started automatically. For Linux systems the init process is responsible for starting or stopping the system services (see section 2.6.3). Thus, in general the init process is also a system service that provides the ability of starting other services. Examples for typical system services are `syslogd`, `getty` or `sshd` [93], [99].

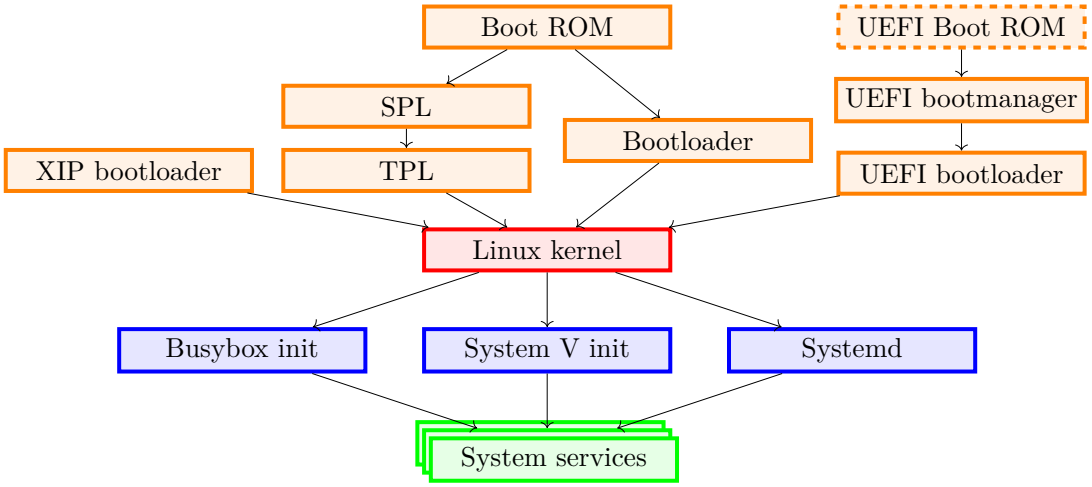


Figure 2.8: Incomplete excerpt of different Linux startup sequences. It contains the distinct methods to start the Linux kernel (orange boxes) and different implementations of the init processes (blue boxes). As the UEFI Boot ROM is not necessarily required for each UEFI implementation its frame is drawn dashed.

2.7 Software updates

As mentioned in section 2.5.2, at the time, when MROMs or PROMs were used, the only way to update the software of embedded devices, was the physical chip replacement [100], [101]. However, since EPROM-, EEPROM- or Flash-memory has been available and widely used, it is no longer required to replace the entire chip. Instead, the chip can easily be reprogrammed. In particular, as EEPROM- and Flash-memory can be reprogrammed electrically and require no UV-light to erase the chip, but also because EPROM chips can usually be reprogrammed only a few hundred times (compared to at least the few thousand cycles EEPROMs provide or at least the hundreds of thousand cycles Flash-memory provides), most memory used in microcontrollers and embedded devices with update functionality is electronically erasable [59].

When embedded devices first used EEPROM- or Flash-memory and required new software updates, a dedicated programming interface was used, which was suited to change the program memory content. With the introduction of CPSs in which devices are connected to other devices, remote software updates have become feasible. In this case, either the software update can be received over an active Internet connection or it can be distributed between the devices in the network. If the device to be updated is a mobile device connected through wireless communication technologies, no physical connections are required to apply software updates. The terms Over-the-Air (OTA) or Firmware OTA (FOTA) are used for wireless remote firmware updates [101], [102].

If software updates are applied on the field (e.g. remote- or OTA-updates), the update routine must provide several features: 1. novelty, 2. integrity, 3. authenticity, 4. com-

patibility and 5. fallback. *Novelty* ensures that the software update mechanism can't be used to downgrade the device. *Integrity* and *authenticity* ensure, that the software update is provided from a trustworthy source, is unmodified and does not contain any malicious code. The term *compatibility* checks that the software update is compatible with the device (in terms of device type as well as boot integrity). *Fallback* means that a proper mechanism is implemented which ensures that once an update fails, the device can go back into a functional state. All these terms are important, because the device should install only compatible updates that have been released for this particular device. Furthermore, if the device has enabled boot integrity checks, the update should also pass the boot integrity test during startup. Therefore, before the update is applied, the update routine must ensure that future boot integrity checks succeed, as incompatible updates and a failing integrity checks may lead to a malfunction device. While boot integrity can be ensured quite easily special care must be taken if the device boots from fully encrypted memory where a unique key per device must be used. In this case, the update must be specifically prepared for each separate device. Additionally, the update routine may fail due to power interruption or that the update itself contains several bugs. In this case, the device must be able to return to the prior working state [102].

Keeping the software up to date is very important for CPS and especially for SCCPS. As software updates often contain bug-fixes that are safety- or security related. However, devices that are driven by simple microcontrollers are often not capable of running a full-fledged OS and thus often rely on very device- and application-specific software (see also section 2.6). A software update mechanism for these devices is generally highly platform dependent (in terms of the installed software and the system startup process) and often individual update mechanism must be developed [90], [93], [89]. Some examples of common approaches are introduced in [103], [104], [105].

Instead of these approaches, the following three approaches are more general but rely on devices capable of running at least a full-fledged OS like Linux.

Symmetric image update relies on two system images, each containing the OS and the root file system with the applications. The bootloader checks a flag which indicates what system image should be used during startup. Once an update is due, the update mechanism installs it to the inactive image. After the update has been done, the bootloader flag is changed to the new updated partition. Future updates will be applied alternating between the two images [93].

Asymmetric image update is an approach which tries to reduce the storage which is required by the symmetric image update. It contains only one full operational system image and a non-operational but functional rescue image. When applying an update, the system must be started in the rescue mode. If the update succeeds, the bootloader selects the operational system image during the next boot up. If the update fails, the rescue mode remains active to apply the update again. The drawback compared to the symmetric image approach is that the recovery

image doesn't contain any operational code and therefore the device remains non-operational after a failed update, until an update succeeds [93].

Atomic file updates rely on redundant copies of the root file system in a single base file system. At boot time, one of the copies is selected (e.g. by using `chroot`). Once one of the copies is selected as the root file system, the others can be changed. Furthermore, it is possible, to save space, because unchanged files between several copies can be linked at file system level [93].

Since these approaches are highly platform dependent, as they rely on the bootloader as well as the OS, these concepts must be implemented individually for each platform. Existing implementations of these concepts for Linux systems are Mender (uses the Symmetric image update), SWUpdate (uses Symmetric and Asymmetric images) and libOSTree (uses Atomic file updates). Additionally, all these implementations can be used in the Yocto Linux build system. All of them are explained in more detail in [93].

CHAPTER 3

System model and threat analysis

In the previous chapter, automation systems, CPSs and SCCPSs are introduced. Additionally, some state-of-the-art security mechanisms for software, hardware and communication protocols are explained there. In this chapter, a short introduction to threat analysis is given. Furthermore, a threat analysis is performed on a fictitious SCCPS.

3.1 Terminology

In order to explain threat analysis, it is required to define the following terms:

Adversary: An adversary is an entity, that utilizes a vulnerability to realize a threat. Sometimes it is also called attacker [106], [107].

Asset: An asset is a valuable item of interest, which an adversary aims at or which must be protected from an incorrect and unauthorized use by the adversary. This item may be abstract (like company's reputation, or the safety of people) or concrete (like the content of a database) [108], [106], [107].

Attack path: An attack path is the condition sequence which is required to achieve an attack goal. Without mitigation it forms a vulnerability [106].

Condition: A condition is an action or weakness present in an attack path [106].

Entry point: An entry point (also access point) is the intersection point between the modeled system and the world, i.e. it provides access to the assets. A system and the corresponding assets can only be accessed and therefore attacked if it has entry points. (It is important to keep in mind, that the term entry point also involves exit points.) [108], [106], [109].

External entity: An external entity is located outside of the scope of the modeled system. But it correlates with the modeled system. This correlation is realized via the entry points of the modeled system [110], [106].

Risk: The risk is a "[...] characterization of the danger of a vulnerability or condition" [106].

Security weakness: Security weakness is an unsatisfactory mitigation of a threat, which typically results in a vulnerability [106].

Threat: A threat exists, if it is possible, that an attack on a specific asset is successful. Thus, a threat may be the goal of adversaries [108], [106].

Threat tree: A threat tree can be used to visualize the attack path of a threat. The root of a threat tree is the actual threat that is visualized [106].

Trust level: A trust level is a specification of an external entity, which describes who has access to an asset using a specific entry point. A trust level contains authentication methods and privileges that the entities of a particular level require [108], [106].

Vulnerability: "A vulnerability is a system property that violates an explicit or implicit security policy" [108]. In particular, a vulnerability defines an attack path that introduces a realized threat, due to insufficient mitigation [106].

In addition to this terminology, the following two acronyms are often used in the context of threat models:

STRIDE: Spoofing, Tampering, Repudiation, Information disclosure, DoS and Elevation of privilege (STRIDE) is a classification of threats which stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service (DoS) and Elevation of Privilege (EoP) [106].

DREAD: Damage potential, Reproducibility, Exploitability, Affected users and Discoverability (DREAD) is a risk ranking method, in terms of a vulnerability or a condition. It stands for Damage potential, Reproducibility, Exploitability, Affected users and Discoverability [106].

3.2 Methodology

As mentioned in section 2.2.2, SCCPS are very complex and tend to become even more complex particularly due to new possibilities in device interconnection and due to new technological and functional advances. This increasing complexity may be a problem, as adding more devices as well as relying on more functionality also increases the attack surface of the system [106], [111]. This includes features like remote administration or

dependencies to cloud services that may lead to new threats. Therefore, the following paragraph states that threat modeling can help to handle this complexity and to mitigate the threats that are implied by this complexity.

In the past, the system developers used firewalls or proxies as mitigation measures, aiming at protecting the system from adversaries. However, these approaches have not helped against architectural or logic flaws in the system. In addition, other vulnerabilities like buffer overflows have not been addressed. In particular, in the software development domain, code reviews are very popular, but because they tend to be slow, boring and the probability that vulnerabilities are missed is very high, a more structural approach which keeps track of the architecture is demanded. Additionally, as security is an important part of reliability, most customers expect or at least prefer secure systems. The gathered data from the sensors of a SCCPS as well as the actuators may be valuable assets for adversaries. Thus, in particular SCCPSs must ensure a certain degree of security to protect these assets. In order to address these requirements, threat modeling is used: Its main goal is evaluation and documentation of possible threats that are related to the system. In particular, for security specifications and later for security testing (like penetration testing), these threat models can be utilized, as the threat model works out many security related properties of the system like the entry points, as well as external dependencies and enumerates possible attack paths. In addition, the security weakness can be extracted from threat models by looking at the unmitigated threats or insufficient mitigation methods compared to total threats. Thus, the risk of a threat should be considered (i.e. is it more or less hazardous that a threat is exploited). This information can be used to identify high-risk areas that are more valuable for adversaries and therefore appropriate candidates for further reworking by system developers (e.g. rework protocol parsers, access control or session management) [112], [113], [106].

Workflow in threat modeling
The typical workflow in threat modeling is based on four questions [107]:

1. How is the system defined?
2. What are possible goals of an adversary?
3. What are the mitigation methods for these goals?
4. Is the threat analysis complete?

The following section will take these four questions as the basis for the methodology.

3.2.1 How is the system defined?

Threat modeling should be started at early steps of the system design workflow. However, it should be carried out at least when all features of the system are clearly defined [113], [106]. This ensures that a definite answer to the question “How is the system defined?”

can be given [107]. For SCCPS this question can be answered by considering the utilized hardware and software and conceiving the connectivity of the system:

Hardware: For an accurate threat model, it is very important to list all devices that are part of the system. For an automation system, this involves not only control devices, PLCs, HMIs, sensors or actuators, but also connected servers, workstations and network coupling devices must be considered. Additionally, external devices that are not part of the system, but connected to the same networks, must be identified.

Software: The software running on the previously identified devices must be considered too. This involves stand-alone applications, user space applications, libraries and the OSs. Additionally, the boot sequence (involving bootloaders as well as boot ROMs) must be considered too. Furthermore, possible software update paths must be identified for each device. As software updates can help to strengthen the system security by fixing known vulnerabilities but they may be used by adversaries as entry points too.

Connectivity: The connectivity of the system describes how all devices of the system are interconnected. In particular, for distributed systems like CPSs (and SCCPSs), that consist of several interconnected control devices it is important to consider the applied networking technologies. This includes the networking technology and the applied protocols as well as the involved security/cryptographic mechanisms. The term "connectivity" involves not only network connectivity, also expansion ports like USB need to be considered as they represent possible entry points too.

3.2.2 What are the possible goals of an adversary?

As threats are goals of an adversary, the first step is identifying the threats against a system.

3.2.2.1 Comparison of threat discovering methods

There exist several different approaches that help to identify threats. Some of them will be introduced in this section:

Brainstorming: One method is brainstorming, in which a group of different persons sit together and try to detect different threats. Depending on the formation of the group and the working time, some important threats may be missed. This is problematic, as adversaries require only one security flaw, to compromise the system [113], [106].

Attack libraries: Attack libraries that contain a huge collection of possible threats can also be used. One example for such a library is the Common Attack Pattern Enumeration and Classification (CAPEC) list with hundreds of attack patterns in

several dozens of groups. Another example is the annually published Open Web Application Security Project (OWASP) top ten list. These attack libraries mainly cover very popular threats. However, as mentioned above, some important threats to the system are missing. This especially applies to very specific threats to the respective system. Additionally, if the list contains hundreds or even thousands of threats, it may be very time consuming to iterate over all possible threats [114], [107].

STRIDE: However, threats can also be detected by using the STRIDE method. As the acronym STRIDE defines the opposites of the security goals mentioned in section 2.3.1, this method is useful for finding security threats [107]:

Spoofing violates **authentication**

Tampering violates **integrity**

Repudiation violates **non-repudiation**

Information disclosure violates **confidentiality**

DoS violates **availability**

EoP violates **authorization**

This STRIDE method simply enumerates the STRIDE categories and ensures, in contrast to the brainstorming approach, that threats from a broad number of categories are taken into account. Even, when applying this method, several threats can remain undetected. Also, it is hard to find an exit criterion. A selective variation of the STRIDE approach named STRIDE-per-element can be used, to overcome some of these problems. This is an extension, in which STRIDE threats are applied to certain elements of the system (e.g. External entities, processes, data flows or data stores) [110], [107].

Data flow approach: An even more structured approach in which all entry points for each use case and access level are enumerated is the data flow approach. This means, that the correlating assets are considered for each entry point. If each entry point is observed, this approach should lead to complete threat models, as all possible attack goals or threats can be enumerated [106]. Therefore, in order to gain a feeling of assets and trust levels from an adversary viewpoint it is important to understand the system and its entry points. In particular, as each system relies on special design decisions and external dependencies. However, also the specific usage scenario plays an important role. Thus, these properties are collected and form an important part of the threat model, as violating them may also lead to vulnerabilities. As threat enumeration may be hard to track for engineers, additional Data Flow Diagrams (DFDs) can be created. These DFD visualize the data flows in the system and can be used to increase the understandability of the data flows. This approach is described in detail in [106]. As the data flow approach is used as the primary threat discovering method in this thesis, the following section summarizes the most important parts:

3.2.2.2 Data flow approach for finding threats

The data flow approach shows where the respective system is prone to security vulnerabilities and follows two principles: 1. An adversary needs a way to interact with the system and 2. assets that are of interest to the adversary must exist [106]. The data flow approach requires to list the trust levels, entry points (incl. exit points) and assets as a first step.

Trust levels: The first requirements that are considered are the trust levels. They are access groups that act as placeholders for a set of permissions and can be given to the entities which interact with the system (user groups). The list of trust levels controls who is permitted to access an entry point or a protected resource [106], [108]. In table 3.1, the trust levels of control devices that are part of a SCCPS can be seen.

Entry points: As already stated in section 3.2.2.1 entry points play an important role for the data flow approach, as they define the interfaces for control- and data flow to the system. However, they may be considered as the attack points to the system by adversaries [106], [108]. For threat modelers it is therefore very important to take all forms of entry points into account. For a software system some entry point examples are (shared) libraries or APIs, files residing on the file system, configurations, environment variables, further input data like standard I/O, data transferred between other systems or network services like Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP) or Network Time Protocol (NTP). Security critical actions or data transformations must be determined for each entry point. If all entry points were considered, a complete threat model can be created [106]. It is important to keep in mind that the term "entry point" also involves exit points, as they may also handle sensitive data [106], [108]. A very common example for exit points are backups. Due to their close relation to entry points, exit points may be combined with entry points in the same tables. In table 3.2, the entry points of control devices that are part of a SCCPS can be seen.

Assets: Assets are the resources, for which the system must ensure that they are used only in authorized and intended ways [106], [108]. In principle, assets are the cause for attacking a system because they are vital to the system, to the system owners as well as to the adversaries. Sometimes, assets are not physically tangible, because the safety of people or the manufacturers reputation are also part of the systems assets. Therefore, it is important to determine the risk of a threat or vulnerability by considering the possible damage adversaries would cause [106]. In table 3.3, the assets of control devices that are part of a SCCPS can be seen.

3.2.2.3 Collecting background information

Another important part of identifying goals of adversaries is defining usage scenarios, external dependencies, system assumptions and security notes [106]. All these topics can be summarized to collecting background information:

Usage scenario: A usage scenario defines the intended way to use the system. This includes security related design decisions that the system withstands and decisions that have no mitigations [106]. Additionally, it is also important to consider the trust levels identified in section 3.2.2.2, as the usage scenario in general differs depending on the trust level [115]. A usage scenario for control devices is shown in section 3.4.4.

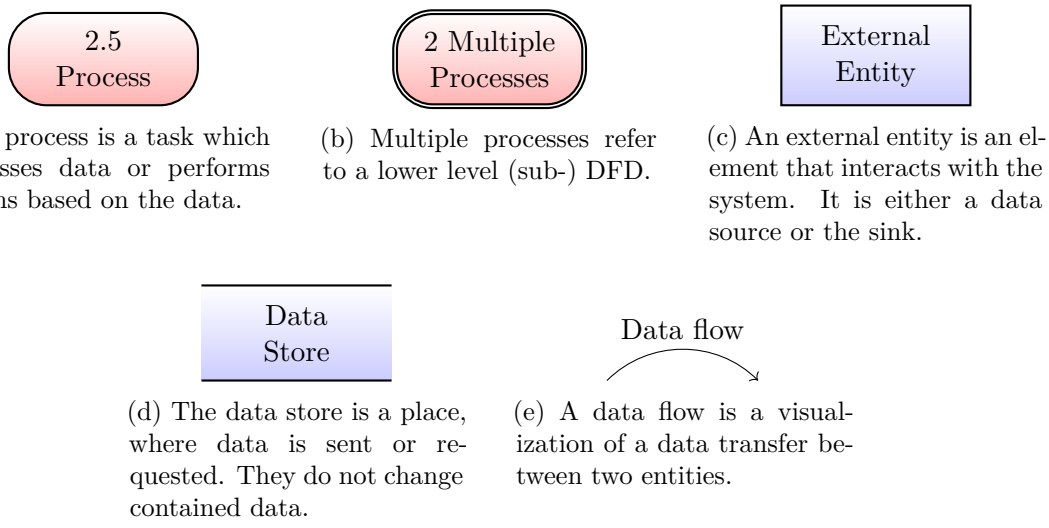
External dependencies: External dependencies are all such requirements that are not in the control of the system developers and that are located outside of the system. It is hard or even impossible to mitigate threats from these external dependencies [106]. Software systems particularly rely on external dependencies like the OS, databases or other services like web servers [116]. A list of external dependencies for control devices can be seen in section 3.4.5.

Implementation assumptions: Implementation assumptions are used very early during the development cycle. They include all assumptions that are not yet implemented, but should be integrated into the final product, as they are a requirement to keep the system secure [106]. Additionally, this list of implementation assumptions is also important if the system must be modified in the future, as these modifications may have implications for these assumptions [116]. A list of implementation assumptions for control devices is described in section 3.4.6.

Security notes: Security notes provide security relevant information to the users of the system (warnings or special guarantees). Thus, this information is required to build and operate the modelled system. Therefore, these security notes may be added to the product documentation [106]. Additionally, security related trade-offs that were made during the development need to be listed in the security notes (e.g. regarding backward compatibility or specific business reasons) [106]. The security notes for control devices are listed in section 3.4.7.

In order to gain a better overview of the data flows in a system, DFDs, Unified Modeling Language (UML) or flow charts can also be used. All of these provide a visualization of the data and the system processes. One of the benefits of DFDs is that they provide different model hierarchies and system views. In particular, as multiple DFDs can be used to describe the data flows of a system, each with different detail levels or even different system parts. As an example, an overall context diagram and several lower level detail diagrams can be created [106].

There are different shapes used in DFDs. They are shown and explained in figure 3.1.



(a) A process is a task which processes data or performs actions based on the data.

(b) Multiple processes refer to a lower level (sub-) DFD.

(c) An external entity is an element that interacts with the system. It is either a data source or the sink.

(d) The data store is a place, where data is sent or requested. They do not change contained data.

(e) A data flow is a visualization of a data transfer between two entities.

Figure 3.1: The shapes of a DFD [106].

DFDs are a good starting point for threat modeling, as they show the processes that interact with data and because adversaries can't attack the system software without supplying data [106]. In figure 3.4, the DFD of control devices of an example SCCPS can be seen.

3.2.2.4 Discovering threats from previously gathered information

By using this previously collected information (trust levels, entry points, assets, external dependencies, implementation assumptions, security notes and a usage scenario) threats against the system can be identified. In particular, enumerating the gathered assets and identifying possible attack goals is a good method to find threats [106]. Specifically, because a threat can't exist without a target asset [106]. In [106], Swiderski and Snyder recommend the following questions for each asset, to extract threats out of the assets:

- Is it possible for adversaries to change the system control flow?
- Are adversaries able to retrieve restricted information?
- Are data manipulations by adversaries possible?
- How likely is it that a system failure is caused?
- Can additional user rights be gained?
- Are spoofing methods possible in order to obtain access to assets on behalf of other users?
- Are there ways to access assets without any access control checks?

- Is it possible to obtain information about past accesses to the asset?

All found threats can be classified through the six STRIDE categories. It is even possible, that a particular threat can be grouped into several STRIDE categories at the same time. Furthermore, threats can be prioritized by this classification. In order to link the threats together with the collected prior information, each threat should refer to the corresponding entry points, assets and additional background information [106].

3.2.2.5 Analyzing threats with threat trees

The threats found in the last step (section 3.2.2.4) must be analyzed to ensure that the modelled system is mitigated against these threats. For this task, threat trees[1] can be used, as they visualize the conditions that have led to a threat on an attack path. Thus, a threat tree contains a root node (the threat) and at least one child condition. Furthermore, each child can itself have several child conditions. In order to simplify complex threat trees, it is also possible to collapse them and represent leaves or subtrees as a combined condition in a new tree. Once there exists a path without mitigation from a leaf condition to the root threat, adversaries may be able to affect the threat [118], [106]. An example threat tree can be seen in figure 3.23.

To characterize the risk of a vulnerability, the DREAD method can be used. As already mentioned, DREAD is an acronym of five categories that are used to express security risk in terms of numeric values [118], [106]. Swiderski and Snyder suggest using a limited range of risk classification (e.g. a three-level range with 1. Edge case 2. Common case 3. Default case [106]).

3.2.3 What are the mitigation methods for these threats?

As each of the identified threats has to be addressed individually, the classification from section 3.2.2.5 can be used to prioritize important threats. For each threat one of the following approaches should be applied [107]:

Mitigation: Threat mitigation makes it harder for adversaries to utilize the threat [107]. This can be achieved in particular with implementation changes like using cryptography or a redesign of the system.

Elimination: Threat elimination can be realized by reducing system functionality. If the threat doesn't exist anymore, because the feature which lead to the threat has been removed, adversaries are no longer able to utilize the threat [111], [107]. Even though this approach seems promising, it is often not feasible, as features are implemented for particular purposes.

[1] Sometimes, threat trees are also called attack trees [117].

Transferring: Transferring the risks to customers, OS or other products is also a possibility for threat modelers [107]. In particular, for cases in which threat elimination is not possible.

Accepting: Accepting risks is another possibility [107]. In particular, if it is very unlikely that a threat is utilized by adversaries.

Thus, the first approach which should be considered by system designers is threat mitigation, as it doesn't rely on feature reduction, risk accepting or risk transfer.

3.2.4 Finalize the threat analysis

As soon as the four steps stated before have been taken, the system should be compared with the threat model again. This step is very important to make sure that no threats are missing. In particular, as new mitigation methods can lead to new threats, that should be addressed. Additionally, if the threat model was created before the system development has been finished, the threat model may no longer be accurate anymore. Once this additional investigation results in no new threats, the threat model can be finalized. The same reasoning applies for hazards too. Particularly for SCCPS where a hazard analysis should also be applied. However, a hazard analysis is out of the scope of this thesis.

3.3 System model

In section 3.2.1, it has been stated that the system definition is the starting point for a new threat model. Thus, this section contains the system definition of a fictitious SCCPS and defines the boundaries of the threat model that will be introduced in section 3.4. This fictitious system is defined as a distributed SCCPS that consists of several control devices, in which each of them is connected to several sensors and actuators. Additionally, some control devices may be connected to the Internet and share this Internet connection with other devices in the SCCPS. In figure 2.3, a simple example of this system model is visualized.

3.3.1 Sensors and actuators

The sensors and actuators are essential elements of the SCCPS. Because the sensors gather data for the control devices and the actuators are controlled by the control devices. As mentioned in section 2.1.1, these sensors and actuators form the field level. In this model, each field device corresponds to a particular control device and a control device is responsible for several sensors and actuators at the same time. These connections between the control device and the sensors and actuators can be established in various methods and protocols. As stated in section 2.1.3, even if some field devices are capable of IP-based communication this feature can't be assumed in general. Instead several wireless protocols, wired serial protocols or fieldbus protocols represent common

examples for these connections. Furthermore, it is also possible to rely on a combination of different communication methods between the control devices and the field devices. Therefore, and due to the fact that each of the communication methods generally has different security considerations, it is hard to determine a basic security level for this heterogeneous set of fieldbus protocols (see section 2.4). Due to this, the sensors and actuators are not part of the generic threat model that will be introduced in section 3.4. Nevertheless, these sensors and actuators should be considered in a final threat model once the system model is complete.

3.3.2 Control devices

Due to the shift from a three-level to a two-level automation pyramid, that is explained in section 2.1.1, the sensors and actuators mentioned above form the field level. The control devices, that are introduced in this section, form the upper level above the field level. In order to reduce the complexity of the system model, each of the control devices is of the same type. These control devices are based on a SoC capable of running Linux and appropriate user space tools. Thus, it is required that the CPU is well supported by Linux. However, defining the involved hardware components exactly for this model is not required. Therefore, it is sufficient to determine that volatile memory, as well as non-volatile (persistent) mass memory is required. In particular, because the non-volatile mass memory may be realized by NAND flash that is in general not XIP capable (see section 2.6.1.2), it is desired that the SoC must contain a boot ROM that initializes the booting sequence. Apart from these specifications, each of these control devices contains a small graphic display and several LEDs to visualize control-, sensor- or process data. Furthermore, with a numpad and several additional buttons, it is possible to control the device locally. Additionally, each control device must support a local clock to provide the current date and time to the software services. In order to get a reliable local date and time source, a combination of several external time sources can be used. Examples for external time sources are NTP-servers that work on top of UDP/IP, Global Positioning System (GPS) or Low Frequency (LF)-sender (e.g. DCF77) [119]. Additionally, this time source must be coupled with a buffer battery assisted real time clock that is included in each control device. The buffer battery of the real time clock ensures that even if the system has been restarted, a time synchronization is not necessarily required.

3.3.2.1 System partitions of a control device

The persistent memory and the volatile memory of a control device must store the following customizable components:

Boot image: The boot image contains the bootloader. During startup, the bootloader is invoked from the boot ROM in the SoC. It is developed from an external organization that ensures that it is compatible with the boot ROM and the Linux kernel. The boot image is used read only and it is stored in the persistent memory, throughout the entire boot process.

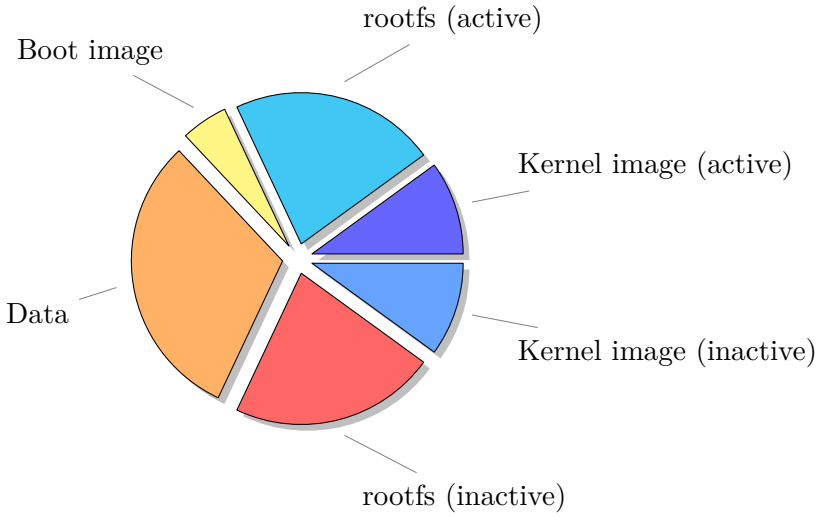


Figure 3.2: Symmetric partition layout: The rootfs and the kernel image both exist twice. While the boot image and the data partition both exist only once.

Kernel image: The kernel image contains the Linux kernel and additional device drivers, as well as the device tree which describes the involved hardware (see section 2.6.2). The kernel image is invoked by the bootloader. Similar to the boot image, it is stored in the persistent memory and used read only during the whole boot process.

Root-File system (rootfs): This system partition contains all user space applications or configurations needed during normal operation. It is mounted by the kernel (read only) and located in the persistent memory.

Data: The data partition is used as a read- and writeable partition, where additional configurations and system log files can be stored persistently during normal operation. The data partition is also part of the persistent memory.

Tempfs: The tempfs partition is a temporary file system, which is read- and writeable as well but it relies on volatile memory instead of the persistent memory.[2]

It is apparent that persistent memory partitions that contain executables are used read only. Thus, it is not possible to modify executable files during normal operation. Even though this restriction exists for security reasons, it limits future software update mechanisms. To overcome this limitation, the symmetric image update method is used (see section 2.7). In this case, the kernel image and the rootfs are stored twice in the persistent memory (see also figure 3.2): The first pair of kernel- and rootfs images contains

[2]Tempfs is mainly used for two reasons: 1. It is usually faster than persistent memory (e.g. EEPROM- or flash) and 2. permanent writes to the volatile memory (e.g. RAM) avoid the risk of wear-out, that is typical for EEPROM or flash memory [93].

the active code that is executed at device startup, while the second pair contains the inactive fallback images. Each time the first pair is active, and the update procedure starts, the inactive second pair can be used read- and writeable. Therefore, the update procedure is able to update the content of the inactive kernel image and the inactive rootfs partition. After both the inactive kernel image and the inactive rootfs have been updated and verified successfully, the bootloader is notified and uses this updated pair during the next system restart as the active pair. It should be taken into account that the bootloader can't be updated by the update mechanism. Thus, if the bootloader requires an update, the whole persistent memory must be replaced by a maintenance user.

As an additional security mechanism, only verified images can be executed. To achieve this image verification, the approach stated in section 2.5.3.3 is used. Thus, the boot ROM verifies the bootloader before it is invoked, and the bootloader verifies the kernel before it is executed. Finally, the kernel verifies the rootfs before it is mounted and the init process is started. Thus, all steps that are involved in the boot process are verified before they are executed. These verification checks make sure that each boot step is signed by an external trusted image signing service. Therefore, the boot ROM contains the public keys that relates to the private keys that are used by the trusted image signing services to sign the images.

3.3.2.2 Control device connectivity

All control devices of the fictitious SCCPS are interconnected in a mesh topology and use IP-based communication. There are no restrictions as to which underlying physical layers must be used, as long as they are capable of IP (either IPv4 or IPv6) traffic. Furthermore, each control device requires a static IP (thus, no DHCP is required). Additionally, some of the control devices have direct access to the Internet and are able to act as a gateway for other control devices without direct Internet access. Thus, this fictitious SCCPS is an example IoT application as all control devices in general are able to use cloud services. The cloud service must be directly accessible from a static IP too. Therefore, and due to the static IPs of each control device, there is no DNS required. In particular, because the DHCP service as well as the DNS service can lead to further threats. Omitting these threats results in a simpler threat model.

Each control device should be able to create a confidential communication channel that provides integrity as well as authenticity. This channel is required for secure M2M-communication between two control devices as well as for communications between control devices and arbitrary cloud services.

To authenticate the communication endpoints in the secure channel, cryptographic certificates are used. These certificates ensure that the communicating devices can authenticate themselves without human interaction, as these certificates are issued from an external CA that is trusted across all devices in the SCCPS. Therefore, no control device requires knowledge of all trusted public keys. Instead, each control device trusts

each public key that is cryptographically signed by a trusted CA (see section 2.3.4). These certificates can be updated via the regular software update mechanism. However, this update contains not only the certificates, the CRL is installed in this manner as well. Each endpoint of the communication channel is itself responsible for validating the certificates and considering the CRL. In addition to the certificate, a unique device ID must be sent. In order to ensure that the device ID, the certificates and the public keys are transmitted fully encrypted, the DH key agreement protocol is performed to derive a common key (see section 2.3.2.4). This key is used as a symmetric encryption key (see section 2.3.2.2). In order to prevent using the same symmetric key too long, PFS is used (see section 2.3.2.5). If the received certificate and the public key are decrypted successfully by the DH key, the authenticity and the permissions of the associate can be checked by the certificate and the device ID. In case of an invalid certificate, the secure communication channel can't be established. If all these checks are successful, the communication channel (secure session) is established and further messages between these endpoints can be sent via this secure channel. In order to protect the endpoints against replay attacks, each message that traverses this channel contains a sequence number that can be used to determine if a message with the same sequence number has already been sent. In order to ensure message integrity and message authentication, at least the encrypted data and the sequence number must be protected by a Keyed-Hash MAC (HMAC). This HMAC is an integrity and authentication check value that is generated out of several constants, the message and a private key. It has to be included in each transmitted message in order to allow receivers to check the message. The HMAC approach is explained in more detail in section 4.2.4.

In contrast to the secure channel that has been explained for the communication between the field devices and the control devices, no security mechanisms are specified. In figure 3.3, an example of this SCCPS in the form of a fire alarm system is depicted. The connected sensors and actuators as well as all components of the PKI are left out, to keep the figure simple. In this example, some FACPs are connected via dedicated connections, while others share the network infrastructure with office-PCs. Furthermore, the connections are established via different technologies like Ethernet, Fiber, Wi-Fi, EIA-485, EIA-232 or an existing office LAN installation. Additionally, some control devices are connected via backbone connections. As already stated, all these connections are capable of IP traffic. In section 5.2.1, an approach can be seen, in which some of these technologies gain the ability, to drive the Internet protocol suite. Figure 3.3 displays that several control devices can be grouped together to form clusters, as it may be useful to logically combine several FACPs. An example cluster contains all FACPs that are located inside a building. Furthermore, all clusters are connected via an IP-based backbone ring network (black). All three clusters use different FACP interconnection facilities: Cluster 1 utilizes only dedicated Ethernet, Fiber, Wi-Fi or EIA-485 connections, Cluster 3 reuses the existing office LAN infrastructure and Cluster 2 combines both: it reuses an existing office LAN infrastructure and relies on dedicated connections between the control devices. Additionally, Cluster 2 and Cluster 3 have direct access to the Internet. This Internet access can be used to send reports to the facility management

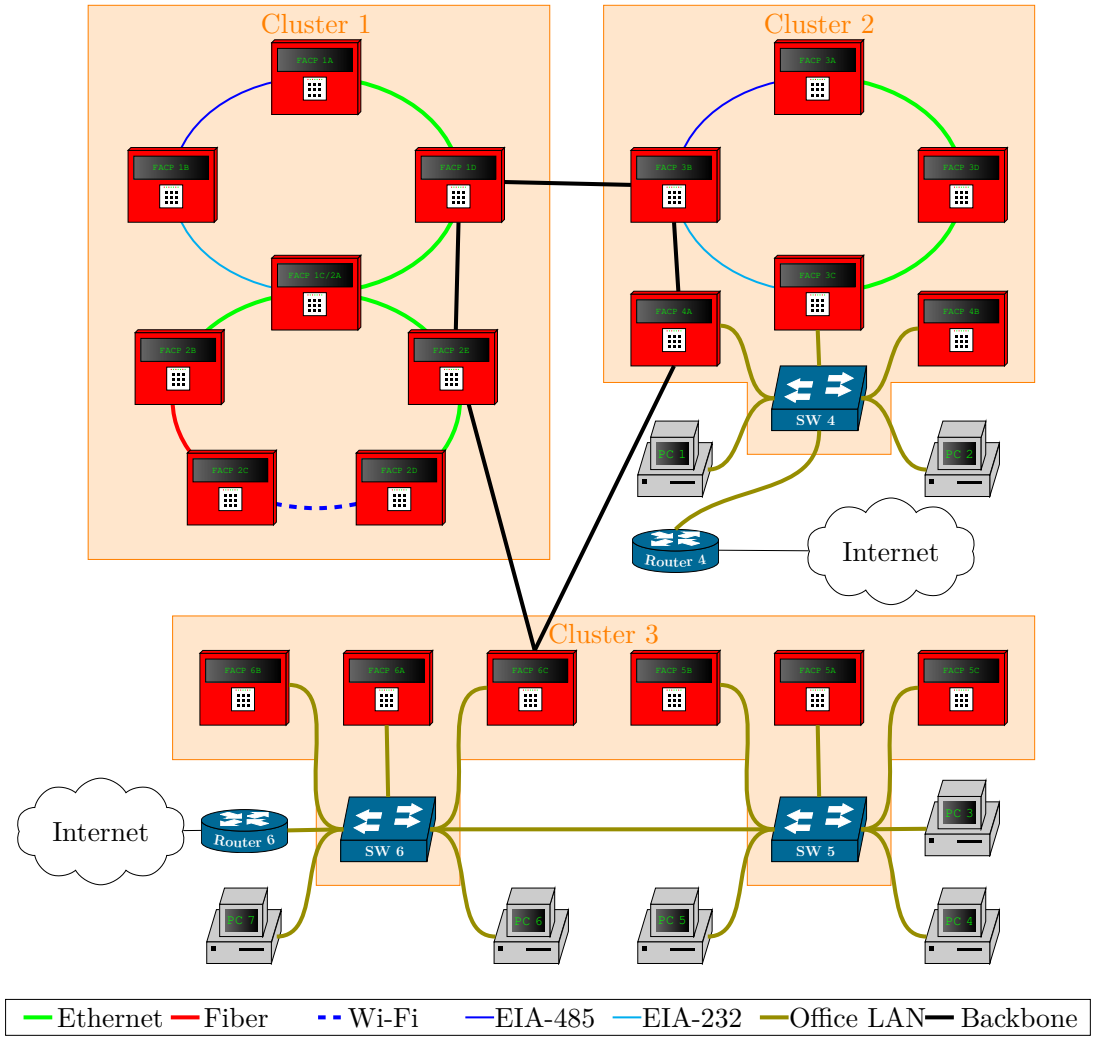


Figure 3.3: Example of a fire alarm system, consisting of three clusters. Each cluster contains several FACPs.

and alarms to the fire department. Additionally, this Internet connection can be shared via a dedicated backbone connection with Cluster 1.

3.3.2.3 Local access to the control device

In order to restrict the local access to specific system-, control- or visualization-functions of a control device, it provides a local user authentication method. This local user authentication relies on a PIN, that must be entered on the keypad to authenticate as a legitimate user on the device. Additionally, also authentication token can be used to authenticate the token owner (see also section 2.5.4). If no local authentication is performed, the local user has only access to a restricted subset of control and visualization

features. This is very important for SCCPS in which it may be necessary, to trigger alarms without authentication. In order to simplify the permissions of each user, several user roles are introduced. Each user role defines a set of permissions on the device and each local user account is related to exactly one user role. In this model, there exist five roles:

1. Maintenance role: Users with this role have the highest privileges, as they can apply software updates, have access to all system internals, can change configurations and may create other users. Typically, these users are only used by certified personnel or employees with service agreements. Therefore, the owner of the system may not have maintenance user rights.
2. Administration role: Users with this role are typically the highest users, the system owner controls. These users have access to some system internals and configurations and can create other users.
3. Authenticated role: This is the standard role for all users that have to interact with the system. They have neither access to system internals nor can create users. They may also have limited data and control access.
4. Emergency role: Users with this role are members of emergency force units which require access to all safety relevant data of the system.
5. Anonymous user role: All users of the system that are not authenticated on the system rely on this role. They have access to emergency stops or trigger alarms but should not have access to system internals.

Additionally, all services on the device itself are started automatically from virtual users with an additional user role. This role is called local system role and users with this role have the permissions to start or stop all device services. Since there are no real people behind these users, they are called virtual users. As the services should be started automatically, users of this role do not require any user interaction or authentication, since they are directly involved by the init process at system startup (see section 2.6.3). The permissions of this role are also very high, as they need access to the log files, configurations, executables, network, or other peripherals. Examples for these services are: A secure communication service (is responsible for establishing secure communication channels to other control devices and to cloud services), an authentication service (handles local as well as remote user authentication requests), an update service (that is responsible for software updates) or a control service (which gathers the data from the sensors, visualizes the data or controls the actuators).

3.3.2.4 Remote access to the control device

Since all control devices of a SCCPS are already interconnected to exchange data, this network can be reused also for remote-control purposes. This requires similar considerations in terms of user administration and user permissions as the already mentioned

local users. Therefore, remote users rely on the same roles as local users except the anonymous user role which does not exist in the remote context. Remote-control should not be possible without authentication. Instead, the anonymous user role is used for the communication from and to field devices. In particular, because the system model doesn't have any requirements for the utilized fieldbus protocols in terms of authentication facilities. Since remote authentication does not require physical presence (which is required for local authentication) remote authentication uses signed asymmetric cryptographic keys instead of numerical codes (see section 2.3.2.3). In addition to the local system role there exist an analogous remote system role which is used by virtual users. This remote system role is required for M2M communication (see section 3.3.2.2) and users with this role rely also on signed asymmetric cryptographic keys for authentication.

3.4 Threat model

In this section, a threat model for the fictitious system model introduced in section 3.3 is presented. This threat model involves only the control devices. Other devices like field devices or network coupling devices that are part of a SCCPS are not covered. This threat model will be created by the methodology introduced in section 3.2. In particular, this threat model is created by using the data flow approach that is introduced in section 3.2.2.2.

3.4.1 Trust levels

Creating a list of trust levels is the first step of the data flow approach (section 3.2.2.2) to create a threat model. These trust levels are based on the user roles that are introduced in section 3.3.2.3 and section 3.3.2.4. All trust levels for this threat model are collected in table 3.1. It is required, that all user levels are equal on each control device and each user is able to authenticate on each control device that is part of the SCCPS. All users must have access to as little features as possible and as much features as necessary.

Table 3.1: Trust levels of a control device. It is easy to see that all users work either local or remote on the device. However, keep in mind that a remote trust level does not necessarily have the same rights than the corresponding local trust level.

TL-ID	Trust level	Description
1	Remote user	Remote users have no physical access to the device. However, they are able to control the control device or communicate with it via its network interfaces.
1.1	Remote standard user	A remote standard user is the default user level for employees or users that utilize the system remotely. Therefore, the access functions depend highly on their workflow.

Table 3.1: (continued)

TL-ID	Trust level	Description
1.2	Remote administrator	A remote administrator is the highest operational access level and has access to management features like user administration, retrieving sensitive system data or completely restart the control device.
1.3	Remote maintenance	A remote maintenance user has the highest remote access level and is used only from maintenance users like the device manufacturer or maintenance contractors. These users have at least the same permissions as a remote administrator and some additional permissions such as updating the device.
1.4	Remote emergency user	A remote emergency user can be used by emergency forces. These users have access to all relevant data and controls that are required to plan and perform emergency operations.
1.5	Remote system user	A remote system user is used for cryptographically secured M2M communications. Therefore, users of this role are in general used for automated data transfer between control devices or periodic backups from control devices to the cloud.
1.6	Remote unreliable user	A remote unreliable user is used for M2M communication that is not (sufficiently) cryptographically secured. Thus, this trust level is used for devices that communicate with the control devices via insecure communication. It is important to distinguish between remote system users and remote unreliable users due to the reduced trustability of remote unreliable users. An example is the communication between control devices and field devices via uncertain fieldbus protocols.
2	Local user	Local users have physical access to the device. Therefore, they can interact with the physical input and output interfaces on the system (displays, buttons, keyboards, etc.).
2.1	Local standard user	A local standard user is the default user level for employees or users that utilize the system locally.

Table 3.1: (continued)

TL-ID	Trust level	Description
2.2	Local administrator	A local administrator is the highest operational access level and has access to management features like user administration or displaying sensitive system data.
2.3	Local maintenance	A local maintenance user is the highest local access level and is used only from maintenance users like the device manufacturer or maintenance contractors. These users have at least the same permissions as a local administrator and some additional permissions such as updating the device.
2.4	Local emergency user	A local emergency user can be used by emergency forces. These users have access to all relevant data and controls that are required to plan and perform emergency operations.
2.5	Local system user	The system operates on its own trust level. It has access to the network interfaces, other IoT-devices and periphery and is required for regular device operation.
2.6	Local anonymous user	A local anonymous user has access to all features that are locally accessible without authentication. This involves local alarm triggers, control-LEDs, status monitors and opening of emergency doors.

3.4.2 Entry points

The next step of the data flow approach is creating a list of entry points (see also section 3.2.2.2). This entry points list contains interfaces of a control device. This list can be seen in table 3.2. Each entry in this list refers to at least one trust level that is defined in table 3.1. Only these trust levels are permitted to access the entry point.

Table 3.2: Entry points of control devices. The column **TL-ID** defines which trust levels use this entry point.

EP-ID	Entry point	Description	TL-ID
1	Network interface		
1.1	Fieldbus	Fieldbus systems are used to connect the SCCPS with the field devices. As mentioned in section 3.3.1, for this fictitious SCCPS it is assumed, that each communication to and from these fieldbus devices is not cryptographically secured. Thus, the fieldbus devices can use only the remote unreliable user trust level.	1.6, 2.3, 2.5
1.2	IP connection	IP connections are typically used for remote-control connections, control device intercommunication and Wide Area Network (WAN) connections. As these communications rely on the confidential communication channel (with integrity and authenticity facilities), all remote trust levels except the remote unreliable users can access the control devices via this entry point. The physical layer may be any technology that is capable of IP communication (e.g. Wi-Fi, Ethernet or Fiber).	1.1, 1.2, 1.3, 1.4, 1.5, 2.5
2	Local administration/maintenance/debug interface	Local administration, maintenance or debug interfaces are special interfaces like Joint Test Action Group (JTAG), serial or console interfaces that can be used for debug, administration or maintenance purposes.	2.2, 2.3
3	Device interaction		
3.1	Notification LED	The notification LED is an exit point that is used to visualize system state information or occurred errors.	2
3.2	Display	The display is an exit point which visualizes system state and control information. It may be used to display sensor values, administration menus, control menus, logging information and much more.	2
3.3	Button	Buttons are used as a simple user input, to switch between display views, operation modes or control purposes.	2

Table 3.2: (continued)

EP-ID	Entry point	Description	TL-ID
3.4	Keypad	The keypad is an advanced user input possibility that can be used for authentication (authentication PIN) and control purposes.	2
3.5	Authentication token	The control devices support also authentication token as an alternative to authentication PINs.	2
4	Internal hardware		
4.1	Power supply	The power supply (including the wires) is required to run the control devices. Only local maintenance users have access to it.	2.3
4.2	TBM	TBMs like TPMs, HSMs or HAB protect the RoT for the boot process in software. They can be used particularly to initially check the integrity of the bootloader (5.2) which is located in the persistent memory. Thus, TBMs represent a very sensitive entry point. Additionally, many TBMs (like TPMs or HSMs, see section 2.5.3.1) can be used to store private keys and apply cryptographic operations with these stored keys.	2.3, 2.5
4.3	Device firmware	This entry point involves the firmware of device components like chipset, networking (e.g. 5G, Wi-Fi) or other periphery devices that are integrated in the control devices. The device firmware is part of the component and never gets executed by the CPU (in contrast to the driver (5.6) that is executed by the CPU). Some components don't support firmware updates. However, if they are supported, only maintenance users with physical access have the permissions to update it.	2.3
4.4	Persistent memory	The persistent memory contains the software (bootloader, OS, drivers, programs and configurations) that is executed on the CPU as well as additional data like log messages. Only local maintenance users and local system users have direct access to the persistent memory.	2.3, 2.5

Table 3.2: (continued)

EP-ID	Entry point	Description	TL-ID
4.5	Expansion port	Expansion ports like USB or serial interfaces can be used to expand the device storage, create backups or for diagnostic purposes. These expansion ports can be used by local maintenance users or admins and the system.	2.2, 2.3, 2.5
5	Software		
5.1	Code in boot ROM	The boot ROM contains code that is executed at system start. Since it is stored in a ROM, it can't be modified. Only the system user has access to the code as it has to execute it.	2.5
5.2	Bootloader	The bootloader gets invoked by the boot ROM. It is used read only and stored in the persistent memory, but it can't be updated in the course of the device update routine. The bootloader can be updated only from local maintenance users, if the persistent memory is replaced.	2.3, 2.5
5.3	OS	The OS is stored twice on the persistent memory (one active and one inactive kernel partition) and gets called by the bootloader. Only the local system user has read access to the active kernel partition in which the current OS is stored. The inactive partition can be updated by the device update routine from each maintenance user.	1.3, 2.3, 2.5
5.4	System applications	System applications are stored in the rootfs partitions. Since the rootfs partitions exist twice in the persistent memory, only the inactive partition can be modified by the update routine that is triggered from maintenance users. The active rootfs partition can't be modified and only the local system user has access to the applications on the active rootfs partition.	1.3, 2.3, 2.5

Table 3.2: (continued)

EP-ID	Entry point	Description	TL-ID
5.5	System libraries	System libraries can be used from any user space application. Only the local system user has read access to the libraries on the active partition. Thus, they can be modified only by the update routine and can be used only from local system users.	1.3, 2.3, 2.5
5.6	Driver	The drivers are stored in the rootfs partitions. Similar to shared libraries and system applications they can be modified only by a system update and are accessible only from local system users.	1.3, 2.3, 2.5
5.7	System configuration	The system configuration is distributed over the rootfs and the data partitions. While the system configuration in the rootfs can't be modified (except by a software update), the part that is stored in the data partition can be read and modified by administration, maintenance or local system users.	1.2, 1.3, 1.5, 2.2, 2.3, 2.5
5.8	Local control application	The local control application is used from the local users of the system to control the control device. It uses the display and the LED for visualization and the buttons and the keypad as input devices. It can be used from each local user.	2
5.9	Remote control application	The remote and control application is the remote analogue to the local control application. It is used from remote users to control the device.	1
5.10	Local authentication service	As users need to authenticate on the system, a possible entry point is the local authentication service. It uses the display and the LED for visualization and the buttons and the keypad as input devices as well as the authentication token. Each local user requires access to it.	2
5.11	Remote authentication service	As users need to authenticate on the system, a possible entry point is the remote authentication service. Each remote user requires access to it.	1

Table 3.2: (continued)

EP-ID	Entry point	Description	TL-ID
5.12	Software update service	The software update service should be able to update the OS, the applications and its configuration either locally or remotely on the control device.	1.3, 2.3
5.13	Secure connection service	The secure connection service establishes a secure connection to other control devices and to cloud services. This secure connection is required for the software update, the remote authentication service and the remote control service.	1.3, 2.3

3.4.3 Assets

After the list of entry points the assets of the control device are examined in table 3.3. Similar to the previous tables containing the entry points and the trust levels this table of assets contains only assets that relate to control devices.

Table 3.3: Assets of a control device. Column TL-ID defines the trust levels with access to the asset.

A-ID	Asset	Description	TL-ID
1	Control device components	Assets that relate to the components of a control device.	
1.1	Device hardware	The device hardware involves TBMs, CPU, persistent memory, volatile memory, data lines and all other components inside the device case. Once adversaries have access to the device hardware, they are able to manipulate the control device. Thus, only local maintenance employees have the permissions to access device internals.	2.3
1.2	Power supply	This asset involves all wires, plugs, batteries, Uninterruptible Power Supplies (UPSs) or Power Supply Units (PSUs) that are responsible for powering the device. Only local maintenance employees have the permissions to access to device internals.	2.3

Table 3.3: (continued)

A-ID	Asset	Description	TL-ID
1.3	Communication wires or plugs	This involves all plugs or wires used to interconnect two control devices or connect a control device with sensors or actuators. Furthermore, connections from and to network coupling devices and WAN connections are also concerned. Only local maintenance employees have the permissions to access to the plugs and wires.	2.3
1.4	Firmware	This involves the firmware of some components of a control device. Typically, network interfaces as well as the CPU contains firmware. If firmware updates are possible, only local maintenance employees have the permissions to perform these updates.	2.3
2	Network communication	Assets that relate to the network communication are in this asset category.	
2.1	Message data	Messages that are transmitted or received during a normal communication are valuable assets. Not only the message payload, also the message metadata like sender or receiver may be useful for adversaries. Only system users and maintenance users have the permissions to access the transmitted messages.	1.3, 1.5, 2.3, 2.5
2.2	Communication endpoints	For adversaries, also the communication endpoints can be helpful assets. However, only maintenance users and the system users have access to the information which communication endpoints are used by a control system.	1.3, 1.5, 2.3, 2.5
2.3	Network utilization	During regular operation, the network traffic utilization should be kept low to avoid high latency (in particular if Carrier Sense Multiple Access (CSMA) is used). Therefore, each network should be capable of handling all messages that occur during normal operation. Thus, a possible asset for adversaries may be the goal to drive the network utilization high.	1.3, 1.5, 2.3, 2.5

Table 3.3: (continued)

A-ID	Asset	Description	TL-ID
2.4	Participate in IP communication	The communication between the control devices is IP-based as well as the remote-control connections. Furthermore, the connections between the control devices and the WAN is also IP based. If adversaries are able to participate in IP communication, they may intercept or inject messages. Only maintenance and system users have the permission to access to the IP communication.	2.3, 2.5, 1.3, 1.5
3	Cryptographic assets	Assets that relate to cryptography.	
3.1	Cryptographic credentials	The utilized cryptographic keys (except public asymmetric keys) are probably the most obvious cryptographic assets for adversaries. As they are used to verify, sign, encrypt or decrypt data (see also section 2.3.2). In particular, because each remote user uses cryptographic authentication keys this asset affects each user role.	1, 2
3.2	TBMs	Because the TBMs built into the control device are able to store confidential data and use particular algorithms, they are possible assets for adversaries. Only local system users and local maintenance users should be able to access the TBM.	2.3, 2.5
3.3	Authentication Token	As each local user can be authenticated by an authentication token. This may be a valuable target too.	2
4	Execute arbitrary applications	Regularly, the device executes only verified software. However, the goal of adversaries may be the execution of arbitrary software.	2.5
5	Manage devices	This category contains assets that relate to remote or local device control.	
5.1	Managing control device	Either local or remote users can be used to control devices. The trust level limits the possibilities that can be controlled. In particular managing control devices can be a valuable asset for adversaries.	1, 2

Table 3.3: (continued)

A-ID	Asset	Description	TL-ID
5.2	Managing sensors and actuators	Each sensor and actuator is assigned to a particular control device. Typically, the actuators are controlled by the control devices (of a SCCPS) depending on the values of the sensors and the internal state of the control devices. Thus, the actuators as well as the sensors may be important assets for adversaries too. Only users of the local system role have the permissions to control these sensors and actuators. Additionally, also local maintenance users are able to manage sensors and actuators of the SCCPS.	2.3, 2.5
6	User administration	Assets that relate to user administration on a control device.	
6.1	Access to user credentials	The user credentials (usernames, passwords, PINs, keys or authentication token) that are needed to authenticate users either remotely or locally on the control device are very important for each control device. Thus, they may also be important assets for adversaries.	1, 2
6.2	Access to personal data	Some personal data may be stored for each user account (like the full name, email, post address or the phone number). Gathering this data may also be valuable for adversaries. Each user has access to its own personal data. Depending on the trust level, it may be possible to request additional data of other users.	1, 2
6.3	User management	The user management facilities of control devices can be used to create, modify or delete users. Thus, adversaries may target it. In particular, the possibility to create or remove user accounts may be useful for adversaries. Only maintenance and administration users are able to create or remove users.	1.2, 1.3, 2.2, 2.3

Table 3.3: (continued)

A-ID	Asset	Description	TL-ID
7	Boot ROM	The boot ROM is the first component that gets invoked if the system starts. Its purpose is verifying the bootloader as well as invoking it. Thus, the boot ROM may be a valuable target for adversaries.	2.5
8	Data in the persistent memory	Assets that relate to the stored data in the persistent memory.	
8.1	Bootloader	The bootloader is stored in the persistent memory and starts the OS. As the bootloader initializes the volatile memory (see section 2.6.1.2) and selects which of the symmetric OS-partitions should be started (see section 2.7), this asset is very important. Thus, it may be a useful asset for adversaries. As the bootloader is automatically started at system startup, the local system user needs access to it. Additionally, the local maintenance users require access to it too.	2.3, 2.5
8.2	OS	The OS is also stored in the persistent memory and may be a valuable asset for adversaries, since it forms the basis for all further applications. In particular, if the OS contains confidential components. Only the local system user and local maintenance users have access to the OS-image.	2.3, 2.5
8.3	Applications	The applications running on the control devices define the functionality of the devices. This involves tasks like collecting data from the connected fieldbus devices, communicate with other control devices and interacting with the users. Thus, the utilized applications may be profitable assets for adversaries. These applications are part of the rootfs and are located in the persistent memory. Only the local maintenance users and the local system user have access to the applications.	2.3, 2.5

Table 3.3: (continued)

A-ID	Asset	Description	TL-ID
8.4	Configuration	The configurations are located in the rootfs and in the data partition. The configuration is used to adjust the behavior of the OS and the applications. Therefore, it is a helpful asset for adversaries as it may contain confidential information. Maintenance users and administration users as well as the system users have access to the configurations.	1.2, 1.3, 1.5, 2.2, 2.3, 2.5
8.5	Logs	The system logs are stored in the data partition. If they contain confidential information, the logs can be helpful for adversaries too. Maintenance users and administration users as well as the system users have access to the logs.	1.2, 1.3, 1.5, 2.2, 2.3, 2.5
9	Services	This involves all services that are provided by a control device. This involves the secure connection service, the authentication service, the update service or the control service. Stopping a service may be a valuable asset for adversaries.	1, 2

3.4.4 Usage scenario

In this section, a usage scenario of the control devices will be defined.

Even though this threat model relies on the system definition from section 3.3. Thus, it doesn't cover all possible threats of a control device. In particular, as the main focus of this work lies at the following three topics:

- Trusted boot mechanism for control devices.
- Secure communication between two control devices and a secure remote-control channel for control devices.
- Secure update mechanism for control devices.

Therefore, the usage scenario is described by the following items:

1. Even though the field devices (sensors or actuators) are part of the SCCPS, any threats regarding these as well as the communication to or from these field devices are not handled in the threat model.

2. The network coupling devices as well as the firmware/software running on them are out of scope of this threat analysis. These devices are examined as transparent communication members that are responsible to forward the received packages to the correct receiver.

3. The control devices are partly connected to other devices in office LANs (see figure 3.3). Any threats regarding office devices are not part of this threat analysis.

4. Some control devices may have direct access to the Internet due to cloud connectivity requirements. The connections to and from the cloud services are cryptographically secured. However, the cloud service is not covered by this threat analysis.

5. The utilized PKI includes the CA that issues the certificate for each device. There exist some threats that relate to the CA, but the main focus of this threat analysis are the control devices and not the components of the PKI.

6. There exist some threats that relate to physical harming of the control devices (like wire unplugging, wire cutting, device disassembling, powering off devices or damaging the hardware) but these threats are not the main focus of this threat analysis.

7. The software and the hardware that is utilized on the control devices is not specified exactly. Therefore, threats regarding the concrete hardware or software cannot be covered in this threat model.

8. As the OS, bootloader, library dependencies (in particular cryptographic libraries) and the applications rely on the latest security patches of the system, it is assumed that the system is kept up to date. Therefore, new updates are applied fast to retrieve the latest bug fixes of known vulnerabilities.

9. The maintenance user roles (trust level 2.3 or 1.3) should be used only from device manufacturer or maintenance contractors.

10. The administrator roles (trust level 2.2 or 1.2) should also be used carefully, because the permissions of this role are very high.

11. Emergency force units use the emergency roles (trust level 2.4 or 1.4). As they have to be able to use the control devices in case of emergency. Thus, the control devices must be installed in locations that are easy to access for emergency force units.

12. Users of the control device can only be added from existing users with the administrator or maintenance role (2.2, 2.3, 1.2, 1.3).

13. All control devices use fixed IP. Dynamic IP services like DHCP should also be possible but these mechanisms can lead to additional threats that can be prevented if fixed IPs are used.

14. Because fixed IPs are used, DNS servers can be omitted too. This requires that the cloud service can be accessible via a static IP. Thus, for this threat model all threats regarding the DNS can be omitted.

15. The control devices require firewalls that are configured conservative (i.e. only services that are absolutely required can be used).

3.4.5 External dependencies

The external dependencies of the control device:

1. A very obvious external dependency is electricity disruption. In particular, because the system requires electricity to work.

2. The loss of Internet connectivity is not as critical as electricity disruption. As the system is able to remain in an operational state. However, it is obvious that in this case software updates can't be downloaded from the Internet. Also cloud services can't be accessed anymore and emergency calls can't be sent via the Internet.

3. The office LAN (and its security) that is reused to interconnect the control devices is an external dependency.

4. The cloud service is another external dependency that has to be considered.

5. The building where the SCCPS is installed should be safe and secure. It should protect against natural disasters like earthquakes or thunderstorm.

6. Network coupling devices are required for the local network. These devices ensure that the control devices are interconnected and can communicate with each other.

7. The SCCPS depends on the correct function of the hardware, that is built into the control devices (e.g. TBM, boot ROM, CPU, volatile or persistent memory).

8. The authentication token that are used by the users to authenticate on the control device.

9. The TBM that is used as RoT during the boot process.

10. The boot ROM that is used as the first stage that is started at boot up.

11. The bootloader, the OS (Linux) and software libraries (in particular cryptographic libraries) are external dependencies, as these components are created by external organizations.

12. To support certificates with expire dates, the control devices rely on the current date and time (for this task, no high precision clock is required). Thus, each control device contains a buffer battery assisted real time clock and the possibility to synchronize it via multiple sources.

13. The CA is used to issue certificates that are required to establish secure connections.
14. The image signing service generates a software image signature that allows the control devices to install these images.

3.4.6 Implementation assumptions

The implementation assumptions for the control device:

1. All network connections between control devices, are capable of IP communications. Therefore, IP-specific security measures can be assumed (see section 3.3.2.2).
2. The PKI operates in a secure environment and issues only valid certificates for each particular device. Furthermore, certificates can be issued only by legitimated users of this CA.
3. Each secret cryptographic key is used once and is stored securely in the control device (either with a TBM, an encrypted storage or it is a temporary key in the tempfs).
4. An efficient and secure cryptographic library is utilized. It supports effective hash algorithms, as well as symmetric and asymmetric cryptography.
5. Software updates are applied sufficiently often, but at least if security critical vulnerabilities can be fixed.
6. In this system model, certificates are distributed via software updates. Thus, it must be guaranteed that the certificate update is run sufficiently often, such that certificates do not expire.

3.4.7 Security notes

The security notes for the control devices:

1. The wires and plugs of the complete system should not be publicly accessible.
2. Restrict access to the devices that are part of a SCCPS (e.g. control devices, field devices, network coupling devices).
3. Keep track of the range of the used wireless networks. Maybe limit the access to places in the wireless range.
4. User authentication differs between local and remote users: Local users require a PIN code, or an authentication token combined with a PIN code, while remote users rely on asymmetric cryptographic keys.
5. System updates can only be installed from users with the maintenance trust level (2.3, 1.3).

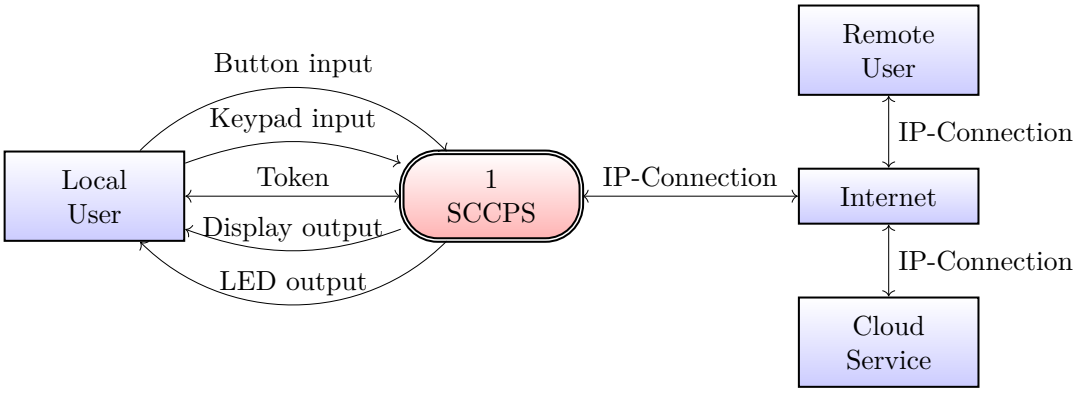


Figure 3.4: Context DFD: Visualization of interaction between remote- and local users as well as cloud services and a SCCPS.

3.4.8 DFDs

The DFDs for the SCCPS are shown in the figures 3.4-3.8:

Figure 3.4 gives a short overview of the technologies the users and cloud services can use to interact with the control devices: Local users can interact with the control devices via buttons, a keypad, LEDs and a display that are mounted at the case. Additionally, each control device supports token for authentication purposes (e.g. USB, NFC). Remote users and cloud services use IP (either IPv4 or IPv6) connections to communicate with the control devices in the SCCPS.

Figure 3.5 shows in contrast to figure 3.4, what kind of data is transmitted to the individual components of a SCCPS. Thus, the shape SCCPS in figure 3.4 is split into its components (control devices, sensors and actuators). It is important to mention that each control device collects sensor data from all sensors that are connected to it and provides this data to all other control devices in the SCCPS. Thus, local users connected to control device 1.1 are able to see the sensor data from sensor 1.S2 that is connected to control device 1.2. The same principle holds for remote users that are connected to control device 1.2 but request sensor or control data from sensors and actuators that are connected to control device 1.1.

In figure 3.6, the authentication methods of control devices are shown. In this example, the SCCPS consists of two control devices (1.1 and 1.2) and a field device (1.F1) that could be either a sensor or an actuator. The control devices itself authenticate each other via asymmetric keys that were signed by a trusted CA. Remote users and cloud services are authenticated the same way. Local users can be authenticated on each control device via a PIN or a combination of an authentication token and a PIN. In particular, if an authentication token is used, it is important to rely on an additional associated PIN too. Thus, even if the token gets stolen, adversaries are not able to use the token without knowing the associated PIN. The authentication of field devices is unspecified in this system model, as it is related to the utilized fieldbus and the involved devices.

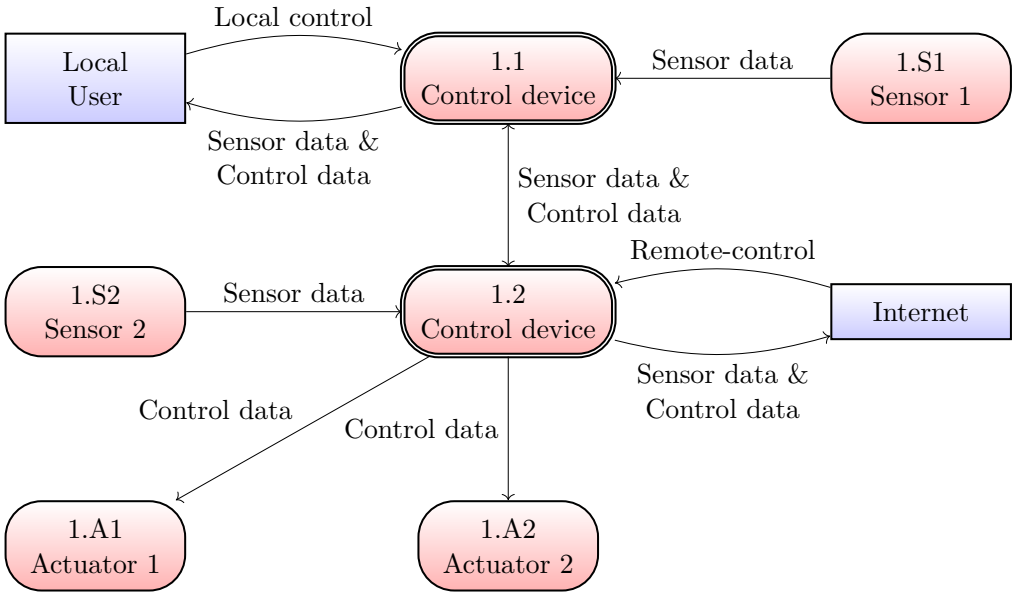


Figure 3.5: Level 0 DFD: Visualizes the data flow between the components of a SCCPS, the users and the Internet.

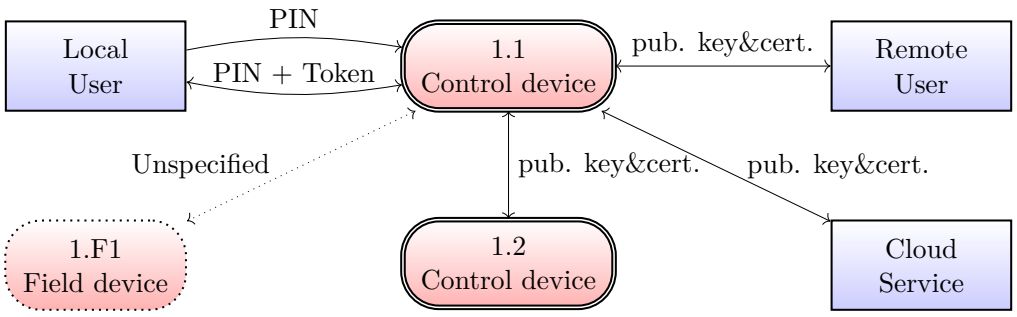


Figure 3.6: Level 0 DFD: Shows the different authentication methods that are supported by each control device in a SCCPS.

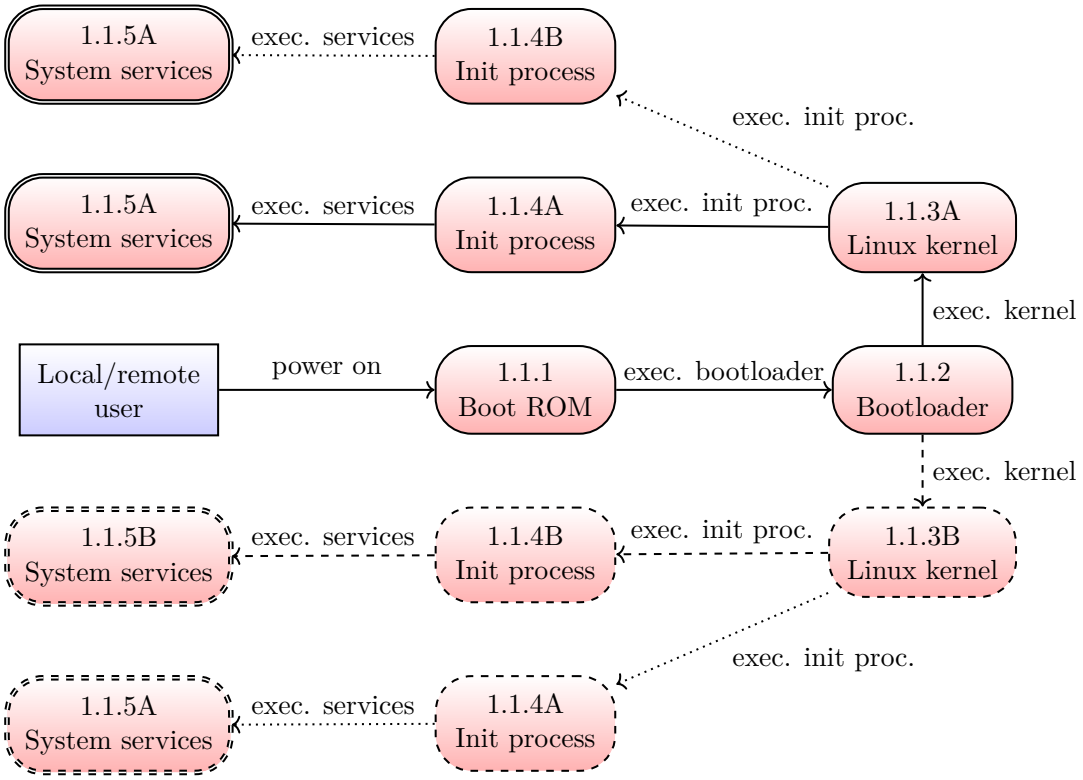


Figure 3.7: Level 1 DFD: Visualization of the trusted boot sequence of a control device (1.1).

In figure 3.7, the boot process of a control device is shown: At first, a local or a remote user triggers the system start sequence. This involves executing the code in the boot ROM. The next step is executing the bootloader that is stored in the persistent memory. As each control device contains two Linux kernels (A and B) and two root partitions (A and B), the bootloader selects which Linux kernel should be started. The dashed data flow denotes the inactive kernel selection. Once the Linux kernel has been started, it decides which root partition should be mounted. Depending on this decision, either the init process A as well as the system services A or the init process B as well as the system services B are started. The dotted data flow denotes the inactive selection. This selection can be changed, if a software update was applied on a particular rootfs and kernel partition.

Establishing a secure communication session between two control devices is shown in figure 3.8. It visualizes what metadata is transmitted in order to create a secure communication channel. It is important to keep in mind that sending and receiving regular messages via this secure communication channel is only possible, if this channel has been setup successfully before. This is denoted by dotted lines in figure 3.8 (i.e. process 1.1.14 has been executed successfully). Be aware that this proposed secure communication ap-

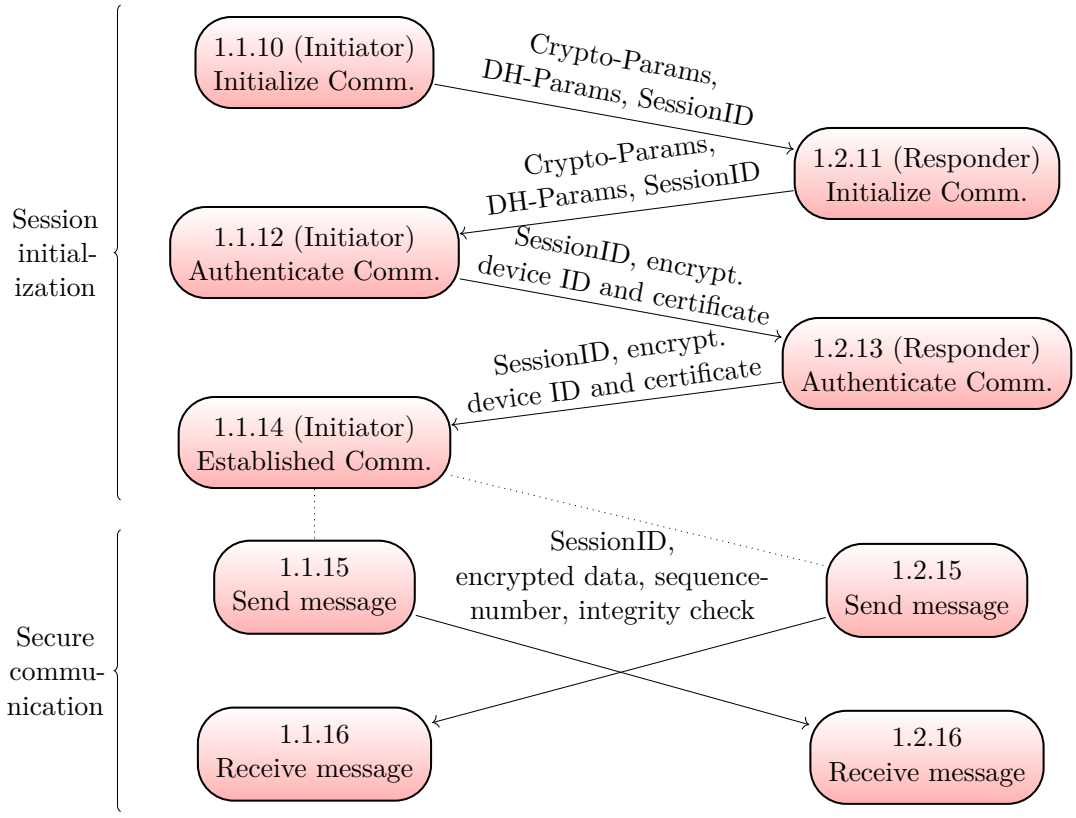


Figure 3.8: Level 1 DFD: Visualization of a secure communication between two control devices (1.1 and 1.2).

proach relies on the connectivity model, that is explained in section 3.3.2.2. In order to use a confidential communication channel between two control devices, that supports integrity and authentication, it must be established first. At first, the communication must be initialized. This step involves agreement on a shared DH key as well as the involved cryptographic algorithms. If several channels can be established concurrently, also the session IDs must be shared. The second step is the authentication step: This involves the transmission of the device ID and the certificate that has been signed by a trusted CA. In this step, the data is already transmitted encrypted by the common DH key that was created in the step before. If both endpoints have validated the received data successfully, the communication is established, and regular messages can be transmitted via this communication channel. These regular messages contain the session ID, encrypted data, a sequence number to prevent replay attacks and an integrity check value. As an additional security step, the encryption key should be changed regularly and should not depend on previous encryption keys.

In figure 3.9, the update process of control devices is shown. All updates must be triggered by either local or remote maintenance users. Once the update has been down-

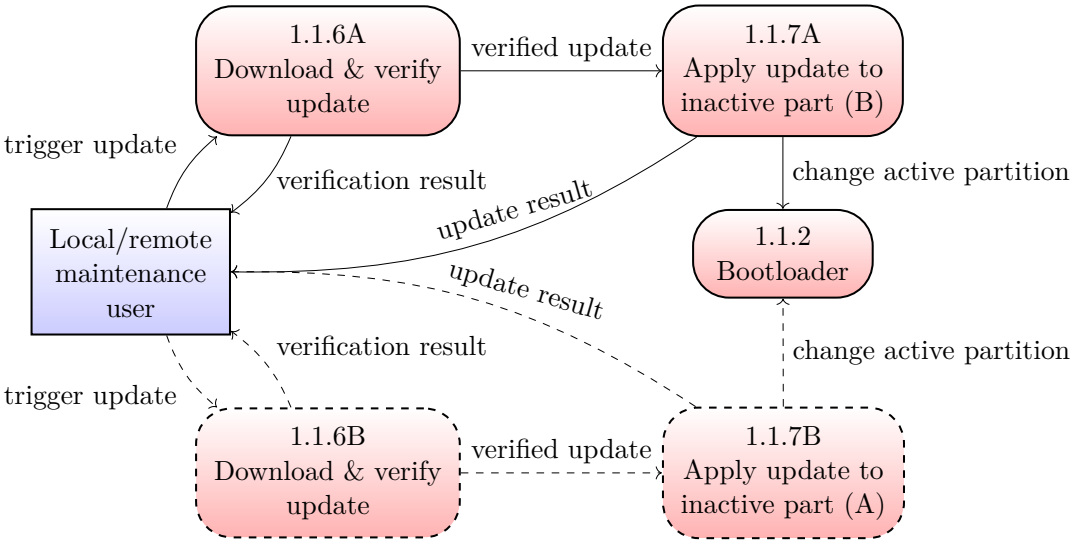


Figure 3.9: Level 1 DFD: Visualization of the update process of a control device (1.1).

loaded, it is verified. This verification step checks if the update is valid and compatible with the control device. If these checks fail, the update is aborted. Otherwise the update is applied onto the inactive kernel image and onto the inactive rootfs. If this step succeeded, the user gets notified and the bootloader resets the active partition to the currently updated partition. In figure 3.9, the dashed processes and data flows denote the inactive system parts. Note that an update renews only the kernel and the rootfs. If the bootloader must be updated the persistent memory must be replaced (see section 3.3.2.1). Thus, the next time, the system starts, the bootloader chooses this updated kernel and rootfs partitions.

3.4.9 Threats

In the following section, the threats of the control devices are listed.

These threats are related to the previously collected information (e.g. assets or entry points). In section 3.4.10, the threat trees for these threats can be found. These threat trees visualize the threat conditions that are described in the threat description. In particular, if one threat relies on other threats, the mitigation must take care of all these threats.

Table 3.4: Threat: Exploit control device software

ID:	1
Name:	Exploit control device software

Table 3.4: (continued)

Description:	As each control device runs Linux and several user space tools, each of these services, libraries or programs can potentially be exploited. In general, these exploits are specific to each part of the software. Thus, each software component that is executed on the device can lead to potential threats and must therefore be considered particularly. These threats can be exploited, if adversaries use a flawed service or use a service that itself relies on flawed components.
Threat tree:	Figure 3.10
Example:	These threats can be found in libraries, drivers, the OS or several system services. Buffer overflows, DoS or EoP are some examples that may lead to these threats. In particular, if binary blobs are used, where the source is not available, it can be hard to fix security flaws.
STRIDE class:	S, T, I, D, E
Mitigated:	The system model in section 3.3 does not specify the utilized software. Thus, it depends on the concrete software implementation whether this threat is mitigated or not.
Known mitigation:	Rely on software that is well supported and receives bug fixes to critical errors fast. Decide, if closed source binary blobs are necessary. Keep the software up to date to prevent exploits and reduce the attack surface. Run the software with the least possible access rights.
Background information:	External dependency: 11, Implementation assumptions: 5
Entry points:	5.x
Assets:	4, 6.x, 7, 8.x

Table 3.5: Threat: Get confidential credentials

ID:	2
Name:	Get confidential credentials
Description:	For the security of a SCCPS and its control devices it is important to keep authentication credentials or encryption keys confidential and prevent adversaries from getting them. Thus, several security concepts rely on the confidentiality of these credentials. If adversaries are able to get them, they may be able to access restricted parts of the system. Adversaries may be able to get secret credentials by social engineering. Another possibility is guessing these credentials. It can be the case, that adversaries already have access to the system and the credentials can also be stolen.

Table 3.5: (continued)

Threat tree:	Figure 3.11
Example:	In particular, if passwords or PINs can be chosen by the users it is very likely that they choose poor passwords, as they are much easier to remember [120]. Thus, in combination with dictionaries or leaked password databases adversaries may be able to guess credentials.[3] Adversaries may be able to spy these credentials while they are typed in (capture keypress events). Additionally, system logs and transmitted messages (if messages are transmitted unencrypted) may contain confidential information as well as displays may visualize these. Sometimes adversaries have already access to the control device (e.g. if the adversary is an employee). In this case, it may be hard to mitigate against this threat. Also keep in mind that TBMs or authentication token are able to store credentials too.
STRIDE class:	S, R, I, E
Mitigated:	It is important that randomly generated PINs and keys (asymmetric and symmetric) are used. The system model in section 3.3 does not specify confidential credential handling guidelines for users. Thus, it depends on the security awareness of the users whether this threat is mitigated or not.
Known mitigation:	For the final SCCPS, a guideline regarding secret credentials has to be worked out which has to be obeyed by each user. In particular, encryption keys as well as authentication PINs should have a particular length. Additionally, authentication token in combination with a PIN should be preferred over a PIN alone.
Background information:	External dependencies: 8, 9 Implementation assumptions: 3, 4 Security note: 4
Entry points:	1.x, 2, 3.4, 3.5, 4.2, 5.x
Assets:	3.x, 6.1

Table 3.6: Threat: Loss of confidential credentials

ID:	3
Name:	Loss of confidential credentials

[3] As stated in [121], with dictionaries (like the Rockyou.com list) and slightly modifying it by common password patterns, it is also possible, to guess a huge number of passwords.

Table 3.6: (continued)

Description:	Because credentials like symmetric keys, private asymmetric keys or PINs are very important, it can be a big issue if these get lost. In particular, if devices where these keys are stored get physically damaged (e.g. fire, earthquake) or malware as well as adversaries are able to delete these credentials. Furthermore, these credentials may be forgotten and not noted somewhere else. Additionally, accidental data loss can happen.
Threat tree:	Figure 3.12
Example:	Loosing private asymmetric keys can lead to fatal problems. For instance, if the private key of a CA gets lost it is not possible to issue certificates anymore. However, regular users may not be able to remember the correct PIN to authenticate on the device.
STRIDE class:	D
Mitigated:	yes
Known mitigation:	Keep multiple copies of important credentials (like the private key of the CA that issues the certificates of the system). Systems should accept backup credentials too (the use of multiple CAs which are supported by the system). Provide facilities to reset the PIN, if the user authenticity is checked.
Background information:	External dependency: 8
Entry points:	1.x, 4.2, 5.x
Assets:	3.x, 6.1

Table 3.7: Threat: Bypass control device authentication

ID:	4
Name:	Bypass control device authentication
Description:	A user may be able to use system services without a proper authentication check. This can be achieved by exploiting the authentication service. Another approach is exploiting a particular system service which can be used to inherit the permissions. Additionally, an authentication token (external dependency) may be exploited. In particular, if this token has some known vulnerabilities.
Threat tree:	Figure 3.13
Example:	Bypassing the authentication service may be achieved by SQL-injections or buffer overflows [122]. However, also system services can be exploited such that adversaries are able to use the system with the permissions of the service.
STRIDE class:	S, T, R, I, E

Table 3.7: (continued)

Mitigated:	The system model in section 3.3 does not specify the concrete authentication service and related system services. Thus, it depends on the concrete software implementation whether this threat is mitigated or not.
Known mitigation:	If system services run in isolated environments (e.g. containers or virtual machines), it gets harder for adversaries to break out of this environment [123]. Keep the software up to date to prevent exploits and reduce the attack surface.
Background information:	External dependencies: 8
Entry points:	1.x, 2, 3.4, 3.5, 4.2, 5.x
Assets:	5.x, 6.x, 9

Table 3.8: Threat: Undesired privileged access to control device

ID:	5
Name:	Undesired privileged access to control device
Description:	Once adversaries are able to get confidential credentials, they are able to use these to log into the device. Additionally, adversaries may be able to bypass the authentication.
Threat tree:	Figure 3.14
Example:	If adversaries are able to guess login credentials, they may be able to log in as regular users or even administrator.
STRIDE class:	S, R, I, E
Mitigated:	yes
Known mitigation:	With stolen or spied keys or PINs, adversaries are not distinguishable from legitimate users. Thus, use certificates with expiration date and use authentication token that can be disabled if they get lost. Additionally, change PINs frequently. Keep the software up to date to prevent exploits and reduce the attack surface. Be sure that the threats 2 and 4 are mitigated.
Background information:	Implementation assumptions: 3, 4
Entry points:	1.x, 2, 3.4, 3.5, 4.2, 5.x
Assets:	5.x, 6.x, 9

Table 3.9: Threat: Undesired physical access to a control device, its wires or plugs

ID:	6
Name:	Undesired physical access to a control device, its wires or plugs

Table 3.9: (continued)

Description:	Adversaries may have direct access to control devices, wires or plugs, because they have permissions to access the location where they are installed. For systems that rely on wireless connections, it is sufficient to get in the wireless range. Getting access to restricted areas can be achieved by physical violence or social engineering. Additionally, adversaries may already have access, because the location is publicly accessible (like telephone wires) or adversaries are employees.
Threat tree:	Figure 3.15
Example:	If adversaries do not have the permissions to get in the area where the SCCPS components are installed, this can be achieved by social engineering: It may be sufficient to masquerade as engineer or repairman, to obtain access to restricted areas. Another possibility for adversaries is to seduce someone with access to the target to do particular actions for them. However, also key reproducing or lock picking can be used, to attain access to the target.
STRIDE class:	I, D
Mitigated:	The system model in section 3.3 does not specify the concrete system installation. Thus, it depends on the installation whether this threat is mitigated or not.
Known mitigation:	Restrict access to potential attack targets only to trustworthy persons. Also keep in mind that not only the control devices itself, also the connection and power cables, are part of the safety critical infrastructure. If wireless communication is used, places in the wireless range should be considered too, as well as SCCPS components that are publicly accessible.
Background information:	External dependencies: 1, 2, 3, Security notes: 1, 2, 3
Entry points:	1.x, 2, 3.x, 4.x
Assets:	1.x, 2.x, 3.2, 6.x, 7, 8.x

Table 3.10: Threat: Physical control device manipulation/damage

ID:	7
Name:	Physical control device manipulation/damage
Description:	Once adversaries have physical access to control devices, they may be able to manipulate these devices too: This manipulation can target each component of the device, e.g. display, circuit or buttons. It may also be the case that the control device gets physical damage from the environment.

Table 3.10: (continued)

Threat tree:	Figure 3.16
Example:	Adversaries may be able to destroy or misuse the expansion or connection ports like USB. Additionally, adversaries may be able to replace the persistent memory, the boot ROM or the TBM. Furthermore, the display may also be a target for adversaries.
STRIDE class:	T, D
Mitigated:	The system model in section 3.3 does not specify the concrete device and its case. Thus, it depends on the final device whether this threat is mitigated or not.
Known mitigation:	Mitigate against physical damage as well as modifying the circuit. This can be achieved by robust cases that are hard to break as well as case intrusion detection systems. Also modifying the circuit can be mitigated physically.
Background information:	External dependencies: 5, 7, 9, 10, Security note: 2
Entry points:	3.x, 4.x
Assets:	1.1, 1.4, 3.2, 6.x, 7, 8.x

Table 3.11: Threat: Control device wires/plugs manipulation

ID:	8
Name:	Control device wires/plugs manipulation
Description:	Once adversaries have physical access to the wires or plugs of a control device, they may be able to manipulate them too: This can have huge impact in the network connectivity. In particular, if star or tree network topologies are used.
Threat tree:	Figure 3.17
Example:	Adversaries may be able to disconnect all Ethernet connections of a control device. Additionally, they may be able to unplug or plug in external USB devices.
STRIDE class:	I, D
Mitigated:	The system model in section 3.3 does not specify the concrete system installation. Thus, it depends on the installation whether this threat is mitigated or not.
Known mitigation:	Mitigation against power loss can be achieved by using a UPS. Redundant network connections help to mitigate against connection loss. Additionally, the control device software should be able to operate temporary without any active connections to other control devices.
Background information:	External dependencies: 1, 2, 3, Security note: 1

Table 3.11: (continued)

Entry points:	1.x, 2, 4.1, 4.5
Assets:	1.2, 1.3, 2.x

Table 3.12: Threat: Undesired remote-control device access

ID:	9
Name:	Undesired remote-control device access
Description:	Control devices of a SCCPS can be accessed via remote IP connections. This involves Local Area Network (LAN)-connections, tunnels over other networks or Internet connections.
Threat tree:	Figure 3.18
Example:	To save network installation costs, the components of a SCCPS are able to share the LAN installation with the existing office infrastructure. Thus, office PCs and SCCPS components are in the same physical network. In that case, office PCs may be able to use control device services of the SCCPS. Another example is a SCCPS where some control devices have direct access to the Internet. This may be used, if some control devices directly rely on cloud services.
STRIDE class:	I, D
Mitigated:	The system model in section 3.3 does not specify the concrete network installation and cloud connections. Thus, it depends on the installation whether this threat is mitigated or not.
Known mitigation:	Operate the SCCPS components in separate networks where no other devices are involved. If the network infrastructure has to be shared with an existing office infrastructure, use firewalls or VPN tunnels to isolate the SCCPS. VPNs can also be used for connections to cloud services.
Background information:	External dependencies: 2, 3, 4
Entry points:	1.x
Assets:	5.x, 6.x, 9

Table 3.13: Threat: Undesired use of local control device service

ID:	10
Name:	Undesired use of local control device service

Table 3.13: (continued)

Description:	Control devices of a SCCPS provide several services that are accessible locally. Using a particular local service requires that adversaries have physical access to the control device and have achieved undesired privileged access to the control device. It also involves the undesired use of the admin/debug interface.
Threat tree:	Figure 3.19
Example:	If adversaries are able to use a local control device service, they use the display, notification LEDs, buttons and the keypad to interact with the control device. Another example involves the use of the debug interface to change the control flow of a program.
STRIDE class:	S, I, D, E
Mitigated:	yes
Known mitigation:	Restrict authorized users of the control device services to keep the attack surface as small as possible. Do not grant access to services, if this access is not really needed. Disable debug interfaces (like JTAG) in production environments.
Background information:	Implementation assumptions: 3, 4, Security note: 4
Entry points:	2, 3.x
Assets:	5.x, 6.x, 9

Table 3.14: Threat: Undesired use of remote control device service

ID:	11
Name:	Undesired use of remote control device service
Description:	Control devices of a SCCPS provide several services that are accessible remotely. Using a particular remote service requires that adversaries have remote access to the control device and have achieved undesired privileged access to the control device.
Threat tree:	Figure 3.20
Example:	If adversaries are able to use a remote device service, they use an IP connection to interact with the control device.
STRIDE class:	S, I, E
Mitigated:	yes
Known mitigation:	Restrict authorized users of the control device services to keep the attack surface as small as possible. Do not grant access to services, if this access is not really needed.
Background information:	External dependencies: 2, 3, 4, Security note: 4
Entry points:	1.2
Assets:	5.x, 6.x, 9

Table 3.15: Threat: Undesired use of control device service

ID:	12
Name:	Undesired use of control device service
Description:	Control devices of a SCCPS provide several local or remote services. Using a particular device service requires that adversaries are able to use either remote device services or a local device service.
Threat tree:	Figure 3.21
Example:	Adversaries may get access to control device services by social engineering.
STRIDE class:	S, I, E
Mitigated:	yes
Known mitigation:	Restrict authorized users of the control device services to keep the attack surface as small as possible. Do not grant access to services, if this access is not really needed.
Background information:	
Entry points:	1.2, 2, 3.x
Assets:	5.x, 6.x, 9

Table 3.16: Threat: Exploit update mechanism

ID:	13
Name:	Exploit update mechanism
Description:	To keep the control devices up to date each of these devices provides a software update mechanism. This mechanism can be exploited by adversaries to downgrade the software to a version which contains known bugs. Other approaches for adversaries may target the obligatory image verification (which is applied before the software update is started) or the software upgrade routines itself. Additionally, adversaries may be able to change the active partitions.
Threat tree:	Figure 3.22
Example:	A possible goal for adversaries may be the prevention of further software updates. In particular, if updates are applied automatically without any feedback from each affected device, this can lead to huge problems. This goal may be achieved if a previously installed software update gets never executed because adversaries prevent the bootloader from selecting it as the active partition.
STRIDE class:	T, I, D

Table 3.16: (continued)

Mitigated:	The system model in section 3.3 does not specify the concrete update service and image verification implementation. Thus, it depends on the concrete software implementation whether this threat is mitigated or not.
Known mitigation:	Downgrading images can be mitigated, if the software updates contain a strictly increasing versioning scheme. Thus, it is not allowed to install a software version, that is older than the current installed version. To mitigate the "Change active partition" vulnerability, the bootloader should be able to decide which partitions should be chosen. It prefers partitions that contain newer updates, but if they fail to start repeatedly, the bootloader falls back to the backup partitions. Software updates should only be applied by users with maintenance trust level.
Background information:	Implementation assumptions: 5, Security note: 5
Entry points:	5.13, 5.12
Assets:	8.2, 8.3, 8.4

Table 3.17: Threat: Modify system partitions

ID:	14
Name:	Modify system partitions
Description:	If all system configuration files are stored in the read only mounted rootfs, it would not be possible to change the behavior of the system services without applying any software updates. Thus, it is necessary to change the system configuration via a configuration service that stores this configuration in an additional read- and write-able partition. However, also the write-protection of the OS or rootfs partition may be exploited by adversaries. Additionally, the update mechanism can be exploited such that it is possible to modify the system partitions. Adversaries may also be able to modify the persistent memory if they have physical access to control devices.
Threat tree:	Figure 3.23
Example:	Examples for malicious system configurations are wrong filenames, wrong TCP or UDP ports, changes regarding the user authentication or switching to extensive logging (which can lead to wear out if NAND is used). Additionally, adversaries may be able to replace the entire persistent memory or use in circuit programming to overwrite it. The latter can also lead to the wear-out effect, that is explained in section 3.3.2.1.

Table 3.17: (continued)

STRIDE class:	T, I, D
Mitigated:	yes
Known mitigation:	Keep as much configuration as possible in the read only part of the partition (as this part gets validated at startup). Use image verification before the partition is executed, to detect if a partition has been modified.
Background information:	External dependency: 11
Entry points:	1.2, 2, 3.x, 4.x, 5.12
Assets:	8.x

Table 3.18: Threat: Undesired signing of image

ID:	15
Name:	Undesired signing of image
Description:	Adversaries may be able to sign a boot partition (image). This can be achieved, if adversaries are able to get the confidential private key that is used to sign the image. Another possibility is exploiting the regular image signing service (which is located outside of the SCCPS). Thus, adversaries may be able to introduce malicious images that are signed as if they would be regular images.
Threat tree:	Figure 3.24
Example:	A malicious employee with access to the image signing service may be able to produce signed image that contain harmful software and that would possibly be accepted by the update service.
STRIDE class:	T, D
Mitigated:	yes
Known mitigation:	Make sure that the private key that is used to sign the images is kept secret. Pay special attention to the image signing service as each image that is signed by this instance can be installed on the control devices as it contains a valid signature. Make sure that only trusted entities are able to create a valid image signature.
Background information:	External dependency: 14, Implementation assumptions: 3, 4
Entry points:	
Assets:	8.x

Table 3.19: Threat: Bypass image verification

ID:	16

Table 3.19: (continued)

Name:	Bypass image verification
Description:	Regularly, each control device verifies the boot partitions before they are executed (see section 3.3.2.1). Thus, if adversaries are able to bypass the image verification, they would be able to execute arbitrary code on the device. To bypass the image verification, several possibilities exist: Exploiting the verification part of the boot stages. This includes parts of the boot ROM (and TBM), the bootloader or the OS. Another possibility is exploiting (validated) system services such that unvalidated code is executed. Furthermore, adversaries may be able to disable or replace the boot ROM or TBM.
Threat tree:	Figure 3.25
Example:	Because there are many components involved, that verify the boot images before they are used (boot ROM, bootloader, Linux kernel), each component implementation can have its own security flaws that can be exploited. This exploit may be used to execute arbitrary software via network, boot from an expansion port (e.g. USB-stick) or from an altered partition.
STRIDE class:	T, D
Mitigated:	yes
Known mitigation:	Use only components that contain no known vulnerabilities regarding the image verification. If vulnerabilities get published, try to fix the issues. Keep the software up to date to prevent exploits and reduce the attack surface (not always possible for issues in the boot ROM). Additionally, it should be difficult for adversaries to get access to the boot ROM. One possibility is putting the boot ROM in the same package as the SoC, such that it can't be replaced easily. Additionally, altering the persistent memory of a control device should be difficult too.
Background information:	External dependencies: 9, 10, 11, 14
Entry points:	4.2, 4.3, 4.4, 4.5, 5.x
Assets:	4, 8.x

Table 3.20: Threat: Exploit image verification

ID:	17
Name:	Exploit image verification

Table 3.20: (continued)

Description:	Before executables in the boot images can be executed, each partition is verified (as mentioned in threat 16). To exploit this image verification, adversaries may be able to bypass this image verification, invalidate the image (with the goal that image verification fails) or forge the image in a way that image verification succeeds. However, all these possibilities require that the system partitions are modified.
Threat tree:	Figure 3.26
Example:	One example is the invalidation of both redundant images (i.e. both OS images). This behavior can be used to drive a DoS attack as it is not possible to boot from invalid images. However, adversaries may also be able to modify the system partitions and bypass the image verifications.
STRIDE class:	T, D
Mitigated:	yes
Known mitigation:	In particular, if DoS should be prevented, it should be hard for adversaries to modify the system partitions. Thus, the update mechanism should be implemented such that a valid set of partitions should not be updated as long as an invalid partition set exists. Use hard cryptographic signing algorithms to prevent forging of images.
Background information:	External dependencies: 9, 10, 11, 14
Entry points:	4.2, 4.3, 4.4, 4.5, 5.x
Assets:	4, 7, 8.x

Table 3.21: Threat: Exploit boot sequence of control devices

ID:	18
Name:	Exploit boot sequence of control devices
Description:	The boot sequence consists of four stages: The boot ROM (and TBM), the bootloader, the Linux kernel and the init process. Each stage consists of its own executable code that is stored in the persistent memory or in the boot ROM.
Threat tree:	Figure 3.27
Example:	Adversaries may be able to exploit the init process to start particular services with higher permissions than necessary.
STRIDE class:	T, D
Mitigated:	The system model in section 3.3 does not specify the concrete implementation of the boot sequences. Thus, it depends on the software implementation whether this threat is mitigated or not.

Table 3.21: (continued)

Known mitigation:	Keep the boot stages simple, reduce the dependency to the runtime configuration which could eventually be exploited by invalid input. Use compile configuration if possible. Reduce the reliance to user input during boot.
Background information:	External dependencies: 9, 10, 11
Entry points:	4.4, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7
Assets:	7, 8.1, 8.2, 8.3

Table 3.22: Threat: Modify boot process of control devices

ID:	19
Name:	Modify boot process of control devices
Description:	Depending on the concrete hardware, adversaries may be able to modify the boot process of control devices such that these devices are able to boot from external sources. This requires physical control device manipulation. Another possibility is exploiting the boot sequence or exploiting the image verification.
Threat tree:	Figure 3.28
Example:	Adversaries may be able to use a USB-stick as an external boot medium.
STRIDE class:	T, D
Mitigated:	The system model in section 3.3 does not specify the concrete implementation of the boot sequences. Thus, it depends on the software implementation whether this threat is mitigated or not.
Known mitigation:	Don't allow booting from external interfaces like JTAG, USB or Ethernet.
Background information:	External dependencies: 9, 10, 11
Entry points:	4.2, 4.3, 4.4, 4.5, 5.x
Assets:	7, 8.1, 8.2, 8.3

Table 3.23: Threat: Modify executed software of control devices

ID:	20
Name:	Modify executed software of control devices
Description:	Software that is executed on control devices (i.e. the bootloader, the OS or the system services) can be modified by either exploiting the update mechanism, modifying the boot process or by modifying the system partitions.
Threat tree:	Figure 3.29

Table 3.23: (continued)

Example:	If adversaries are able to modify the executed software, the possibilities are immeasurable. In particular, because the affected control devices can either be controlled by the adversaries or they can participate in the regular fieldbus and IP traffic.
STRIDE class:	T, D
Mitigated:	yes
Known mitigation:	Keep the software up to date to prevent exploits and reduce the attack surface. Additionally, prevent the execution of files on the data partition as well as on the tempfs partition[4].
Background information:	External dependencies: 9, 10, 11
Entry points:	1.2, 2, 3.x, 4.x, 5.x
Assets:	8.x

Table 3.24: Threat: Exploit firmware

ID:	21
Name:	Exploit firmware
Description:	Several components of a control device are driven by a firmware. Because the firmware is highly connected with the hardware, it typically accesses the hardware at a lower level. Thus, adversaries that are able to exploit the firmware may be able to perform unexpected attacks that are unrelated to the utilized software.
Threat tree:	Figure 3.30
Example:	Exploiting firmware involves components like network interfaces (Wi-Fi, Ethernet, ...) as well as Graphics Processing Units (GPUs) or CPUs. Exploiting the firmware of network interfaces may be used to sniff all outgoing or incoming traffic. Other examples relating to CPUs are exploits like Meltdown [FR12] or Spectre [FR13], [FR14].
STRIDE class:	T, D
Mitigated:	The system model in section 3.3 does not specify the utilized hardware and the corresponding firmware. Thus, it depends on these components whether this threat is mitigated or not.

[4] A mount point without execution privileges can be achieved by the `noexec` mount flag in Linux [FR11].

Table 3.24: (continued)

Known mitigation:	Mitigation against firmware exploits can be a cumbersome task. In particular, if the firmware has been developed by a third party and no updates are provided. In this case, it is often unfeasible to fix known vulnerabilities. However, even if a firmware update is available, it may be hard or even impossible to update the firmware in the field.
Background information:	External dependencies: 9, 10
Entry points:	4.3
Assets:	1.4, 7

Table 3.25: Threat: Undesired issue of certificates

ID:	22
Name:	Undesired issue of certificates
Description:	Adversaries may be able to issue valid but undesired certificates. This can be achieved, if adversaries are able to get the confidential private key of the CA. Other possibilities are bypassing the authentication check or exploiting the certificate request service.
Threat tree:	Figure 3.31
Example:	In general, CAs issue certificates only if a previous authentication check was successful. However, adversaries may be able to bypass this authentication check and thus they may be able to get a valid certificate.
STRIDE class:	T
Mitigated:	yes
Known mitigation:	Make sure that the private key of a CA is kept secret. Additionally, get the latest security fixes for the software that issues these certificates.
Background information:	External dependency: 13, Implementation assumptions: 2, 3, 4
Entry points:	
Assets:	2.4

Table 3.26: Threat: Accept revoked or expired certificates

ID:	23
Name:	Accept revoked or expired certificates

Table 3.26: (continued)

Description:	Certificates contain an expiration date. Thus, they are only valid as long as the expiration date has not been passed. Thus, if control devices do not run with the current date and time, it is possible that expired certificates are accepted. Furthermore, already issued and still valid certificates can be revoked by CRLs. If control devices do not have the latest CRLs, it is possible that revoked certificates are accepted.
Threat tree:	Figure 3.32
Example:	If adversaries are able to prevent future updates of CRLs, revoked certificates would be accepted.
STRIDE class:	E
Mitigated:	yes
Known mitigation:	Propagate CRL as fast as possible to ensure that all control devices have the most recent CRL. Each control device should use date and time synchronization to provide the current local time to the software services. For a fail-safe synchronization, multiple time sources must be used (see section 3.3.2 for examples).
Background information:	External dependencies: 12, Implementation assumptions: 6
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.27: Threat: Install an undesired device

ID:	24
Name:	Install an undesired device
Description:	This threat requires physical access to device, wires or plugs and involves not only foreign devices that are installed in the network. It also includes cases in which adversaries use the expansion ports of control devices to install malicious devices. Once these devices are installed, adversaries may be able to damage other devices, or extract confidential information. In particular, if the communication is realized unencrypted.
Threat tree:	Figure 3.33
Example:	A malicious network sniffer can be installed. In particular, if wireless connections are used, this attack can be very promising for adversaries.
STRIDE class:	I, D
Mitigated:	The system model in section 3.3 does not specify the utilized hardware and infrastructure. Thus, it depends on these whether this threat is mitigated or not.

Table 3.27: (continued)

Known mitigation:	Reduce the number of expansion ports to a minimum or make them hard to access. Prevent the access to the network infrastructure to prevent installation of foreign devices in the same network. Use secure network protocols (see threat 25).
Background information:	Implementation assumption: 1, Security notes: 1, 3
Entry points:	1.x, 2, 4.5
Assets:	2.x

Table 3.28: Threat: Undesired participation in network communication

ID:	25
Name:	Undesired participation in network communication
Description:	Adversaries may be able to participate in the network communication (wired and wireless) if they are able to install a foreign device or they are able to modify or exploit the software that is executed on the control devices.
Threat tree:	Figure 3.34
Example:	Participating in the network communication can involve sniffing the network traffic as well as transmitting malicious packages or establishing connections to other control devices.
STRIDE class:	I, D
Mitigated:	yes
Known mitigation:	Use cryptographic protocols with certificates that ensure that connections are only established if the remote devices have a trusted certificate and that messages are transmitted encrypted and authenticated. Also take care of private keys and ensure that certificate requests are valid. Additionally, use anti-reply mechanisms in the communication protocols.
Background information:	External dependencies: 12, 13, Implementation assumptions: 1, 3, 4
Entry points:	1.x, 2, 4.5, 5.x
Assets:	2.4, 5.x

Table 3.29: Threat: Reverse engineer control device

ID:	26
Name:	Reverse engineer control device

Table 3.29: (continued)

Description:	This involves the hardware as well as the software. In particular, because adversaries may be able to reverse engineer the control device. This is possible, either if adversaries are able to get physical access to the device or if they are able to access or modify the executed software.
Threat tree:	Figure 3.35
Example:	Once adversaries have physical access to the device, they may be able to disassemble the hardware. Another example involves modifying or exploiting the running software to get access to the program code that is executed.
STRIDE class:	I
Mitigated:	The system model in section 3.3 does not specify the utilized hardware. Thus, it depends on the hardware whether this threat is mitigated or not.
Known mitigation:	Use persistent memory encryption to prevent reading of the program memory. Prevent physical access to device internals.
Background information:	
Entry points:	1.x, 2, 3.x, 5.x
Assets:	1.1, 3.x, 6.x, 8.x

Table 3.30: Threat: Undesired pass of the certificate check

ID:	27
Name:	Undesired pass of the certificate check
Description:	As the network communication relies on certificates, a valuable goal for adversaries may be a certificate check that is passed. Since the public key and the certificate are not required to be private, adversaries are able to pass the certificate check once they are in possession of the private key. Additionally, the certificate check may be exploited by adversaries.
Threat tree:	Figure 3.36
Example:	If the certificate check component in the software contains security flaws, adversaries may be able to pass the certificate check even if adversaries don't have a certified asymmetric key pair.
STRIDE class:	S, E
Mitigated:	Yes
Known mitigation:	Use strong asymmetric keys, don't reuse private keys, keep the private keys secret (e.g. by storing them into a TBM or authentication token). Use only well tested implementations of the certificate checking algorithms. Keep the software up to date.

Table 3.30: (continued)

Background information:	External dependency: 12, Implementation assumptions: 3, 4
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.31: Threat: Undesired establishing of secure communication

ID:	28
Name:	Undesired establishing of secure communication
Description:	To establish a secure communication, adversaries must be able to participate in the network communication and pass the certificate check. Only if these conditions are fulfilled, it is possible to establish a secure communication.
Threat tree:	Figure 3.37
Example:	An example may be a foreign device, that was installed by adversaries. This device uses a certified asymmetric key pair for authentication.
STRIDE class:	S
Mitigated:	Yes
Known mitigation:	Make it hard for adversaries to participate in the network communication. Keep the certified private keys secret (e.g. by storing them into a TBM or authentication token). Use only well tested implementations of the certificate checking algorithms. Keep the software up to date.
Background information:	External dependency: 12, Implementation assumptions: 1, 3, 4
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.32: Threat: Exploit message de-/encryption

ID:	29
Name:	Exploit message de-/encryption
Description:	Adversaries may be able to de- or encrypt the ciphered messages that are sent via the network. As mentioned in section 3.3.2.2, the symmetric encryption keys are derived via DH for each connection independently. However, adversaries may be able to exploit the encryption or key derivation algorithm. Additionally, they may be able to get the confidential encryption keys.
Threat tree:	Figure 3.38

Table 3.32: (continued)

Example:	Adversaries may be able to predict the derived keys, if the DH key agreement protocol has not been correctly implemented.
STRIDE class:	I
Mitigated:	yes
Known mitigation:	Derive new symmetric keys each time the connection is established. Change the symmetric keys after some time and ensure that these new keys do not relate to the previous keys (PFS). Keep these temporary keys in the RAM only, don't store them in the persistent memory. Once the connection is closed, delete all related symmetric encryption keys. Use only well tested implementations of the key agreement and encryption algorithms. Keep the software up to date.
Background information:	
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.1

Table 3.33: Threat: Exploit HMAC

ID:	30
Name:	Exploit HMAC
Description:	As mentioned in section 3.3.2.2, a HMAC is included in each transmitted message. Adversaries may be able to generate a valid HMAC for a particular message, if they are able to get the confidential signing key or if they can exploit the HMAC check.
Threat tree:	Figure 3.39
Example:	If the HMAC check contains some security flaws, adversaries may be able to exploit the HMAC checking such that altered messages get accepted.
STRIDE class:	S, T
Mitigated:	yes
Known mitigation:	Keep asymmetric private keys that are used to sign the messages private. Use only well tested implementations of the HMAC checking algorithms. Keep the software up to date.
Background information:	
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.34: Threat: Exploit secure communication

ID:	31
Name:	Exploit secure communication
Description:	To exploit a secure communication, two control systems must already have initialized a secure communication. After this step succeeded, adversaries may be able to exploit this communication. It requires that adversaries are able to participate in network communication as well as they are able to exploit the HMAC and the message de- or encryption.
Threat tree:	Figure 3.40
Example:	Adversaries may be able to install a foreign device between two communicating control devices such that they are able to participate in the network communication. If they are able to exploit the HMAC checks as well as the message de-/ and encryption, because they got the secret keys, they are able to de- or encrypt the transmitted messages and they can even send messages that would be accepted by both control devices.
STRIDE class:	S, T, I
Mitigated:	yes
Known mitigation:	Use well tested implementations of HMAC and (a)symmetric cryptography algorithms. Keep the software up to date to prevent exploits and reduce the attack surface.
Background information:	Implementation assumptions: 1, 3, 4
Entry points:	1.x, 2, 4.5, 5.x
Assets:	2.x, 5.x

Table 3.35: Threat: Flooding the network (DoS)

ID:	32
Name:	Flooding the network (DoS)
Description:	Flooding the network with useless or invalid packets may be a possibility for adversaries to drive a DoS attack. In particular, if adversaries have the control over a malicious device. This malicious device must be able to utilize an important fraction of the network capacity. In wireless networks also jammers fall in this category. Thus, it is not necessarily required to implement the network protocol in order to flood the targeted network.
Threat tree:	Figure 3.41
Example:	Adversaries may be able to highjack a network coupling device that can be used to send permanently packages.
STRIDE class:	D

Table 3.35: (continued)

Mitigated:	yes
Known mitigation:	Flooding the network is hard to mitigate. Even though, network coupling devices (like switches or routers) are not part of the modeled systems, they may be able to detect invalid packets and drop them before they are sent to the destination.
Background information:	External dependency: 6, Security note: 3
Entry points:	1.x
Assets:	2.3

Table 3.36: Threat: Undesired capture of transmitted packets

ID:	33
Name:	Undesired capture of transmitted packets
Description:	Even if the message payload is transmitted encrypted, adversaries may be able to derive important information from the unencrypted metadata that is contained in a regular packet.
Threat tree:	Figure 3.42
Example:	If adversaries are able to sniff the traffic between several control devices, they are able to get the sender, receiver or sequence number of the packets as well as the information how long the captured packet is. Furthermore, they may be able to check if the length or the destination of some packages relate to several environmental events.
STRIDE class:	
Mitigated:	yes
Known mitigation:	For this system model, the unencrypted metadata is no security issue. Make sure that no message is sent unencrypted.
Background information:	
Entry points:	1.x
Assets:	2.1, 2.2

Table 3.37: Threat: Undesired replay of captured packets

ID:	34
Name:	Undesired replay of captured packets

Table 3.37: (continued)

Description:	If adversaries are able to capture transmitted packets, they could also try to replay them later. As the system model already contains a sequence-based reply protection (see section 3.3.2.2) and this sequence number is protected by the HMAC, adversaries must be able to generate a new HMAC or exploit the HMAC check each time the sequence number has been changed.
Threat tree:	Figure 3.43
Example:	Adversaries may be able to capture a message that opens the door lock. Once the door lock is closed again, they may be able to reopen the door by sending the captured "open door" message again. If this sequence is a small integer (e.g. only a few bits wide) it is even possible that counter overruns occur more frequently. In this case, it may not be necessary that adversaries must calculate a new HMAC.
STRIDE class:	S, T
Mitigated:	yes
Known mitigation:	The message includes an incrementing sequence counter (with enough bit width) in each packet. And receivers don't accept consecutive messages that contain the same sequence counter values as previous messages. Because the sequence number is part of the HMAC calculation, each time the sequence is changed, also the HMAC must be recalculated.
Background information:	
Entry points:	1.x
Assets:	2.3, 2.4, 8.1

Table 3.38: Threat: Undesired decryption of captured packets

ID:	35
Name:	Undesired decryption of captured packets
Description:	If adversaries are able to capture transmitted packets, they could also try to decrypt them. This requires that message de- or encryption can be exploited.
Threat tree:	Figure 3.44
Example:	Adversaries may be able to get the secret symmetric encryption key. Then they are able to decrypt the messages as long as the symmetric encryption key is not changed.
STRIDE class:	I
Mitigated:	yes

Table 3.38: (continued)

Known mitigation:	Change the symmetric encryption key after some time and at least each time, a new connection is established (PFS) by a key agreement protocol. Such that captured keys can be used only for a few messages. If possible, store the temporary symmetric keys only in the volatile tempfs and not in the persistent memory partitions.
Background information:	Implementation assumptions: 3, 4
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.1, 2.2

Table 3.39: Threat: MITM

ID:	36
Name:	MITM
Description:	Adversaries may be able to perform a MITM attack (see section 2.3.4) if they are able to participate in the network communication and they are able to route the targeted traffic through a malicious device. By performing a MITM attack, adversaries are able to capture all packages that pass the malicious device. However, if adversaries are able to establish a secure communication, they may also be able to read or modify the transmitted messages.
Threat tree:	Figure 3.45
Example:	A possible target for adversaries is a dedicated Ethernet connection between two control devices. If they are able to connect a malicious device (can also be a malicious router or switch) in between these control devices, they are able to capture all traffic that traverses this connection.
STRIDE class:	S, R, I, D
Mitigated:	yes
Known mitigation:	Each control device identifies itself with a certificate. Thus, adversaries require a certified private key before they are able to communicate in a secure way. Additionally, it is possible to log if common communication endpoints have changed their certificates.
Background information:	External dependencies: 6, 12
Entry points:	1.x
Assets:	2.x

Table 3.40: Threat: Undesired drop of packets

ID:	37
Name:	Undesired drop of packets
Description:	This threat relates to the MITM threat 36. As it requires a device that acts in between two communicating control devices and selectively drops several packets or simply drops all packets.
Threat tree:	Figure 3.46
Example:	An example for this threat may be a malicious network coupling device (i.e. switch), that randomly drops packets that should be passed to other control devices.
STRIDE class:	R, D
Mitigated:	yes
Known mitigation:	Dropping random packets may be mitigated at higher protocol levels. Thus, the protocols must be designed such that dropped packages can be detected, such that the sender is able to resend the dropped packet.
Background information:	External dependency: 12
Entry points:	1.x
Assets:	2.4, 5.x

Table 3.41: Threat: Undesired altering of packets

ID:	38
Name:	Undesired altering of packets
Description:	This threat relates to the MITM threat 36. As it requires a device that acts in between two communicating control devices and that is able to alter several packets that are passing through. If adversaries aren't able to establish a secure communication and can't exploit a secure communication to regular control devices, the HMAC check would fail. However, if either establishing or exploiting a secure communication is possible, regular control devices may not notice the malicious MITM device.
Threat tree:	Figure 3.47
Example:	An example for this threat may be a malicious control device, that modifies packets which should be passed to other control devices.
STRIDE class:	S, T, R
Mitigated:	yes

Table 3.41: (continued)

Known mitigation:	Each control device identifies itself with a certificate. Thus, adversaries require a certified private key before they are able to communicate in a secure way. Additionally, it is possible to log if common communication endpoints have changed their certificates.
Background information:	External dependency: 12, Implementation assumptions: 3, 4
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.42: Threat: Undesired sending of new packets

ID:	39
Name:	Undesired sending of new packets
Description:	If adversaries are able to establish or exploit a secure communication to regular control devices, they may be able to send packets that are accepted by regular control devices. And these regular control devices may not notice that these messages originate from a malicious device.
Threat tree:	Figure 3.48
Example:	An example involves adversaries that are able to send particular commands to regular control devices.
STRIDE class:	S
Mitigated:	yes
Known mitigation:	Each control device identifies itself with a certificate. Thus, adversaries require a certified private key before they are able to communicate in a secure way. Additionally, it is possible to log if common communication endpoints have changed their certificates.
Background information:	External dependency: 12, Implementation assumptions: 3, 4
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.4, 5.x

Table 3.43: Threat: DoS by generating high crypto workload

ID:	40
Name:	DoS by generating high crypto workload

Table 3.43: (continued)

Description:	In particular, asymmetric cryptography like the DH key agreement protocol can lead to an increased computational effort [48]. Thus, adversaries may be able to enforce the target to do these expensive computations.
Threat tree:	Figure 3.49
Example:	One example involves DoS by permanent connection initialization requests. As the initialization requests require the generation of DH parameters, this may be a valuable target. In particular, if many consecutive requests are sent.
STRIDE class:	D
Mitigated:	yes
Known mitigation:	Allow only a certain number of connection initialization requests in a particular time period. Provide the possibility to ignore these requests from particular devices (blacklist). Even though, keep in mind that this can still lead to DoS, as long as many connection initialization requests are performed.
Background information:	
Entry points:	1.2, 5.13, 5.9, 5.11, 5.12
Assets:	2.3, 2.4, 9

Table 3.44: Threat: Breaking control device functionality

ID:	41
Name:	Breaking control device functionality
Description:	Breaking the functionality of a particular control device can have a big impact to the complete SCCPS. This can be achieved in several ways: One obvious way involves physical manipulation, as well as unplugging the power supply or battery and installing malicious devices that prevent regular operation of the control device. Additionally, if adversaries may be able to exploit the firmware or use a system service that can be exploited, or they are able to modify the software that is executed.
Threat tree:	Figure 3.50
Example:	"USB Killer" is an example for a malicious device that can be used on a USB port and can destroy the device or the port it is plugged in: It pulls electricity out of the USB port and stores it in capacitors. If the voltage on the "USB Killer" reaches a certain level it sends the stored electrical energy in a single burst back to device [45]. Another example are adversaries that try to break the control device by applying physical damage.

Table 3.44: (continued)

STRIDE class:	D
Mitigated:	The system model in section 3.3 does not specify the concrete device and system installation. Thus, it depends on the installation whether this threat is mitigated or not.
Known mitigation:	Keep the software up to date to prevent exploits and reduce the attack surface.
Background information:	External dependencies: 1, 2, 3, 5, 7
Entry points:	1.x, 2, 3.x, 4.x, 5.x
Assets:	9

3.4.10 threat trees

This section contains the threat trees for the threats in section 3.4.9.

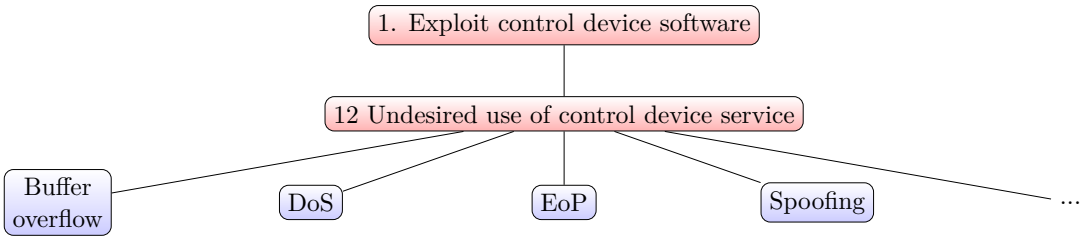


Figure 3.10: Threat tree to threat 1: Exploit system software

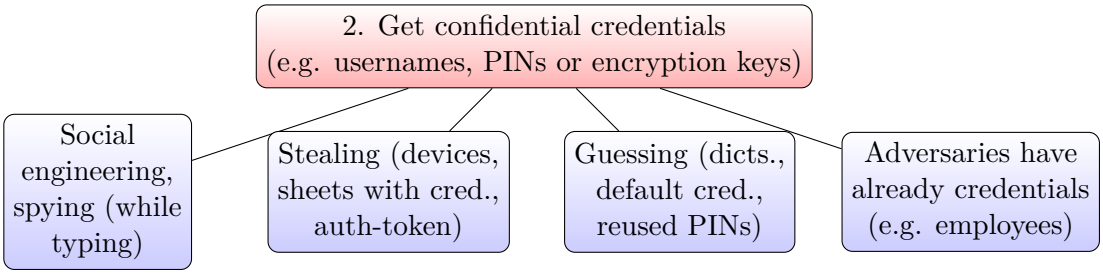


Figure 3.11: Threat tree to threat 2: Get confidential credentials

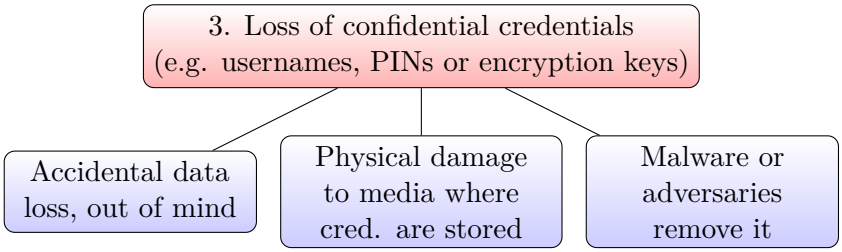


Figure 3.12: Threat tree to threat 3: Loss of confidential credentials

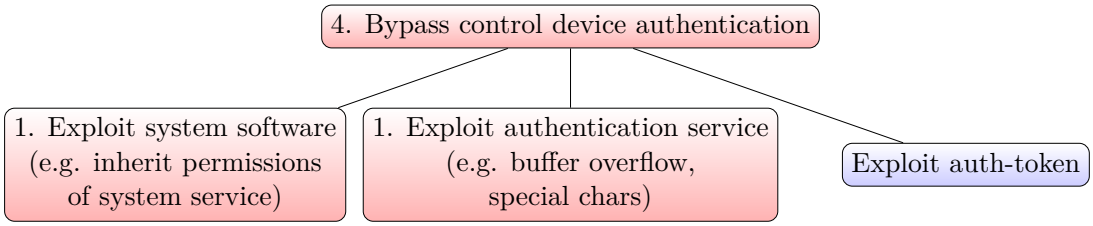


Figure 3.13: Threat tree to threat 4: Bypass control device authentication

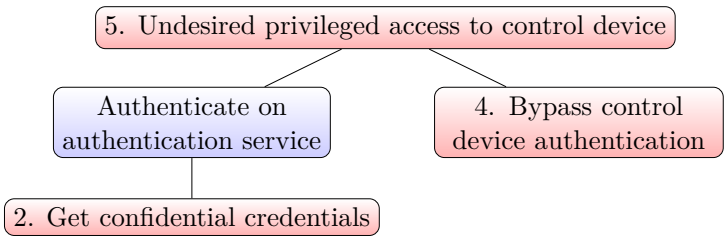


Figure 3.14: Threat tree to threat 5: Undesired privileged access to control device

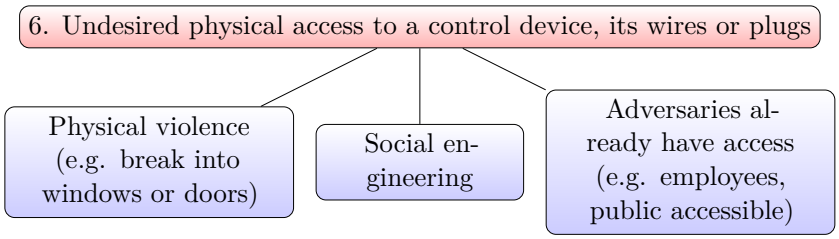


Figure 3.15: Threat tree to threat 6: Undesired physical access to a control device, its wires or plugs

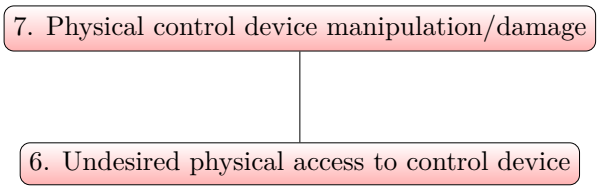


Figure 3.16: Threat tree to threat 7: Physical control device manipulation/damage

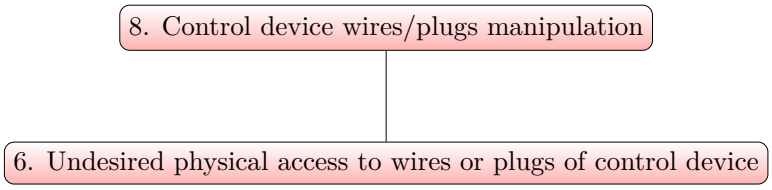


Figure 3.17: Threat tree to threat 8: Control device wires/plugs manipulation

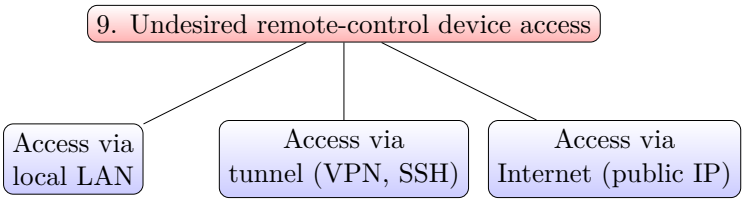


Figure 3.18: Threat tree to threat 9: Undesired remote-control device access

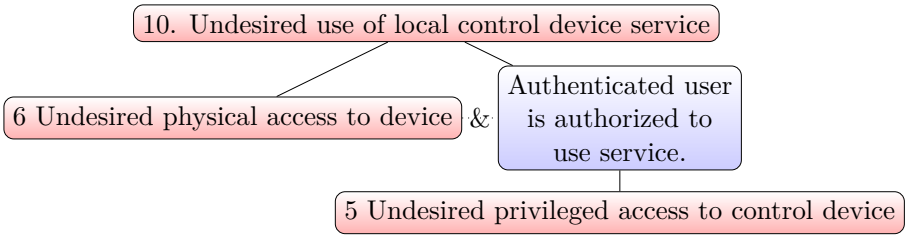


Figure 3.19: Threat tree to threat 10: Undesired use of local control device service

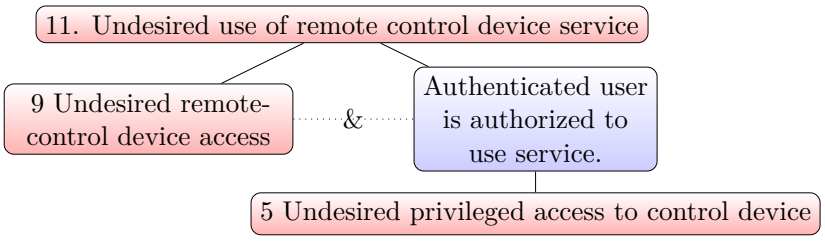


Figure 3.20: Threat tree to threat 11: Undesired use of remote control device service

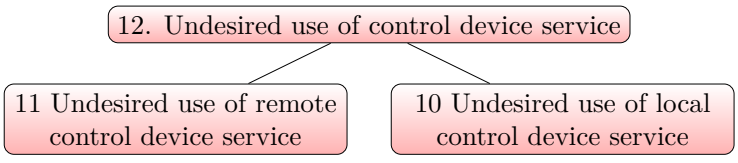


Figure 3.21: Threat tree to threat 12: Undesired use of control device service

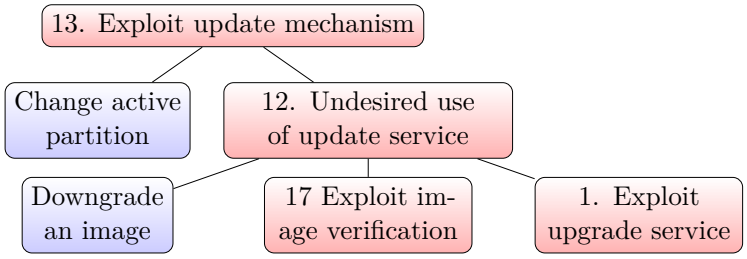


Figure 3.22: Threat tree to threat 13: Exploit update mechanism

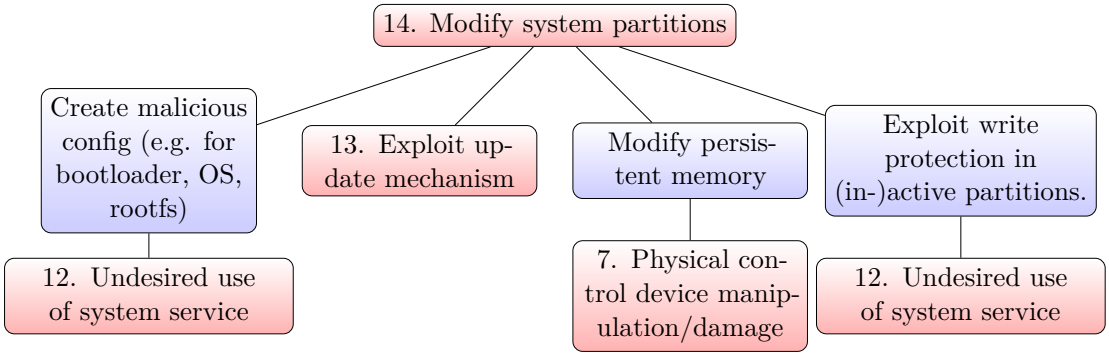


Figure 3.23: Threat tree to threat 14: Modify system partitions

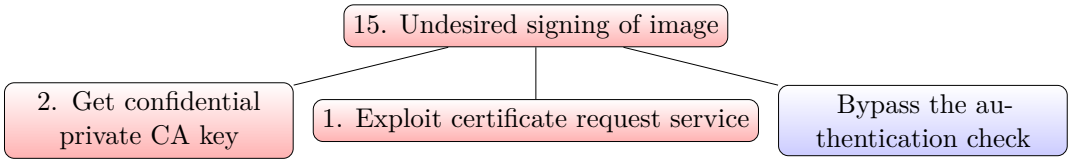


Figure 3.24: Threat tree to threat 15: Undesired signing of image

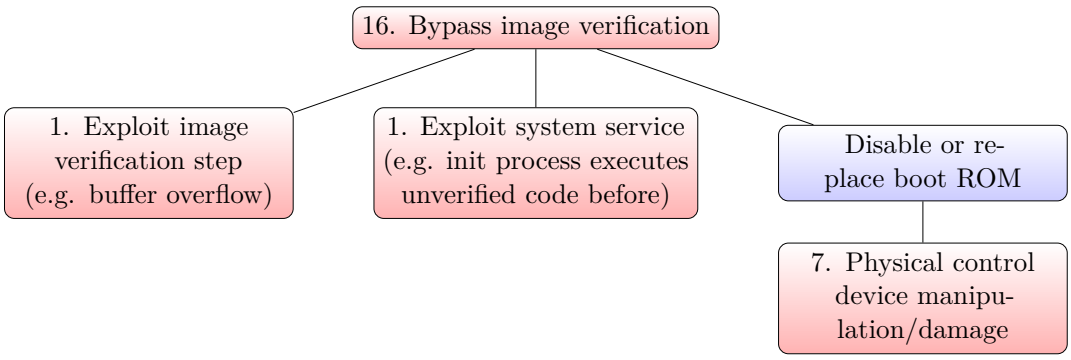


Figure 3.25: Threat tree to threat 16: Bypass image verification

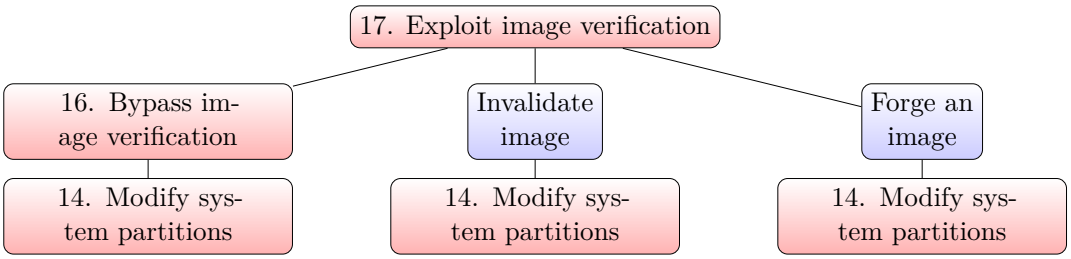


Figure 3.26: Threat tree to threat 17: Exploit image verification

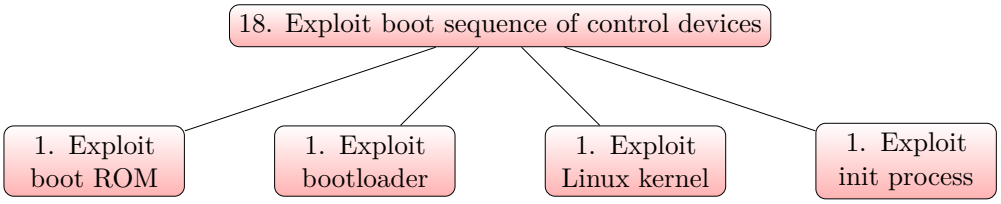


Figure 3.27: Threat tree to threat 18: Exploit boot sequence of control devices

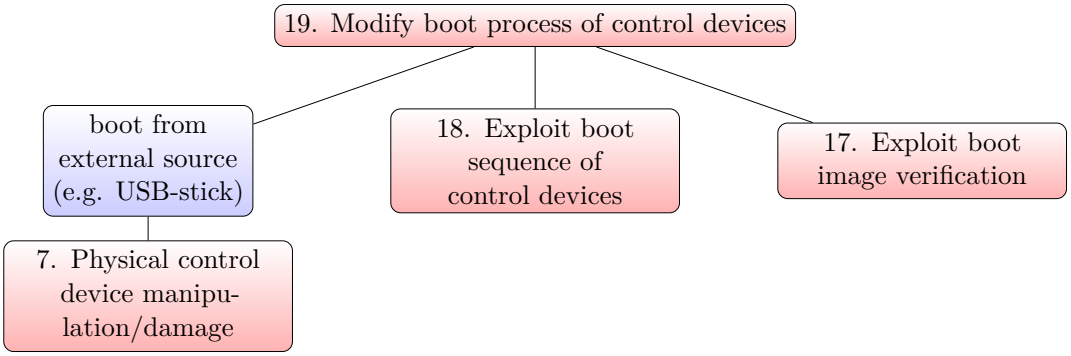


Figure 3.28: Threat tree to threat 19: Modify boot process of control devices

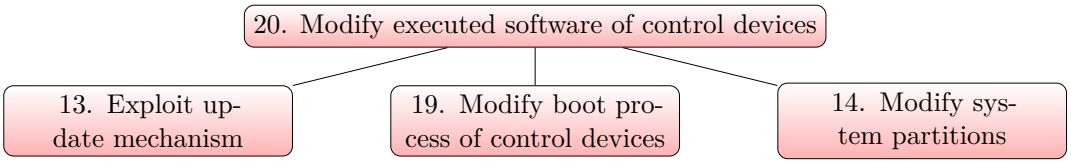


Figure 3.29: Threat tree to threat 20: Modify executed software of control devices

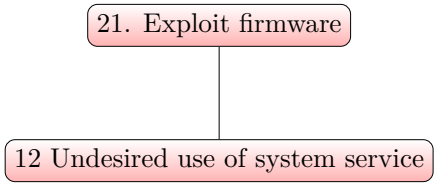


Figure 3.30: Threat tree to threat 21: Exploit firmware

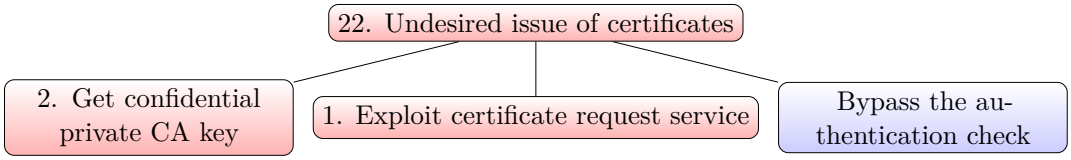


Figure 3.31: Threat tree to threat 22: Undesired issue of certificates

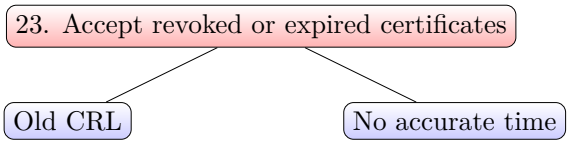


Figure 3.32: Threat tree to threat 23: Accept revoked or expired certificates

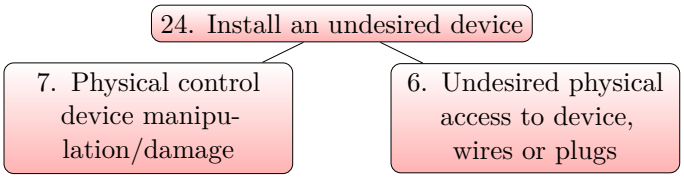


Figure 3.33: Threat tree to threat 24: Install an undesired device

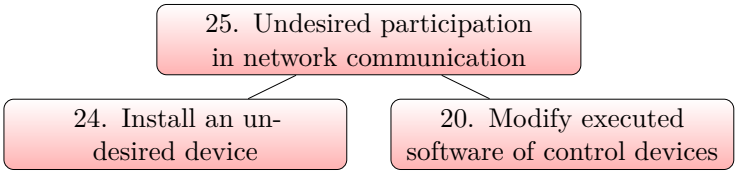


Figure 3.34: Threat tree to threat 25: Undesired participation in network communication

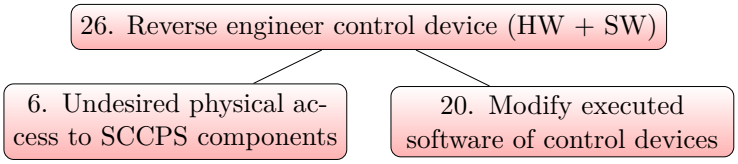


Figure 3.35: Threat tree to threat 26: Reverse engineer control device

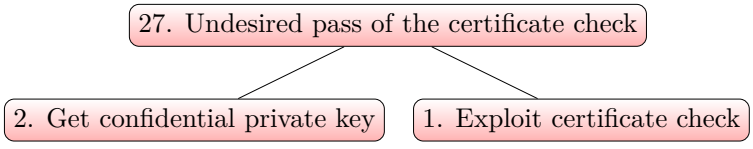


Figure 3.36: Threat tree to threat 27: Undesired pass of the certificate check

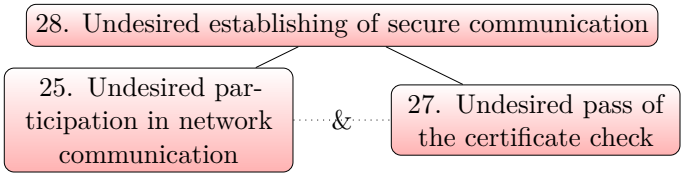


Figure 3.37: Threat tree to threat 28: Undesired establishing of secure communication

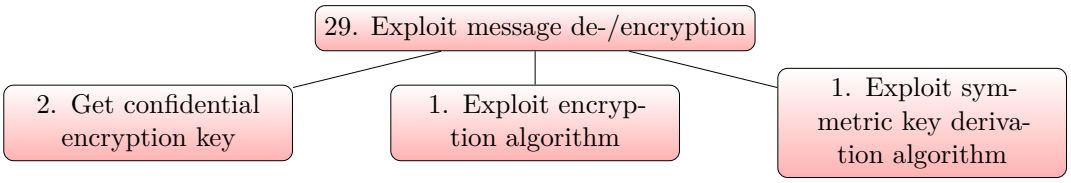


Figure 3.38: Threat tree to threat 29: Exploit message de-/encryption

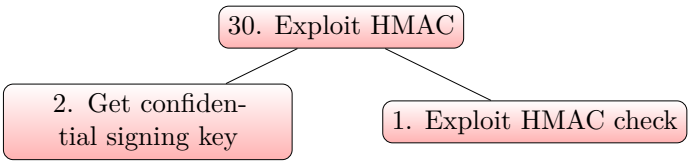


Figure 3.39: Threat tree to threat 30: Exploit HMAC

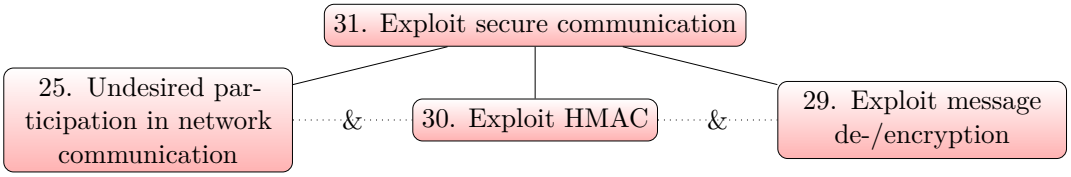


Figure 3.40: Threat tree to threat 31: Exploit secure communication

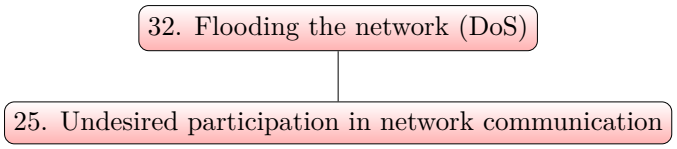


Figure 3.41: Threat tree to threat 32: Flooding the network (DoS)

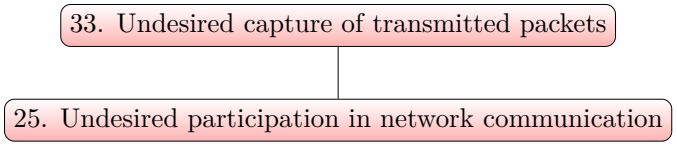


Figure 3.42: Threat tree to threat 33: Undesired capture of transmitted packets

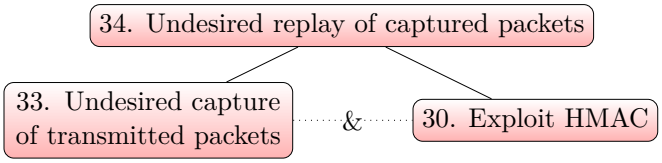


Figure 3.43: Threat tree to threat 34: Undesired replay of captured packets

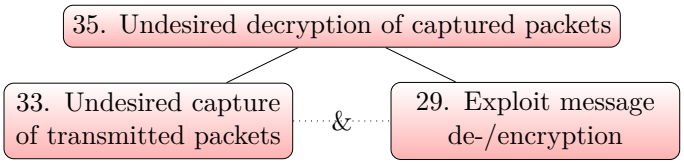


Figure 3.44: Threat tree to threat 35: Undesired decryption of captured packets

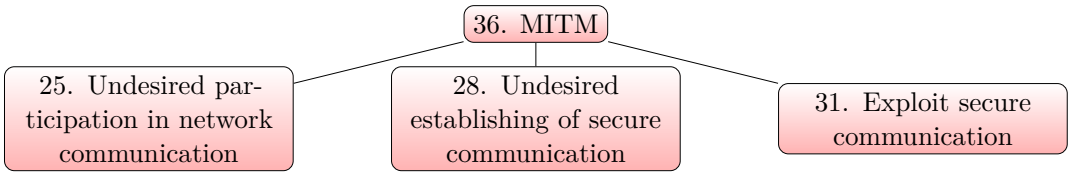


Figure 3.45: Threat tree to threat 36: MITM

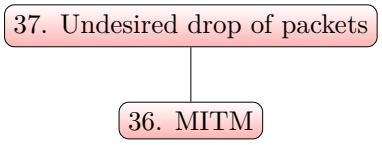


Figure 3.46: Threat tree to threat 37: Undesired drop of packets

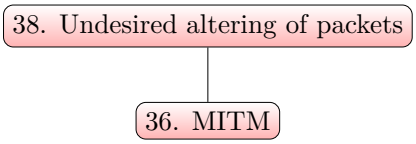


Figure 3.47: Threat tree to threat 38: Undesired altering of packets

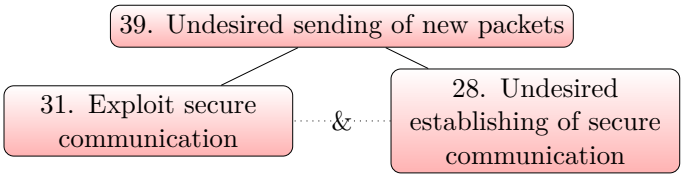


Figure 3.48: Threat tree to threat 39: Undesired sending of new packets

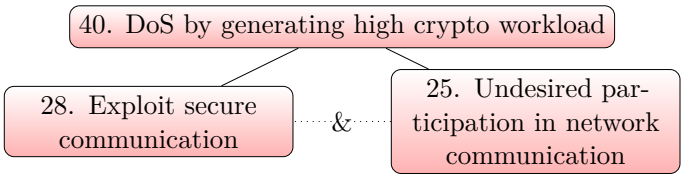


Figure 3.49: Threat tree to threat 40: DoS by generating high crypto workload

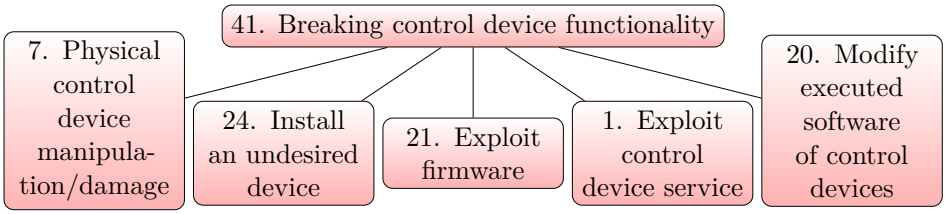


Figure 3.50: Threat tree to threat 41: Breaking control device functionality

CHAPTER 4

Design

In this chapter, several approaches to harden the control devices of a SCCPS in terms of security, reliability, integrity and confidentiality are discussed: The first approach (section 4.1) introduces a trusted booting mechanism for the control devices of a SCCPS. This ensures the integrity of the software that is executed on the device [79]. In section 4.2, an approach for a secure communication channel between all involved control devices as well as between a control device and a particular cloud service is presented. Finally, a secure update mechanism for control devices is introduced in section 4.3.

4.1 Trusted boot approach

For a SCCPS, it is very important to restrict the software that is executed on the involved devices. This is especially important as the purpose of embedded devices (in particular processors) is the execution of any installed software. Thus, the goal of the trusted boot concept is that nobody is able to alter the software that is executed on these devices. In particular, this concept is often also referred to as a secure booting mechanism [62].

The trusted boot mechanism must be designed in a way, that future software updates are supported. Thus, it is not sufficient to store the software in a ROM which can't be modified later. Instead, later software updates should be applicable in the future (see section 4.3).

4.1.1 The boot ROM and the TBM forms a hardware RoT

The trusted boot approach relies on the trusted boot introduction that is given in section 2.5.3.3. Additionally, it is based on the system model in section 3.3 and uses the findings of the threat model in section 3.4. Therefore, control devices must have a hardware-based TBM and a boot ROM, that both form the RoT. It must be implemented in hardware because hardware-based RoTs are harder to attack [79]. In order

to use the trusted boot concept, it is required that the TBM is supported by the boot ROM. Thus, this boot ROM is not only responsible for loading the first instructions of the bootloader into the SRAM (see section 2.6.1.2), it is also responsible for verifying the bootloader image. It is not strictly required that the TBM and the boot ROM are two distinct elements. Since both relate together it is no problem if both are implemented in the same component as long as this hardware RoT is able to verify its own integrity as well as the integrity of other software components [79]. In order to mitigate threat 16, the boot ROM as well as the TBM should be part of the SoC.

The image verification approach is a digital signature-based check (see section 2.3.3.1). In order to boot a specific image, it requires that a suitable signature is generated by trusted entities (like the device manufacturer or maintenance contractors). To create a valid signature, it is required to generate a hash value of the boot image. Next, this hash value is encrypted by the private key of a trusted entity. This encrypted hash forms the digital signature of the image. As this signature is required at boot up, it must be stored beside the boot image in the persistent memory (mitigates threat 15).

In order to verify the signature of a specific image, the hash of the complete image must be computed. Then the corresponding public key of the trusted entity must be used to decrypt the signature (which is stored beside the image). Only if the decrypted signature matches the computed hash of the image, it is guaranteed that the image has not been modified (mitigates threat 14).

Therefore, all trusted public keys that are able to sign a boot image must be available at boot up. The hardware RoT must be capable of handling these public keys. This requires that additional cryptographic data must be stored in the hardware RoT in a way that no one is able to change it. For redundancy reasons, it must be possible to store the required data for two independent public keys in the hardware RoT. Thus, the boot ROM and the TBM should be either one-time programmable (OTP) or individually produced with the required data (like MROMs). As two or more public keys must be supported by the hardware RoT, it must also be possible to revoke some of these static keys in the hardware RoT permanently (e.g. from local or remote maintenance users). This revocation feature is helpful if secret private keys have been revealed, as adversaries may be able to use these revealed keys to create valid signatures. In order to achieve these requirements for a hardware RoT, several approaches exist. Some of them are proposed here:

Store trusted keys: An obvious approach involves a TBM that is able to store a trust list (see section 2.3.4.2) with up to n public keys from trusted entities ($n > 1$). Thus, if many large keys must be stored, this approach requires much space in the TBM. To revoke one of these public keys, it must be possible to disable one of them. This can be achieved by field programming fuses that permanently disable a specific key slot.

Hash of trusted keys: This approach does not store the actual trust list in the TBM. Instead, a hash over a list of n trusted keys is generated and stored in the TBM

($n > 1$). As this hash over n public keys typically doesn't require that much space that would be required if these n public keys are stored directly in the TBM, more space is saved. As a drawback of this approach all n supported public keys must be stored alongside the image because the hardware RoT must be able to calculate the hash over all these keys at boot up. If this calculated hash value corresponds with the hash value stored in the TBM, it is guaranteed that the same set of trusted public keys has been used. Of course, this approach supports the same method as before to disable a specific key slot. However, it must be kept in mind that even if a public key is revoked, this key must still be stored alongside the image. Otherwise, the hardware RoT will not be able to generate the required hash at boot up anymore.

The hash of trusted keys approach is visualized in figure 4.1: The persistent memory contains the boot image, the signature and a set of trusted public keys. The hash of these public keys is compared against the hash in the TBM. If both hashes match, the next step is the verification of the boot image signature. As the signature uses key slot 2, it must be verified against "Public key 2". If the signature verification succeeds, the boot image is executed.

CA based approach: Another approach uses a CA. This CA creates a certificate for each public key of the trusted image signing entities. Thus, only the key of this signing CA must be stored in the TBM. In order to verify the image signature, the certificate as well as the actual public key must be stored alongside the image. In order to revoke one of the certified public keys, it is also required to store a list of revoked public keys or their hashes in an OTP memory. Once a specific public key appears there, it is not trusted anymore, even if the CA issued a valid certificate for this key. Thus, this approach is much more extensive than simply burning field programmable fuses.

In general, all these approaches can be used to provide a hardware RoT. As long as there do not exist any known security flaws in the TBM or boot ROM. Thus, users of the system can unconditionally trust the hardware RoT.

4.1.2 Trusted Linux boot process

Another requirement for control devices is that each must consist of hardware that is capable of executing a Linux system. The Linux start procedure is explained in detail in section 2.6 and consists of four stages: 1. Initial booting sequence, 2. starting the Linux kernel, 3. starting the init process and 4. starting the system services. For this trusted boot approach, the procedure is similar. The DFD in figure 3.7 already gives an overview of the proposed trusted Linux boot sequence. As all boot partitions are stored on the device, it is not required to boot from external interfaces like JTAG, USB or Ethernet. Thus, it is not allowed to boot from these interfaces (mitigates threat 19).

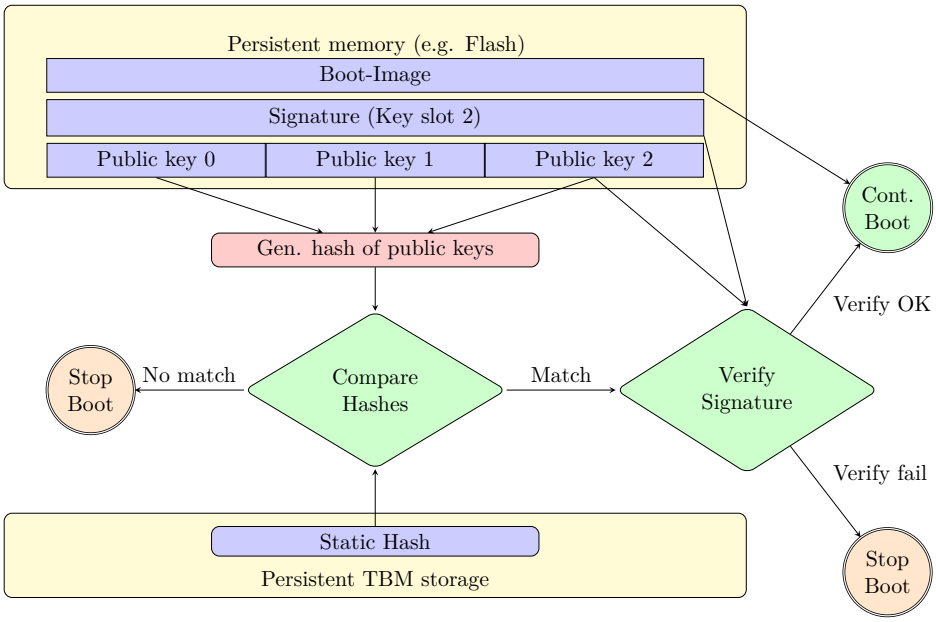


Figure 4.1: Simplified visualization of the "hash of trusted keys" boot image verification approach from section 4.1.1.

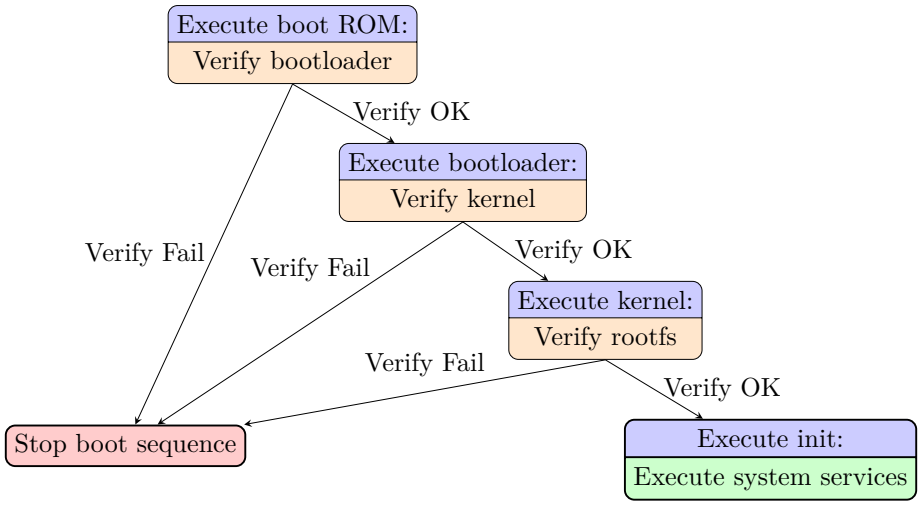


Figure 4.2: Phases of the trusted Linux boot process.

Additionally, the boot phases are visualized in figure 4.2 similar to the trusted boot DFD in chapter 3:

1. Once the user powers on the device, the first phase is started. It consists of the static boot ROM (incl. TBM) and the bootloader which is stored in the persistent memory. The unconditionally trusted boot ROM and the TBM verify their own integrity and the integrity of the bootloader. If all these verifications succeed, the next step (i.e. the bootloader) is started.

2. The previously verified bootloader verifies the Linux kernel image and starts it, if the verification succeeds.

3. The previously verified Linux kernel verifies the root file system and starts the init process, if the verification succeeds.

4. The init process in the verified rootfs starts all required services in the rootfs.

As mentioned in section 3.3.2.1, the bootloader, the kernel partition as well as the rootfs partition are used read only. These partitions can only be modified during a regular software update procedure (see section 4.3).

As the first boot phase is trusted unconditionally, the next boot phases can be verified and executed using the first boot phase. Additional boot phases can be executed if they have been verified successfully in previous boot phases. Thus, this concept is also called chain of trust, as each phase checks the integrity of the next phase before the execution. If these checks are implemented correctly, this chain ensures that the code in execution can be trusted as it has not been tampered [124]. Special care has to be taken on each check since the whole trusted boot mechanism relies on the strength of each integrity check. If one of these components contains security flaws, the whole chain may be broken.

Since each of these verification checks is unrelated to the other verification checks, it is not required that all components (bootloader, kernel and rootfs) are signed with the same key. Hence, the public key that is required for the next boot step can be included in the current step. This means that the bootloader contains the public key to verify the kernel image, and the kernel image itself contains the public key to verify the rootfs. Furthermore, in contrast to the hardware RoT which has to support several public keys at the same time, the bootloader image and the kernel image may contain only one public key. Thus, if a key must be revoked, it is sufficient to apply a software update with kernel and rootfs images that were signed by other private keys.

If one of these verifications fail, the complete boot process must be interrupted. This interruption is required even if the device becomes non-operational. As the possible severity of executing unverified software is supposed to be higher than a non-operational device. This must be kept in mind for the software update mechanism (see section 4.3) as this mechanism must ensure that updated software always passes the integrity checks

that are performed at boot up. Thus, if appropriate hashing and cryptography algorithms are used and the private keys remain private it is impracticable for adversaries to modify the content of a specific image (bootloader, kernel or rootfs) without invalidating the complete image.

Some SoC solutions provide convenient software debugging interfaces (like JTAG). Even though this interface can be very useful during software development, it is important that this interface is disabled in operational environments. Otherwise, adversaries may be able to use this debugging interface to manipulate the program flow and bypass the image verification checks.

4.1.3 Symmetric boot images

Because the proposed software update mechanism relies on a symmetric image approach (see also section 4.3.1 and figure 3.2), the kernel and the rootfs are stored twice in the persistent memory. Thus, as mentioned in section 2.7, the bootloader controls which of both redundant kernel images should be started: If both kernel partitions contain the same software version, it does not matter, which partition is selected. However, if one partition contains a newer software version, because a software update was recently installed, this partition must be selected. Additionally, if the bootloader detects that the verification of one kernel image fails, it has the possibility to select the second kernel image (even if its software version is older). Therefore, as long as the integrity check of one kernel image succeeds, the bootloader is able to start a kernel. The same approach can be used for the rootfs, as the rootfs is stored twice in the persistent memory. This means, that the kernel checks the integrity of one rootfs image and if the check fails, it verifies the second rootfs image. Thus, as long as the verification of a rootfs image succeeds, it can be mounted and the init process can be started. This procedure is also shown in the DFD figure 3.7: The bootloader selects which kernel image should be used and the active kernel selects, which rootfs partition should be used (mitigates threat 13). Therefore, there exist four different ways to boot the system (also shown in figure 3.7).

The approach can also help in cases in which the software update process fails (e.g. because of power loss) and leaves an invalid partition behind. However, it also helps in cases in which adversaries were able to modify the content of a specific partition. Thus, it is possible that a control device remains operational even if the verification of a kernel or rootfs partition fails.

4.1.4 Encrypted boot

Even though trusted boot reduces the possible attack surface, it does not help against threat 26. Thus, if the confidentiality of the complete boot process (involving the bootloader, the kernel and the rootfs) is another mandatory requirement, the hardware RoT of each control device requires additional facilities. The hardware RoT (and in particular the TBM) of a control device must provide at least the possibility to store a confidential key. This key can either be a symmetric or an asymmetric private key. As long as the

key itself is kept private. Thus, it should in general not be possible to access this key from any application. Instead, the key should only be available in the hardware RoT via a well-defined cryptography interface:

- If confidential symmetric keys are used, the hardware RoT interface must provide the possibility to encrypt and decrypt a specific data blob with the confidential key.
- If confidential asymmetric private keys are used, the hardware RoT interface must provide the possibility to decrypt a specific data blob with the confidential private key. Additionally, the public key to the confidential private key must be available.

Furthermore, it is very important that each device contains its own private key. Otherwise, if the key gets leaked, adversaries may be able to reuse the key on several devices. Thus, it is not sufficient, if the same key is distributed over several control devices. Instead, either at the chip fabrication step or during the first installation of the control device a random key must be generated and used as confidential key. A possibility may be an OTP memory in the hardware RoT that is programmed at the chip manufacturing process.

4.1.4.1 Software updates

In particular, for the software update deployment it would be a challenging task, if each control device gets a distinct update image. Thus, intermediate keys are used: In figure 4.3, this concept is visualized. The intermediate key IK is a symmetric key that is used to decouple the device specific keys (AK or BK) from the universal applicable boot images. That implies, that all encrypted images or encrypted data that is included in an update package can be encrypted by the same symmetric intermediate key IK. Therefore, the intermediate key IK must not be stored in plaintext in the persistent memory. Thus, the intermediate key IK is encrypted by the confidential device specific keys (AK or BK) and stored on the machine. Typically, these device specific keys (AK or BK) are stored in the hardware RoT on the device: Thus, if these device specific keys are asymmetric private keys, the intermediate key is encrypted by the public key. If these keys are symmetric keys, the intermediate key may be encrypted by a cryptographic interface of the hardware RoT. (In order to prevent anyone to get access to this confidential key.) Finally, this encrypted intermediate key may be stored on the device or even in the update package (similar to the signatures in section 4.1.1).

In general, the same intermediate key can be used for several devices. Thus, in case of software updates, the same images can be distributed over several control devices. However, if this intermediate key gets leaked, adversaries may be able to encrypt the boot image of several devices. Therefore, the intermediate key should be changed at least for each software update. In figure 4.4, the encrypted boot concept with intermediate keys is visualized: The persistent memory contains the encrypted boot image and the

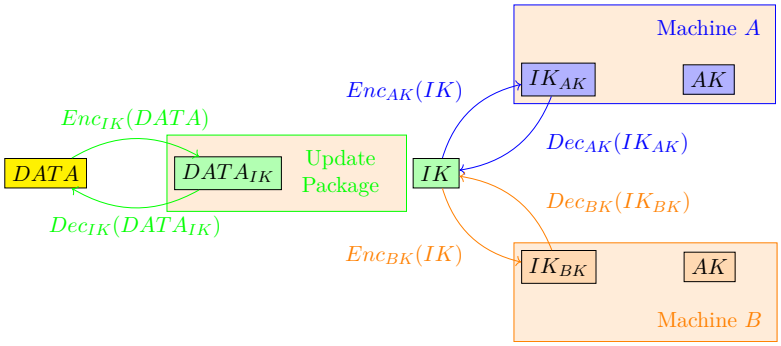


Figure 4.3: Simplified visualization of the intermediate keys concept that is used for an update package (see section 4.1.4.1).

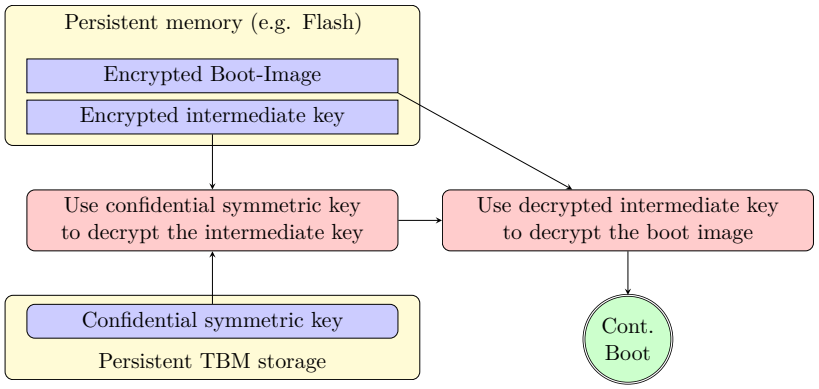


Figure 4.4: Simplified visualization of the encrypted boot approach (see section 4.1.4).

encrypted intermediate key. The intermediate key gets decrypted by the cryptography interface of the hardware RoT. Next, this key is used to decrypt the encrypted boot image. Even though, this encrypted boot concept must not be used without the trusted boot, in this figure it is omitted, to keep the image simpler.

4.1.4.2 Encrypted Linux boot sequence

The encrypted Linux boot sequence is very similar to the trusted Linux boot sequence and relies on it: Each component (bootloader, Linux kernel and rootfs) is encrypted. The boot ROM verifies and decrypts the bootloader image. The verified and decrypted bootloader image contains the public keys to verify the Linux kernel as well as a symmetric key to decrypt the encrypted Linux kernel image. Thus, the bootloader is able to verify and decrypt the Linux kernel. The Linux kernel image also contains public keys and the encryption key to verify and decrypt the encrypted rootfs. When the Linux kernel is started, it uses these public keys to check the signature and the encryption key to decrypt the rootfs. This process is visualized in figure 4.5.

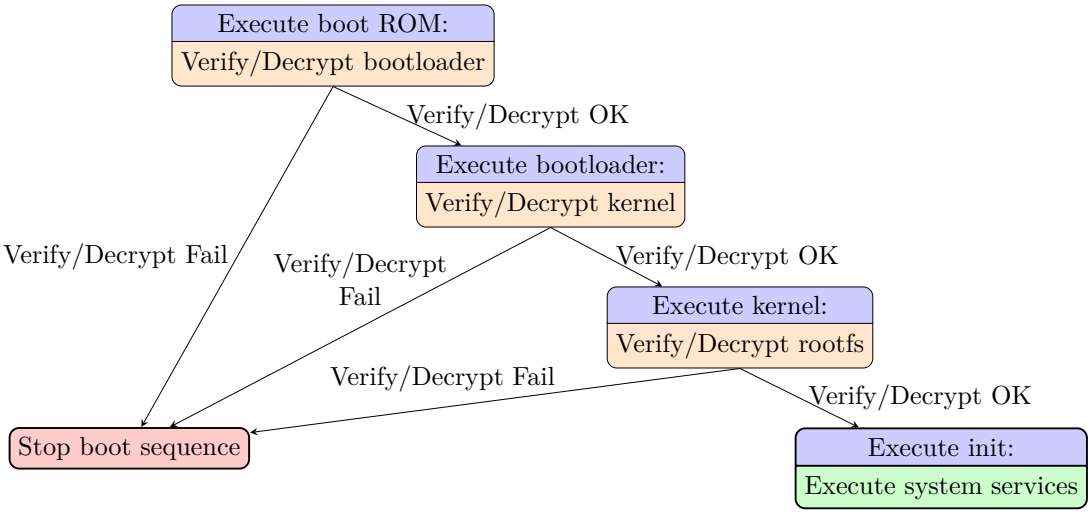


Figure 4.5: Phases of the encrypted Linux boot process.

4.1.5 Requirements summary

The following list summarizes the requirements for the proposed trusted boot approach:

1. The hardware RoT acts as a security anchor and consists of boot ROM and TBM.
2. The hardware RoT must be capable of trusted boot. Additionally, it must be able to store at least two public signing keys that are required to verify the signature of the bootloader.
3. Once the trust list (required to verify the signatures) is stored in the hardware RoT, it should not be possible to change the keys anymore.
4. The trust list must be accessible from the hardware RoT (i.e. it is stored in the persistent memory or in the TBM).
5. The image signature must be stored beside the boot image in the persistent memory.
6. It must be possible for local or remote maintenance users to revoke at least one entry in the trust list (that is stored in the hardware RoT).
7. The bootloader image and the kernel image must be capable of trusted boot. This requires that they are able to verify the next boot step by a public key that is contained in the bootloader and in the kernel image.
8. The boot ROM must be capable of starting a bootloader, which itself must be able to start the Linux kernel.

9. If the integrity checks of the software fail, the system should stop the start sequence.

10. A symmetric image layout of the kernel image and the rootfs image is required. Thus, the bootloader and the kernel must be able to select one of both partitions (newer software versions should be preferred). Additionally, if the verification of an image fails the bootloader and the kernel should try to start the redundant partitions.

11. The software version of the kernel image should be stored in a way such that the bootloader is able to detect the version easily. The same rule applies for the rootfs such that the kernel is able to detect the version.

12. Debug interfaces like JTAG as well as external boot interfaces like USB or Ethernet must be disabled.

13. The boot ROM as well as the TBM should be part of the SoC.

Additionally, if encrypted boot is required beside the trusted boot mechanism, the following list summarizes the additional requirements:

1. The confidential private keys (required for the bootloader decryption) must be accessible from the hardware RoT. It should not be possible to access this key from the application as the key should be stored in the TBM.

2. The hardware RoT must provide a well-defined interface to decrypt (and encrypt) a specific data blob using the confidential private key that is stored in the TBM.

3. Each device gets its own confidential private key.

4. The encrypted intermediate key must be stored beside the image (just like the image signature).

5. The bootloader image and the kernel image must be capable of encrypted boot. This requires that they are able to decrypt the next boot step by a symmetric key that is contained in the bootloader/kernel image.

4.2 Secure communication approach

As a SCCPS consists of several control devices, it is required that these devices are able to communicate in a secure manner. Additionally, some control devices also have direct Internet access. This Internet access is primarily used to access cloud services. Thus, the goal of the secure communication concept is that each traffic between two control devices as well as the communication between control device and cloud services is confidential and each participant is required to authenticate. Furthermore, the integrity of each sent packet must be guaranteed.

The secure communication approach relies on the communication model defined in section 3.3.2.2. Thus, the control devices in the SCCPS are connected in a mesh topology and communicate with each other in an IP (either IPv4 or IPv6) network. IP was selected because of its large distribution (since all Internet traffic is sent via IP [52]) and its increasing importance in automation systems and CPS (see section 2.1.1). It is not required that all devices are connected via Ethernet, Fiber or Wi-Fi. In particular, because in section 5.2.1 a concept is introduced which can be used to drive the IP protocol over EIA-485, EIA-232 or other networks. Furthermore, each control device as well as the cloud service require a static IP. Due to this static configuration there are neither external DHCP nor DNS services required to establish a M2M connection. Instead, the actual IP addresses must be configured during the system deployment.

In section 2.4.2, several security protocols that work on top of an IP layer are introduced. An obvious difference between these security protocols is the layer in which they operate on. For this secure communication approach, an additional security layer for secure communication is introduced on top of the Internet layer. This security layer takes advantage of the addressing, networking and routing facilities of the IP protocol. Another benefit of a security layer below the transport layer is that a system service or application is able to rely on secure connections to other devices transparently just by opening a regular network socket. E.g. in Linux systems, the IP communication up to layer 4 is managed by the networking subsystem [57]. Hence, this secure connection can be used from all user space applications that are capable of regular IP traffic without considering any authentication or encryption properties. All traffic to and from each control device gets transparently encrypted and authenticated by this additional security layer. Concepts that rely on TLS transfer the responsibility for establishing a secure connection to the application. Thus, the application must consider authentication or encryption properties on its own (see also figure 2.7).

4.2.1 Session based communication

As shown in the DFD figure 3.8 and explained in section 3.4.8, the secure communication approach is based on sessions. Thus, before any encrypted packets can be sent, it is required to initialize a secure session. This can be achieved by a system service that is responsible for the secure communication and executed at system startup. In order to initialize a secure session, it is required to send at least four messages (see figure 3.8). In these four messages, the parameters of the session are determined and the communication endpoints are authenticated (mitigates threat 25). Once this system service has successfully initialized a session with the communication partner, the applications are able to communicate confidentially and authenticated via these sessions (see figure 3.8). To distinguish between several concurrent sessions, each session is identified by a unique session ID. Therefore, once a session is established all further messages must contain the corresponding session ID (mitigates threat 31).

Additionally, each message that belongs to an established session must contain a sequence number. The sequence number is a strictly increasing counter value that is required as

a protection against replay attacks. Once a packet is received, the receiver is able to determine if a message with this sequence number was already received. In this case, the receiver is allowed to drop the packet, as it is required that no regular message in an established session contains the same sequence number (mitigates threat 34).

Adversaries may be able to send many session initialization requests originating from several IPs with the goal of starting a DoS attack (mitigates threat 40). Additionally, adversaries may be able to send new session initialization requests, with the goal of closing established sessions. The secure communication approach should support a mitigation against these threats. Furthermore, the secure communication session initialization should be designed such that not much computational processing power is required to generate the initial messages.

4.2.2 Device authentication

The first main requirement of the secure communication protocol is the authentication of communicating devices (e.g. control devices and the cloud service). Therefore, during the initialization phase of a new session, it is required that each involved device authenticates itself. As mentioned in section 3.3.2.2, a certified asymmetric public authentication key is required for authentication (see figure 3.8).

Thus, it is required that each control device owns a distinct private authentication key and a corresponding public key. This private authentication key must not be stored unencrypted on the device. If the encrypted boot approach is used (see section 4.1.4.2) in which the complete boot chain (incl. rootfs) is encrypted, the private authentication key can therefore be stored directly in the rootfs. However, in particular if the same rootfs image is used for several devices, and for software updates, this approach can be challenging. Thus, another possibility is reusing the persistent confidential key that is stored in the hardware RoT and also required for the encrypted boot approach. This confidential key can be used to encrypt the private authentication key. Finally, the encrypted private authentication key can be stored in an unencrypted partition of the control device (e.g. the data partition that is introduced in section 3.3.2.1). Furthermore, if no encrypted boot is used, HSMs, TPMs or authentication token may also be able to store the private authentication key in a secure manner (mitigates threats 27 and 28).

The public authentication key must be signed by a CA that is trusted throughout the whole SCCPS. Since certificates typically have a limited validity period, it is required that the certificates are updated regularly via the software update (see section 4.3.2). The certificate and the public authentication key can be stored anywhere (e.g. unencrypted in the data partition of each control device). For the trusted CAs, a trust list (see section 2.3.4.2) must be stored on the device. As this trust list consists of several public keys, it is not required to encrypt the list. However, undesired modifications of this list should be prevented. Thus, the verified rootfs can be used. It should be possible to change the contents of the trust list by applying a software update. Besides the trust list, also the CRL (see section 2.3.4.1) must be updated regularly via a software update.

In addition to the public authentication key and the certificate, a device ID is required for authentication (see figure 3.8). The device ID can be used to perform a device specific access control. Thus, even if a control device trusts the authentication keys of all other control devices, it is possible to limit the communication to specific device IDs. In particular, if a control device consists of two or more network interfaces, it can be useful to use different device IDs for each network interface. Finally, this device ID should be transferred only encrypted.

4.2.3 Message encryption

In order to provide a confidential communication channel between two communicating devices, it is required to encrypt the payload of each transmitted message (see figure 3.8). As this secure communication approach operates on top of the Internet layer, the link layer as well as the Internet layer are not encrypted. Thus, the IP and the MAC headers are transmitted in plaintext. Furthermore, the secure communication protocol may consist of additional data that must be transmitted unencrypted. However, starting from the transport layer (e.g. UDP or TCP) the transmitted data is fully encrypted. Also, during the session initialization, the device ID, the public key and the certificate are encrypted (see figure 3.8).

As this secure communication protocol relies on PFS (see section 2.3.2.5), ephemeral keys are used. These ephemeral encryption keys are generated by the DH key agreement protocol between the communicating devices during the session initialization step. These negotiated keys are getting invalid after a specific timeout or at least if the secure session is closed. Once the ephemeral keys are invalid, they should be deleted to make sure that no one else gets access to these keys. Otherwise, adversaries may be able to decrypt transmitted packets from a specific session. Therefore, a good memory location for these ephemeral keys is the volatile memory that is deleted at the system reboot (mitigates the threats 29 and 35).

4.2.4 Message integrity and message authentication

The message encryption from section 4.2.3 encrypts only the message data from the transport layer with the key from DH. Other message components like the IP addresses, the sequence number or the session ID are not encrypted. To authenticate these components, the HMAC based message integrity and message authentication is elected. As mentioned in section 3.3.2.2, a Keyed-Hash MAC (HMAC) is a Message Authentication Code (MAC) that uses a specific PSK K and some data D (the message) that should be protected. HMACs are defined in equation (4.1) [RFC2104].

$$H_n(K'_n \oplus opad_n, H_n(K'_n \oplus ipad_n, D)) \tag{4.1}$$

In this formula $H_n()$ is a cryptographic hash function that takes all its arguments and creates a hash of length n bytes. The constant $opad_n$ is defined as $opad_n = 0x36^n$.[1] The

[1] The notation $0x36^n$ means n consecutive bytes of $0x36$. E.g. for $n = 4$: $0x36^4 = 0x36363636$

constant $ipad_n$ is defined as $ipad_n = 0x5c^n$. The operand $\oplus$ is a byte wise logic XOR operation and the operand | is the byte wise logic OR operation. K'_n is the representation of the PSK K. As it is required that the length of K'_n is n bytes, K'_n is calculated by equation (4.2) [RFC2104], [125]:

$$K'_n = \begin{cases} K & length(K) = n \\ H_n(K) & length(K) > n \\ K|0x0^n & \text{otherwise (padding with 0)} \end{cases} \tag{4.2}$$

In order to use this HMAC procedure, it is required that each communicating device pair uses the same key K. This key K must also be an ephemeral key (like in section 4.2.3). Thus, the key agreement must also be done during the session initialization and an ephemeral key should be used only for a specific session. Additionally, these keys should be changed after a specific timeout. Even though, the encryption keys and the HMAC keys may be related, they must not be the same. For the HMAC generation, the data must contain the session ID, the sequence number and the encrypted payload. The HMAC procedure mitigates the threats 30, 38 and 39.

However, if HMAC and encryption facilities are used they can lead to problems if middleboxes are used. According to [RFC3234], is a middlebox "[...] defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host." In particular, if Network Address Translation (NAT) middleboxes are utilized that typically modify the protocol headers of the transport layer (like the UDP header) these modifications are not possible anymore as the entire transport layer is encrypted and protected by an HMAC. Thus, as long as no NAT middlebox has knowledge of the utilized secret authentication and encryption keys they are not able to modify the headers of the transport layer without violating the HMAC check. Therefore, NAT middleboxes that are installed between two communicating devices are not supported by this entire secure communication approach as it is unreasonable to propagate the secret communication keys.

4.2.5 Requirements summary

The following list summarizes the requirements for the proposed secure communication approach:

1. The secure communication layer relies on IP (either IPv4 or IPv6) and is session based.
2. No middleboxes between two communicating devices are allowed.
3. This layer is a transparent security layer above the Internet layer. It can be used from the user space applications.
4. It is required to use static IPs (i.e. no DHCP servers are required).

5. It is required to use IPs and no host names (i.e. no DNS servers are required).
6. A system service establishes a secure communication channel to the communication endpoint.
7. The session ID must be included in each transmitted packet.
8. An increasing sequence number must be included in each transmitted packet.
9. A certificate containing a device ID and signed by a trusted CA is required for authentication.
10. The device ID, the certificate and the public authentication key must be encrypted before they are transmitted during the session initialization.
11. Robustness against DoS attacks with spoofed session initialization requests is required.
12. The private authentication key has to be stored in a secure storage (e.g. in the encrypted rootfs, a TBM, a TPM or a HSM).
13. The trust list which contains all trusted CAs must be stored in the verified rootfs.
14. The certificates must be kept up to date via software updates.
15. The CRL must be updated via software updates.
16. DH key agreement and PFS is required for ephemeral key generation.
17. Different ephemeral keys for message encryption and message authentication must be used (HMAC).
18. At least the data from the transport layer and above must be encrypted.
19. Parts of the session ID, the sequence number and the encrypted payload must be included in the HMAC calculation.

4.3 Secure updates approach

It is very important to provide a secure software update mechanism for all control devices of a SCCPS, because the software and the components of a PKI (certificates, trust lists or CRLs) should be kept updated on these control devices.

Secure updates can be triggered either by local users with the local maintenance user role (see section 3.3.2.3) or by remote users with the remote maintenance user role (see section 3.3.2.4). Users with other roles should not be able to perform software updates. Thus, the system is not capable of automatic software updates, as there is at least

a maintenance user required that monitors the software update process of all safety critical CPS components.

The software updates can be installed from local administration interfaces (entry point 2), expansion ports (entry point 4.5) or via secure network connections (see section 4.2) to cloud services.

As these network connections are already authenticated and fully encrypted, it is possible to download the software update via simple protocols like the Trivial File Transfer Protocol (TFTP) (see [RFC1350]) that do not rely on additional authentication or encryption facilities. Even though, downloading via the FTP or the HTTP is also possible.

4.3.1 Symmetric kernel and rootfs partitions

As mentioned in section 4.1.3 and section 3.3.2.1, a symmetric boot partition approach is used. Thus, there exist two kernel partitions and two rootfs partitions on the persistent memory. At system startup, the bootloader selects one of both kernel partitions and the kernel selects one of both rootfs partitions as the active partitions. The unselected kernel partition and rootfs partition are both inactive. Thus, these inactive partitions can be used as the target for the software update. For example, if the rootfs should be updated it can simply be installed onto the inactive rootfs partition without interfering the current active rootfs partition. Even if the software update fails, the control device remains operational and it is also possible to restart the boot process. If the software update could be successfully applied, during the next restart of the system, the kernel must select the previously updated rootfs partition (as it prefers newer software versions). Finally, if this system boot succeeds, the software update can be applied onto the other rootfs partition as well (mitigates threat 20).

4.3.2 Software update package

A software update package is a bundle of files that contains partition images, regular files and additional metadata. In order to reduce the workload during the software update, the partition images must be included in the package such that they can be applied directly onto the destination partition. Thus, if the encrypted boot approach is used (see section 4.1.4), the partition images must already be encrypted by the correct intermediate key. As the bootloader and the kernel must both be able to detect the software version of a partition, they are able to determine which version should be started at boot up. Thus, this version info must also be contained in the partition images.

If a bootloader image is included in the software update package the bootloader signature must also be included, as well as additional data that is required to verify the bootloader signature (e.g. the trust list if the “hash of trusted keys” approach in section 4.1.1 is used). Furthermore, if the encrypted boot approach (section 4.1.4) is used the intermediate key must also be contained in the metadata of the software update package. If a kernel image is included in the software update package only the kernel image signature has to

be included, as the public verification key and the decryption key (if the encrypted boot approach is used) must be included in the bootloader image. The situation is similar for the rootfs which only requires the image signature, as the kernel image already includes the public verification key and the decryption key.

Additionally, it may be helpful if the rootfs contains the timestamp when the software update package was generated, as this information can be used to restrict the local time of the control device, as the time synchronization can lead to further threats like threat 23. Therefore, this packaged timestamp represents a lower boundary for the system time. Thus, if rootfs partition updates are applied sufficiently often, the timestamp of the last update can be used as a lower local time boundary for certificate expiration checks without the need to rely on external time synchronization services. Even though, the best solution would be a synchronization service for the local time that can be used by all control devices of a SCCPS. In addition to the time point that may be included in the software update package, the device authentication property of the secure communication channel requires that the CRL and the list of trusted CAs is included in the rootfs (see section 4.2.2). The software update package may also contain a new certificate of the public authentication key, which is also required for the secure communication channel. Additionally, each software update package must contain version information for each partition image and regular file that is included in the software update (mitigates threat 13). This version information is required to ensure that no software downgrades are performed. The version information must also be stored in the metadata of a software update package. If a SCCPS consists of multiple control device types, it can be helpful to include also a list of targeted device types. Hence, the software update routine is able to determine if the software update fits the device type.

In particular, if the encrypted boot approach (section 4.1.4) is used, the confidential intermediate key must be included in the software update package. Thus, some components of the package must be encrypted by a software update key that is stored on each control device using the same procedure as the private authentication key in section 4.2.2. Therefore, this software update key can be used to decrypt the encrypted components of a software update package before they are installed.

Similar to the trusted boot approach (see section 4.1) in which each boot partition must be signed by a trusted entity, each software update package must be signed by a trusted entity too. Therefore, the rootfs partition must contain either a trust list of entities whose signatures are accepted or a trusted CA issues certificates for trusted entities and software update packages are only accepted if they were signed by an entity with a valid certificate. In the latter case, the certificate must be included in the software update packet and the public key of the CA and a CRL must be included in the rootfs (mitigates threat 13).

4.3.3 Software update procedure

The following software update procedure must be executed on the control device that should be updated. This procedure follows the DFD in figure 3.9.

1. Before a software update can be applied, the user must be authenticated. Because the user authentication is out of the scope of this approach, it is assumed that it works as expected and allows only maintenance users to trigger the update.
2. The software update package must be downloaded to the control device (e.g. from the network or a USB-stick).
3. The signature of the software update package must be verified (either by a trust list or the certificates). If these verification checks fail, the update is aborted.
4. If the software update package contains encrypted components, they must be decrypted by the private software update key. This symmetric key is either stored in the encrypted rootfs, a TBM, a TPM or a HSM. If the decrypt steps fail, the update is aborted.
5. The list of targeted device types that is included in the software update package must be compared with the actual device type of the control device. If the current device type is not in the list of targeted device types, the update is aborted.
6. The version information of each component in the software update package must be compared with the version that is installed on the device (downgrade check). If the version of at least one component in the package is lower or equal to the installed component, the complete update is aborted.
7. The files that are contained in the software update package are installed to their destination. The actual destination is included in the metadata of the package.
8. The partition images that are contained in the software update package are installed to their destination partition. In case the partition image is a kernel or a rootfs image, the inactive partitions are selected.
9. If the intermediate key of the encrypted bootloader is included in the software update package, it must be encrypted by the confidential symmetric key in the TBM. This encrypted key must be stored in the persistent memory.
10. Once all files and images have been installed, the system must be rebooted.
11. During the boot process, the bootloader chooses the newest installed kernel partition (if the verification check succeeds) and the kernel uses the newest installed rootfs partition (if the verification check succeeds).
12. If the system and all system services could be started successfully, the same software update procedure must be applied to the second partitions too.

Note: As long as the software update package doesn't contain a bootloader image the device should still be operational, even if the software update procedure fails (due to the symmetric image approach). However, if the software update procedure fails during the bootloader update (e.g. due to a power-failure), the device may not be able to start the kernel. Thus, the bootloader should only be updated if it is absolutely necessary. Therefore, these software update packages must be installed with caution.

CHAPTER 5

Proof of concept

In this chapter, existing techniques are presented that show the feasibility of the approaches that are introduced in chapter 4. The proof of concept utilizes an i.MX7Dual SoC from NXP. The i.MX7Dual contains a dual core ARM Cortex A7 processor that operates at 1.2GHz speed. It was selected as it supports all hardware requirements from chapter 4. The i.MX7Dual is used on an i.MX7DSABRE board from NXP. This board contains 1GB volatile DDR3 memory and supports non-volatile block addressable NAND flash (e.g. SD cards, MMC cards and eMMC). Additionally, the Linux kernel is well supported on this board.

5.1 Trusted boot

The trusted boot approach is introduced in section 4.1. It contains a list of mandatory requirements that must be satisfied in order to provide the trusted boot facility. HAB (which is introduced in section 2.5.3.3) and in particular HAB version 4 (HABv4) is one option that is capable of all hardware requirements that are needed to implement the trusted boot approach. Thus, HAB is selected as the basis for this trusted boot concept, as HABv4 fulfills all hardware RoT requirements for the trusted boot approach (see section 4.1.5), the i.MX7Dual SoC with HAB support has been chosen as the target platform. As the bootloader U-Boot works well with HABv4, it is selected as the target bootloader for the trusted boot concept.

In section 5.1.1, the bootloader verification with HAB is explained. The verified bootloader that verifies the kernel image is introduced in section 5.1.2. In section 5.1.3, the verified kernel image verifies the rootfs before it is mounted. Finally, in section 5.1.4, some ideas regarding an encrypted boot chain are presented.

5.1.1 Bootloader verification (by hardware RoT)

The HABv4 feature that is implemented in several i.MX processors of NXP Semiconductors consists of an on-chip ROM that is not only responsible for loading the initial program image from the boot medium, it provides also the ability to authenticate the images before they are executed. In order to check the authenticity of the boot images, digital signatures are used. Thus, the boot ROM can be used as a RoT that is able to authenticate the first boot stage [FR10].

HABv4 relies for the boot image verification on image signatures that were created by trusted private Rivest-Shamir-Adleman (RSA) keys (up to 4096Bit[1]). Thus, the boot ROM verifies the image signature by the corresponding trusted public RSA keys. Boot images can be signed by different RSA keys, the image size can vary, and they can be stored in different memory locations. Thus, it is required to configure the boot ROM during startup. For this purpose, the Command Sequence File (CSF) must be used. It contains all instructions and parameters that are required by the boot ROM to successfully authenticate the signed boot image.

Because HAB is optional on the i.MX7Dual processor, the boot ROM must also be able to start the boot image without the CSF. Thus, important boot configuration parameters must not be stored in the CSF. Therefore, the boot ROM requires the Image Vector Table (IVT), the Device Configuration Data (DCD) table and additional boot data. The DCD contains configuration settings for some system peripherals at startup, the boot data contains the position and the size of the boot image and the IVT contains the start addresses of the DCD, the boot data, the IVT and the CSF. All the data must be stored beside the boot image in a particular layout [FR16]. The memory layout for an i.MX7Dual boot image is visualized in figure 5.1.

For authenticated boot with HABv4 it is not sufficient to create a signature of the image data alone as the boot configuration must be verified too. Thus, it is required that the CSF, the IVT, the DCD, the boot data and the actual boot image are signed by trusted RSA keys (see also figure 5.1) as all these components contain sensitive boot data. To create the signature of the IVT, the DCD, the boot data and the boot image, the private RSA key IMG_n is used. For the signature of the CSF the private RSA key CSF_n is used. Even though it is possible to use one private RSA key to sign all these components (the CSF, the IVT, the DCD, the boot data and the boot image), from a security perspective it is recommended to use the CSF_n and IMG_n keys [FR10].

In order to check the signature of all boot components, it is required that the signature was created by trusted private RSA keys. In order to propagate trusted RSA keys, HABv4 uses a table of up to four Super Root Keys (SRKs). Each SRK SRK_n is a RSA key and has two subordinate keys: An IMG_n key and a CSF_n key ($n \in \mathbb{N}_0, 0 \leq n \leq 3$). This means that each SRK key SRK_n acts as a CA and creates a certificate for each corresponding IMG_n and CSF_n key. In [FR10], this structure is called PKI tree and

[1] See the manual of the *Code-Signing Tool* [FR15].

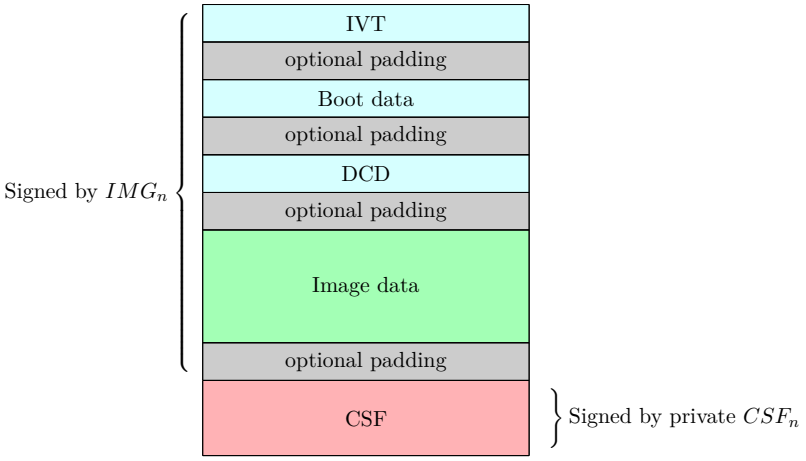


Figure 5.1: Simplified memory layout of the i.MX7Dual boot image components [FR16], [FR10].

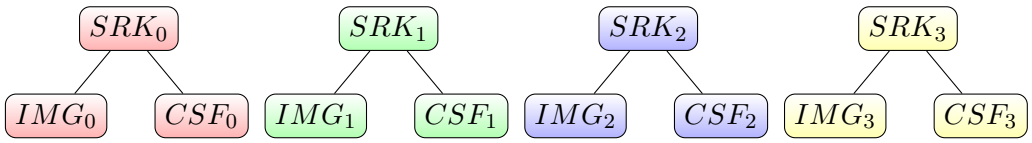


Figure 5.2: PKI tree structure of all RSA signature keys in the PKI tree [FR10].

it is visualized in figure 5.2. Instead of storing all four public SRK keys directly in the boot ROM, only the fingerprint (hash) of all four keys is stored. Therefore, the public key of each SRK must be contained in the CSF. As each SRK (SRK_0 to SRK_3) acts as a CA, it is not required that the SRK table contains any IMG_n or CSF_n public keys. Instead, the certificate of the utilized IMG_n key and the certificate of the CSF_n key are included in the CSF. Thus, at boot up it is possible to check if these certificates were correctly signed by the corresponding SRK_n [FR10].

To generate the fingerprint of all trusted public SRKs it is required to calculate the hash of each public SRK key (SRK_0 to SRK_3). All these hashes are put together in a hash array, that is finally hashed again (this concept can be seen in figure 5.3). Therefore, the hash of the hash array is a fingerprint of all four SRKs. During system deployment, this fingerprint of trusted public SRK keys can be programmed permanently by burning e-fuses. Once these fuses have been burnt successfully, only RSA keys IMG_n and CSF_n that were signed by these trusted SRKs are supported. All other image or CSF signing keys are untrusted and images that have been signed with these untrusted keys would not be accepted by the boot ROM [FR10].

During the boot process the boot ROM performs the same calculations: It reads the SRK table from the persistent memory, calculates the hash of each SRK, stores them in a hash array and calculates the fingerprint. Only if the fingerprint is equal to the

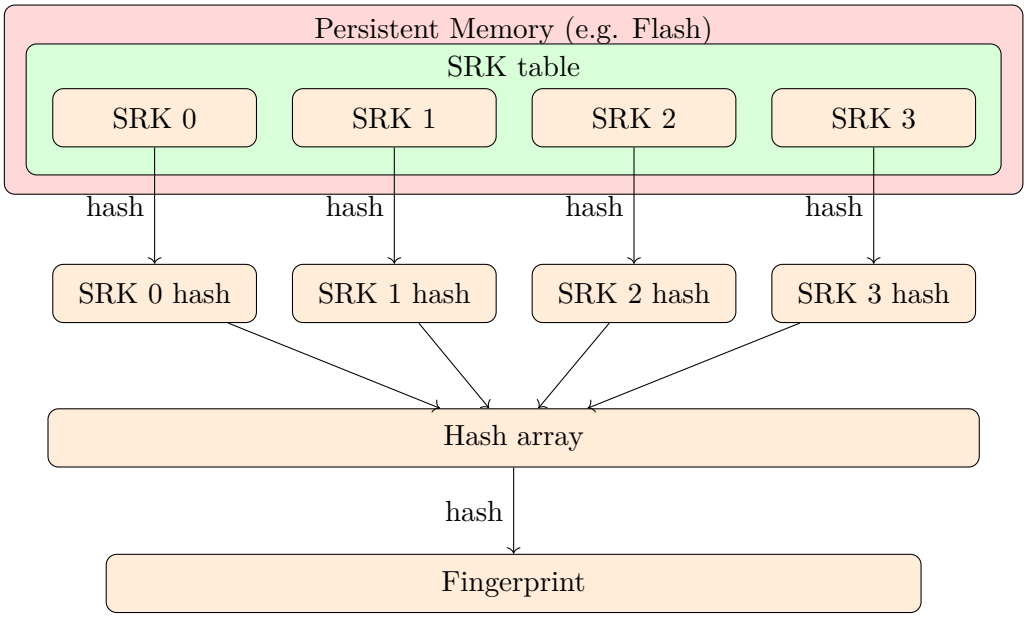


Figure 5.3: Generation of the SRK hash out of the SRK table [FR10].

fingerprint that was burned permanently into the device, the SRK table is accepted. Next, the certificates of the CSF key CSF_n and the image key IMG_n (which are stored in the CSF) are validated. If these certificates have been signed by one of the trusted SRK keys in the SRK table, these keys are trustworthy. Therefore, IMG_n can further be used to verify the integrity of the bootloader image, IVT, DCD and the boot data and CSF_n can be used to verify the integrity of the CSF [FR10].

As the fingerprint of all four SRKs in the SRK table must be burned into the device during the deployment, it is not possible to change the trusted SRKs afterwards. Thus, it is very important to keep the SRKs private, prevent unauthorized access to them and keep them on safe places, as no one would be able to sign future bootloader images if all SRKs get lost. Even though, it is possible that a SRK gets compromised. In this case, HAB provides the possibility to revoke the first three SRKs (SRK_0, SRK_1 and SRK_2). Therefore, there exist three independent e-fuses that correspond to the three SRKs. By burning the SRK disable e-fuse, the corresponding SRK key will be revoked permanently. Thus, during the boot process the boot ROM also checks which SRKs have been revoked. Even if the image and CSF verification would succeed, it refuses to boot images that were signed by revoked RSA keys [FR10].

In July 2017, NXP announced two critical secure boot vulnerability reports for some of their i.MX devices: ERR010872 [FR17] and ERR010873 [FR18]. ERR010872 describes a vulnerability in which the serial downloader can be used to modify code such that unauthorized images can be executed. As it does not affect the utilized i.MX7Dual that is used in this proof of concept, it is not explained furthermore [FR17]. ERR010873

explains a vulnerability in which special crafted certificates can be used to bypass the signature verification. Thus, with these certificates it is possible to bypass the image verification and execute unauthorized images (see threat 3.30). Even though there exists an update of the boot ROM for newly assembled i.MX7Dual devices, already manufactured devices may already contain this security flaw. Because the boot ROM can't be updated, NXP suggests that software updates are verified independently from HAB before they are going to be installed [FR18]. Additionally, the physical access to these devices should be limited and if possible, only revisions of the i.MX7Dual that already fixed ERR010872 should be utilized.

Thus, the capabilities of the HAB based hardware RoT can be summarized to:

- Up to four permanent public RSA keys (SRKs) form the basis for the image signature checks.
- Three of the four permanent SRKs can be revoked.
- The next boot step (bootloader) can be executed only if it has been signed by trusted RSA keys.
- If the boot signature check fails, the start sequence is interrupted.
- HAB provides the possibility to update the bootloader image as long as the updated image signatures are stored in the CSF.

In [FR10], several examples regarding signed image creation are presented. It even contains guidance how signed U-Boot (which will be introduced in section 5.1.2) bootloader images can be created.

5.1.2 Linux kernel verification (by bootloader)

Once all required instructions of the boot ROM were executed and the HAB component of the i.MX7Dual has successfully verified the bootloader image, the bootloader is started. As mentioned in section 5.1.1, for this proof of concept the bootloader U-Boot[2] has been selected. It was chosen as it is free software that is licensed under GNU General Public License version 2 (GPLv2), it supports the verification of kernel images (if Flattened Image-Tree (FiT) images are used) and the image verification possibilities are well documented. Additionally, U-Boot is under active development[3] and creating HAB compatible bootloader images is also well documented in [FR10].

U-Boot supports with FiT images the same concept that is used for the hardware description format Flattened Device Tree (FDT). However, instead of hardware description, all images that are required to start Linux are contained in this FiT image [FR19]. The

[2] `https://www.denx.de/wiki/U-Boot/WebHome`

[3] According to the GitHub release page `https://github.com/u-boot/u-boot/releases`, U-Boot has been released the last three years at least all two or three months.

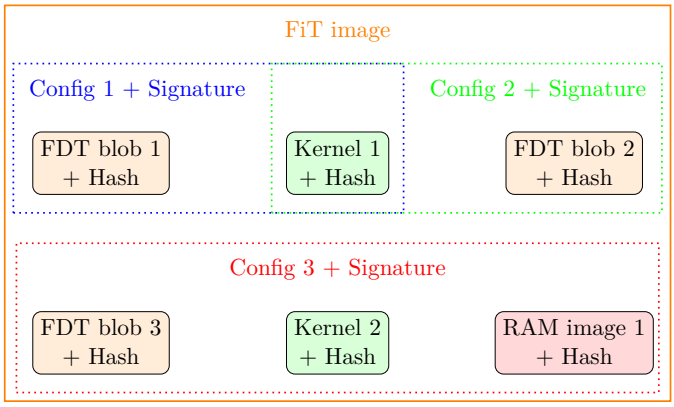


Figure 5.4: Schematic of a FiT image consisting of three configurations that share two kernel images, three FDT blobs and a RAM image.

images that are typically included in a FiT image are FDT blobs, kernel images or RAM disk images [FR20]. To support multiple devices with one FiT image, each FiT image supports multiple kernel images, FDT blobs or even RAM disk images. In order to select properly which sub images should be selected at boot up, each FiT image contains additional configurations. A configuration is a selection of at least a kernel image and optional other images (e.g. FDT blobs or RAM disk images) that can be used together. Thus, the bootloader must select the right configuration at boot up.

Each FiT sub image or configuration can be protected by a hash or digital signature. Which hash or signature algorithm should be used is configurable (e.g. `md5`, `sha1` for hashing algorithms and `sha1,rsa2048` or `sha256,rsa4096` for signatures [FR20], [FR21]). Even though, signed images provide protection against many threats if the configurations are not signed it is still possible for adversaries to change the configurations such that incompatible images are combined. Therefore, it is important that at least each configuration is signed by a trusted private RSA key. As mentioned in [FR21], it is sufficient that each sub image is hashed (and not signed) and the configuration (containing the hashes of the used sub images) are signed by a trusted private RSA key. It should be kept in mind that the RSA keys that were used for HAB should not be reused to sign the FiT image configurations. Instead, as the scope of keys should be limited, different key pairs should be used to create FiT image signatures [126].

An example visualization of a FiT image can be seen in figure 5.4: This FiT image consists of three signed configurations. In contrast to the signed configurations, the images are only protected by hashes. Each configuration contains at least a kernel image. Whereas Config 1 and Config 2 share the same kernel image, Config 3 uses an additional RAM image.

To create a valid FiT image, the image settings must be specified in an image source file (typically ending with `.its`). It contains meta-information about the whole FiT image including a textual image description. Additionally, also the sub images can be

specified. This involves the destination of each sub image, the hash or signature and some additional metadata like a textual description, the image type, the device architecture, compression settings, or image addresses. The configurations can be defined by specifying the contained images and an optional textual description of the configuration [FR20]. [FR22] and [FR21] contain several examples of signed image source file configurations. Once the image source file (`*.its`) has been created and all sub image files are available, the final FiT image (typically ending with `.itb`) can be created. This requires the two tools `mkimage` and `dtc`. `mkimage` is a tool that creates images for the U-Boot bootloader. It is shipped with the U-Boot package[4]. `dtc` is the Linux device tree compiler that is hosted by kernel.org[5]. Instructions how to create a FiT image with `mkimage` and `dtc` are available in [FR23].

In order to check the signature of the FiT image configurations, the public RSA key that was used to sign the FiT configurations must be included in the U-Boot image. In [FR21], it is suggested to store the public RSA key in the control FDT of U-Boot. In the BeagleBone example in [FR24], this requirement is achieved by calling the `mkimage` commando with the control Device Tree Blob (DTB) of U-Boot (see `-K` argument in [FR25]). To build U-Boot with the FiT requirements, U-Boot must be compiled with `CONFIG_FIT` enabled. Also, the flags `CONFIG_FIT_SIGNATURE` and `CONFIG_RSA` must be enabled (to support RSA signatures in FiT images). To enable the control FDT of U-Boot, the flag `CONFIG_OF_CONTROL` should be enabled too [FR21].

It is important that in cases in which U-Boot detects that the signature of the selected FiT boot configuration is invalid, the boot sequence of this FiT image configuration must be stopped. As the approach in section 4.1 relies on symmetric images, the bootloader is able to check the signature of the symmetric image too. If the verification check of this second image succeeds, this second image can be selected as boot up source. Thus, the bootloader must be able to boot from two independent images. As it is already possible to select the FiT boot image manually via the U-Boot console (see [FR24]), no limitations exist that would prevent the automation of this process.

Thus, the bootloader capabilities can be summarized to:

- The U-Boot image is capable of trusted boot, as it is able to verify FiT kernel images (and DTB or ramfs images) by a public key that is contained in the bootloader image.
- The kernel image signature is stored beside the kernel image in the FiT image.
- If the integrity check of a kernel image succeeds, U-Boot starts the kernel.
- As there are two kernel images installed, the selection of the most recent kernel version at startup remains an open issue. Nevertheless, the metadata in the FiT

[4] ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2
[5] https://git.kernel.org/pub/scm/utils/dtc/dtc.git

image can be used to store the required image version information that can be used by this selection mechanism.

5.1.3 Rootfs verification (by Linux kernel)

Once the bootloader U-Boot has verified and started the FiT image (that contains a valid Linux kernel image), the verified Linux kernel can be started. As the signatures of U-Boot and the Linux kernel have been verified successfully, the remaining element in the boot chain is the root file system. Before any system services that are stored in this rootfs are executed or any data is read, it must be verified that the data is signed correctly.

As mentioned in the introduction to chapter 5, block addressable NAND flash is used to store the symmetric system images (like the kernel images or rootfs partitions). Thus, the dm-verity concept can be utilized for this proof of concept: dm-verity works on top of the device mapper component of the Linux kernel and can be used to verify the integrity of read only partitions [FR26]. As rootfs partitions can be rather huge partitions on the block devices, it can take some time if the integrity of the whole block device must be checked before the partition is mounted. Thus, dm-verity uses a hash tree (Merkle tree) based verification approach that verifies the integrity of requested blocks on demand. Therefore, as long as no blocks are read out of the dm-verity protected partition, no integrity checks are performed on the blocks of the file system [FR27].

The tree-based verification structure works as follows: Each of the N blocks in the partition is hashed by a configurable hash algorithm: $B_0 = hash(block_0)$, $B_1 = hash(block_1)$, ..., $B_{N-1} = hash(block_{N-1})$. The hashes of n ($n << N$) blocks B_0, B_1, ..., B_{n-1} are concatenated and hashed again (intermediate level 1 hash): $H_{0_0} = hash(B_0|B_1|...|B_{n-1})$, $H_{0_1} = hash(B_n|B_{n+1}|...|B_{2n-1})$, ... And even m hashed hashes H_{x_y} are concatenated and hashed again (intermediate level 2 hash): $H_0 = hash(H_{0_0}|H_{0_1}|...|H_{0_m})$, ... This procedure is repeated until a single hash remains. This remaining hash is called the root hash. It forms the fingerprint of all blocks in the root file system. A change in a single block would also change the root hash. Once a hash mismatch of an arbitrary block is detected, it is possible to trigger an immediate restart and prevent that the mismatched block is read [FR27].

The hash tree concept is visualized in figure 5.5. In this visualization, the partition consists of 32768 blocks. For each of these blocks a hash is calculated. 128 of these block hashes are combined to an intermediate level 1 hash. Out of these intermediate level 1 hashes two intermediate level 2 hashes are calculated. And out of both intermediate level 2 hashes the root hash is calculated. The hash tree must be pre-calculated and stored in the dm-verity partition. The dm-verity partition layout is shown in figure 5.6. It starts with the superblock of the file system. After the superblock, the actual file system data blocks are stored. The last file system block is followed by the metadata of dm-verity which itself is followed by the hash tree [127].

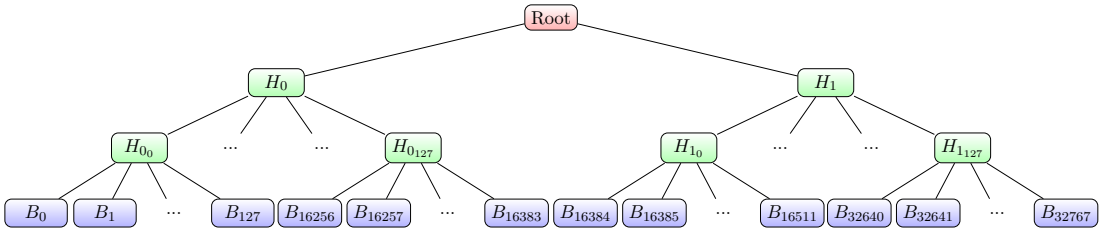


Figure 5.5: Visualization of a dm-verity hash tree. Each node represents a hash value [127], [FR27]. The green nodes and the root node are hashes of concatenated hashes whereas the blue nodes are hashes of file system blocks.

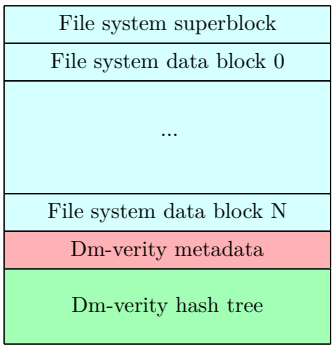


Figure 5.6: Layout of dm-verity partitions [127].

dm-verity was initially developed for Chrome OS and introduced in Linux 3.4. It requires that the kernel configuration option `CONFIG_DM_VERITY` is enabled. With the 4.4 release of Android, dm-verity was also introduced for Android. In contrast to the Chrome OS implementation (which is part of the Linux implementation), the Android implementation requires RSA signatures. Thus, the public RSA key, that is required to verify the signature, is included in the boot partition. The signature is created for the complete dm-verity mapping table which contains device locations, offsets, the root hash and the salt. The mapping table as well as the signature are both stored in the dm-verity metadata block [127], [FR27].

For the trusted boot concept of the i.MX7Dual, a similar approach can be used: The already verified kernel image must contain the public RSA key of a trusted image signing entity. This public key can then be used to check the integrity of the mapping table before the dm-verity file system is used. As mentioned in section 5.1.2, it is important that the utilized RSA key is different to the keys that are used in section 5.1.1 and section 5.1.2. If the mapping table signature is invalid, the dm-verity partition must not be used, and a reboot should be triggered[6]. As the symmetric image approach is used, the kernel is also able to select one of both rootfs partitions. If the kernel is able to read

[6]In order to implement these verity signature checks the “dm-verity tools” from Nikolay Elenkov can be used: `https://github.com/nelenkov/verity`

the metadata of the rootfs partitions, it is able to select the most recent rootfs partition with a higher priority. Additionally, as a block hash mismatch leads to a system reboot, the kernel is able to select the second rootfs partition at the next startup.

Thus, the dm-verity based rootfs verification features can be summarized to:

- The kernel image is capable of trusted boot, as it is able to verify the rootfs partition by a public key that is contained in the kernel image.
- The rootfs image signatures are stored in the dm-verity metadata.
- If the integrity checks of the dm-verity signature succeed, the rootfs partition can be mounted.
- As there are two rootfs partitions installed, the selection of the most recent partition at startup remains an open issue. Nevertheless, this information can be stored in the dm-verity metadata.

5.1.4 Encrypted Boot

The previous subsections cover detailed explanations of the trusted boot concept as it is the main focus of this section. Anyhow, this subsection introduces some rough ideas how the encrypted boot concept could be implemented, as the i.MX7Dual provides the hardware facilities for it. Furthermore, the following encrypted boot concept relies on the trusted boot concepts of section 5.1.1, section 5.1.2 and section 5.1.3.

One of the benefits of recent HABv4 devices like the i.MX7Dual is that it provides the requirements for an encrypted boot chain (at least devices with HABv4.1 support are required), as the HABv4.1 ROM provides encryption and decryption facilities [FR28]. This encrypted boot facility relies on the trusted boot facility of HAB. Therefore, the memory layout is similar and depicted in figure 5.7. It can be seen that the CSF is stored unencrypted as it contains the instructions that are required to decrypt the bootloader image. Additionally, the IVT, the boot data and the DCD are also stored unencrypted, as they are required by the CSF. It is sufficient to sign these sections, as no confidential data is contained in them. The memory layout of the trusted boot image differs slightly from the memory layout of the encrypted boot image: It contains the Data Encryption Key (DEK) blob which covers the symmetric Advanced Encryption Standard (AES) encryption key DEK that is required to decrypt the encrypted U-Boot image. The DEK is stored encrypted in the DEK blob as it is a very important confidential property of the encrypted boot image. The length of the DEK is variable and can be either 128Bit, 196Bit or 256Bit [FR28].

The key that is used to encrypt the DEK is the OTP Master Key (OTPMK) which is unique for each device and created during the chip manufacturing process. This key can only be accessed by the Cryptographic Acceleration and Assurance Module (CAAM) of the processor. Thus, the CAAM ensures that the OTPMK is kept confidentially as the

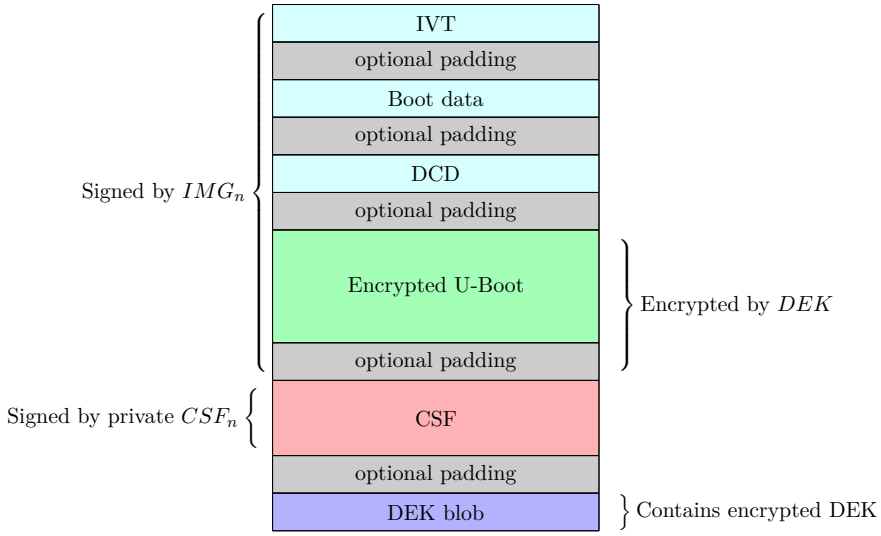


Figure 5.7: Simplified memory layout of the encrypted i.MX7Dual boot image components [FR28].

DEK is decrypted during the boot process in a secure memory partition that can be accessed only by the CAAM. Because each device contains a different OTPMK, each DEK blob contains different data, even if the same DEK is used across multiple devices [FR28]. More information on encrypted boot with HABv4 can be found in [FR28].

Once U-Boot has been successfully decrypted, it gets executed when the boot ROM finishes. As U-Boot already supports AES-128 decryption (in cipher block chaining (CBC) mode), it can be used to decrypt the FiT image (containing the kernel and the DTB) before it is started [FR29]. Thus, once the bootloader U-Boot has initialized the DRAM controller it can utilize the DRAM to store the decrypted kernel image. In particular, as the kernel images typically have sizes of a few megabytes they fit easily in the DRAM as the i.MX7DSABRE board consists of 1GB DRAM. Thus, at boot up U-Boot copies the encrypted FiT image to a distinct address in the DRAM and decrypts it there. If the validation step that has been described in section 5.1.2 succeeds, the described kernel image can be executed. As the U-Boot image is already encrypted in the persistent memory, the AES decryption key can be included in the U-Boot image.

A similar approach can be used for the kernel image; which then contains the encryption key for the rootfs. In contrast to the kernel image which fits without any problems in the DRAM and can be decrypted there, the size of the rootfs partitions can be much bigger than 1GB. Thus, it may not be possible to decrypt the complete dm-verity partition. Therefore, the dm-crypt module can be used. Like dm-verity, it works on top of the device mapper component of the Linux kernel. Therefore, it relies on block devices too. It is designed particularly for large file systems that should be fully encrypted. The file encryption facilities operate transparently as the file system works on top of the

dm-crypt module [FR30]. More information on dm-crypt can be found in [FR30].

This encrypted boot concept can be summarized to:

- The OTPMK is a confidential private key that is accessible only from the CAAM module of the processor. Applications do not have access to this key.
- During the manufacturing process a unique OTPMK is generated for each i.MX7-Dual processor.
- The CAAM provides an interface that is used to decrypt (and encrypt) the U-Boot image.
- The DEK acts as an intermediate key that is used as encryption key for the U-Boot image.
- Both the U-Boot image and the kernel image contain the encryption key for the next boot step and use these keys for decryption.

5.1.5 Evaluation

This subsection compares the properties of the trusted boot concept, that is introduced in this section, with the requirements in section 4.1.5. Thus, it covers the bootloader verification, the kernel verification and the rootfs verification. Additionally, the optional encrypted boot requirements are covered too.

The bootloader verification via HABv4 has been introduced in section 5.1.1. It forms a hardware RoT and contains a boot ROM that is executed at system startup (requirement 1). HABv4 is used to verify the bootloader signature, that is stored in the CSF next to the image, before it is executed (requirements 2, 5 and 8). If the verification fails, the start sequence is interrupted (requirement 9). To verify the bootloader, four constant RSA keys can be used and three of them can be revoked (requirements 2 and 6). Once the RSA keys have been stored in the HABv4 components, the boot ROM is able to use them. However, it is not possible to alter these keys anymore (requirements 3 and 4). All components of HABv4 are combined in one SoC with the processor (requirement 13). Additionally, the i.MX7Dual platform supports the configuration of boot devices and interfaces and allows the complete deactivation of the JTAG interface [FR16] (requirement 12).

The trusted bootloader U-Boot verifies the kernel image with the digital signature that is included in the symmetric FiT images (requirement 5). If the first signature check fails, the second (symmetric) kernel image is verified (requirement 10). Only if both image verifications fail, the start sequence is interrupted (requirement 9). As the FiT image already contains metadata that belongs to the image, it is possible to implement an image selection mechanism that prefers the most recent installed image at bootup (requirement 11).

Similar reasoning holds for the trusted kernel that verifies the rootfs via dm-verity, where the signature is included in the dm-verity metadata (requirement 5). Thus, if this signature check fails, the second (symmetric) rootfs partition can be verified (requirement 10). If both verifications fail, the start sequence is interrupted (requirement 9). Therefore, U-Boot as well as the Linux kernel are both capable of trusted boot, as they are both able to verify the next boot step (requirement 7). Thus, this concept (including HABv4, U-Boot as well as the Linux kernel) fulfills all requirements of the trusted boot concept from section 4.1.5.

Besides the trusted boot concept HABv4 additionally fulfills the encrypted boot requirements in section 4.1.5: OTPMK is the confidential private decryption key that can't be accessed by applications (requirement 1). Therefore, HABv4 provides the CAAM interface that uses the OTPMK to decrypt and encrypt any data (requirement 2). The OTPMK is created for each i.MX7Dual processor during the manufacturing process and is therefore unique for each device (requirement 3). Since the intermediate key DEK is used for several devices, it is stored encrypted beside the image (requirement 4). As the decryption key for the kernel image can be included in the encrypted bootloader and the decryption key for the rootfs partition can be included in the encrypted kernel image, the secure storage of these keys is no problem. Furthermore, U-Boot already supports AES decryption and the kernel module dm-crypt can be used for encrypted rootfs partitions (requirement 5).

5.2 Secure communication

In this section, a secure communication concept for the control devices of a SCCPS is introduced. It is based on the approach in section 4.2. Thus, it relies on IP based communication. Since legacy control devices may utilize connections that are not capable of IP, a proof of concept is presented that introduces IP over serial EIA-232 connections in section 5.2.1. Therefore, even if legacy connections do not support IP communication innately, it may be possible to upgrade these connections to support IP. Finally, section 5.2.2 contains the IP based proof of concept of the approach mentioned before.

5.2.1 IP-based communication for non-IP capable protocols

As mentioned in section 4.2, IP communication is required for the secure communication approach. Nevertheless, there exist many protocols that are not intended for IP communication. Common examples are point-to-point (PTP) connections between two endpoints. Thus, the addressing and routing facilities are typically not required for these communications as only two devices are communicating together. Even though, existing PTP infrastructure may already exist and therefore it may also be required to reuse this infrastructure. In particular devices that utilize this legacy infrastructure can benefit, if regular IP packages can be sent, as the approach in section 4.2 can be applied. Therefore, in this section introduces a concept that enables IP communication for PTP connections like a serial EIA-232 interface.

5.2.1.1 TUN device

As each control device runs Linux, the Linux networking subsystem is responsible for the IP communication. In the context of the Internet protocol suite (see section 2.4.1), the kernel is responsible for data processing from the data link layer (Linux Ethernet network device drivers) up to the transport layer. The transport layer (layer 4) is typically accessible from the user space via the "Portable Operating System Interface (POSIX) Sockets API" [57], [IEEE 1003.1g]. The user space is typically responsible for the application layer [57].

However, to enable link layer or Internet layer communication via serial connections (e.g. EIA-232), the TUN/TAP-drivers can be used [FR31]. With these drivers it is possible, to create virtual network adapters in the user space and process data from the link layer or Internet layer within the user space. The virtual adapters operate either as point-to-point adapters (network TUNnel (TUN)) or as virtual Ethernet adapters (network Terminal Access Point (TAP)) [128].

As this concept targets PTP communications based on IP, a virtual TUN device satisfies all requirements. The TUN device can be created and managed via the `netdevice` interface (see [FR32]) of the Linux kernel. Additionally, also the interaction of the kernel with the TUN device can be handled programmatically. As all TUN related operations can be managed from applications in the user space, it is possible to create system services that ensure that the TUN device is created at startup and configured with the correct IP configuration. As TUN devices are virtual network interfaces, they can have several IPv4 and IPv6 addresses assigned. However, in contrast to physical interfaces, no link layer is used. Therefore, packets that originate or target a TUN interface do not contain data from the link layer. Thus, no Ethernet header is contained, and no MAC address is assigned to these TUN interfaces [FR32].

5.2.1.2 PTP protocol specific interface adapter

Once, the TUN device has been created, the packets that traverse the TUN interface must be forwarded to the PTP endpoint. Thus, a PTP protocol specific interface adapter is introduced: It ensures that at least the transport layer payload and some data from the Internet layer (metadata) of each IP packet are transmitted via the serial PTP connection. It is important that enough Internet layer information is transmitted, as the protocol specific interface adapter of the receiving endpoint must be able to restore the original IP packet. As this adapter is the link between the TUN device and the physical PTP connection, it depends on the requirements of the utilized PTP protocol and must in general be developed for each PTP protocol separately.

For the proof of concept both, the TUN device and the PTP protocol adapter are combined in the "IP over serial" system service that is started automatically at startup. Thus, it handles TUN device creation and TUN device configuration as well as adaption of the IP packages to the message format of the utilized PTP protocol. Thus, by using

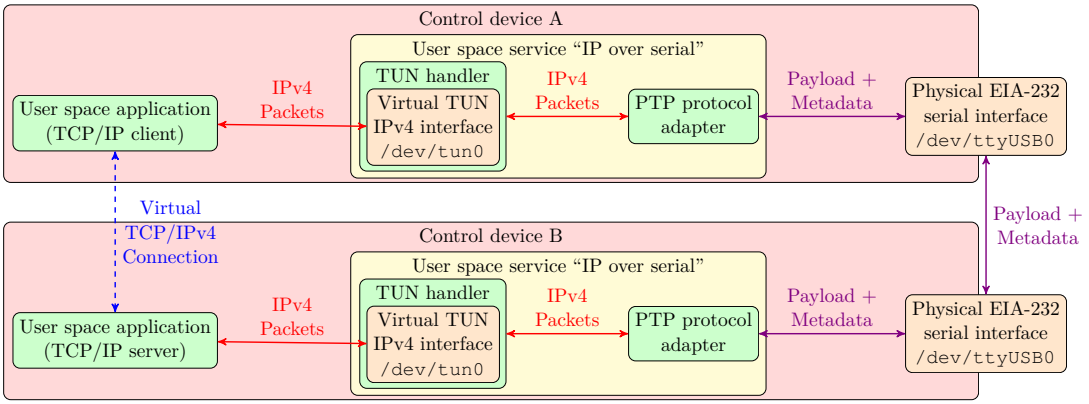


Figure 5.8: A virtual TCP/IP connection between two control devices utilizing serial EIA-232 interfaces and virtual TUN devices.

this approach, it is possible to rely on IP communication even without an IP capable physical network adapter.

In figure 5.8, this concept is shown as an example: The user space application in control device A wants to establish a TCP connection to the control device B. Thus, it sends the required IPv4 packets to the TUN device. The data is captured by the TUN handler and transmitted to the PTP protocol adapter. Next, the PTP protocol adapter extracts the TCP data (TCP header and TCP payload) and some IPv4 header fields (metadata) from the IPv4 packets. The extracted data is than sent via the serial interface to control device B. Once the PTP protocol adapter receives this data (TCP data and IPv4 metadata) from the serial interface, it reassembles the original IPv4 packet and sends it to the TUN handler. Lastly, the TUN handler passes the reassembled IPv4 packets via the virtual TUN device to the user space application (TCP server). For packets that traverse the devices in the reverse direction, the same principles apply.

5.2.1.3 Implementation

For the proof of concept, the PTP connection between two control device endpoints is realized by a serial EIA-232 connection. The EIA-232 connection is utilized by the termios interface (see [FR33]) which implements the POSIX standard [IEEE 1003.1]. While Universal Asynchronous Receiver Transmitter (UART) frames in general are able to process data with five to nine bits, one start bit, one or two stop bits and an optional parity bit (see [129]), for this example the EIA-232 interface is configured to use eight data bits, one stop bit and no parity bit. Additionally, the baud rate is configurable at least between 38400 and 230400. Even though, these data transfer rates are not overwhelming, they can be sufficient for some connections between control devices of a SCCPS. In particular, if only a few control and measure data are transferred. However, if high bandwidth is required (i.e. audio or video streaming) or several packets are routed via this serial connection, other technologies like Ethernet or Fiber should be used.

The implementation was tested with Debian GNU/Linux 10 with Linux kernel 4.19 and a "Digitus DA-70156" USB to serial converter. This converter is available in Linux 4.19 via the device file `/dev/ttyYSB0` and the TUN device was created at `/dev/tun0`. A functional software implementation can be found at `https://git.auto.tuwien.ac.at/theses/sccps-security-concepts`. It currently supports only IPv4 traffic that can be tunnelled via this serial connection. However, in general also IPv6 packets should not be a problem. Instead of particular IPv4 header fields, the complete IPv4 packets (including the TCP header and payload) are transmitted via the serial connection.

The protocol specific adapters can be exchanged without modifying the user space application. For the application developers, this leads to a decoupling of the communication technologies, as all traffic can be addressed by IP. Therefore, even if the serial EIA-232 connection will be replaced by Ethernet in the future, the underlying technology is transparent for the application as long as it is capable of IP.

5.2.1.4 Further improvements

A further improvement for the "IP over serial" user space service is an additional support for IPv6 packets, as currently only IPv4 packets are supported.

In particular, if the payload size of a packet is small compared to the IP header, it can be beneficial to use an IP header compression mechanism (like in 6LoWPAN [RFC4944], [130]) to reduce the packet size. Some possibilities are:

- If only IPv4 or only IPv6 packets are transmitted, the IP version field can be omitted.
- If the transmitted packets originate and target only the endpoints of the two PTP connection, the source- and destination addresses can be omitted (keep in mind, that this disables the routing facility.)
- If the number of possible IP addresses can be restricted (e.g. by a common routing prefix or subnet) redundant address information can be omitted.
- If only a unique transport protocol like UDP or TCP is supported, the "next header" field (IPv6) or the "protocol" field (IPv4) can be omitted.
- If fixed values for the IPv6 fields "traffic class" or "flow label" as well as the IPv4 fields "IP header length" or "type of service" are used, these fields can be omitted.

For example, if only IPv4 packets without optional header fields are transmitted and no routing functionality is required, the IPv4 addresses can left out. Thus, the version field, the header length and the time to live field as well as the IPv4 addresses can be omitted. Therefore, instead of transmitting 20 bytes of IPv4 header data, it is sufficient to transmit only 10 bytes and restore the original header on the receiving endpoint. For

IPv6 packets the header information can be reduced by 32 bytes, if both IPv6 addresses are omitted.

5.2.2 Secure IP based communication between CSs

This section introduces a proof of concept for an IP based secure communication between control devices. The secure communication fulfills all requirements from section 4.2.5. Additionally, with the concept from section 5.2.1, the application area can be extended to cover protocols that are innately not capable of IP.

5.2.2.1 IPSec as secure communication

In section 2.4.2, several secure communication approaches for the Internet protocol suite have been introduced. The proof of concept relies on IPSec to satisfy the secure communication requirements from section 4.2.5. IPSec has been developed by the IETF and works on top of IPv4 and IPv6. It is defined in several RFCs, starting with [RFC4301]. As these RFCs are publicly available proposed standards, everybody is able to implement IPSec. Thus, there exist several independent implementations that are more or less compatible (e.g. from Cisco, Microsoft or Checkpoint [47]) [52].

For the proof of concept, the strongSwan[7] implementation of IPSec is utilized. It originates from the FreeS/WAN project which implemented IPSec for Linux. It supports many platforms (like Linux, Windows, macOS or Android) and is released under GPLv2 [131], [47].

As mentioned in section 2.4.2, IPSec consists of three components: IKE, AH and ESP [47].

IKE The IKE component is used from two communicating endpoints to agree on encryption algorithms, authentication algorithms, DH groups, authenticate the endpoints and create encryption keys for AH or ESP. Currently, there exist two versions of IKE: IKE version 1 (IKEv1) and IKE version 2 (IKEv2). IKEv2 has been developed to simplify the IPSec setup. However, it contains also some additions to IKEv1 [47].

AH The AH protocol can be used to check the authenticity and integrity of transmitted messages. These checks include several IP headers (like the IP version, data length, protocol or source address [RFC4302]).

ESP The ESP protocol provides authenticity, integrity and confidentiality of transmitted messages. In contrast to AH, these checks do not include any IP headers [47].

[7] https://www.strongswan.org/

5.2.2.2 IKEv2

Both AH and ESP rely on the negotiated parameters (authentication algorithm, encryption algorithm, the DH group and ephemeral encryption keys) from IKE. All the negotiated parameters as well as the destination IP are combined by IKE in an unidirectional Security Association (SA). Thus, for bidirectional communication it is required to create two SAs. Each SA has a particular lifetime. Therefore, if a SA expires, IKE creates a new SA with new key material. IKE is in contrast to ESP or AH no protocol on its own, as it works on top of UDP and typically uses port 500. IKE supports several possibilities to authenticate the communicating endpoints. Examples are PSKs as well as RSA signatures or Extensible Authentication Protocol (EAP) (which is only supported in IKEv2). As the authentication requirements in section 4.2.5 require certificates combined with a unique device ID, the RSA signature approach is used for the proof of concept. Because IKEv2 is a new version of IKE that simplifies the SA initialization and authentication and supports more features as the predecessor IKEv1, the proof of concept utilizes IKEv2 [RFC7296], [47].

In general, messages in IKEv2 occur in pairs (initiation and response). In order to initialize an IKEv2 SA, an `IKE_SA_INIT` message pair must be sent. This message pair is used to negotiate on the Security Parameters Indexes (SPIs) and the cryptographic algorithms (`SAi1`, `SAr1`). Additionally, it contains the ingredients for a DH key agreement (`Kei`, `Ker`, `g`) as well as nonces (`Ni`, `Nr`) that are required for the generation of keys in the next step. After the `IKE_SA_INIT` messages have been transmitted, both communicating endpoints are able to generate the same symmetric authentication and encryption keys out of the already transmitted and agreed parameters `Kei`, `Ker`, `g`, `Ni` and `Nr`. These keys can be used to authenticate and (partly) encrypt the second message pair `IKE_AUTH` which is used to transmit the identities (`IDi`, `IDr`), the certificates (or certificate chain) (`CERTi`, `CERTr`), a new SA (`SAi2`, `SAr2`) and some additional traffic and authentication related data. This new negotiated SA characterize the security properties and the utilized protocols (ESP or AH). In figure 5.9, the IKEv2 initialization sequence is depicted. As required by section 4.2.5, this initialization sequence is typically performed by a system service in the user space (i.e. an IKE-daemon) which creates the required SAs at startup. Additionally, this service is responsible for the renewing of SAs. The strongSwan implementation that is used in this proof of concept uses the IKE daemon Charon for this purpose [RFC7296], [47].

Therefore, IKEv2 supports all secure session requirements that are introduced in section 4.2.5: It supports a certificate and device ID based authentication and encrypts the certificate as well as the device ID before both are transmitted. Thus, the IKEv2 daemon Charon must have access to a list of trusted CAs and the software update procedure must ensure that the CRL and renewed certificates are updated frequently on the device. IKEv2 uses a DH key agreement-based procedure to generate distinct ephemeral keys which are used for message authentication and encryption. Furthermore, it provides a protection against a huge number of `IKE_SA_INIT` requests from spoofed IPs (DoS attack): If there are already many `IKE_AUTH` messages outstanding, a responder

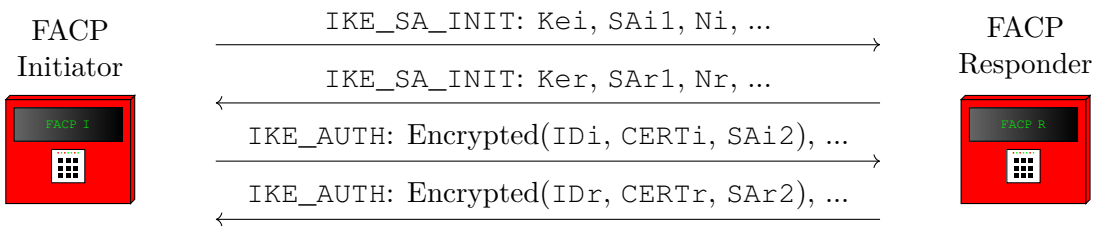


Figure 5.9: The IKEv2 initialization sequence that generate the SA that is used from ESP or AH.

is able to reject new `IKE_SA_INIT` requests with a notification payload. Once the initiator receives the rejected `IKE_SA_INIT` with the notification payload, it sends a new `IKE_SA_INIT` request that contains this notification payload. Thus, using these notification payloads this DoS threat is mitigated, as adversaries that spoofed an IP address do not receive this notification payload. Additionally, IKEv2 requires that this notification payload is hard to predict, and it is easy to determine for the responder if the notification payload is valid, even if the notification payload is not stored [47].

5.2.2.3 ESP

As mentioned in section 2.4.2, it is required that either the AH protocol or the ESP protocol is chosen for each IPSec connection. Section 4.2.5 states that it is required to encrypt at least the data from the transport layer and above. Thus, the security features of AH are not sufficient to fulfill these requirements. Therefore, in the proof of concept IPSec with ESP is used, as it can be used to encapsulate, encrypt and authenticate the data from the transport layer and above. An ESP packet consists of several fields (which are shown in figure 5.10) [RFC2406], [47]:

SPI: The SPI field in combination with the destination IP is required to identify the SA that corresponds to the ESP packet.

Sequence number: The sequence number field is a monotonic increasing number that is set by the sender and helps the receiver to detect messages that have been replied.

IV: If the encryption algorithm requires an Initialization Vector (IV) (e.g. CBC algorithms) it is stored in the IV field that is located before the payload field.

Payload: The payload field contains the actual message information that must be sent to the receiver.

Padding: As the encryption algorithms are typically block ciphers, it is often required that the payload is filled upon the next block size. This padding is added to the padding field.

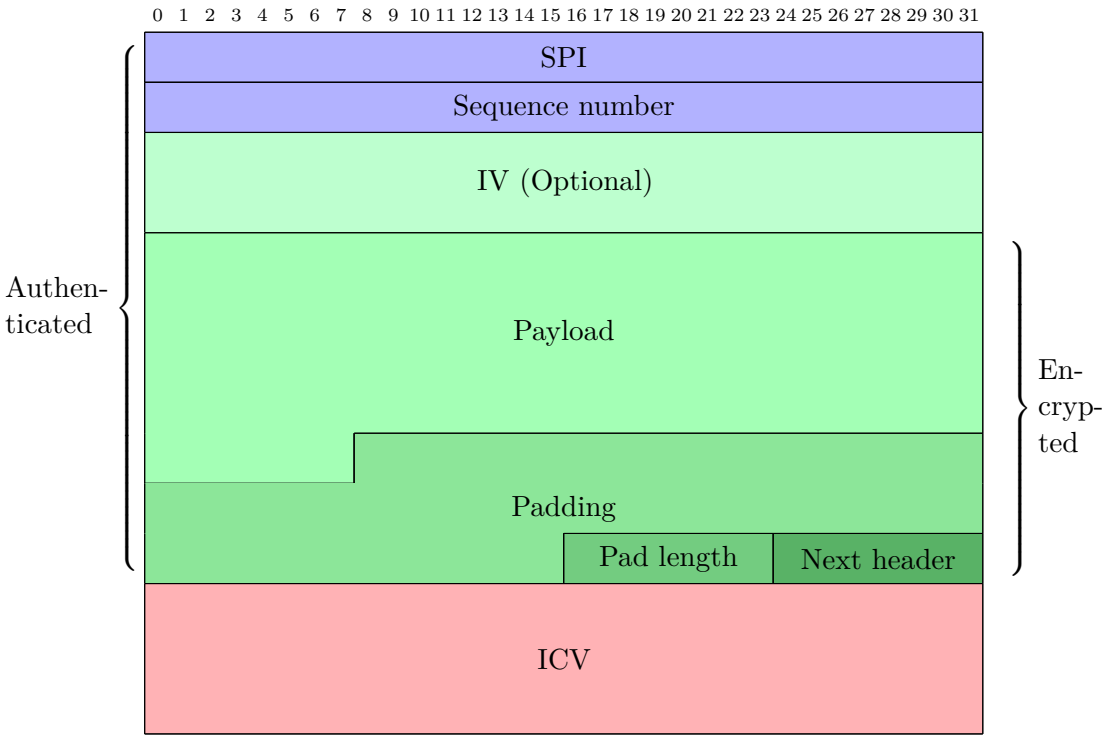


Figure 5.10: Visualization of an ESP packet [RFC2406], [47].

Padding length: The field padding length denotes the length of the padding field.

Next header: The field next header contains the protocol number of the payload (i.e. 17 for UDP).

ICV: The Integrity Check Value (ICV) contains the 96 highest bits of the HMAC that is used to authenticate the ESP packet.

Thus, as required by section 4.2.5, each ESP packet contains a sequence number and the session ID (SPI). Additionally, the ESP protocol encrypts the fields payload, padding and next header before the message is sent (see also figure 5.10). Once these fields have been successfully encrypted, the message authentication (ICV) is calculated. The ICV is calculated by an HMAC procedure which involves all previously encrypted fields, the IV, the SPI and the sequence number field. Thus, ESP packets do not include any IP header fields in the ICV calculation (in contrast to AH packets) [RFC2406], [47].

ESP can be used in two different transmission modes: The transport mode and the tunnel mode. Both modes are shown in figure 5.11 [47]. In the transport mode, the ESP header is included between the Internet layer (e.g. IPv4) and the transport layer (e.g. UDP). Thus, the IP header must be modified slightly before the ESP packet (encapsulating the original UDP payload) can be included (e.g. by adjusting the total length, the protocol and the checksum fields in the IP header). Whereas the tunnel mode wraps the original

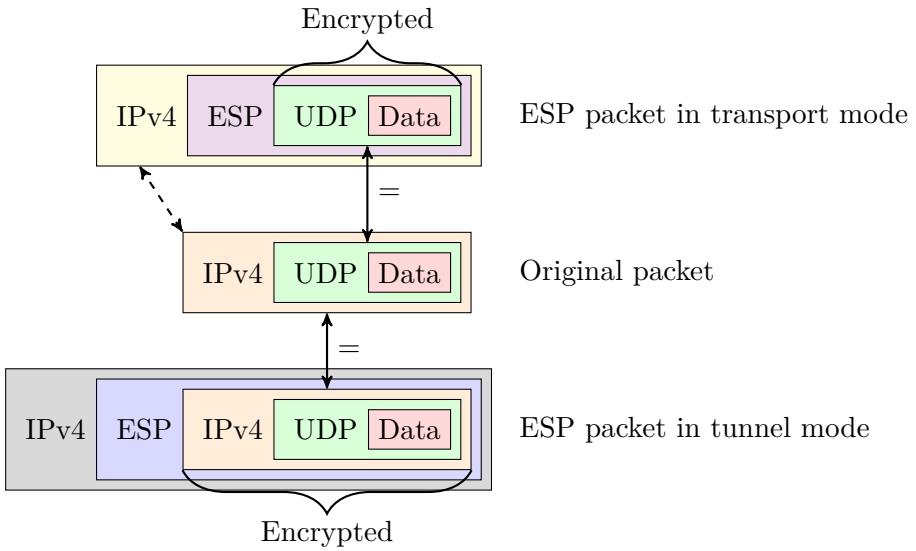


Figure 5.11: Difference between the ESP transport mode and the ESP tunnel mode using IPv4 packets and UDP payload: In transport mode the ESP packet contains the original UDP packet whereas ESP packets in the tunnel mode contain the original IPv4 packet [47].

IP packet without any modifications and encloses it with the ESP and a new IP header. In the tunnel mode, the enclosed IP header may have other source- and destination addresses as the encapsulated IP packet, whereas in the transport mode the source- and destination addresses of the IP header are not changed. To meet the requirements in section 4.2.5 it is sufficient to use ESP in transport mode as it provides data encryption for the transport layer and above.

5.2.3 Evaluation

IPSec and in particular the strongSwan IPSec implementation that is used in this proof of concept fulfills all the requirements from section 4.2.5, as it is a session based security layer above the Internet layer (either IPv4 or IPv6) that can be used from all user space applications transparently (requirements 1 and 3). Furthermore, section 5.2.1 introduces a concept that enables IP based communication over other protocols like EIA-232. Thus, it may be possible to increase the applicability of this secure communication concept. As this concept relies on static IPs, no DHCP servers and no DNS servers are required, as both can lead to additional vulnerabilities (requirements 4 and 5). Additionally, NAT middleboxes are not supported as the ESP header encrypts and authenticates the transport layer (requirement 2). However, as long as the devices get a public IP address, NAT is not required anymore, as this secure communication layer is fully IPv6 compatible. Thus, obtaining public IPv6 addresses is generally no problem.

As IPSec is session based, each ESP and IKEv2 package contains the session ID that corresponds to the particular session (requirement 7). Besides this session ID, each message contains an ascending sequence number (requirement 8). Furthermore, the payload of each message is encrypted. As IPSec works on top of the Internet layer, all ESP messages encrypt the data from the transport layer and above (requirement 18). Furthermore, each ESP packet is authenticated by an HMAC. This HMAC computation covers amongst others the session ID, the sequence number and the encrypted payload (requirement 19).

This concept relies on the strongSwan IPSec implementation, which provides a system service that is responsible for a secure communication channel between the communication endpoints (requirement 6). The strongSwan service uses a device ID and a certified public authentication key (all these components are encrypted before they are transmitted) in order to authenticate the device at the communication endpoint (requirement 10). It is important that the certificate has been issued by a CA that is trusted by the endpoint (requirement 9). As each control device is a possible endpoint, it is required that each control device contains a list of trusted CAs. The concept in section 5.1 can be used to store a trusted list of CAs in the rootfs (requirement 13). Besides the list of trusted CAs, a CRL must be added to the rootfs. Via the software updates concept in section 5.3, the CRL and the trust list can be updated (requirements 14 and 15).

StrongSwan can be configured such that DH key agreement with PFS is used at session initialization (requirement 16). It is also possible to use ephemeral keys for message encryption and message authentication (requirement 17) and strongSwan provides a protection against spoofed session initialization requests (requirement 11). An example strongSwan configuration for two hosts that use ESP in transport mode can be seen at `https://git.auto.tuwien.ac.at/theses/sccps-security-concepts`.

The only requirement in section 4.2.5 that is still open claims that the private authentication key must be stored in a secure storage (requirement 12). This requirement can be addressed, if the encrypted boot concept in section 5.1.4 is used. It consists of a fully encrypted rootfs partition where the authentication key can be stored securely. However, even if the trusted boot concept from section 5.1 without the encrypted boot facility is used, the OTPMK (which has been introduced in section 5.1.4) encryption facility of the i.MX7Dual can be used in order to encrypt or decrypt the authentication key. Thus, the encrypted authentication key can be stored in an unencrypted data partition. By using the CAAM interface at startup, it is possible to decrypt the private authentication key before the strongSwan services are executed.

5.3 Secure Updates

As stated in section 2.7, it is required to keep the control devices of a SCCPS up to date. Additionally, the secure communication approach in section 4.2 relies on the software update feature of a control device to keep the CRL up to date and renew the certificates. Thus, the update scheme is an essential concept of secure control devices. For the

proof of concept, the SWUpdate mechanism (that has been introduced in section 2.7) has been chosen as it already provides support for symmetric image updates. It can be extended easily by Lua plugins that allow custom update package processing. Furthermore, SWUpdate is FOSS[8] and released under the GPLv2 [93]. Thus, everybody is allowed to customize it according to its needs.

The software update procedure that has been introduced in section 4.3.3 requires that the software update package contains individual files as well as complete partition images. All these components in the update package are referenced in the "sw-description" file, which is part of the SWUpdate package and contains the metadata of the package. Thus, the bootloader image, the kernel image (FiT image) and the rootfs image are included in the update package as complete images. This implies that the complete image must be updated, even if only one file in this image must be updated, whereas updates of individual files that are located in the data partition (like encrypted configuration files, certificates or encrypted CRLs) can be updated on a file basis. SWUpdate fulfills the requirements as it supports update packages that contain multiple image files as well as individual files [93], [FR34]. As it is responsible for the update package installation, SWUpdate must be installed on each control device.

Due to the different update package components (any combination of files or images), it is important that the update package contains individual version information for each component. In a SWUpdate package, this version information can be stored in the "sw-description" file. In order to fulfill the requirements in section 4.3.3, this version information of all included components must be compared with the version information of the installed components before any updates are applied. Thus, if the packaged component version is lower than the installed component version, the complete update package must be rejected as it is not allowed to downgrade a distinct software component. As this requirement is currently not supported in SWUpdate, it must be implemented [FR34]. Additionally, each update package must contain a list of supported device types. Thus, if a package is installed on an unsupported device type, the software update must be aborted. SWUpdate provides this functionality with the `hardware-compatibility` attribute that can be specified in the "sw-description" file [FR34].

In order to allow only software updates from trusted entities, the "sw-description" file of a software update package must be signed at least by a trusted entity. As this "sw-description" file of each update package can take the hashes of each included image or file, the signature involves the components of the software update package too. Thus, the software update service requires access to a list of trusted CAs that are allowed to sign a update package (similar to the IKEv2 daemon Charon in IKEv2, section 5.2.2.2). Note that, this list of trusted CAs can also be updated by software updates. If the update package contains a bootloader image, the HAB API can be utilized from the update installer to verify the contained bootloader image signature. This API can be used to check the bootloader signature before it is installed [FR10]. Therefore, the installation of invalid bootloader images can be prevented.

[8] https://github.com/sbabic/swupdate/

As stated in section 4.3.3, the secure update approach requires support for encrypted components. SWUpdate already provides this functionality for all files and images that are included in the update package. Thus, the symmetric encryption key for the software update must be stored on the device either in the rootfs image or in the data partition. For the proof of concept, this encryption key is stored in the data partition. As the OTPMK encryption feature of the i.MX7Dual can be utilized in this approach, this symmetric encryption key can be stored encrypted in the data partition of the control device (see section 5.2.3) [FR35]. Additionally, if the encrypted boot approach from section 5.1.4 is used, the encryption key for the bootloader image must be encrypted by OTPMK, too. As mentioned there, this encrypted bootloader encryption key is stored beside the image in the DEK blob.

Because symmetric rootfs partitions and symmetric kernel images are used on the control devices, the update service must ensure that FiT images or a rootfs images are installed at the respective inactive partitions. Additionally, the bootloader verifies the signatures of both FiT images at boot up. If both are verified, it selects the most recent kernel image. The same approach is used from the kernel and both rootfs partitions.

5.3.1 Installation procedure

To install software updates, it is required that users authenticate themselves on the device. For security reasons, only authenticated users with the maintenance user role are permitted to apply a software update. Thus, other users (that are introduced in section 3.4.1) must not be able to access the secure software update installer. Therefore, each authenticated user must be either a local user (authenticates itself on the device via a PIN or an authentication token) or a remote user (authenticates itself on the device via asymmetric cryptographic keys). Once the user has been authenticated successfully, the update package must be downloaded to the control device. It can be downloaded either from the network (convenient for remote users) or from external devices like USB-sticks. Once the update package has been downloaded successfully, the signature of the package must be verified. If the update package contains a bootloader image, the HAB API can be utilized from the installer to verify the contained bootloader image signature [FR10]. Additionally, if the update package contains a kernel or rootfs image, the signatures of these images must be verified, to make sure that the device is able boot from these new images. Not all of these checks are supported by SWUpdate, thus these must be implemented by a Lua plugin or by extending the SWUpdate source code. Only if all these checks succeed, the update can be applied in a safe and secure manner. As a software update package typically contains boot images, it is required to restart the device once the software update has been applied successfully. Additionally, as a symmetric image approach is used, the software update procedure must be started again to make sure that the software update is applied to the redundant partitions too.

As the bootloader image is not stored symmetrically on the device, a failed update of the bootloader can lead to a bricked device. Therefore, bootloader updates can be applied

by local users only, as they have physical access to the control devices and can react accordingly if the bootloader fails.

5.3.2 Evaluation

The software update concept in this section fulfills the technical pre-requisites of the software update approach in section 4.3.3. Therefore, it is possible to implement a secure update procedure that follows the defined procedure in section 4.3.3. Even though, SWUpdate must be extended to fit this procedure. In particular, if the NXP i.MX7Dual reference platform is used and HAB or OTPMK facilities should be utilized. As SWUpdate currently doesn't support the hardware specific peculiarities of this platform.

The user authentication method as well as the distribution of the software update packages are out of the scope of this thesis. However, these components must be part of the final concept (procedure steps 1 and 2). The signature check of the software update packages can be handled with a list of trusted CAs and a CRL similar to the secure communication approach in section 5.2 (procedure step 3). Furthermore, if the software update package contains encrypted components and the encrypted boot facility from section 5.1.4 is utilized, the HAB and OTPMK features from the NXP i.MX7Dual may be used to decrypt and verify the package (procedure step 4). Additionally, it may be required to encrypt the intermediate decryption key if encrypted components are included (procedure step 9). In order to prevent the installation of incompatible software updates or inhibit software downgrading, each software update package must contain a list of supported devices as well as version information of the contained software. SWUpdate already provides this hardware compatibility check, but a software downgrade check must be implemented (procedure steps 5 and 6). SWUpdate also supports the installation of individual files that are included in the software update package (procedure step 7) as well as the installation of symmetric rootfs or kernel images that are installed into the appropriate inactive partitions (procedure step 8).

Once the update process finished successfully, the system must be rebooted (procedure step 10). If the secure boot concept in section 5.1 has been implemented correctly and a rootfs or kernel image has been updated, the previously inactive and therefore updated rootfs or kernel images must be selected during the boot procedure (procedure step 11). Thus, the software update can be applied on the inactive (previously active) rootfs or kernel images again (procedure step 12). Thus, combined with the hardware facilities of the NXP iMX7Dual platform and the previous concepts regarding trusted boot and secure communication, a secure update procedure for SCCPSs can be implemented. All three concepts together complete the security measures for SCCPSs in this thesis.

CHAPTER 6

Conclusion

6.1 Main contribution

The aim of this thesis is to introduce three security concepts for several components of a SCCPS. These three security concepts are trusted boot, secure communication and secure updates. Even though the targeted devices of the concepts are control devices of a SCCPS, they may be applied to other devices as well (if the underlying devices fulfill all requirements of the concepts). As the three concepts are key security features, they may be suitable for many CPSs and particularly for many SCCPSs. Especially, as the concepts already mitigate many possible threats for CPSs, provided that they are implemented correctly. The thesis covers a state-of-the-art analysis of IoT devices as well as CPSs and addresses the foundation of the necessary security concepts. Furthermore, a threat analysis based on an accurate threat model is performed. Additionally, the security concepts and their requirements are introduced. Finally, the thesis concludes with a proof of concept that implements the three security concepts for a concrete platform.

There already exist many research topics for IoT and CPSs, including approaches for the development of security concepts as well as security analysis:

In [132], possible security issues of automation IoT are analyzed in a layered approach. This layered approach is based on four layers: 1. A sensors and actuators layer, 2. a networking layer 3. a data processing layer and 4. an application layer. Thus, these layers are structurally similar to the layers in RAMI 4.0. Based on the layers, different threat types are identified, and possible mitigations are introduced. Therefore, this approach is different from the approach in this thesis where the threats are identified based on the entry points of the system model.

In [1], the following attack categories for industrial devices are listed: 1. “Espionage”, 2. “DoS”, 3. “Replay Attacks” and 4. “Deception attacks”. However, this paper uses the same layers as [132] in order to define which layers are affected by these attack

categories and how these threats may be mitigated. Furthermore, a threat and risk analysis methodology is introduced.

In [133], a different perception of the security of IoT hardware is given. It covers many threats regarding the hardware of IoT devices. These include anti-piracy methodologies for intellectual property cores, hardware trojans that are included in these intellectual property cores, supply voltage scaling as well as frequency modulation. These threats are not covered by the threat model in this thesis, as the system model in section 3.3 has been developed for an abstract system model that does not contain concrete hardware. Thus, if the concepts of this thesis are used, an additional threat analysis is required once the hardware is determined.

In [134], secure processors and their features regarding hardware protections, protected boot and protected execution environments are introduced. Thus, it basically compares the security features of modern secure processors independent of the underlying architecture and lists common vulnerabilities as well as countermeasures. For example, memory attacks that are mitigated by address space layout randomization.

The article in [135] presents networking related threats. These include eavesdropping, man in the middle attacks (including replay attacks) or privacy attacks. The security concepts covered in this thesis all mitigate these threats. Furthermore, the covered concepts provide the security objectives of confidentiality, integrity and authenticity that are also treated in [135].

6.2 Summary of the introduced concepts

In order to analyze these security concepts, a threat analysis of the proposed concepts is performed in chapter 3. Therefore, a high-level system model is created in section 3.3. It covers the three concepts trusted boot, secure communication and secure updates. Based on this system model, the trust levels, entry points and assets of the system are identified (as well as the usage scenario, external dependencies and implementation assumptions). The findings are the basis for the threats that are determined for this system. Finally, it is checked whether all threats that are identified in the threat model are mitigated.

As this threat model has been developed very generally, it does not rely on a concrete implementation of the proposed concept (i.e. the proof of concept in chapter 5). Therefore, even if different implementations of the approach in chapter 4 are utilized, the threat model should be applicable as well. However, this also implies that a new threat model must be created as soon as the concrete implementation is determined.

Based on the threat model, the following three security concepts are developed.

A trusted boot concept for Linux: The concept is introduced in section 4.1 and ensures that only certified software is executed on the device. The concept depends on the Linux boot sequence. In addition to being Linux capable, the device must provide a hardware RoT that is able to verify the first boot stage in the boot

process (i.e. the bootloader). Once this stage has been verified, the other stages (i.e. kernel and rootfs) can be verified by the previously verified stages. This concept is therefore called trusted boot chain. Thus, the hardware RoT is a key component for the developed trusted boot concept.

Besides the trusted boot concept, an optional encrypted boot concept is introduced. It works similarly to the trusted boot chain. Thus, at first the bootloader is decrypted. Then the kernel is decrypted by the decrypted bootloader. Finally, the kernel decrypts the rootfs.

Due to the hardware RoT requirement, the trusted boot concept and the encrypted boot concept are highly dependent on the hardware. Therefore, section 5.1 introduces a proof of concept that utilizes the hardware RoT capabilities of the HAB feature, that is provided by an NXP i.MX7Dual processor. As the secure update concept relies on a symmetric partition approach for the kernel and the rootfs, the trusted boot concept must be able to verify both kernel images and both rootfs partitions at startup. If the verification of both partitions succeeds, the bootloader must be able to select the most recent kernel image and the kernel must be able to select the most recent rootfs partition at startup.

A secure communication concept: The concept is introduced in section 4.2. It is generally independent of the OS or device capabilities. The only requirement is that the device is capable of IPv4 or IPv6, as this concept operates on top of the IP layer. Thus, from a current perspective, the secure communication approach seems future proven for SCCPSs, as IP based communication is on the rise for CPSs (see section 2.1.4). However, if IP based communication is currently not possible, section 5.2.1 shows an approach to enable IP based communication via other protocols like EIA-232. It may be possible to adapt the approach for other communication protocols as well.

Because this concept is session based, a session to the communication endpoints must be established before any messages can be sent. To establish a secure communication session, a device ID and a certified public authentication key are used. It is important that the authentication key is certified by a CA that is trusted by the endpoints. Once the session is established (and the endpoints are authenticated), encrypted as well as authenticated messages can be sent. Thus, the IP payload (including the transport layer and above) is encrypted. For the message authentication, an HMAC is used. Amongst others, it covers the encrypted payload, the session ID and the sequence number of the message.

As the IPSec protocol fulfills all requirements of the secure communication concept, this standardized protocol is used for the proof of concept in section 5.2.2. It relies on the strongSwan IPSec implementation. Even though this concept is platform independent in general, it relies on a secure storage of the private authentication key. Thus, either the previously mentioned encrypted boot concept or a different encrypted storage mechanism is required in order to provide a secure key storage.

A secure update concept: The proposed concept ensures that the device can be updated in the field and is introduced in section 4.3. Thus, it is very important for safety critical devices, as software vulnerabilities can be fixed even if the SCCPS is already in production. The concept relies on a symmetric partition approach. This means that the kernel image as well as the rootfs partition are stored twice in the persistent memory. At startup, one of both kernel images and one of both rootfs partitions are selected. These are the active partitions and the secure update concept ensures that the update is applied only to the inactive partitions. Thus, the active partitions remain unchanged. Therefore, the device remains usable even if the update fails as the active partitions are not modified.

Due to the symmetric kernel and rootfs partitions, it is required that the trusted boot concept is capable of an intelligent partition selection mechanism. As the trusted boot concept already verifies the authenticity of both components, the verification also checks if the partitions are valid and the software update has been applied successfully.

The proof of concept in section 5.3 relies on the free software SWUpdate as it already provides support for symmetric partition layout and it is easily extendable according to the needs of the secure update concept.

6.3 Further work

If the control devices of a SCCPS do not use IP based communication to communicate with each other, the introduced secure communication concept is not applicable. Therefore, section 5.2.1 introduces a concept for Linux devices that enables IP based communication via serial EIA-232 connections. Thus, it may be possible to adapt this concept for other protocols as well, in order to provide IP based communication for more components of a SCCPS.

The secure communication approach should be applied to connections to the cloud as well. However, as the Internet already uses IP based communication, the technological requirements are met. However, additional work may be required for confidential and authenticated communication at the field level, if a fieldbus is used (like connections between control devices and sensors or actuators). Even though, as mentioned in section 2.1.4, a shift to IP communication can be observed for future field devices (WSNs).

As the threat model in chapter 3 is very generic, it must be extended for the concrete implementation of the trusted boot approach, the secure communication concept and the secure update procedure, as each concrete implementation of these security measures can lead to new threats.

Additionally, an anomaly detection system as proposed in [136] may also be an appropriate countermeasure against attackers.

6.4 Implementation and source code

An example configuration of the trusted boot proof of concept in section 5.1 that uses HAB, FiT and dm-verity is available at `https://git.auto.tuwien.ac.at/theses/sccps-security-concepts`. This repository also contains an example configuration for the secure communication concept (utilizing strongSwan) and an implementation of the IP over EIA-232 concept that has been introduced in section 5.2.1.

List of Figures

List of Tables

Glossary

adversary An adversary is an entity, that utilizes a vulnerability to realize a threat. Sometimes it is also called attacker [106], [107]. 3, 4, 19, 43–52, 66–71, 75, 80–85, 87–109, 118, 122, 123, 128, 129, 142, 155, 175, 176

ARM TrustZone A TEE developed from *ARM*. 31, 32

asset An asset is a valuable item of interest, which an adversary aims at or which must be protected from an incorrect and unauthorized use by the adversary. This item may be abstract (like company's reputation, or the safety of people) or concrete (like the content of a database) [108], [106], [107]. 43–45, 47, 48, 50, 51, 66–71, 79–108, 164, 173, 175, 176

attack path An attack path is the condition sequence which is required to achieve an attack goal. Without mitigation it forms a vulnerability [106]. 43–45, 51, 175–177

authentication Authentication consists of two parts; one is responsible for entity authentication (i.e. to ensure the correct identity of an entity) and the other part is responsible for message authentication. Authentication ensures therefore the identity and origin of communication [43], [42], [8]. 17, 47, 175

authorization Authorization is preventing illegitimate entities from doing actions, that require special legitimation (access control) [8], [37]. 17, 47, 175

availability Availability ensures, that authenticated entities have data access each time they needed it. This means, that the system needs to be in a functional state. Stopped systems may be highly secure, but they are not functional and therefore no entities are able to request data [8], [40]. 16, 47, 175

condition A condition is an action or weakness present in an attack path [106]. 43, 44, 51, 79, 99, 175

confidentiality Confidentiality means, that protected data is kept secret, as long as no authorized entity has successfully authenticated itself. It is important, that authentication is not part of this property [41], [42]. 16, 17, 47, 122, 175

entry point An entry point (also access point) is the intersection point between the modeled system and the world, i.e. it provides access to the assets. A system and the corresponding assets can only be accessed and therefore attacked if it has entry points. (It is important to keep in mind, that the term entry point also involves exit points.) [108], [106], [109]. 43–48, 50, 51, 61–66, 79–108, 132, 164, 173, 176

error "An error is the part of the system state that may lead to a failure" [3]. 3, 7, 8, 14, 62, 80, 176

external entity An external entity is located outside of the scope of the modeled system. But it correlates with the modeled system. This correlation is realized via the entry points of the modeled system [110], [106]. 44, 47, 176

failure A failure is "[...] an event that occurs when the delivered service of a system deviates from correct service" [3]. 3, 14, 22, 32, 50, 135, 176

fault "A fault is the cause of an error" [3]. 3, 14, 176

integrity Integrity ensures, that information was not altered or modified. If the information is transferred, integrity ensures that the content was not changed by some entities between the sender and the receiver [41], [42]. 16, 17, 39, 47, 176

non-repudiation Non-repudiation is a combination of integrity and authentication. It allows an entity to be sure, that another particular entity is responsible for a particular action [44], [42]. 17, 47, 176

risk The risk is a "[...] characterization of the danger of a vulnerability or condition" [106]. 14, 44, 45, 48, 51, 52, 54, 176

security weakness Security weakness is an unsatisfactory mitigation of a threat, which typically results in a vulnerability [106]. 44, 45, 176

threat A threat exists, if it is possible, that an attack on a specific asset is successful. Thus, a threat may be the goal of adversaries [108], [106]. 6, 14, 43–52, 55, 71–73, 79–81, 83–86, 89, 92–94, 96–98, 105, 108, 118, 119, 122, 127–130, 132, 133, 141, 142, 155, 164, 166, 175–177

threat tree A threat tree can be used to visualize the attack path of a threat. The root of a threat tree is the actual threat that is visualized [106]. 44, 51, 79–116, 169, 170, 176

trust level A trust level is a specification of an external entity, which describes who has access to an asset using a specific entry point. A trust level contains authentication methods and privileges that the entities of a particular level require [108], [106]. 44, 47–50, 59–62, 66, 68, 72, 74, 89, 164, 173, 176

vulnerability "A vulnerability is a system property that violates an explicit or implicit security policy" [108]. In particular, a vulnerability defines an attack path that introduces a realized threat, due to insufficient mitigation [106]. 20, 43–45, 47, 48, 51, 72, 74, 82, 89, 91, 95, 140, 141, 157, 175–177

Acronyms

2FA two-factor authentication. 34

6LoWPAN IPv6 over Low Power WPAN. 13, 152

AES Advanced Encryption Standard. 146, 147, 149

AES-NI AES New Instructions. 34

AH Authentication Header. 26, 153–156

AI artificial intelligence. 7

API Application Programming Interface. 31, 48, 159, 160

BACS building automation and control system. 7, 8, 12, 14, 24

BIOS basic input/output system. 32, 33, 37

CA certificate authority. 20–23, 55, 56, 72, 74, 75, 78, 82, 95, 111, 113, 119, 128, 131, 133, 138, 139, 154, 158, 159, 161, 165

CAAM Cryptographic Acceleration and Assurance Module. 146–149, 158

CAPEC Common Attack Pattern Enumeration and Classification. 46

CBC cipher block chaining. 147, 155

CENELEC Comité Européen de Normalisation Électrotechnique. 12

CIA confidentiality, integrity and availability. 16, 17

CPS cyber-physical system. vii, ix, 2, 3, 6, 13–15, 24, 35, 38–40, 43, 46, 127, 163, 165

CPU Central Processing Unit. 28, 29, 35–37, 53, 63, 67, 73, 94

CRL certificate revocation list. 20, 22, 56, 96, 113, 128, 131, 133, 154, 158, 159, 161

CRTM Core Root of Trust for Measurement. 33

CSF Command Sequence File. 138–141, 146, 148

CSMA Carrier Sense Multiple Access. 67

DCCP Datagram Congestion Control Protocol. 26

DCD Device Configuration Data. 138, 140, 146

DEK Data Encryption Key. 146–149, 160

DFD Data Flow Diagram. 47, 49, 50, 75–79, 119, 121, 122, 127, 134, 169

DH Diffie-Hellman. 18, 19, 56, 78, 99, 100, 107, 129, 131, 153, 154, 158

DHCP Dynamic Host Configuration Protocol. 48, 55, 72, 127, 130, 157

DNS Domain Name System. 48, 55, 73, 127, 131, 157

DoS Denial of Service. 44, 47, 80, 92, 101, 106–108, 114, 116, 128, 131, 154, 155, 163, 170, 174

DRAM Dynamic RAM. 28, 35–38, 147

DREAD Damage potential, Reproducibility, Exploitability, Affected users and Discoverability. 44, 51

DTB Device Tree Blob. 143, 147

DTLS Datagram TLS. 24, 26

EAP Extensible Authentication Protocol. 154

EEPROM Electrically EPROM. 29, 30, 33, 39, 54

EFI Extensible Firmware Interface. 37

EoP Elevation of Privilege. 44, 47, 80, 108

EPROM Erasable PROM. 29, 39

ERP enterprise resource planning. 9

ESP EFI System Partition. 37

ESP Encapsulating Security Payload. 26, 153–158, 171

EtherCAT Ethernet for Control Automation Technology. 12

FACP fire alarm control panel. 14, 15, 56, 57

FDT Flattened Device Tree. 141–143

FF Foundation Fieldbus. 12, 24

FIDO Fast Identity Online. 35

FIP Factory Instrumentation Protocol. 12

FiT Flattened Image-Tree. 141–144, 147, 148, 159, 160, 167, 170

FOSS Free and Open Source Software. 35, 159

FOTA Firmware OTA. 39

FTP File Transfer Protocol. 27, 132

GP GlobalPlatform. 31, 32

GPLv2 GNU General Public License version 2. 141, 153, 159

GPS Global Positioning System. 53

GPU Graphics Processing Unit. 94

HAB High Availability Boot. 33, 37, 63, 137, 138, 140–142, 146, 159–161, 165, 167

HABv4 HAB version 4. 137, 138, 146–149

HMAC Keyed-Hash MAC. 56, 100, 101, 103, 105, 114, 115, 129–131, 156, 158, 165, 170, 173

HMI human machine interface. vii, ix, 3, 8–10, 46

HOTP HMAC-Based One-time password. 34

HRNG Hardware Random Number Generator. 34

HSM Hardware Security Module. 30, 31, 33, 37, 63, 128, 131, 134

HTTP Hypertext Transfer Protocol. 26, 132

HTTPS Hypertext Transfer Protocol Secure. 23, 26

I²C Inter-Integrated Circuit. 30

IC integrated circuit. 11

ICMP Internet Control Message Protocol. 25

ICT information & communication technology. 8, 9

ICV Integrity Check Value. 156

IEEE Institute of Electrical and Electronics Engineers. 12, 27

IETF Internet Engineering Task Force. 27, 153

IKE Internet Key Exchange protocol. 26, 153, 154

IKEv1 IKE version 1. 153, 154

IKEv2 IKE version 2. 153–155, 158, 159, 171

IoT Internet of Things. 1, 13, 14, 24, 30, 55, 61, 163, 164

IP Internet Protocol. vii, ix, 5, 12, 13, 25, 26, 28, 52, 55, 56, 62, 68, 72–75, 86, 87, 94, 110, 127–131, 149–157, 165–167

IPSec Internet Protocol Security. 6, 24, 26–28, 153, 155, 157, 158, 165

IPv4 IP version 4. 26, 55, 75, 127, 130, 150–153, 156, 157, 165

IPv6 IP version 6. 26, 55, 75, 127, 130, 150, 152, 153, 157, 165

IV Initialization Vector. 155, 156

IVT Image Vector Table. 138, 140, 146

JTAG Joint Test Action Group. 62, 87, 93, 119, 122, 126, 148

L2F Layer 2 Forwarding. 27

L2TP Layer 2 Tunneling Protocol. 27

LAN Local Area Network. 86, 110

LF Low Frequency. 53

LIDAR light detection and ranging. 8

M2M machine-to-machine. 14, 55, 59, 60, 127

MAC Medium Access Control. 27, 129, 150

MACSec MAC Security. 27

MITM man-in-the-middle. 4, 20, 104, 105, 115, 170, 174

MROM Mask ROM. 29, 39, 118

NAT Network Address Translation. 130, 157

NFC near-field communication. 34, 75

NTP Network Time Protocol. 48, 53

OP-TEE Open Portable - TEE. 31

OS operating system. ix, 23, 28, 31, 35, 40, 41, 46, 49, 52, 63, 64, 66, 70–73, 80, 89, 91–93, 111, 165

OSI Open Systems Interconnection. 11, 25

OTA Over-the-Air. 39

OTP one-time programmable. 118, 119

OTP one-time password. 34, 123

OTPMK OTP Master Key. 146–149, 158, 160, 161

OWASP Open Web Application Security Project. 47

PFS perfect forward secrecy. 19, 56, 100, 104, 129, 131, 158

PGP Pretty Good Privacy. 27, 28

PKI public key infrastructure. 20, 22–24, 26, 56, 72, 74, 131, 138, 139, 170

PLC programmable logic controller. 3, 8–10, 46

POSIX Portable Operating System Interface. 150, 151

PPP Point-to-Point Protocol. 27

PPTP Point-to-Point Tunneling Protocol. 27

PROFIBUS Process Field Bus. 12, 24

PROFINET Process Field Network. 12

PROM Programmable ROM. 29, 39

PSK Pre-shared key. 19, 129, 130, 154

PSU Power Supply Unit. 66

PTP point-to-point. 149–152

RA registration authority. 20–22

RAM Random Access Memory. 28, 29, 54, 100, 142

RAMI 4.0 Reference Architecture Model for Industry 4.0. 1, 10, 163

REE Rich Execution Environment. 31

RFC Requests for Comments. 13, 27, 153

RNG Random Number Generation. 30

ROM Read-only Memory. 4, 5, 28, 29, 33, 35–37, 39, 46, 53, 55, 64, 70, 73, 77, 85, 91, 92, 111, 112, 117–119, 121, 124–126, 138–141, 146–148

rootfs Root-File system. 54, 55, 89

RoT Root of Trust. 32–34, 63, 73, 117–119, 121–126, 128, 137, 138, 141, 148, 164, 165

RSA Rivest-Shamir-Adleman. 138–143, 145, 148, 154, 170

RTM Root of Trust for Measurement. 32, 33

RTR Root of Trust for Reporting. 32, 33

RTS Root of Trust for Storage. 32, 33

SA Security Association. 154, 155

SCCPS safety critical CPS. vii, ix, 3–6, 14–16, 40, 43–46, 48, 50, 52, 55, 56, 58, 59, 62, 69, 71, 73–76, 80, 81, 84, 86–88, 90, 107, 113, 117, 126–128, 131, 133, 149, 151, 158, 161, 163, 165, 166, 169

SFTP SSH File Transfer Protocol. 27

SGX Software Guard Extensions. 31

SoC System on Chip. 29, 36, 53, 91, 118, 122, 126, 137, 148

SPI Security Parameters Index. 154–156

SPI Serial Peripheral Interface bus. 30

SPL Secondary Program Loader. 37, 39

SRAM Static RAM. 28, 35–37, 118

SRK Super Root Key. 138–141, 170

SSH Secure Shell. 27, 28, 110

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, DoS and Elevation of privilege. 44, 47, 51, 80–88, 90–108

TAP network Terminal Access Point. 150

TBM Trusted Boot Module. 33, 34, 37, 63, 66, 68, 73, 74, 81, 85, 91, 92, 98, 99, 117–119, 121, 122, 125, 126, 131, 134

TCG Trusted Computing Group. 30

TCP Transmission Control Protocol. 26, 28, 89, 129, 151, 152

TCP/IP TCP/IP. 12, 25, 26, 151

TDMA time-division multiple access. 12

TEE Trusted Execution Environment. 31–33, 37, 175

TFTP Trivial File Transfer Protocol. 132

TLS Transport Layer Security. 23, 24, 26–28, 127

TOTP Time-Based One-time password. 34

TP Trusted Platform. 30–32

TPL Ternary Program Loader. 37, 39

TPM Trusted Platform Module. 30–33, 37, 63, 128, 131, 134

TUN network TUNnel. 150–152

TXT Trusted Execution Technology. 31

U2F Universal Second Factor. 34, 35

UART Universal Asynchronous Receiver Transmitter. 151

UDP User Datagram Protocol. 26, 28, 89, 129, 130, 152, 154, 156, 157

UDP/IP UDP/IP. 12, 25, 26, 53

UEFI Unified EFI. 32, 33, 37, 39

UML Unified Modeling Language. 49

UPS Uninterruptible Power Supply. 66, 85

VPN virtual private network. 27, 86, 110

WAN Wide Area Network. 62, 67, 68

WPAN wireless PAN. 12, 13, 24

WSN wireless sensor network. 12, 13, 166

XIP execute in place. 35, 36, 39, 53

References

[1] A. I. Elkhawas and M. A. Azer, "Security perspective in rami 4.0", in *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, IEEE, 2018, pp. 151–156. DOI: 10.1109/ICCES.2018.8639235.

[2] M. Illarramendi, L. Etxeberria, X. Elkorobarrutia, and G. Sagardui, "Increasing dependability in safety critical cpss using reflective statecharts", in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds., Springer International Publishing, 2017, pp. 114–126, ISBN: 978-3-319-66283-1. DOI: 10.1007/978-3-319-66284-8_11.

[3] T. Warns, *Structural Failure Models for Fault-Tolerant Distributed Computing*. Vieweg+Teubner Verlag, 2010, ISBN: 978-3-834-81287-2. DOI: 10.1007/978-3-8348-9707-7.

[4] T. e. a. Bijlsma, "Security challenges for cooperative and interconnected mobility systems", in *Critical Information Infrastructures Security: 8th International Workshop, CRITIS 2013, Amsterdam, The Netherlands, September 16-18, 2013, Revised Selected Papers*, E. Luiijf and P. Hartel, Eds. Springer International Publishing, 2013, pp. 1–15, ISBN: 978-3-319-03963-3. DOI: 10.1007/978-3-319-03964-0_1.

[5] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution", in *Design Automation Conference*, ACM, 2010, pp. 731–736, ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837461.

[6] M. Patton, E. Gross, R. Chinn, S. Forbis, L. Walker, and H. Chen, "Uninvited connections: A study of vulnerable devices on the internet of things (iot)", in *2014 IEEE Joint Intelligence and Security Informatics Conference*, IEEE, 2014, pp. 232–235, ISBN: 978-1-4799-6364-5. DOI: 10.1109/JISIC.2014.43.

[7] E. Mitchell and J. S. Park, "Noah for iot: Cybersecurity strategy for home applications", in *Proceedings of the 2017 International Conference on Security and Management*, Las Vegas, USA: CSREA Press, 2017, pp. 68–74, ISBN: 978-1-601-32467-2. [Online]. Available: http://csce.ucmss.com/cr/books/2017/LFS/CSREA2017/SAM9734.pdf (visited on 08/01/2019).

[8] T. Novak, "Functional safety and system security in building automation and control systems : A common approach", PhD thesis, TU Wien, TUW, 2008. [Online]. Available: https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-21828 (visited on 08/01/2019).

[9] H. Fourati, *Multisensor Data Fusion: From Algorithms and Architectural Design to Applications*. CRC Press, 2016, ISBN: 978-1-315-21499-3. DOI: 10.1201/b18851.

[10] M. Hager, P. Gromala, B. Wunderle, and S. Rzepka, "Affordable and safe high performance vehicle computers with ultra-fast on-board ethernet for automated driving", in *Advanced Microsystems for Automotive Applications 2018*, J. Dubbert, B. Müller, and G. Meyer, Eds., Springer International Publishing, 2018, pp. 56–68, ISBN: 978-3-319-99761-2. DOI: 10.1007/978-3-319-99762-9_5.

[11] T. Sauter, "The three generations of field-level networks, Evolution and compatibility issues", *IEEE Transactions on Industrial Electronics*, vol. 57, no. 11, pp. 3585–3595, 2010. DOI: 10.1109/TIE.2010.2062473.

[12] S. Bush, *Smart Grid: Communication-Enabled Intelligence for the Electric Power Grid*. Wiley, 2014, ISBN: 978-1-119-97580-9. DOI: 10.1002/9781118820216.

[13] T. Macaulay and B. L. Singer, *Cybersecurity for Industrial Control Systems: SCADA, DCS, PLC, HMI, and SIS*. Auerbach Publications, 2012, ISBN: 978-0-429-163807. DOI: 10.1201/b11352.

[14] J. Beyerer, A. Pak, and M. Taphanel, Eds., *Deterministic Industrial Network Communication: Fundamentals*, Karlsruher Schriften zur Anthropomatik / Lehrstuhl für Interaktive Echtzeitsysteme, Karlsruher Institut für Technologie ; Fraunhofer-Inst. für Optronik, Systemtechnik und Bildauswertung IOSB Karlsruhe, KIT Scientific Publishing, Karlsruhe, 2018, ISBN: 978-3-7315-0779-6. DOI: 10.5445/KSP/1000081314.

[15] W. Kastner, G. Neugschwandtner, S. Soucek, and H. M. Newman, "Communication systems for building automation and control", *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1178–1203, 2005. DOI: 10.1109/JPROC.2005.849726.

[16] T. Sauter, S. Soucek, W. Kastner, and D. Dietrich, "The evolution of factory and building automation", *IEEE Industrial Electronics Magazine*, vol. 5, no. 3, pp. 35–48, 2011. DOI: 10.1109/MIE.2011.942175.

[17] T. Sauter, "The continuing evolution of integration in manufacturing automation", *IEEE Industrial Electronics Magazine*, vol. 1, no. 1, pp. 10–19, 2007. DOI: 10.1109/MIE.2007.357183.

[18] ——, "Integration aspects in automation - a technology survey", in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, IEEE, 2005, pp. 255–263, ISBN: 978-0-780-39401-8. DOI: 10.1109/ETFA.2005.1612688.

[19] S. Seifried, "Reliable control network gateways, A case study for knx and zigbee", Master's Thesis, TU Wien, TUW, 2015. [Online]. Available: https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-78656 (visited on 08/01/2019).

[20] N. Fallenbeck and C. Eckert, "It-sicherheit und cloud computing", in *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung ů Technologien ů Migration*, T. Bauernhansl, M. ten Hompel, and B. Vogel-Heuser, Eds. Springer Fachmedien Wiesbaden, 2014, pp. 397–431, ISBN: 978-3-658-04681-1. DOI: 10.1007/978-3-658-04682-8_20.

[21] M. Neubauer and C. Stary, *S-BPM in the Production Industry: A Stakeholder Approach*. Springer International Publishing, 2016, ISBN: 978-3-319-48465-5. DOI: 10.1007/978-3-319-48466-2.

[22] R. Heinze, C. Manzei, and L. Schleupner, *Industrie 4.0 im internationalen Kontext: Kernkonzepte, Ergebnisse, Trends*. Beuth Verlag GmbH, 2017, ISBN: 978-3-410-27602-9. [Online]. Available: https://www.beuth.de/de/publikation/industrie-4-0-im-internationalen-kontext/272185844 (visited on 08/01/2019).

[23] A. Corradi, L. Foschini, C. Giannelli, R. Lazzarini, C. Stefanelli, M. Tortonesi, and G. Virgilli, "Smart Appliances and RAMI 4.0: Management and Servitization of Ice Cream Machines", *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1007–1016, 2018. DOI: 10.1109/TII.2018.2867643.

[24] T. Meudt, M. Pohl, and J. Metternich, "Die Automatisierungspyramide - ein Literaturüberblick", TU Darmstadt, 2017. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:tuda-tuprints-62982 (visited on 08/01/2019).

[25] H. Flatt, S. Schriegel, J. Jasperneite, H. Trsek, and H. Adamczyk, "Analysis of the cyber-security of industry 4.0 technologies based on rami 4.0 and identification of requirements", in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2016, pp. 1–4, ISBN: 978-1-5090-1314-2. DOI: 10.1109/ETFA.2016.7733634.

[26] Z. Ma, A. Hudic, A. Shaaban, and S. Plosz, "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, 2017, pp. 153–159, ISBN: 978-1-5386-2244-5. DOI: 10.1109/EuroSPW.2017.65.

[27] T. Sauter, "Fieldbus system fundamentals", in *Industrial Communication Technology Handbook, Second Edition*. CRC Press, 2015, pp. 1–1–1–50, ISBN: 978-1-315-21548-8. DOI: 10.1201/b17365-2.

[28] M. Felser and T. Sauter, "The fieldbus war: History or short break between battles?", in *4th IEEE International Workshop on Factory Communication Systems*, IEEE, 2002, pp. 73–80, ISBN: 978-0-780-37586-4. DOI: 10.1109/WFCS.2002.1159702.

[29] M. Felser, "Real-time ethernet for automation applications", in *Industrial Communication Technology Handbook, Second Edition*. CRC Press, 2017, pp. 17–1–17–22, ISBN: 978-1-315-21548-8. DOI: 10.1201/b17365-18.

[30] L. Zheng, "Industrial wireless sensor networks and standardizations: The trend of wireless sensor networks for process autometion", in *Proceedings of SICE Annual Conference 2010*, IEEE, 2010, pp. 1187–1190, ISBN: 978-1-424-47642-8. [Online]. Available: https://ieeexplore.ieee.org/document/5602869 (visited on 08/01/2019).

[31] M. d. l. A. C. Leon, J. I. N. Hipolito, and J. L. Garcia, "A security and privacy survey for wsn in e-health applications", in *2009 Electronics, Robotics and Automotive Mechanics Conference (CERMA)*, IEEE, 2009, pp. 125–130, ISBN: 978-0-7695-3799-3. DOI: 10.1109/CERMA.2009.47.

[32] A. Vidács and R. Vida, "Wireless sensor network based technologies for critical infrastructure systems", in *Intelligent Monitoring, Control, and Security of Critical Infrastructure Systems*, E. Kyriakides and M. Polycarpou, Eds. Springer Berlin Heidelberg, 2014, pp. 301–316, ISBN: 978-3-662-44159-6. DOI: 10.1007/978-3-662-44160-2_11.

[33] E. A. Lee, "Cyber physical systems: Design challenges", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-8, 2008. [Online]. Available: http://www2.eecs.berkeley.

edu/Pubs/TechRpts/2008/EECS-2008-8.html (visited on 08/01/2019).

[34] N. Jazdi, "Cyber physical systems in the context of industry 4.0", in *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, IEEE, 2014, pp. 114–126, ISBN: 978-1-479-93731-8. DOI: 10.1109/AQTR.2014.6857843.

[35] A. Sonalker and E. Griffor, "Evolving security", in *Handbook of System Safety and Security*, E. Griffor, Ed. Elsevier Science, 2017, pp. 67–82, ISBN: 978-0-128-03773-7. DOI: 10.1016/b978-0-12-803773-7.00004-8.

[36] W. Granzer, F. Praus, and W. Kastner, "Security in building automation systems", *IEEE Transactions on Industrial Electronics*, vol. 57, no. 11, pp. 3622–3630, 2010. DOI: 10.1109/TIE.2009.2036033.

[37] T. Novak, "Embedded security in safety critical automation systems", in *26th International System Safety Conference*, Curran Associates, 2008, ISBN: 978-1-615-67364-3.

[38] C. Schwaiger and A. Treytl, "Smart card based security for fieldbus systems", in *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.03TH8696)*, vol. 1, IEEE, Sep. 2003, 398–406 vol.1, ISBN: 978-0-780-37937-4. DOI: 10.1109/ETFA.2003.1247734.

[39] A. Treytl, T. Sauter, and C. Schwaiger, "Security measures in automation systems-a practice-oriented approach", in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, IEEE, 2005, pp. 847–855, ISBN: 978-0-780-39401-8. DOI: 10.1109/ETFA.2005.1612762.

[40] M. Meyers and S. Jernigan, *Mike Meyers' CompTIA Security+ Certification Guide, Second Edition (Exam SY0-501)*. McGraw-Hill Education, 2017, ISBN: 978-1-260-02637-5. [Online]. Available: https://books.google.at/books?id=YFFBDwAAQBAJ (visited on 08/01/2019).

[41] M. Hafner and R. Breu, *Security Engineering for Service-Oriented Architectures*. Springer Berlin Heidelberg, 2008, ISBN: 978-3-540-79538-4. DOI: 10.1007/978-3-540-79539-1.

[42] S. Oriyano, *Penetration Testing Essentials*. Wiley, 2017, ISBN: 978-1-119-23530-9. DOI: 10.1002/9781119419358.

[43] O. S. Faragallah, E.-S. M. El-Rabaie, F. E. A. El-Samie, A. I. Sallam, and H. S. El-Sayed, *Multilevel Security for Relational Databases*. CRC Press, 2014, ISBN: 978-0-429-09054-7. DOI: 10.1201/b17719.

[44] J. Stapleton, *Security without Obscurity: A Guide to Confidentiality, Authentication, and Integrity*. CRC Press, 2014, ISBN: 978-0-429-16770-6. DOI: 10.1201/b16885.

[45] M. Ciampa, *CompTIA Security+ Guide to Network Security Fundamentals*. Cengage Learning, 2017, ISBN: 978-1-337-28878-1. [Online]. Available: https://books.google.at/books?id=LMw2DwAAQBAJ (visited on 08/01/2019).

[46] R. Oppliger, *SSL and TLS: Theory and Practice*. Artech House, 2009, ISBN: 978-1-596-93447-4. [Online]. Available: https://books.google.at/books?id=FGWWmAEACAAJ (visited on 08/01/2019).

[47] R. Spenneberg, *VPN mit Linux: Grundlagen und Anwendung virtueller privater Netzwerke mit Open-Source-Tools*. Pearson Deutschland, 2010, ISBN: 978-3-827-32515-0. [Online]. Available: https://os-s.de/buecher/vpn_buch2.pdf (visited on 07/30/2019).

[48] M. Gregg, *CASP CompTIA Advanced Security Practitioner Study Guide: Exam CAS-002*. Wiley, 2014, ISBN: 978-1-118-93084-7. [Online]. Available: https://books.google.at/books?id=LKPCBwAAQBAJ (visited on 08/01/2019).

[49] J. Tiller, *A Technical Guide to IPSec Virtual Private Networks*. CRC Press, 2001, ISBN: 978-0-429-22517-8. DOI: 10.1201/9780203997499.

[50] Y. Frankel, D. W. Kravitz, C. T. Montgomery, and M. Yung, "Beyond identity: Warranty-based digital signature transactions", in *Financial Cryptography*, R. Hirchfeld, Ed., Springer Berlin Heidelberg, 1998, pp. 241–253, ISBN: 978-3-540-64951-9. DOI: 10.1007/bfb0055487.

[51] J. Stapleton and W. Epstein, *Security without Obscurity: A Guide to PKI Operations*. CRC Press, 2016, ISBN: 978-0-429-16033-2. DOI: 10.1201/b19725.

[52] S. Turner and R. Housley, *Implementing Email and Security Tokens: Current Standards, Tools, and Practices*. Wiley, 2008, ISBN: 978-0-470-38142-7. [Online]. Available: https://books.google.at/books?id=4SD34D8Jvc0C (visited on 08/01/2019).

[53] J. Buchmann, E. Karatsiolis, and A. Wiesmaier, *Introduction to Public Key Infrastructures*. Springer Berlin Heidelberg, 2013, ISBN: 978-3-642-40656-0. DOI: 10.1007/978-3-642-40657-7.

[54] H. Kim, Y. Cho, S. Jin, and S. M. Chung, "Advanced public key infrastructure for internet security, Second edition", in *Encyclopedia of Library and Information Science*, M. Dekker, Ed., 2003, pp. 82–89, ISBN: 978-0-824-72075-9. [Online]. Available: https://books.google.at/books?id=ef1YDwAAQBAJ (visited on 08/01/2019).

[55] T. Sauter and A. Treytl, "Security in industrial communications", in *Industrial Communication Technology Handbook, Second Edition*. CRC Press, 2015, pp. 29–1–29–23, ISBN: 978-1-315-21548-8. DOI: 10.1201/b17365-31.

[56] W. Köhler, "Simulation of a KNX network with EIBsec protocol extensions", Master's thesis, TU Wien, TUW, 2008. [Online]. Available: https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-24571 (visited on 08/01/2019).

[57] R. Rosen, *Linux Kernel Networking: Implementation and Theory*. Apress, 2014, ISBN: 978-1-430-26196-4. DOI: 10.1007/978-1-4302-6197-1.

[58] C. Meinel and H. Sack, *Internetworking: Technological Foundations and Applications*. Springer Berlin Heidelberg, 2013, ISBN: 978-3-642-35391-8. DOI: 10.1007/978-3-642-35392-5.

[59] S. P. Skorobogatov, "Semi-invasive attacks – A new approach to hardware security analysis", University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-630, Apr. 2005. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf (visited on 08/01/2019).

[60] K. Borgeest, *Elektronik in der Fahrzeugtechnik: Hardware, Software, Systeme und Projektmanagement*. Springer Vieweg, 2014, ISBN: 978-3-8348-1642-9. DOI: 10.1007/978-3-8348-2145-4.

[61] F. Kesel and R. Bartholomä, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs: Einführung mit VHDL und SystemC*. De Gruyter, 2013, ISBN: 978-3-486-74715-7. DOI: 10.1524/9783486747157.

[62] N. Pohlmann and H. Reimer, *Trusted Computing: Ein Weg zu neuen IT-Sicherheitsarchitekturen*. Vieweg+Teubner Verlag, 2008, ISBN: 978-3-8348-0309-2. DOI: 10.1007/978-3-8348-9452-6.

[63] C. Mitchell, "What is trusted computing?", in *Trusted Computing*, C. Mitchell, Ed., Institution of Engineering and Technology, 2005, pp. 1–10, ISBN: 978-1-849-19047-3. DOI: 10.1049/pbpc006e_ch1.

[64] H. Löhr, A.-R. Sadeghi, C. Stüble, M. Weber, and M. Winandy, "Modeling trusted computing support in a protection profile for high assurance security kernels", in *Trusted Computing: Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings*. L. Chen, C. J. Mitchell, and A. Martin, Eds. Springer Berlin Heidelberg, 2009, pp. 45–62, ISBN: 978-3-642-00586-2. DOI: 10.1007/978-3-642-00587-9_4.

[65] M. Schramm and A. Grzemba, "Trusted computing concepts for resilient embedded networks", Academy of Science and Engineering (ASE), USA,ľ ASE 2014, 2014.

[66] J.-E. Ekberg and M. Kylänpää, "Mobile trusted module (mtm) - an introduction", Nokia Research Center Helsinki, Tech. Rep. NRC-TR-2007-015, 2007. [Online]. Available: https://pdfs.semanticscholar.org/450b/e1d34f3a9c4879d8db1350410daae522a604.pdf (visited on 08/01/2019).

[67] G. J. Proudler, "Concepts of trusted computing", in *Trusted Computing*, C. Mitchell, Ed. Institution of Engineering and Technology, 2005, pp. 11–27, ISBN: 978-1-849-19047-3. DOI: 10.1049/pbpc006e_ch2.

[68] E. Gallery and C. J. Mitchell, "Trusted mobile platforms", in *Foundations of Security Analysis and Design IV: FOSAD 2006 / 2007 Tutorial Lectures*, A. Aldini and R. Gorrieri, Eds. Springer Berlin Heidelberg, 2007, pp. 282–323, ISBN: 978-3-540-74809-0. DOI: 10.1007/978-3-540-74810-6_10.

[69] W. Stanek, *Introducing Microsoft Windows Vista*. Microsoft Press, 2006, ISBN: 978-0-735-62284-5. [Online]. Available: https://books.google.at/books?id=-ANomF3oARkC (visited on 08/01/2019).

[70] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not", in *2015 IEEE Trustcom/BigDataSE/ISPA*, Springer, Cham, 2015, pp. 57–64, ISBN: 978-1-4673-7952-6. DOI: 10.1109/Trustcom.2015.357.

[71] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "Trusted execution environments on mobile devices", in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, 2013, pp. 1497–1498, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516758.

[72] A. Umar and K. Mayes, "Trusted execution environment and host card emulation", in *Smart Cards, Tokens, Security and Applications*, K. Mayes and K. Markantonakis, Eds. Springer International Publishing, 2017, pp. 497–519, ISBN: 978-3-319-50498-8. DOI: 10.1007/978-3-319-50500-8_18.

[73] H. Janjua, W. Joosen, S. Michiels, and D. Hughes, "Trusted operations on mobile phones", in *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ACM, 2017, pp. 452–459, ISBN: 978-1-4503-5368-7. DOI: 10.1145/3144457.3144502.

[74] W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015, ISBN: 978-1-4302-6583-2. DOI: 10.1007/978-1-4302-6584-9.

[75] R. Yeluri and E. Castro-Leon, *Building the Infrastructure for Cloud Security: A Solutions View*. Apress, 2014, ISBN: 978-1-430-26145-2. DOI: 10.1007/978-1-4302-6146-9.

[76] K. Kursawe, "The future of trusted computing: An outlook", in *Trusted Computing*, C. Mitchell, Ed., Institution of Engineering and Technology, 2005, pp. 299–304, ISBN: 978-1-849-19047-3. DOI: 10.1049/PBPC006E_ch11.

[77] Z. Zhou and R. Xu, "Bios security analysis and a kind of trusted bios", in *Information and Communications Security: 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007. Proceedings*, S. Qing, H. Imai, and G. Wang, Eds. Springer Berlin Heidelberg, 2007, pp. 427–437, ISBN: 978-3-540-77047-3. DOI: 10.1007/978-3-540-77048-0_33.

[78] N. Lieberknecht, "Application of trusted computing in automation to prevent product piracy", in *Trust and Trustworthy Computing: Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, A. Acquisti, S. W. Smith, and A.-R. Sadeghi, Eds. Springer Berlin Heidelberg, 2010, pp. 95–108, ISBN: 978-3-642-13868-3. DOI: 10.1007/978-3-642-13869-0_7.

[79] H. van Tilborg and S. Jajodia, *Encyclopedia of Cryptography and Security*. Springer Boston, 2011, ISBN: 978-1-441-95905-8. DOI: 10.1007/978-1-4419-5906-5.

[80] R. N. Akram, K. Markantonakis, and K. Mayes, "An introduction to the trusted platform module and mobile trusted module", in *Secure Smart Embedded Devices, Platforms and Applications*, K. Markantonakis and K. Mayes, Eds. Springer New York, 2014, pp. 71–93, ISBN: 978-1-4614-7914-7. DOI: 10.1007/978-1-4614-7915-4_4.

[81] A. Tomlinson, "Introduction to the TPM", in *Smart Cards, Tokens, Security and Applications*, K. Mayes and K. Markantonakis, Eds., Springer International Publishing, 2017, pp. 173–191, ISBN: 978-3-319-50498-8. DOI: 10.1007/978-3-319-50500-8_7.

[82] F. J. Krautheim, D. S. Phatak, and A. T. Sherman, "Introducing the trusted virtual environment module: A new mechanism for rooting trust in cloud computing", in *Trust and Trustworthy Computing: Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, A. Acquisti, S. W. Smith, and A.-R. Sadeghi, Eds. Springer Berlin Heidelberg, 2010, pp. 211–227, ISBN: 978-3-642-13868-3. DOI: 10.1007/978-3-642-13869-0_14.

[83] G. Proudler, L. Chen, and C. Dalton, *Trusted Computing Platforms: TPM2.0 in Context*. Springer International Publishing, 2014, ISBN: 978-3-319-08743-6. DOI: 10.1007/978-3-319-08744-3.

[84] T. Unterluggauer and S. Mangard, "Exploiting the physical disparity: Side-channel attacks on memory encryption", in *Constructive Side-Channel Analysis and Secure Design*, F.-X. Standaert and E. Oswald, Eds., Springer International Publishing, 2016, pp. 3–18, ISBN: 978-3-319-43282-3. DOI: 10.1007/978-3-319-43283-0_1.

[85] V. Raes, J. Vossaert, and V. Naessens, "Development of an embedded platform for secure cps services", in *Computer Security – SECPRE 2017, CyberICPS 2017*, S. K. Katsikas, F. Cuppens, N. Cuppens, C. Lambrinoudakis, C. Kalloniatis, J. Mylopoulos, A. Antón, and S. Gritzalis, Eds., Springer International Publishing, 2018, pp. 19–34, ISBN: 978-3-319-72816-2. DOI: 10.1007/978-3-319-72817-9_2.

[86] I. Loutfi and A. Jøsang, "Fido trust requirements", in *Secure IT Systems*, S. Buchegger and M. Dam, Eds., Springer International Publishing, 2015, pp. 139–155, ISBN: 978-3-319-26501-8. DOI: 10.1007/978-3-319-26502-5_10.

[87] C. Rath, S. Roth, H. Bratko, and T. Zefferer, "Encryption-based second authentication factor solutions for qualified server-side signature creation", in *Electronic Government and the Information Systems Perspective*, A. K and E. Francesconi, Eds., Springer International Publishing, 2015, pp. 71–85, ISBN: 978-3-319-22388-9. DOI: 10.1007/978-3-319-22389-6_6.

[88] C. Eckert, *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter, 2018, ISBN: 978-3-11-056390-0. DOI: 10.1515/9783110563900.

[89] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things", in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2013, pp. 79–80, ISBN: 978-1-4799-0055-8. DOI: 10.1109/INFCOMW.2013.6970748.

[90] P. Liggesmeyer and M. Trapp, "Trends in embedded software engineering", *IEEE Software*, vol. 26, no. 3, pp. 19–25, 2009. DOI: 10.1109/MS.2009.80.

[91] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular cps architecture design based on ros and docker", *International Journal on Interactive Design and Manufacturing (IJIDeM)*, pp. 949–955, 2016. DOI: 10.1007/s12008-016-0313-8.

[92] P. Raghavan, A. Lad, and S. Neelakandan, *Embedded Linux System Design and Development*. CRC Press, 2005, ISBN: 978-0-429-13505-7. DOI: 10.1201/9781420031614.

[93] C. Simmonds, *Mastering Embedded Linux Programming*. Packt Publishing, 2017, ISBN: 978-1-787-28885-0. [Online]. Available: https://books.google.at/books?id=4Hc5DwAAQBAJ (visited on 08/01/2019).

[94] C. Gu, *Building Embedded Systems: Programmable Hardware*. Apress, 2016, ISBN: 978-1-484-21918-8. DOI: 10.1007/978-1-4842-1919-5.

[95] A. Vaduva, A. Gonzalez, and C. Simmonds, *Linux: Embedded Development*. Packt Publishing, 2016, ISBN: 978-1-787-12445-5. [Online]. Available: https://books.google.at/books?id=yIJcDgAAQBAJ (visited on 08/01/2019).

[96] R. Streif, *Embedded Linux Systems with the Yocto Project*. Pearson Education, 2016, ISBN: 978-0-133-44328-8. [Online]. Available: https://books.google.at/books?id=xT7-CwAAQBAJ (visited on 08/01/2019).

[97] R. Smith, *CompTIA Linux+ Study Guide: Exams LX0-101 and LX0-102*. Wiley, 2012, ISBN: 978-1-118-57034-0. [Online]. Available: https://books.google.at/books?id=G-nxDXP6YL4C (visited on 08/01/2019).

[98] M. Jang, *Security Strategies in Linux Platforms and Applications*. Jones & Bartlett Learning, 2010, ISBN: 978-0-763-79189-6. [Online]. Available: https://books.google.at/books?id=BmMcoQL2LI4C (visited on 08/01/2019).

[99] A. González, *Embedded Linux Development Using Yocto Project Cookbook: Practical recipes to help you leverage the power of Yocto to build exciting Linux-based systems, 2nd Edition*. Packt Publishing, 2018, ISBN: 978-1-788-39292-1. [Online]. Available: https://books.google.at/books?id=6NRJDwAAQBAJ (visited on 08/01/2019).

[100] J. Wakerly, "Computer organization and programming", in *Reference Data for Engineers: Radio, Electronics, Computers and Communications*, M. Van Valkenburg and W. Middleton, Eds. Elsevier Science, 2002, pp. 42-1-42–33, ISBN: 978-0-750-67291-7. DOI: 10.1016/B978-075067291-7/50044-3.

[101] R. Khan, K. Ghoshdastidar, and A. Vasudevan, *Learning IoT with Particle Photon and Electron*. Packt Publishing, 2016, ISBN: 978-1-785-88734-5. [Online]. Available: https://books.google.at/books?id=e4FcDgAAQBAJ (visited on 08/01/2019).

[102] F. Kohnhäuser and S. Katzenbeisser, "Secure code updates for mesh networked commodity low-end embedded devices", in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds., Springer International Publishing, 2016, pp. 320–338, ISBN: 978-3-319-45740-6. DOI: 10.1007/978-3-319-45741-3_17.

[103] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat, "Dynamic software update of resource-constrained distributed embedded systems", in *Embedded System Design: Topics, Techniques and Trends*, A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F. J. Rammig, Eds., Springer US, 2007, pp. 387–400, ISBN: 978-0-387-72257-3. DOI: 10.1007/978-0-387-72258-0_33.

[104] J. Jeong and D. Culler, "Incremental network programming for wireless sensors", in *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, IEEE, 2004, pp. 25–33, ISBN: 978-0-7803-8796-6. DOI: 10.1109/SAHCN.2004.1381899.

[105] F. Dressler, M. Strübe, R. Kapitza, and W. Schröder-Preikschat, "Dynamic software management on btnode sensors", in *IN: 4TH IEEE / ACM International Conference On Distributed Computing In Sensor Systems (IEEE/ACM DCOSS 2008): IEEE/ACM International Workshop on Sensor Network Engineering (IWSNE '08), Santorini Island, Greece*, IEEE, 2008, pp. 9–14.

[106] F. Swiderski and W. Snyder, *Threat Modeling*. O'Reilly Media, 2004, ISBN: 978-0-735-63769-6. [Online]. Available: https://books.google.at/books?id=qWjoUuFSmf8C (visited on 08/01/2019).

[107] A. Shostack, *Threat Modeling: Designing for Security*. Wiley, 2014, ISBN: 978-1-118-81005-7. [Online]. Available: https://books.google.at/books?id=YiHcAgAAQBAJ (visited on 08/01/2019).

[108] J. Bürger, J. Jürjens, T. Ruhroth, S. Gärtner, and K. Schneider, "Model-based security engineering: Managed co-evolution of security knowledge and software models", in *Foundations of Security Analysis and Design VII: FOSAD 2012 / 2013 Tutorial Lectures*, A. Aldini, J. Lopez, and F. Martinelli, Eds. Springer International Publishing, 2014, pp. 34–53, ISBN: 978-3-319-10081-4. DOI: 10.1007/978-3-319-10082-1_2.

[109] B. Stackpole and P. Hanrion, *Software Deployment, Updating, and Patching*. CRC Press, 2007, ISBN: 978-0-429-13403-6. DOI: 10.1201/9781420013290.

[110] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems", in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, IEEE, 2017, pp. 1–6, ISBN: 978-1-538-61954-4. DOI: 10.1109/ISGTEurope.2017.8260283.

[111] M. Solomon, *Security Strategies in Windows Platforms and Applications*. Jones & Bartlett Learning, 2013, ISBN: 978-1-284-03166-9. [Online]. Available: https://books.google.at/books?id=hYhSAAAAQBAJ (visited on 08/01/2019).

[112] M. Morana and T. UcedaVelez, *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. Wiley, 2015, ISBN: 978-0-470-50096-5. DOI: 10.1002/9781118988374.

[113] S. Myagmar, A. J. Lee, and W. Yurcik, "Threat modeling as a basis for security requirements", in *Symposium on requirements engineering for information security (SREIS)*, 2005, pp. 1–8.

[114] P. H. Engebretson and J. J. Pauli, "Leveraging parent mitigations and threats for capec-driven hierarchies", in *2009 Sixth International Conference on Information Technology: New Generations*, IEEE, 2009, pp. 344–349, ISBN: 978-1-424-43770-2. DOI: 10.1109/ITNG.2009.24.

[115] N. A. Malik, M. Y. Javed, and U. Mahmud, "Threat modeling in pervasive computing paradigm", in *2008 New Technologies, Mobility and Security*, IEEE, 2008, pp. 1–5, ISBN: 978-1-424-43547-0. DOI: 10.1109/NTMS.2008.ECP.97.

[116] W. A. Conklin, "Threat modeling and secure software engineering process", in *Handbook of Research on Information Security and Assurance*, J. N. D. G. S̃. K. Sharma, Ed. IGI Global, 2009, pp. 415–422, ISBN: 978-1-599-04855-0. DOI: 10.4018/978-1-59904-855-0.ch036.

[117] A. Jürgenson and J. Willemson, "Serial model for attack tree computations", in *Information, Security and Cryptology – ICISC 2009*, D. Lee and S. Hong, Eds., Springer Berlin Heidelberg, 2010, pp. 118–128, ISBN: 978-3-642-14422-6. DOI: 10.1007/978-3-642-14423-3_9.

[118] M. van der Linden, *Testing Code Security*. Auerbach Publications, 2007, ISBN: 978-0-429-18603-5. DOI: 10.1201/9781420013795.

[119] D. Mills, *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. CRC Press, 2011, ISBN: 978-1-315-21815-1. DOI: 10.1201/b10282.

[120] B. Schneier, *Applied Cryptography, Second Edition: Protocols, Algorithms and Source Code in C*. Wiley, 2015, ISBN: 978-0-471-12845-8. DOI: 10.1002/9781119183471.

[121] E.. Tatl, "Cracking more password hashes with patterns", *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 8, pp. 1656–1665, 2015. DOI: 10.1109/TIFS.2015.2422259.

[122] D. Meyer, J. Haase, M. Eckert, and B. Klauer, "A threat-model for building and home automation", in *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, IEEE, 2016, pp. 860–866, ISBN: 978-1-509-02871-9. DOI: 10.1109/INDIN.2016.7819280.

[123] A. Mouat, *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2015, ISBN: 978-1-491-91592-9. [Online]. Available: https://books.google.at/books?id=wpYpCwAAQBAJ (visited on 08/01/2019).

[124] V. Velu, *Mobile Application Penetration Testing*. Packt Publishing, 2016, ISBN: 978-1-785-88869-4. [Online]. Available: https://books.google.at/books?id=Y0XiCwAAQBAJ (visited on 08/01/2019).

[125] M. Rhee, *Internet Security: Cryptographic Principles, Algorithms and Protocols*. Wiley, 2003, ISBN: 978-0-470-85285-9. [Online]. Available: https://books.google.at/books?id=bJJUVNGbrLsC (visited on 08/01/2019).

[126] J. Kelsey, B. Schneier, and D. Wagner, "Protocol interactions and the chosen protocol attack", in *Security Protocols*, B. Christianson, B. Crispo, M. Lomas, and M. Roe, Eds., Springer Berlin Heidelberg, 1998, pp. 91–104, ISBN: 978-3-540-64040-0. DOI: 10.1007/BFb0028162.

[127] N. Elenkov, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014, ISBN: 978-1-593-27641-6. [Online]. Available: https://books.google.at/books?id=-QcvDwAAQBAJ (visited on 08/01/2019).

[128] E. Crist and J. Keijser, *Mastering OpenVPN*. Packt Publishing, 2015, ISBN: 978-1-783-55314-3. [Online]. Available: https://books.google.at/books?id=O-13CgAAQBAJ (visited on 08/01/2019).

[129] J. Langbridge, *Arduino Sketches: Tools and Techniques for Programming Wizardry*. Wiley, 2014, ISBN: 978-1-118-91960-6. DOI: 10.1002/9781119183716.

[130] A. Talukder, N. Garcia, and J. M, *Convergence Through All-IP Networks*. Pan Stanford, 2013, ISBN: 978-0-429-08844-5. DOI: 10.1201/b15463.

[131] M. Rohrer, "Strongswan vpn plasmoid unter kde", Bachelor's Thesis, Hochschule für Technik Rapperswil, HSR, 2010. [Online]. Available: https://eprints.hsr.ch/94/1/strongswan_vpn_plasmaoid_KDE.pdf (visited on 08/01/2019).

[132] P. Varga, S. Plosz, G. Soos, and C. Hegedus, "Security threats and issues in automation iot", in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, IEEE, 2017, pp. 1–6, ISBN: 978-1-509-05789-4. DOI: 10.1109/WFCS.2017.7991968.

[133] A. Sengupta and S. Kundu, "Guest editorial securing iot hardware: Threat models and reliable, low-power design solutions", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3265–3267, 2017. DOI: 10.1109/TVLSI.2017.2762398.

[134] S. Sau, J. Haj-Yahya, M. M. Wong, K. Y. Lam, and A. Chattopadhyay, "Survey of secure processors", in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, 2017, pp. 253–260, ISBN: 978-1-538-63438-7. DOI: 10.1109/SAMOS.2017.8344637.

[135] C. K. Keerthi, M. A. Jabbar, and B. Seetharamulu, "Cyber physical systems(cps): Security issues, challenges and solutions", in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, IEEE, 2017, pp. 1–4, ISBN: 978-1-5090-6622-3. DOI: 10.1109/ICCIC.2017.8524312.

[136] V. Gokarn, V. Kulkarni, and P. Singh, "Enhancing cyber physical system security via anomaly detection using behaviour analysis", in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, IEEE, 2017, pp. 944–948, ISBN: 978-1-5090-4443-6. DOI: 10.1109/WiSPNET.2017.8299901.

Standards & RFCs

[EN16484-2] *Building automation and control systems (bacs), Part 2: Hardware*, Norm, EN ISO 16484-2:2004, 2004.

[EN61158-1] *Industrielle kommunikationsnetze - feldbusse, Teil 1: Überblick und leitfaden zu den normen der reihe iec 61158 und iec 61784*, Norm, DIN EN 61158-1:2015, 2015.

[EN61508] *Functional safety of electrical/electronic/programmable electronic safety-related systems*, Norm, DIN EN 61508, 2010.

[EN62264-1] "Integration von unternehmensführungs- und leitsystemen, Teil 1: Modelle und terminologie", European Norm, Standard, 2014, DIN EN 62264-1.

[EN62890] *Life-cycle management for systems and products used in industrial-process measurement, control and automation (draft)*, Norm, EN 62890: 2016, 2016.

[IEEE 1003.1] "IEEE 1003.1-2001 - portable operating system interface (POSIX(r))", IEEE, Standard, 2001. DOI: 10.1109/IEEESTD.2001.93364.

[IEEE 1003.1g] "IEEE 1003.1g-2000 - protocol-independent interfaces", IEEE, Standard, 2000.

[IEEE 802] "IEEE 802-2014 - standard for local and metropolitan area networks: Overview and architecture", IEEE, Standard, 2014. DOI: 10.1109/IEEESTD.2014.6847097.

[ISO7498-1] "Information technology - open systems interconnection - basic reference model: The basic model", ISO, Standard, 1996.

[RFC1122] R. Braden, "Requirements for internet hosts – communication layers", 1989, RFC 1122. [Online]. Available: https://tools.ietf.org/pdf/rfc1122.pdf (visited on 08/01/2019).

[RFC1350] K. Sollins, "The tftp protocol (revision 2)", 1992, RFC 1350. [Online]. Available: https://tools.ietf.org/pdf/rfc1350.pdf (visited on 08/01/2019).

[RFC2104] H. K. et al., “Hmac: Keyed-hashing for message authentication”, 1997, RFC 2104. [Online]. Available: https://tools.ietf.org/pdf/rfc2104.pdf (visited on 08/01/2019).

[RFC2402] R. A. S. Kent, “Ip authentication header”, RFC 2402. [Online]. Available: https://tools.ietf.org/pdf/rfc2402.pdf (visited on 08/01/2019).

[RFC2406] ——, “Ip encapsulating security payload (esp)”, RFC 2406. [Online]. Available: https://tools.ietf.org/pdf/rfc2406.pdf (visited on 08/01/2019).

[RFC2818] R. K. et al., “Http over tls”, 2000, RFC 2818. [Online]. Available: https://tools.ietf.org/pdf/rfc2818.pdf (visited on 08/01/2019).

[RFC3193] B. Patel, B. Aboba, W. Dixon, G. Zorn, and S. Booth, “Securing l2tp using ipsec”, RFC 3193. [Online]. Available: https://tools.ietf.org/pdf/rfc3193.pdf (visited on 08/01/2019).

[RFC3234] B. Carpenter, “Middleboxes: Taxonomy and issues”, 2002, RFC 3234. [Online]. Available: https://tools.ietf.org/pdf/rfc3234.pdf (visited on 08/01/2019).

[RFC4226] D. M. et al., “Hotp: An hmac-based one-time password algorithm”, 2005, RFC 4226. [Online]. Available: https://tools.ietf.org/pdf/rfc4226.pdf (visited on 08/01/2019).

[RFC4253] E. T. Ylonen C. Lonvick, “The secure shell (ssh) transport layer protocol”, RFC 4253. [Online]. Available: https://tools.ietf.org/pdf/rfc4253.pdf (visited on 08/01/2019).

[RFC4301] B. T. S. Kent K. Seo, “Security architecture for the internet protocol”, RFC 4301. [Online]. Available: https://tools.ietf.org/pdf/rfc4301.pdf (visited on 08/01/2019).

[RFC4302] B. T. S. Kent, “Security architecture for the internet protocol”, RFC 4302. [Online]. Available: https://tools.ietf.org/pdf/rfc4302.pdf (visited on 08/01/2019).

[RFC4880] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, “Openpgp message format”, RFC 4880. [Online]. Available: https://tools.ietf.org/pdf/rfc4880.pdf (visited on 08/01/2019).

[RFC4944] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, “Transmission of ipv6 packets over ieee 802.15.4 networks”, RFC 4944. [Online]. Available: https://tools.ietf.org/pdf/rfc4944.pdf (visited on 08/01/2019).

[RFC6238] D. M. et al., “Totp: An time-based one-time password algorithm”, 2005, RFC 6238. [Online]. Available: https://tools.ietf.org/pdf/rfc6238.pdf (visited on 08/01/2019).

[RFC7296] C. K. et al., “Internet key exchange protocol version 2 (ikev2)”, 2014, RFC 7296. [Online]. Available: https://tools.ietf.org/pdf/rfc7296.pdf (visited on 08/01/2019).

[RFC7668] J. N. et al., “Ipv6 over bluetooth(r) low energy”, 2015, RFC 7668. [Online]. Available: https://tools.ietf.org/pdf/rfc7668.pdf (visited on 08/01/2019).

Further Reading

[FR1] T. Simon, "Critical infrastructure and the internet of things", Global Commission on Internet Governance Paper Series, 2017. [Online]. Available: https://www.cigionline.org/sites/default/files/documents/GCIG%20no.46_0.pdf (visited on 08/01/2019).

[FR2] C. W. Mario Ballano Barcena, "Insecurity in the internet of things", Symantec, Tech. Rep., 2015. [Online]. Available: https://www.symantec.com/content/en/us/enterprise/fact_sheets/b-insecurity-in-the-internet-of-things-ds.pdf (visited on 08/01/2019).

[FR3] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, and K. Haley, "Isr - internet security threat report - volume 21", Symantec, Tech. Rep., 2016. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf (visited on 08/01/2019).

[FR4] R. Dietrich, *Industrial Ethernet, ... from the Office to the Machine*. Harting. [Online]. Available: http://schusterusa.com/wp-content/uploads/2012/12/harting_industrial_ethernet_handbook.pdf (visited on 08/01/2019).

[FR5] T. C. G. TCG, *Tcg specification architecture overview*, version Rev. 1.4, 2007. [Online]. Available: https://trustedcomputinggroup.org/tcg-architecture-overview-version-1-4/ (visited on 08/01/2019).

[FR6] J. E. et al., "The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market", Tech. Rep., 2015. [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/04/GlobalPlatform_TEE_Whitepaper_2015.pdf (visited on 08/01/2019).

[FR7] M. Schramm, *Embedded trusted computing on arm-based systems*, 2014. [Online]. Available: https://www.securityforum.at/wp-content/uploads/2014/05/SF14_Slides_Schramm.pdf (visited on 08/01/2019).

[FR8] S. Gueron, “Intel advanced encryption standard (aes) new instructions set”, Tech. Rep., 2012. [Online]. Available: https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf (visited on 08/01/2019).

[FR9] I. Toshiba America Electronic Components, “Nand vs. nor flash memory, Technology overview”, Tech. Rep., 2006, pp. 1–4. [Online]. Available: http://aturing.umcs.maine.edu/~meadow/courses/cos335/Toshiba%20NAND_vs_NOR_Flash_Memory_Technology_Overviewt.pdf (visited on 08/01/2019).

[FR10] *Secure boot on i.mx 50, i.mx 53, i.mx 6 and i.mx 7 series using habv4*, Application Note AN4581, NXP Semiconductors, 2018. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN4581.pdf (visited on 08/01/2019).

[FR11] *Mount(8) - linux man page*, kernel.org, 2015. [Online]. Available: http://man7.org/linux/man-pages/man8/mount.8.html (visited on 08/01/2019).

[FR12] *CVE-2017-5754*, Available from MITRE, CVE-ID CVE-2017-5754. Jan. 4, 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754 (visited on 08/01/2019).

[FR13] *CVE-2017-5715*, Available from MITRE, CVE-ID CVE-2017-5715. Jan. 4, 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715 (visited on 08/01/2019).

[FR14] *CVE-2017-5753*, Available from MITRE, CVE-ID CVE-2017-5753. Jan. 4, 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753 (visited on 08/01/2019).

[FR15] *Code-signing tool, User’s guide*, NXP Semiconductors, 2018. [Online]. Available: https://www.nxp.com/webapp/sps/download/view_license.jsp?colCode=IMX_CST_TOOL (visited on 08/01/2019).

[FR16] *I.mx 7dual applications processor reference manual*, NXP Semiconductors, 2018. [Online]. Available: https://www.nxp.com/docs/en/reference-manual/IMX7DRM.pdf (visited on 08/01/2019).

[FR17] *ERR010872 ROM: Secure boot vulnerability when using the serial downloader*, Preliminary Erratum Information, NXP Semiconductors, 2017. [Online]. Available: https://community.nxp.com/servlet/JiveServlet/download/334996-3-406867/ERR010872_Secure_Boot_Vulnerability_Erratum_Preliminary_Rev0.pdf (visited on 08/01/2019).

[FR18] *ERR010873 ROM: Secure boot vulnerability when authenticating a certificate*, Preliminary Erratum Information, NXP Semiconductors, 2017. [Online]. Available: https://community.nxp.com/servlet/JiveServlet/download/334996-3-406868/ERR010873_Secure_Boot_Vulnerability_Erratum_Preliminary_Rev0.pdf (visited on 08/01/2019).

[FR19] S. Glass, *U-boot verified boot*, Is shipped with the U-Boot release archive in doc/uImage.FIT/verified-boot.txt, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR20] S. G. Maria Balakowicz, *U-boot new uimage source file format (bindings definition)*, Is shipped with the U-Boot release archive in doc/uImage.FIT/source_file_format.txt, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR21] S. Glass, *U-boot fit signature verification*, Is shipped with the U-Boot release archive in doc/uImage.FIT/signature.txt, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR22] *Example image source file*, Is shipped with the U-Boot release archive in doc/uImage.FIT/sign-configs.its, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR23] *How to use images in the new image format*, Is shipped with the U-Boot release in doc/uImage.FIT/howto.txt, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR24] ——, *Example image source file*, Is shipped with the U-Boot release in doc/uImage.FIT/beaglebone_vboot.txt, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR25] N. Iwamatsu, W. Denk, and S. Glass, *MKIMAGE(1) general commands manual*, kernel.org, 2010. [Online]. Available: https://manpages.ubuntu.com/manpages/cosmic/man1/mkimage.1.html (visited on 08/01/2019).

[FR26] M. P. et.al., *Veritysetup(8) maintenance commands*, kernel.org, 2019. [Online]. Available: http://man7.org/linux/man-pages/man8/veritysetup.8.html (visited on 08/01/2019).

[FR27] *Dmverity*. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity (visited on 08/01/2019).

[FR28] *Encrypted boot on habv4 and caam enabled devices*, Application Note AN12056, NXP Semiconductors, 2018. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12056.pdf (visited on 08/01/2019).

[FR29] M. Vasut, *U-boot verified boot*, Is shipped with the U-Boot release archive in cmd/aes.c, U-Boot. [Online]. Available: ftp://ftp.denx.de/pub/u-boot/u-boot-2019.04.tar.bz2 (visited on 08/01/2019).

[FR30] *Dmcrypt*. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt (visited on 08/01/2019).

[FR31] M. K. et.al., *Universal tun/tap device driver*. kernel.org, 2002. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt (visited on 08/01/2019).

[FR32] *Netdevice(7) linux programmer's manual*, kernel.org, 2017. [Online]. Available: http://man7.org/linux/man-pages/man7/netdevice.7.html (visited on 08/01/2019).

[FR33] *Termios(3) linux programmer's manual*, kernel.org, 2019. [Online]. Available: http://man7.org/linux/man-pages/man3/termios.3.html (visited on 08/01/2019).

[FR34] e. Stefano Babic, *Swupdate: Syntax and tags with the default parser*, DENX Software Engineering. [Online]. Available: https://sbabic.github.io/swupdate/sw-description.html (visited on 08/01/2019).

[FR35] ——, *Symmetrically encrypted update images*, DENX Software Engineering. [Online]. Available: https://sbabic.github.io/swupdate/encrypted_images.html (visited on 08/01/2019).