Roman Gerhard Silberschneider

Replay Resistant Secure Boot for Industrial Internet of Things Devices

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor
Thomas Unterluggauer

Evaluator
Prof. Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, July 2018

This document was written with Vim, is set in Palatino,
compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

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

Date	Signature

Abstract

The importance of the Industrial Internet of Things (IIoT) is increasing rapidly. IIoT devices live in production environments as well as in the public infrastructure and hence typically contain security-relevant software that is responsible for controlling critical tasks. Their deep integration into the environment, however, often allows attackers to gain physical access to IIoT devices. This physical access represents a huge attack surface for these devices. In particular, memory safety attacks, replay attacks, and side-channel attacks are serious threats to the software in the IIoT. For example, an attacker can exploit a memory safety vulnerability in the software to take control of a device. To repair these software vulnerabilities, vendors hence continuously create and deploy security updates. However, an attacker with physical access can revert these updates by using replay attacks, where they force a device to use an old and flawed version of the software. Even further, an attacker can use side-channel attacks to learn data processed on the device. These side-channel attacks, for example, allow to learn cryptographic keys used by mechanisms that aim to protect the confidentiality and authenticity of software executed on these devices, such as secure boot. Eventually, all these attacks threaten the software executed on an IIoT device and therefore the processed data as well. As a result, the whole IIoT system is endangered.

In this thesis, we improve the security of IIoT devices by presenting a bootloader concept that prevents replay attacks. The bootloader uses secure storage, located in a Trusted Platform Module (TPM), to verify that the latest firmware version is being loaded. This verification is based on a cryptographic hash calculated over the encrypted firmware image that represents a unique identifier of the version. This value is stored inside the TPM and compared to the hash of the loaded image on every system boot. After

successful verification, the firmware image is decrypted. To protect the decryption against side-channel attacks, we use a leakage-resilient encryption scheme. We use the modern programming language Rust to prevent against memory safety issues. With a proof-of-concept implementation running on a physical device, we verify the practical feasibility of the bootloader concept.

Keywords: Secure Boot, Replay Attacks, Side-Channel Attacks, Memory Safety, Rust

Kurzfassung

Das Industrielle Internet der Dinge (IIoT) verbreitet sich zunehmend und wird direkt in unsere Umwelt integriert. Aus diesem Grund bieten IIoT Geräte eine große Angriffsfläche. Im Speziellen können Angreifer Schwachstellen in der Software ausnutzen. Solche Schwachstellen werden mit Sicherheitsupdates zwar behoben, jedoch kann ein Angreifer mit physikalischem Zugriff ein eingespieltes Update mit einer sogenannten Replay Attacke rückgängig machen. Weiters kann ein Angreifer Seitenkanalinformationen nutzen um im Gerät verarbeitete Daten zu extrahieren. Besonders interessant sind dabei kryptographische Schlüssel, wie sie zum Beispiel für verschlüsselte Firmware verwendet werden. Mit den genannten Angriffen kann ein Angreifer die volle Kontrolle über ein Gerät übernehmen und so auch das gesamte IIoT System gefährden.

Wir verbessern in dieser Arbeit die Sicherheit von IIoT Geräten indem wir ein Bootloader Konzept zur Verhinderung von Replay Attacken vorstellen. Dafür wird über die verschlüsselte Firmware ein kryptographischer Hash berechnet und in einem sicheren Speicher abgelegt. Als sicherer Speicher dient uns ein Trusted Platform Module (TPM). Bei jedem Systemstart wird der Hash der geladenen Firmware berechnet und mit dem gespeicherten Wert verglichen. Nur bei Gleichheit wird die Firmware entschlüsselt. Um den Entschlüsselungsvorgang gegen Seitenkanalangriffe zu schützen, verwenden wir sogenannte Leakage-Resilient Cryptography. All diese Funktionen sind in Software implementiert und müssen daher gegen Softwareschwachstellen, speziell Speicherzugriffsverletzungen, geschützt werden. Dafür verwenden wir die moderne Programmiersprache Rust, welche Speichersicherheit garantiert. Wir zeigen die Machbarkeit des Konzepts mit einer praktischen Implementierung, welche auf einem pysikalischem Gerät läuft.

Schlüsselwörter: *Secure Boot*, *Replay* Angriffe, Seitenkanalangriffe, Speicherzugriffssicherheit, Rust

Contents

List of Figures

List of Figures

Glossary

AES	Advanced Encryption Standard.
API	Application Programming Interface.
ASLR	Address Space Layout Randomization.
CAN	Controller Area Network.
CBC	Cipher Block Chaining.
CFI	Control Flow Integrity.
CI	Continuous Integration.
CMOS	Complementary Metal-Oxide-Semiconductor.
COOP	Counterfeit Object-Oriented Programming.
CPU	Central Processing Unit.
CS	Chip Select.
DoS	Denial of Service.
DPA	Differential Power Analysis.
DRM	Digital Rights Management.
ECB	Electronic Code Book.
ELF	Executable and Linking Format.
EM	Electromagnetic.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array.
FSBL	First Stage Bootloader.
GC	Garbage Collector.

GND	Ground Voltage Level.
GPIO	General Purpose Input Output.
HMAC	Hash-based Message Authentication Code.
I/O	Input/Output.
IDE	Integrated Development Environment.
IIoT	Industrial Internet of Things.
IoT	Internet of Things.
IP	Intellectual Property.
IV	Initialization Vector.
JOP	Jump-Oriented Programming.
libc	C Standard Library.
LIFO	Last In First Out.
LOP	Loop-Oriented Programming.
MAC	Message Authentication Code.
ME	Management Engine.
MISO	Master In Slave Out.
MMIO	Memory Mapped Input/Output.
MOSI	Master Out Slave In.
NIST	National Institute of Standards and Technology.
nonce	number only used once.
OS	Operating System.
PC	Personal Computer.
PCR	Platform Configuration Registers.
PIC	Position-Independent Code.
PLC	Programmable Logic Controller.

PLL	Phase-Locked Loop.
PMOD	Peripheral Module.
PRG	Pseudo-Random Generator.
PRNG	Pseudo-Random Number Generator.
PTP	PC Client Platform TPM Profile.
RAM	Random-Access Memory.
RILC	Return-Into-libc.
RNG	Random Number Generator.
ROM	Read-Only Memory.
ROP	Return-Oriented Programming.
RSA	Rivest–Shamir–Adleman.
SCK	Serial Clock.
SoC	System on Chip.
SPA	Simple Power Analysis.
SPI	Serial Peripheral Interface.
SROP	Sigreturn-Oriented Programming.
SSH	Secure Shell.
TCG	Trusted Computing Group.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
TPM	Trusted Platform Module.
TPM2	Trusted Platform Module version 2.
TRNG	True Random Number Generator.
TV	Television.
UART	Universal Asynchronous Receiver Transmitter.
USB	Universal Serial Bus.
VDD	Positive Supply Voltage.

W⊕X Write Xor Execute.

XOR Exclusive OR.

1 Introduction

The Internet of Things (IoT) is getting more and more relevant today. The IoT contains *things* that are connected to the Internet. These *things* are interconnected computers, called IoT devices, that are spread all over the world to fulfill tasks of our everyday life.

A modern home is full of IoT devices. Smart Televisions (TVs), smartwatches, smartphones, smart household appliances and smart homes are interconnected and fulfill different tasks of our private lives. IoT devices connect to a cloud via the Internet. The cloud acts as central storage and control point for the IoT system. For example, a smartphone can be used to control the room temperature of a smart home via the Internet, even if the operator is not at home or even on the other side of the planet.

The IoT gets popular not only in the private world but also for the industry. The cloud can be used to monitor, evaluate or control industrial systems. Different industrial computers, so-called Industrial Internet of Things (IIoT) devices, connect to the cloud. These devices are for example found in a power plant, in traffic management or controllers of a production machine. Programmable Logic Controllers (PLCs) are controlling production machines and are responsible for the production progresses. With a connection to the internet, the PLCs turn into IIoT devices and benefit from this connection. The usage of the IIoT offers the possibility to provide customized production. E.g., a factory for sport shoes can fit the product to the customer's needs. Based on a 3D scan of the customer's feet, it is possible to produce a pair of shoes with the perfect fit. The data is transferred over the Internet to the factory where a fully automatic production process fabricates the individual pair of shoes. This concept of individualization is adaptable to many other products. Further, it is possible to monitor the production machine and detect mechanical problems before the machine

fails. Therefore, the analysis is performed in the cloud, and the machine owner gets notified over the Internet. Although there are many benefits of using IIoT systems, there is the risk of broken cybersecurity. Different adversaries are, e.g., cybercriminals, competitors or secret services endanger the IIoT devices. Attackers can use the Internet access to espionage or sabotage on IIoT devices. Further, an attacker can use physical access if the IIoT devices are located in unsafe environments, like smart meters of a smart grid located in a private household or a PLC of a production machine located in a foreign country. IIoT devices contain secret information like domain knowledge of a process that is Intellectual Property (IP) of a company. Further, cryptographic secrets used to authorize against the cloud are stored on the device. Protecting these assets is necessary. Whenever an attacker gains access to the assets, the operator of the IIoT system gets into trouble. For example, the misusage of cloud credentials can threaten the whole IIoT system. Further, the attacker can duplicate IIoT devices or send fake data into the cloud. This behavior risks the production and makes the IIoT system unusable. Therefore, an attack can threaten a whole company and hence the economy. When using IIoT for power grids, attacks endanger the energy supply of whole nations. Today's population depends on the electrical energy. A permanent loss would be a significant threat to our society. For this reason, IIoT systems require strongest protections against cyber attacks.

1.1 Attacks on Industrial Internet of Things (IIoT) Devices

However, IIoT devices are distributed all over the world and therefore cannot always be protected against physical access of an attacker. E.g., production machines can be located in a foreign country, where malicious persons have physical access to the machine. This physical access is a problem for the IIoT device, a computer deeply embedded in the environment.

Like almost every computer, it consists of Central Processing Units (CPUs), Random-Access Memory (RAM) and some storage, like a hard disk or an SD-card. Further, the IIoT device has a network connection to establish a

link to the cloud. On the other hand, the IIoT device is connected to the environment. The IIoT device interacts with the environment using sensors and actors. With a sensor, the device can monitor environmental values from a process. With an actor, the device can influence a process.

A physical attacker can target every part of the device. By opening the housing of the IIoT device, the attacker can tamper with the inner structure, including measuring signals and modifying the device. For example, when attacking the storage medium, the attacker can read or write data and software that is executed by the IIoT device. This attack gives the attacker full access to the assets and therefore needs to be protected. A state-of-the-art mechanism to protect storage is secure boot. Based on cryptography, the integrity and confidentiality of the executed software and stored data are guaranteed. Confidentiality in this context is the protection against unpermitted reads that is reached with encryption. The integrity is reached with Message Authentication Codes (MACs) to prevent the execution of unintended firmware. These two cryptographic mechanisms require secure places to store keys. Therefore, the hardware of the IIoT device needs to support secure boot and therefore verify and decrypt the firmware before execution. In detail, the device starts a bootloader and this bootloader starts the firmware. The bootloader and the firmware are implemented as a software and therefore are at risk of software bugs. These bugs are programming errors that can lead to security vulnerabilities. Vulnerabilities are quite common in modern software as shown in the list of common vulnerabilities and exposures [Cor18]. A particular class of vulnerabilities are memory-safety problems. These problems are caused by unsafe memory handling. For example, a stack-based buffer overflow belongs to this class of vulnerabilities. A vulnerability can be exploited by an attacker so that it can lead to full control over the device. This means that the attacker can execute arbitrary software on the IIoT device and therefore obtains access to the assets. With security updates, vulnerabilities are fixed and the security problem is repaired. However, with a so-called replay attack, an old firmware image can be restored even if secure boot is in place. Due to the fact, that secure boot is based on cryptographic keys that are only once writeable to secure storage in hardware, an old firmware image stays valid. This attack can bring back old vulnerabilities, and allows to exploit the IIoT device.

Another problem resulting from physical access for an attacker are side-channel attacks. The IIoT device leaks information through side channels while processing data. Namely, processing data generates side effects, like timing differences, EM emanation, heat, and power consumption. These side channels allow concluding on the processed data. For example, the decryption of a firmware image while secure boot is a target for a side-channel attack. The attacker wants to get knowledge of the firmware plaintext to extract secrets and IP. In order to do so, the attacker can use a Differential Power Analysis (DPA), a common side-channel attack based on the power consumption of the device, while decrypting. With this attack, it is possible to extract the cryptographic key and further decrypt the firmware image. Again, this gives the attacker permissions to reproduce IIoT devices as well as impersonating the IIoT device when connecting to the cloud.

We conclude that the mitigation of memory-safety, replay and side-channel attacks are a necessary property for building secure IIoT devices. Robust mechanisms are required to ensure these properties. Currently, there is no solution available that prevents all attacks covered in this thesis. IIoT devices need to be secure and trustworthy, otherwise, IIoT devices run into danger to threaten our world.

1.2 Contributions

This thesis covers three main contributions:

1. Replay resistance for operating system updates.
2. Memory safety for the implemented bootloader.
3. Side-channel mitigations for operating system decryption.

We verify the feasibility of these security mechanisms with a proof-of-concept implementation of a secure bootloader. To protect against replay attacks, a cryptographic hash over the encrypted next stage is stored inside a Trusted Platform Module (TPM). The TPM guarantees that this data can not be modified. On every system start, the bootloader then computes a hash over the encrypted next-stage and compares it against the securely stored value. This approach prevents executing old next stage copies and thus

a replay attack. We implemented a bootloader using the modern system programming language Rust. Rust guarantees memory safety without a Garbage Collector (GC) by enforcing a robust ownership model for program variables. To cope with side-channel attacks on the bootloader, we use a leakage-resilient cryptographic mode of operation, such as proposed by Pereira et al. [PSV15], for the encryption of the next stage. However, as discussed by Dobraunig et al. [Dob+17], leakage-resilient encryption schemes are yet vulnerable to side-channel attacks where attackers maliciously alternate the ciphertext. To also protect the decryption at startup, we hence check the hash of the ciphertext before actually performing the decryption. In this respect, this work shows that secure modifiable storage can help to mitigate side-channel attacks during the decryption of boot images.

Our results further indicate that it is possible to secure a device against replay attacks and mitigate side-channel attacks without being in danger of memory safety issues. From our results, we suggest using Rust for new IoT implementations and encounter replay and side-channels as potential threats.

Outline

In Chapter 2, the fundamental property of memory safety is covered, and different attacks and mitigations are explained. Chapter 3 focuses on physical attacks, particularly replay, and side-channel attacks. Moreover, cryptographic primitives are described in the context of side-channel attacks. Basic concepts of secure systems, including secure boot, random number generation, and the TPM, can be found in Chapter 4. Chapter 5 gives details on the actual implementation. Chapter 6 concludes this thesis.

2 Memory Safety

Devices in the Internet of Things (IoT) are computers integrated into an embedded world. These computers run software written by programmers. C and C++ are currently the standard programming languages for low-level programs like operating system kernels, device drivers, or bare-metal applications. Therefore, a large share of software running on IoT devices is written in C/C++. C/C++ is well established and offers easy hardware access. Memory mapped Input/Output (I/O) is used to control interfaces, timers or interrupts. Hardware interrupts can be implemented to get notifications on events so that real-time applications can be programmed. However, C/C++ is error-prone in the sense of dealing with memory. The language does not prevent the introduction of different flaws and therefore expects the programmer to deal with all these problems. Using C/C++, it is easy to implement a program that accidentally allows illegal access to memory. If such an implementation is introduced unintentionally, a so-called software bug occurs. Such a software bug could be exploited by an attacker to threaten the security of the IoT device. An exploitable bug in this context is called a vulnerability. As shown in the list of common vulnerabilities and exposures [Cor18], security vulnerabilities commonly occur in software products.

A particular class of vulnerabilities are memory safety issues, commonly found in applications written in C/C++. Memory safety is a central property in information processing promising that no memory area can be illegally read or written. However, memory safety vulnerabilities often occur in software today. By exploiting memory safety vulnerabilities, an attacker can reach different goals, up to full control over a device. Recent programming languages can assure the absence of memory safety issues, but are not widely used or do not fulfill the requirements of embedded programming. These languages often introduce a massive overhead in execution time

and hardware requirements so that C/C++ was not replaced by now. A promising, new system programming language, which tackles memory safety issues with a novel concept of an ownership model, is Rust, covered in Section 2.3.

Memory safety issues often lead to code execution attacks. This class of attacks allows an attacker to execute malicious software on the device under attack. The attacker hereby crafts a user input which causes a special behavior of a software program. In this way, the attacker exploits a vulnerability and therefore gains more permissions as intended for a user of the program. Memory safety attacks can give an attacker full control over the device, including unauthorized access to data, taking control over the behavior of the system and Denial of Service (DoS) attacks. A so-called DoS attack threatens the availability of a device. For example, this can be achieved by crashing the application. A DoS attack already is a serious problem, but other attacks are more powerful and even more harmful.

Section 2.1 gives a brief overview of common vulnerabilities found in today's software. Section 2.2 covers state-of-the-art attacks and mitigations of memory safety vulnerabilities.

2.1 Vulnerabilities

As already motivated, exploitable software bugs lead to vulnerabilities. There exist many different classes of vulnerabilities that can lead to memory safety attacks. Serious vulnerabilities found in popular software are collected and rated in the list of common vulnerabilities and exposures [Cor18]. This list is extensive and visualizes that vulnerabilities are a severe problem in present information processing systems. In the following, we give an overview of the most relevant types of vulnerabilities.

Buffer Overflow. Stack-based buffer overflows were first shown by Levy [Lev98] in the 90s and enable an attacker to gain unauthorized access to a system by rewriting the stack. In general, a stack is a data structure following the Last In First Out (LIFO) principle. With a `push` operation, data can be

Listing 2.1: Buffer overflow example written in C.

```
1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    char buffer[10];
8    bool is_admin = false;
9    printf("Please enter your name: ");
10   scanf("%s", buffer);
11   printf("Hello %s\n", buffer);
12
13   if(is_admin == true)
14   {
15     printf("Welcome Administrator, here is your shell:\n");
16     system("/bin/sh");
17   }
18   return 0;
19 }
```

placed on the stack and with a `pop` instruction the last *pushed* value can be retrieved. The stack, as used by C/C++ compilers, is a memory region inside the Random-Access Memory (RAM) and is used to store arguments, local variables and return addresses.

User input is typically stored in buffers, meaning every character is stored next to another in a memory range defined while programming. Listing 2.1 shows a small code snippet receiving user input and storing it to the array `buffer`. The code provides a maintenance access if the `is_admin` flag is set to true. Although the code seems correct, the implementation is vulnerable. Missing bounds checks on a memory buffer can lead to a buffer overflow in the variable `buffer`. In the example, `scanf` accepts more characters than can be stored into the buffer. As shown in Figure 2.1, `is_admin` and `buffer` are located on the stack. By writing out of bounds, the variable `is_admin` is overwritten. Doing so, the condition `is_admin == true` in line 13 in

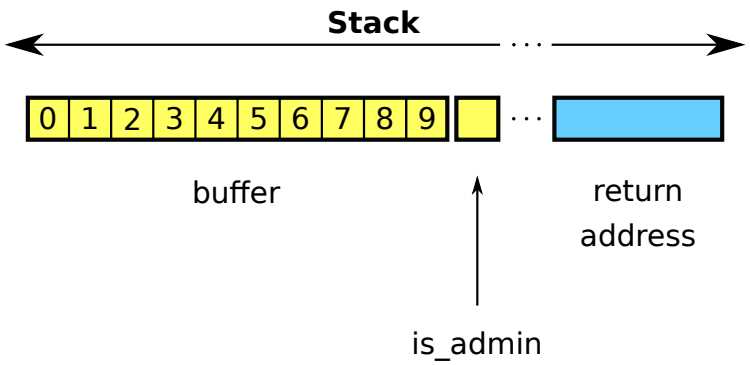

Figure 2.1: Memory layout for the example program shown in Listing 2.1.

Listing 2.2: Console output of Listing 2.1.

```
Please enter your name: ABCDEFGHIJK
Hello ABCDEFGHIJK
$ uname
Linux
```

Listing 2.1 can be triggered to evaluate to true. In this way, the maintenance entrance is unlocked as demonstrated in Listing 2.2, which shows the console output of the program execution. The bug in the example is easy to fix within the `scanf` format string: `scanf("%9s", buffer)`. With this modification, no more than nine characters are accepted. Since strings are zero-terminating in C, only nine characters can be stored in a ten-byte buffer.

Integer Overflow. Integral data types, as used in modern computers, have limited memory footprints, and therefore a restricted range of possible values. For example, an unsigned eight-bit datatype can hold values from 0 to 255. When incrementing over this limit, the variable overflows. The equation $255 + 1$ results in the value 0. A similar behavior occurs when decrementing the variable. Integer Overflows also occur on signed datatypes. This vulnerability is a class of hard-to-find software bugs. If this problem occurs in a length check of a buffer, the attacker is able to write out of bounds. In this case, the attacker can overflow a buffer, as shown before.

Use After Free. A Use After Free attack abuses the dynamic memory management. While programming C or C++, the programmer can allocate memory on the heap, which is the memory region for dynamic memory. The programmer is entirely responsible for initialization, usage, and freeing this memory. If the program frees such a memory element, no automatic mechanism will clear the content nor invalidate any pointer to this memory. By abusing a weak implementation, an attacker can leak information or execute malicious code from these unchecked memory regions.

Type Confusion. A C++ object can contain function pointers located in the virtual method table. As discussed by Jeon et al. [Jeo+17], these function pointers can be abused by type conversion. If an illegal type conversion, an illegal cast, is performed, an unintended function call can arise.

Format Strings. A format string is the format representation used in the C standard library. For example, `printf` uses format strings to determine the output format. If a user can control the format string, e.g., through the user input, the program is in general exploitable. Listing 2.3 shows the usage of the format string of the `printf` function. If `fmt` results from user input, a malicious user can gain arbitrary read and write access to the program memory, which in turn can result in code execution. A vulnerable format string used in `printf` is *Turing complete*, as shown by Carlini et al. [Car+15]. This property means that the exploitation is universal programmable. As a consequence, an attacker can program the vulnerability to perform malicious actions.
Modern compilers are aware of this vulnerability and can output warnings to inform the programmer. Unfortunately, large software projects tend to be full of compiler warnings. In this case, essential warnings can easily be missed.

Listing 2.3: Format string example written in C.

```
1 char* fmt = "%d";
2 printf(fmt, 42);
```

2.2 Attacks and Mitigations

This section covers state-of-the art attacks that allow an attacker to execute code on the device under attack. The attacker gains full control over a system if malicious code is executed. Of course, there exist defensive mechanisms, so-called mitigations, that hinder an attacker from exploiting a vulnerability. New mitigations lead to the development of new attacks and vice versa. The following list contains attacks and mitigations showing the efforts of both sides.

Attack: Shellcode. The simplest case is to write a so-called shellcode through user input to the memory. A common goal when attacking binaries is to get access to the computer's command line, which is also known under the name shell. A shellcode contains machine instructions that the attacked computer executes in order to gain access to the shell. Therefore, the shellcode attack belongs to the group of code injection attacks. Of course, there are mitigations in place that lead to an evolution of this attack.

Mitigation: Write Xor Execute. Write Xor Execute (W⊕X) is a state-of-the-art mechanism that limits the scope of code injection attacks by using memory privileges. The mechanism prohibits that a memory area is both writeable and executable, analogous to the truth table of the Exclusive OR (XOR) function. In that way, an attacker cannot write instructions to the memory and execute it afterwards. This approach mitigates code injection attacks since user-writeable memory regions are no longer executable. A simple shellcode attack is made impossible.

Attack: Return into Libc. As discussed in Section 2.1, a stack-based buffer overflow can overwrite other elements on the stack. Unfortunately, return addresses are also stored on the stack. This behavior is necessary, because whenever a called function returns, the program should continue right after the point, where the function was called. Therefore, before a function is called, the so-called return address is pushed to the stack. However, instead of returning to the calling function, the attacker can use a function

return to call arbitrary functions by overwriting the return address on the stack. Therefore, Return-Into-libc (RILC) belongs to the group of code reuse attacks. Hereby, an attacker typically uses functions of the C Standard Library (libc) to exploit a program. The libc is the standard library on GNU Linux for programs written in C, including functions for file I/O and process execution. These functions are powerful tools for an attacker to gain full control over the system. Even when W⊕X is in place, it is possible to perform a RILC attack. RILC is *Turing complete*, as shown by Tran et al. [Tra+11].

Mitigation: Stack Canary. As a mitigation against buffer overflows, as shown in Section 2.1, the compiler inserts a magic value, the so-called stack canary, between buffers and return addresses on the stack. By exploiting a buffer overflow, the attacker will overwrite the stack canary. An added piece of code checks the canary value before returning from a function because the return address is a common goal for attackers. If the canary value alters, the program will stop its execution. However, a stack canary can be tricked if the program leaks the exact value of the stack canary. In this way, the attacker can read out the stack canary and knows what to write to the critical position.

Attack: Return oriented Programming. Return-Oriented Programming (ROP) is a well-established attack. Existing tools [Gal18; Sch18; Sal17] help attackers to exploit vulnerabilities with ROP. ROP gadgets are automatically searched and the toolchains can orchestrate them together.
ROP is a generalization of RILC, which was described before, and is classified as a code reuse attack. There is no need to call an entire library function, also small chunks of instructions work. If such a chunk ends with a return instruction, it is considered as a so-called ROP gadget. Gadgets can be linked to a so-called ROP chain. When executing a ROP chain, one gadget returns and therefore calls the next gadget. As a result, the attacker can write a program in the shape of a ROP chain, by writing a sequence of addresses of ROP gadgets as return addresses on the stack. Figure 2.2 shows an example ROP chain. The gadgets consist of a few instructions and end with the `ret` instruction. Libraries, like the libc, and the binary itself are

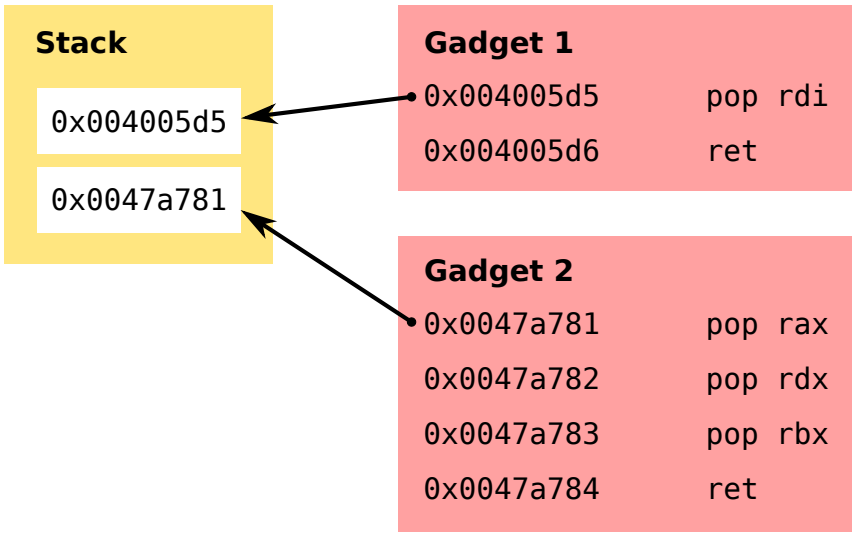

Figure 2.2: Example ROP chain built out of two ROP gadgets. The address shows the location of the gadgets. The stack contains the entry points of each ROP gadget.

mapped to the memory. The gadgets found in the mapped memory can be used for ROP. The program returns to the start address of the gadget and executes the instructions. Another interesting fact is, that for architectures with variable length instructions, like x86, one binary blob can result in different ROP chains. In this way, the number of ROP gadgets is increased. The ROP chain is packed into user input, as so-called payload, and sent to the process under attack. The ultimate goal is to get control over the process. This approach is known as process owning. By process owning, an attacker can, for example, get access to a command line interface on an online server. Using this command line interface the attacker can connect to other services, maybe a local network of the server, or can use other bugs to get more privileges on the server. If a so-called privilege escalation is successful, the attacker can gain full control over the server and all processes running there. Full control means that the attacker can kill every process, delete all data, destroy data, write data, run programs, read all data and especially copy all data to the local computer.

Mitigation: Address Space Layout Randomization. Randomization approaches were invented to prevent code reuse attacks. Address Space Layout Randomization (ASLR) is a state-of-the-art mitigation technique to prevent memory safety attacks. Hereby, the location of essential memory regions is randomized. On every execution of a binary, the position of memory sections, like the stack, is randomized. As the location of the stack changes, the memory addresses are randomized. An attacker cannot rely on the position of a library and is handicapped in performing a code reuse attack.

Mitigation: Position Independent Code. Position-Independent Code (PIC) is a randomization approach to hide memory regions. The compiler removes absolute addresses so that the code runs on arbitrary position. Compared to ASLR, also the position of the main binary is moved. This approach mitigates code injection and code reuse attacks. However, for both ASLR and PIC, an attacker can recompute target locations if the program leaks a memory offset of a location of interest.

Mitigation: Control Flow Integrity. A software program consists of instructions. While execution, different paths of instructions can be taken. These paths form the control-flow graph, which is a graph with finite size. During normal operation, the execution follows the edges of the control-flow graph. When the control-flow graph is left, a vulnerability is exploited, or a fault occurred. A fault can be caused by a defective memory cell or a power glitch. Control Flow Integrity (CFI), proposed by Abadi et al. [ABE05], is a mechanism to ensure that the control-flow graph is never left. Usually, CFI schemes require hardware support integrated into the Central Processing Unit (CPU). The CFI scheme mitigates code injection and code reuse attacks but does not prevent against type confusion vulnerabilities as mentioned in Section 2.1.

Advanced Attacks. Multiple attacks, like Jump-Oriented Programming (JOP) [Ble+11], Sigreturn-Oriented Programming (SROP) [Mab16] and Loop-Oriented Programming (LOP) [Lan+15], were developed to overcome different mitigations. An exciting attack further is Counterfeit Object-Oriented Programming (COOP), proposed by Schuster et al. [Sch+15], that circumvents CFI.

The sum of all attacks against memory safety issues shows that they are a serious threat to current software. Mitigation techniques were bypassed by new types of attacks and never solved the original problem of software containing vulnerabilities. As a result, it is time that a compiler can guarantee memory safety.

2.3 Modern Programming Languages

Most of the vulnerabilities described in Section 2.1 are closely related to the programming languages C and C++. C was introduced in the year 1972, when the security threat situation was different. With the invention of the Internet, much more threats arose. Before the Internet, exploiting a vulnerability on a local computer would mean to attack the own computer. Security was not a big deal for a long time. Nowadays, millions of computers are connected to the Internet and vulnerabilities offer attack capabilities to attackers from all over the world. More recent programming languages like Java, Python or Go offer stronger guarantees for memory safety, but are not that valuable for embedded programming as a system language. These programming languages are focused on higher-level applications.

2.3.1 Rust

Graydon Hoare introduced Rust, a modern programming language, in the year 2010. Today Mozilla Research is the driving force behind the project. Mozilla uses Rust in the currently experimental browser engine Servo [Moz17]. Further, Rust is found in the release of Firefox Quantum, where the CSS rendering engine was rewritten in Rust. Clark [Cla17] praises Quantum CSS for the high performance.

Rust is designed for the same purpose as C/C++, as a system language. It is suitable for applications running on a modern Operating System (OS) as well as embedded programming. Further, Rust guarantees memory safety by design. The guarantee is based on a strong ownership model for program variables, which also protects against data races in concurrent executions. Rust introduces a concept named lifetimes, which replaces a Garbage Collector (GC). Based on the concept of lifetimes, use after free vulnerabilities can be excluded. By using zero-cost abstractions, Rust implementations reach execution times that are competitive with programs written in C/C++.

The language is compiled with the Rust compiler (`rustc`), that itself is written in Rust. The compiler uses an LLVM backend, a state-of-the-art toolchain capable of cross-compilation to many different CPU architectures. The Rust package manager (`cargo`) manages dependencies to different *crates*. A *crate* is a project that could be a either an executable or a library. Dependencies are distributed as source code and automatically fetched from the repository and built locally. It is easy to include external library *crates* into a software project.

As discussed by Wilson [Wil16], integer overflows are not always checked for performance reasons. The current default strategy is to perform the tests in debug builds and skip the tests in performance-optimized release builds. Of course, the programmer can override the default behavior. Another way is to use library functions, offered for all integral types, to perform a check manually.

Unsafe Rust. Low-level programming requires direct access to hardware registers. This requirement can not fulfill the memory safety guarantees of Rust. Unsafe Rust solves this problem. The `unsafe` keyword is used to define blocks that override all security mechanisms. Such a block can be embedded in a Rust program, where the remaining source code will be handled as safe Rust. A good practice is to keep the amount and size of `unsafe` blocks as small as possible.

Embedded Rust. At the time of programming the project behind this thesis, embedded Rust was still in development. Unstable features were required to compile a bare-metal project for an embedded platform. Therefore, a nightly built Rust compiler (`rustc`) and the tool Xargo were required. Xargo is an extension for Cargo with the ability to compile the core library for all kinds of platforms. The Rust developers are currently working on stabilizing these features [Apa18a]. Moreover, the features Xargo provides will be merged into Cargo [Apa18b]. With this work, the Rust developers make Rust even more attractive for embedded programming. These efforts are planned to be finished by the end of 2018 [Tea18c].

3 Physical Attacks

Physical attacks are the class of attacks on a computing device where the attacker has physical access to the device. Because Industrial Internet of Things (IIoT) devices are widespread and operate in unsafe environments, they are in high danger of physical attacks. Therefore we need to protect IIoT devices against physical attacks where an attacker has hands-on access to the device. Because of this access, the attacker can open the housing of the device, replace components, measure signals or introduce faults.

Two families of physical attacks are in the focus of this work. The first family are physical replay attacks. Replay attacks are not only limited to physical attacks and allow an attacker to use obsolete information to bypass security mechanisms. This attack is a critical threat to communications as well as for software running on an IIoT device. The attacker can replay an old software image with a physical attack even if the image is encrypted. The second family of physical attacks are side-channel attacks. Side-channel attacks allow an attacker to break cryptography and gain knowledge of secret information from observing physical device properties. With a side-channel attack, the attacker can leak the encryption key and decrypt the software image.

Section 3.1 gives details on the IIoT device and the different attack vectors. Section 3.2 covers replay attacks and mitigations. Section 3.3 discusses side-channel attacks with the focus on cryptographic algorithms.

3.1 Motivation

IIoT devices are deeply integrated into the environment to fulfill sophisticated tasks. However, this deep integration causes the risk of physical

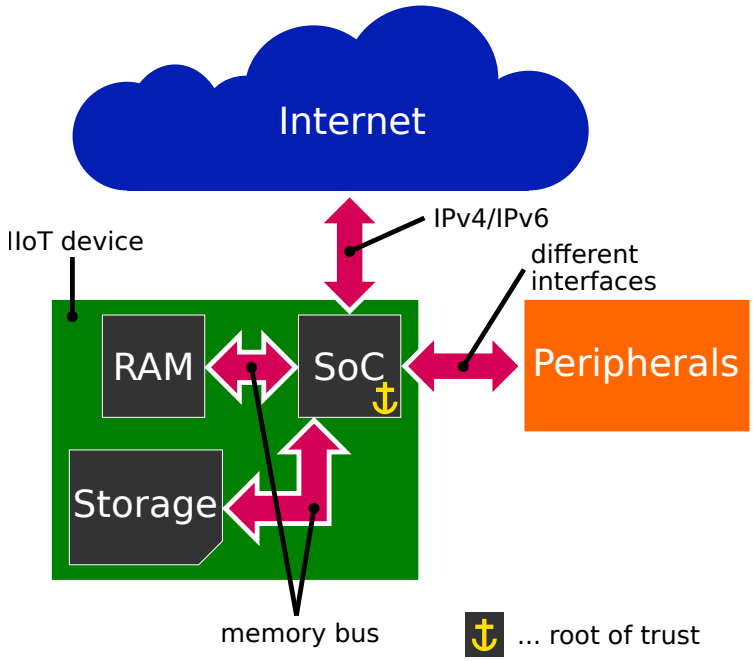

Figure 3.1: General structure of an IIoT device embedded into the environment.

attacks. For example, a device located in a data center is protected by physical barriers, and therefore the attacker needs to break the physical barrier before mounting a physical attack. For an IIoT device, usually it is impossible to protect the device with physical barriers. Attackers can gain physical access to the device and therefore can access and tamper with the physical properties.

The attackers want to steal or manipulate data on the IIoT device or want to clone the device itself. Figure 3.1 shows the basic structure of an IIoT device embedded into its environment and gives a basic overview of what parts can be attacked. On the one hand, the device is connected to the internet, shown as a cloud. Through this connection, the IIoT device communicates with servers that belong to cloud services. In this way, the IIoT device is associated to an overall IIoT system. On the other hand, the IIoT device is connected to peripherals. Through these connections, the device can gain insights or control the physical world. In other words, an IIoT device is a computer that connects the physical world with a cloud backend. It consists of memory

storage, like an SD-card, to save firmware and data. Also, Random-Access Memory (RAM) is part of the IIoT device that is used as a working memory for the computations. Both memory types are connected to a System on Chip (SoC), the core component of the IIoT device, through memory busses. The heart of a SoC are one or multiple Central Processing Units (CPUs), but also memory controllers and interface controllers are included in a SoC. For example, General Purpose Input Output (GPIO), Universal Asynchronous Receiver Transmitter (UART), Controller Area Network (CAN) or Serial Peripheral Interface (SPI) are common interfaces offered by a SoC design.

A physical attacker has the opportunity to attack every single point of the IIoT device. For example, the connection to the cloud can be intercepted. There exist proper standard protocols that ensure protection on this channel. The linkage to the peripherals is another entry point. First, the applied interfaces, e.g., CAN, often implement no security measures at all. Second, physical parameters can be manipulated directly. For example, the temperature that is measured by a temperature sensor can be modified by an attacker. Further, a physical attacker can access the inner structure of an IIoT device, even on the inner structure of the SoC.

In this thesis, we do not encounter attacks on the inner structure of IIoT devices. These attacks are expensive and require sophisticated knowledge of chip structures. Attacks on peripherals are also not considered in this work. For example, protecting a temperature sensor against manipulation is not easy at all in the presence of physical access. Further, we do not encounter attacks on the connection to the cloud since there are well-established standard protocols like Transport Layer Security (TLS).

The considered attacks focus on all components the IIoT device consists of. An attacker can gain access to the memory by tampering with memory busses or getting direct access to a storage medium, e.g., an SD-card. Further, an attacker can measure all kinds of physical properties, like Electromagnetic (EM) emanation, heat, and power consumption. Through all these possibilities, an attacker can reach the goals defined before. Section 3.1.1 gives some examples of practical, physical attacks.

3.1.1 Recent Incidents

A possible attack vector is to manipulate values inside the RAM. By rewriting memory, an attacker can hijack the device. For example, Jacob et al. [Jac+17] proposed to rewrite bootloader arguments to force the device to boot from a malicious network location. The described attack uses hardware trojans to manipulate memory, but unprotected memory is also at risk of being modified by an attacker. To prevent such attacks, memory can be protected with transparent memory encryption and authentication as proposed by Werner et al. [Wer+17].

A simple physical attack is to clone the device by cloning the hardware design and duplicating the firmware. Although there exist countermeasures, still a large number of devices are vulnerable to this type of attack. The attacker can read all secrets from the storage and use them for malicious actions. Similar to this, parts of the firmware can be modified on an unprotected device, so that the device executes software written by an attacker. For example, sensor values that are reported to the cloud can be manipulated.

Muellner and Kammerstetter [MK17] showed on the 34^{th} Chaos Communication Congress that Hoermann BiSecur devices include a cryptographic vulnerability, allowing to deduce the cryptographic key and to clone a remote control for garage door systems. They were able to read out and reverse engineer the firmware. This analysis is a good example of a physical attack. Using the recovered firmware, the cryptographic flaw was detected.

3.2 Replay Attacks

When performing a replay attack, an attacker records a message and replays it later. This type of attack is simple, but powerful, and often used to overcome specific security features. Originally, replay attacks were performed on network communication, but these could also be applied to our everyday life. For example, if a concert ticket is used and not devalued correctly, an attacker can reuse a copy to get access to the event.

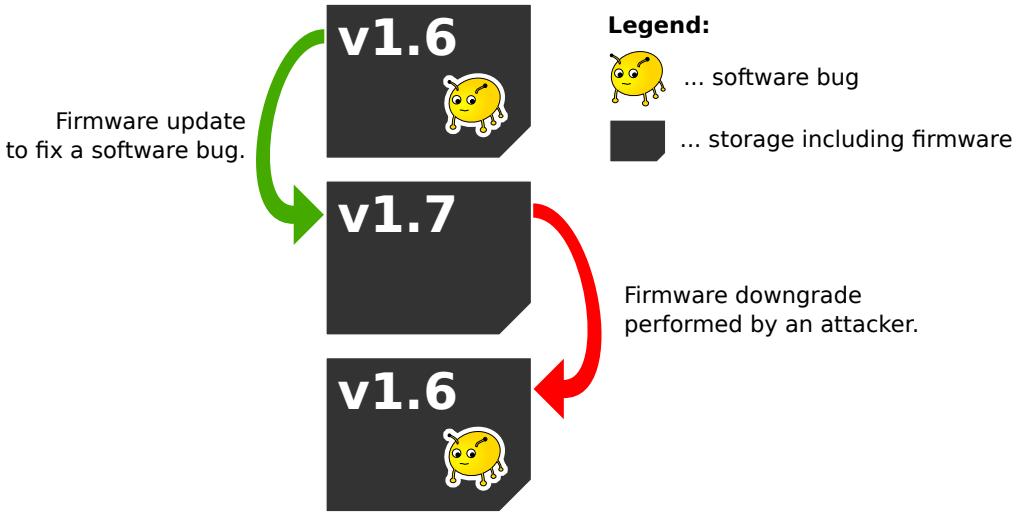

Figure 3.2: Replay Attack on firmware images.

As indicated in Figure 3.2, the same concept can be applied to firmware images. An attacker with physical access can always take copies from the storage and replay it at a time of choice. An old firmware version can be replayed by an attacker with physical access to the device. A replay attack on firmware images is dangerous, because firmware updates often include security updates, where vulnerabilities are removed. Performing the attack brings back the vulnerabilities that can be exploited afterwards. Even if the firmware image is encrypted and the encryption key is not known by the attacker, a replay attack can be accomplished. Further, secure boot, a state-of-the-art mechanism to protect storage discussed in more detail in Section 4.1, does not protect against replay attacks.

A prominent example by Skorobogatov [Sko16] showed how to mirror the storage of an iPhone 5c. The phone could be configured with a security feature which deletes all data if a wrong passcode was entered too often. By replaying the old image one could circumvent this security feature. Namely, it allows to reset the password counter and therefore apply a brute-force attack to crack the passcode. Further, all possible combinations of the passcode can be tested because there is no limit for the number of tries anymore.

Similar to the password counter in the previous example, devices may contain sensitive data like counters for pricing, bank account balance or a usage counter of a machine. In any of these cases, a replay attack has to be considered.

3.2.1 Mitigations

One way to prevent replay attacks is to use a tamper-resistant single-chip computer. In this case, the firmware storage is inside the chip and can not be modified by an attacker. Therefore a built-in firmware update solution is required. When performing an update, the firmware update tool checks if the version is increasing to prevent replay attacks. For example, secure microcontrollers or smart cards fulfill the requirement that memory cannot be manipulated. Physical barriers prevent an attacker from accessing the built-in memory.

Another option is a secure, writeable memory inside the SoC, to store the version number. This number is then matched while boot. A popular implementation of such a secure storage are fuses, a memory that is only writeable once. With this instrument, every version needs to consume one bit. In turn, the number of possible firmware update depends on the number of fuse bits integrated into the SoC. Indeed, not all SoC chips available include custom programmable fuses.

The best solution is an on-chip secure writeable memory to keep track of the software version, but this is rare in current SoC implementations. As a workaround, an external secure element, like a Trusted Platform Module (TPM), can offer tamper-resistant secure storage. In this case, it is essential to pay attention to secure communication between the secure element and the SoC. If the attacker can impersonate the secure element, replay attacks are feasible again.

3.3 Side Channels

A computer leaks information through side channels when processing data. Timing, EM emanation, heat or power are well known physical side channels. For example, if the execution time depends on the processed data, a timing side channel exists. In this case, the time can be measured and under certain circumstances the processed data can be reconstructed. Timing leakage can be mitigated with constant time implementations. If the execution time is independent of the processed data, no timing information can be exploited. In general, the information leakage through side channels depends on implementation characteristics. Two implementations of the same algorithm can have different exploitability.

Side-channel attacks are potentially dangerous for cryptographic algorithms, where secret keys are involved. The goal of an attacker is to learn these keys and to use them later. In this way, an attacker can decrypt ciphertexts, or even impersonate as an authenticated user.

In this thesis, we focus on the power side channel. Based on the power consumption of the device different attacks are applied. The basis of all analysis is the measurement of power (P_{DUA}) on the device under attack. More precisely, a time series of values proportional to the power (P_{DUA}) is required. Since computers are supplied with a constant voltage, the current is such a proportional value to the power as shown in Equation 3.1.

$$P_{DUA} = V_{DD} \cdot I_{DUA} \tag{3.1}$$

$$V_{Shunt} = R_{Shunt} \cdot I_{DUA} \tag{3.2}$$

An oscilloscope is an instrument that allows the measurement of time series. Figure 3.3 shows a basic measurement setup by using an oscilloscope to measure the voltage drop on the shunt resistor. Based on Ohm's law, as shown in Equation 3.2, the voltage drop (V_{Shunt}) on the shunt resistor is proportional to the current, and therefore proportional to the power consumption of the device under attack. By using an oscilloscope with a current probe or a shunt resistor, it is possible to record time series of

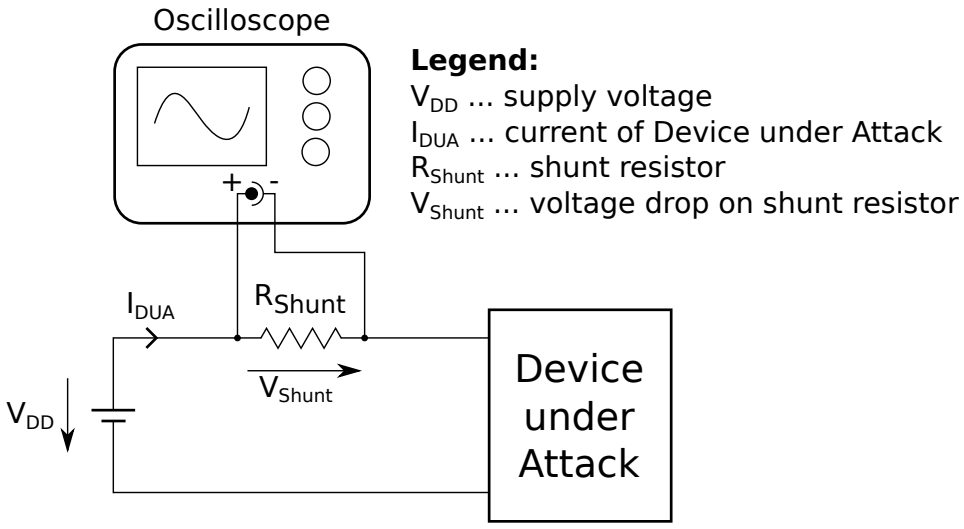

Figure 3.3: Measurement setup for power analysis attacks.

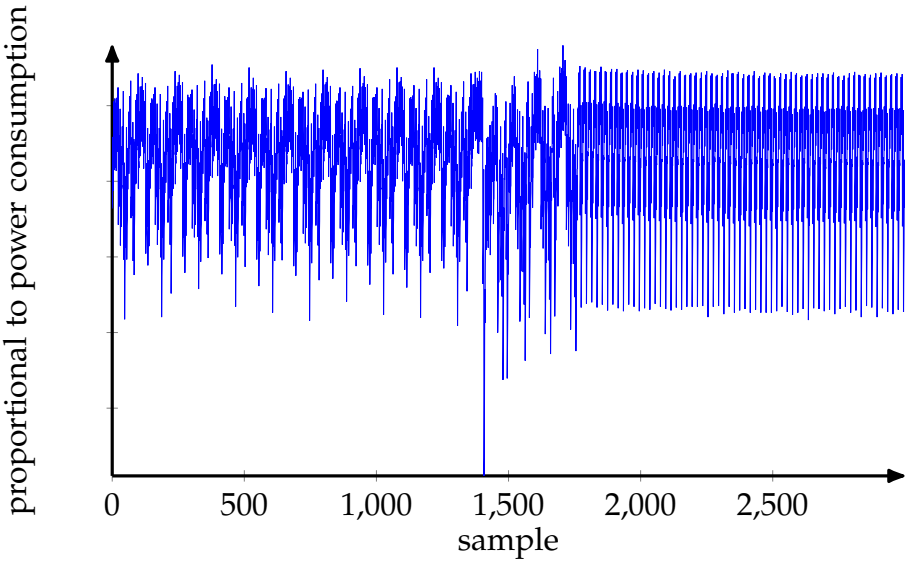

Figure 3.4: Example power trace of an AES encryption on an embedded device.

power values, so-called traces. Generating multiple traces means to repeat the recording with different inputs for the device under attack. Figure 3.4 shows an example power trace. Side-channel attacks perform analysis on these traces to extract the information that is leaked through the power side channel.

Simple Attacks. On the one hand, there are single-trace attacks, where the measurement of a single event is enough to extract the data of interest. For example, Courrège et al. [CFR10] show a Simple Power Analysis (SPA) on an implementation of the Rivest–Shamir–Adleman (RSA) algorithm [RSA78]. Based on the shape of a single power consumption trace [CFR10, Figure 3], the exponent used in the computation can be determined. Squaring uses different parts of a chip than multiplying, and hence generates a distinguishable power consumption. Some implementations produce obvious traces so that the value can be determined by visual inspection only.

Differential Attacks. On the other hand, there are differential attacks. Multiple, up to thousands, of traces are used to expose minimal differences that arise in a computation. One example of differential attacks is Differential Power Analysis (DPA), a state-of-the-art attack proposed by Kocher et al. [KJJ99], based on power side channels. Recording the power consumption of a system while encrypting or decrypting different inputs with the same key generates traces. With these traces, an attacker can deduce the key used. The reason for this is while processing data, the power consumption of the system depends on the processed data. For example, a Complementary Metal-Oxide-Semiconductor (CMOS) transistor consumes more energy, if the state changes. The reason is that the electrical charge of the gate capacity needs to be changed. This event occurs just for a very short time but depends on the data processed. Since a DPA requires multiple traces capturing this event, they need to be aligned. This means that the recording has to start every time at the same point and the executed instructions always happen at the same position in the recorded trace. This can be done by using the oscilloscope trigger input. Of course, an event is required that is used for this trigger. It is possible to improve the alignment in the analysis phase, for

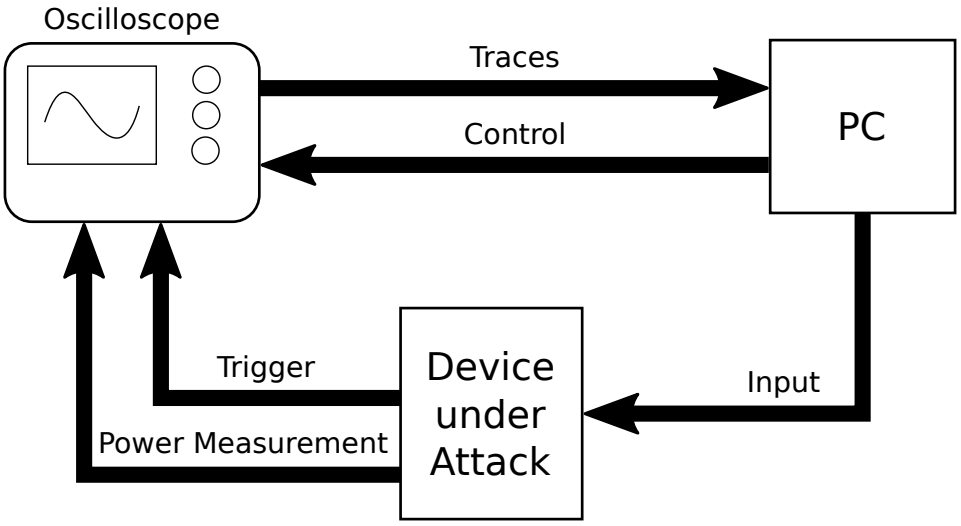

Figure 3.5: Measurement setup for a DPA.

example, by maximizing the correlations between traces by shifting them on the time axis.

3.3.1 Differential Power Analysis (DPA)

DPA is a well established physical attack on cryptographic devices. One advantage is that it is possible to perform this attack with relatively cheap equipment. Namely, an oscilloscope or similar device like the ChipWhisperer [Inc18] is required to record timing power traces. These devices are affordable for a few hundred dollars. Figure 3.5 shows the basic setup of a DPA. It consists of a device under attack, an oscilloscope to measure traces and a Personal Computer (PC) for processing. The PC is the attacker's tool to feed the device under attack with inputs and receive the associated traces from the oscilloscope.

Intermediate Result. The first task to perform in a DPA is to select a function inside the algorithm that uses the secret and a selectable value as an input. The function's output is an intermediate result that is the target of the attack. The power consumed while storing the intermediate result in some memory depends on the value of the intermediate result and therefore, information is leaked.

Measurement. The second step is to record traces. Therefore, the measurement setup is installed, and the cryptographic operation is performed multiple times with different inputs. The attacker needs to inject these inputs, for example, with a PC. Within this process, the power consumption of the device is recorded. Essential for this step is that the operation under attack is inside the time frame of the recorded trace. As mentioned before, proper alignment of the traces is required for the following analysis.

Hypothesis. As a third step, possible intermediate values need to be computed, the so-called hypothesis. Therefore, the selected function under attack is calculated locally for one selected input value and all possible values for the secret. This is continued for all selected input values and results in a matrix with the shape of all possible secret values times all selected inputs.

Power Model. As the fourth step, the power model is applied to the hypothesis. The power model is a function that maps the value of a specific intermediate result to the expected power consumption. A common model is the Hamming weight, which counts the number of bits that are true. This follows the assumption that enabling more bit requires more power.

Compare. The last step is to compare the result of the power model to the hypothesis with the measured data. A popular method is to use a correlation function to compare a vector containing the power model of all selected input values for one particular possible secret value. This vector is correlated with every timestep of all recorded traces. For every timestep, a correlation factor is computed. At the timestep, where the operation of interest is

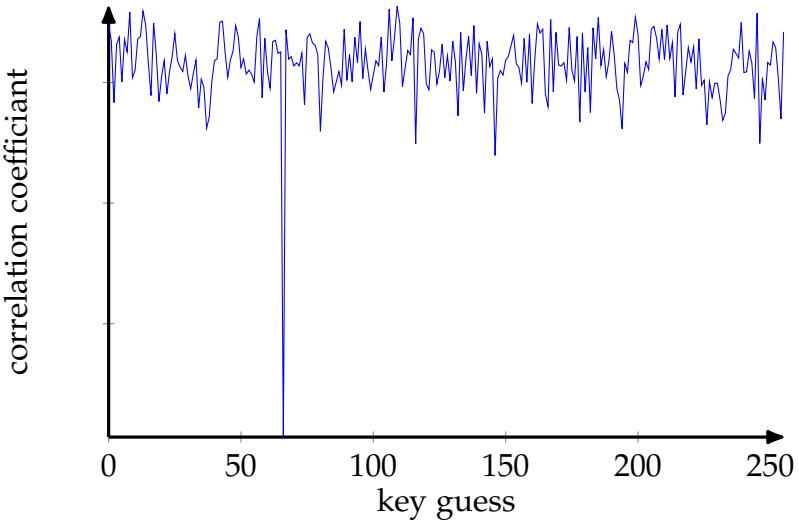

Figure 3.6: Example correlation of a DPA attack on an eight-bit AES implementation. The correct key guess is 66.

performed, the correlation of the correct key will be highest. Depending on the type of correlation, this can also mean the lowest correlation coefficient, as in the following example. Figure 3.6 shows the minimal correlation coefficient of all traces for different key guesses. The correct key guess is clearly visible. Therefore, a secret is extracted out of the recorded power traces.

3.3.2 Advanced Encryption Standard (AES)

Cryptographic algorithms are a common goal for side-channel attacks. The attacker's ambition is to learn the key involved that, for example, can be used to decrypt ciphertexts. Symmetric encryptions are used to encrypt and decrypt big blocks of data. The same key is used for encryption and decryption, therefore this mechanism is called symmetric encryption. The AES is the current standard for symmetric encryption. A competition was started in 1997 to find the best algorithm for the new standard. As the winner of the selection process, the Rijndael algorithm proposed by Daemen and Rijmen [DR98] was standardized in the year 2000 by the

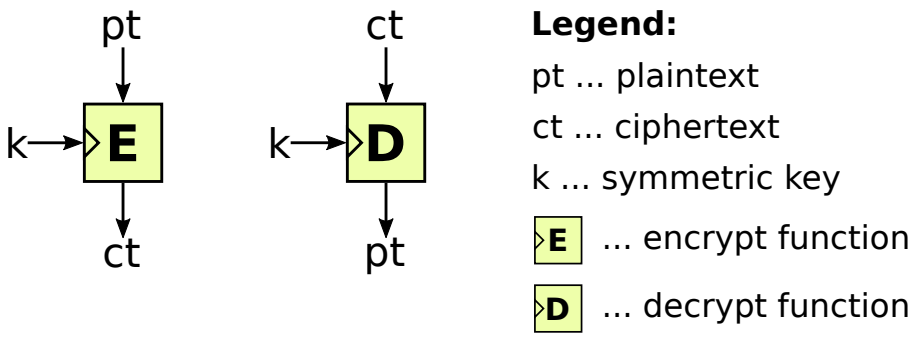

Figure 3.7: Block cipher for encryption and decryption.

National Institute of Standards and Technology (NIST). The AES encryption and decryption functions perform cryptographic operations with a secret key involved. The operations can leak information through side-channels. We take a closer look at the AES and different modes of operation, including a leakage resilient mode.

AES is a block cipher, that means the algorithm operates on fixed-size data blocks. The AES algorithm has a block size of 128 bits and the possible key sizes are 128, 192, and 256 bits. The left part of Figure 3.7 illustrates a block cipher encryption function. The encryption function encrypts the plaintext (`pt`) with the key (`k`) and hence generates a ciphertext (`ct`). Plaintext (`pt`) and ciphertext (`ct`) have the same size as the block size. The decryption function decrypts the ciphertext (`ct`) with the same key (`k`) as used for encryption, shown right in Figure 3.7.

DPA on AES. The AES encryption and decryption functions are often a target of DPA attacks with the goal to extract the secret key. One essential part of the AES algorithm is the S-box. It is a substitution step that can be implemented using a lookup table, as shown in Listing 3.1. Within the AES algorithm, the input plaintext and key get connected with an Exclusive OR (XOR) and then the S-box is applied, as shown in Listing 3.2. This part of the algorithm is a good intermediate result to mount a DPA attack. The measurement is focused on the timestep where the described operation is performed and uses a lot of different input plaintexts. Further, Listing 3.2 is used to generate the hypothesis by calculating this equation for every

Listing 3.1: AES S-box implemented as lookup table in C/C++.

```
uint8_t sbox[256] =
{
  0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01,
  0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, 0xCA, 0x82, 0xC9, 0x7D,
  0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4,
  0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
  0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, 0x04, 0xC7,
  0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2,
  0xEB, 0x27, 0xB2, 0x75, 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E,
  0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
  0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB,
  0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF, 0xD0, 0xEF, 0xAA, 0xFB,
  0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C,
  0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
  0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, 0xCD, 0x0C,
  0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D,
  0x64, 0x5D, 0x19, 0x73, 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A,
  0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
  0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3,
  0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D,
  0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A,
  0xAE, 0x08, 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
  0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A, 0x70, 0x3E,
  0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9,
  0x86, 0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9,
  0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
  0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99,
  0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};
```

Listing 3.2: AES S-box usage implemented in C/C++.

```
intermed = sbox[plaintext ^ key];
```

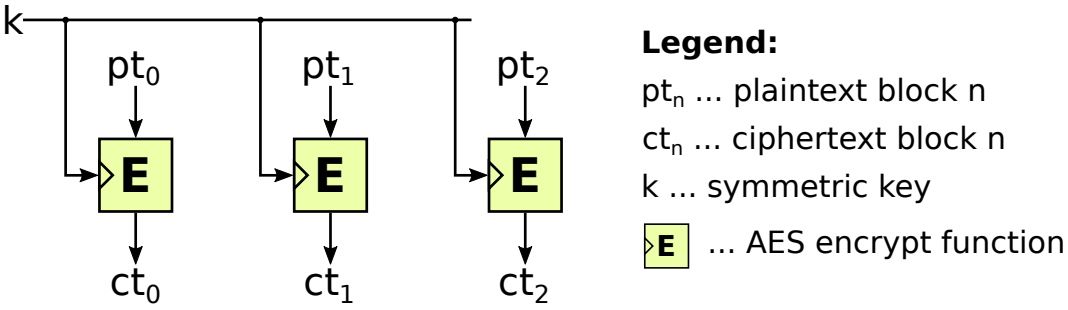

Figure 3.8: Electronic Code Book (ECB) mode for encryption.

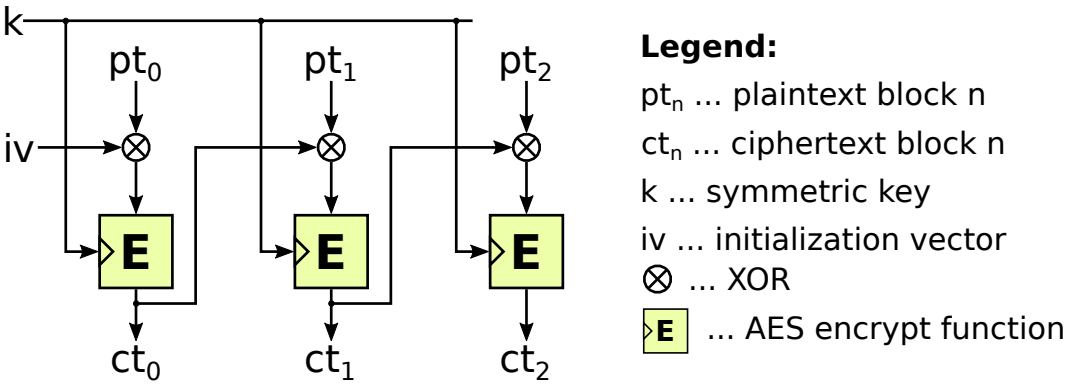

Figure 3.9: Cipher Block Chaining (CBC) mode for encryption.

possible value of the key byte. Next, the power model is applied and compared to the recorded traces using a correlation coefficient. In the end, the correct key guess leads to the highest correlation to the measurement. This procedure is repeated for every key byte, and the attacker finally knows the whole secret key. A DPA is successfully applied to the AES algorithm.

Modes of Operation. Block ciphers, like AES, are used in so-called modes of operations to deal with memory junks bigger than a single block. The most straightforward example is the Electronic Code Book (ECB) mode, where the input is split up into blocks and then fed into the block cipher, as shown in Figure 3.8. The ECB mode is however not recommended since equal plaintext blocks result in the same ciphertext block. This problem is fixed in chained operation modes like the Cipher Block Chaining (CBC)

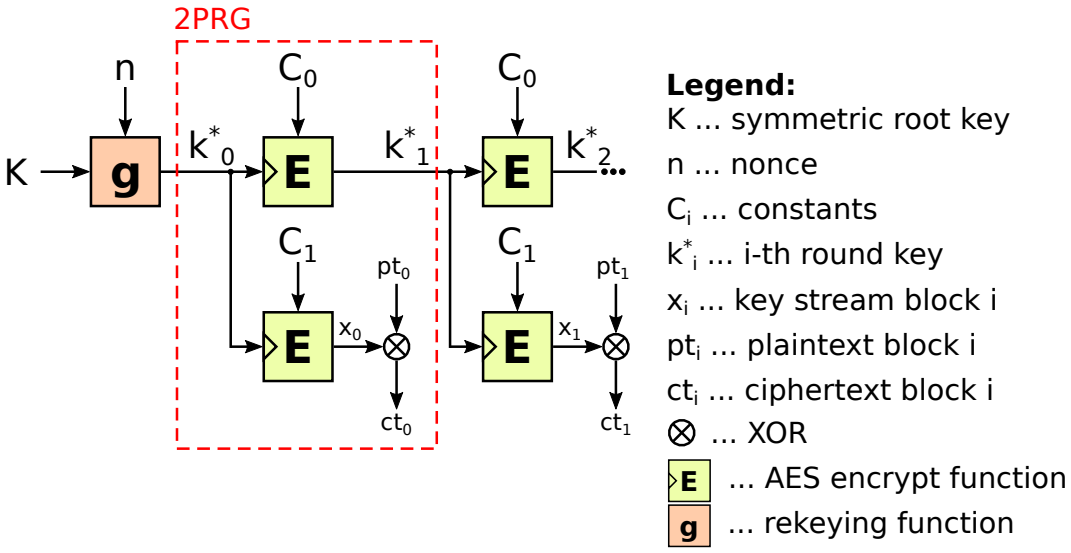

Figure 3.10: Leakage-resilient mode of operation.

mode. In CBC, an initialization vector (iv) is used to get freshness for each ciphertext. With this additional randomness, two identical messages get different initialization vectors and therefore a different ciphertext. As shown in Figure 3.9, every cipher block operates with the same key (k). Therefore the scheme is vulnerable to differential attacks, even if only one ciphertext/plaintext pair is processed because every block creates a trace belonging to the same key (k). By exploiting all the traces, an attacker can gain knowledge of the key.

Leakage Resilience. To overcome DPA vulnerabilities in modes of operations, Pereira et al. [PSV15] proposed a leakage resilient encryption scheme. The underlying construction is shown in Figure 3.10 and uses an AES encryption function as a Pseudo-Random Generator (PRG). This structure of two PRGs driven with one key is called 2PRG. The scheme is a stream cipher, creating a key stream (x_i), the so-called pad. This pad is then applied with the XOR operation on a plaintext to encrypt, or on a ciphertext to decrypt. First, the round key k_1^* is generated by encrypting the constant C_0 with the key k_0^*. Second, a pad, used later as a key stream, is generated by encrypting the constant C_1 with the key k_0^*. As a result, one key, e.g.,

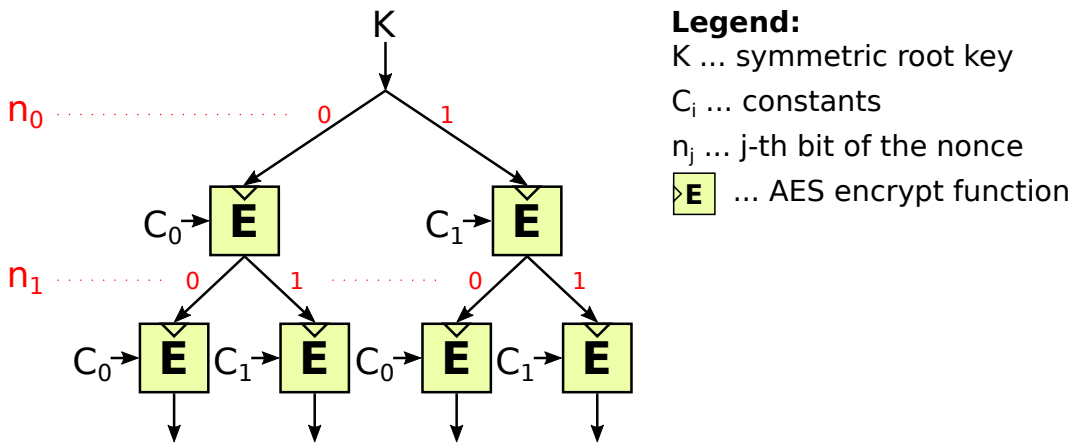

Figure 3.11: Re-keying function (g) used in Figure 3.10.

k_0^* is used no more than twice in an AES encryption function. Therefore, an attacker can observe no more than two different traces using the same key. This property is called 2-limiting. It is a common assumption in the literature [Sta+10] that two traces using the same key are not enough to sucessfully perform a DPA.

A number only used once (nonce) is included in the scheme to get freshness. If the same content is encrypted twice, the ciphertext will differ, since on every encryption a fresh nonce is used. In perspective of side-channel attacks, this means, that the applied round keys (k_0^*, k_1^*, ...) will differ and the attacker can not collect more traces with the same key. The re-keying function g is used to apply the nonce on the root key K. The re-keying function also uses the 2PRG construction using symmetric encryption and is shown in Figure 3.11. Depending on the value of the nonce n, a different path through the graph is taken. Every nonce bit decides whether the constant C_0 or the constant C_1 will be encrypted. The output of the re-keying function is derived from the root key K and used as first round key k_0^*. Every possible key inside the re-keying function is only used in two different encryption runs, particularly for the constant C_0 and the constant C_1. Therefore, the re-keying function is also 2-limited, and the same argument regarding differential side-channel attacks as stated before holds.

3.3.3 Hash function

Hash functions are fundamental and necessary building blocks in today's cryptography. A hash function is a cryptographic one-way function. MD5, SHA-1 and the families of SHA-2 and SHA-3 are well-known hash functions. A big data block, a so-called image x, is mapped to a constant-size hash value h, e.g., with a size of 256 bits for SHA256, by computing $h = H(x)$. For example, when downloading Linux images, hash values of different algorithms are displayed on the webpage to check for transmission errors after downloading. In this way, a hash function fulfills the same requirements as a checksum, but has additional cryptographic requirements. The requirements are *collision resistance*, *preimage resistance* and *second preimage resistance*.

A collision occurs, if it is computationally feasible to find two images $x \neq x'$ that result in the same hash value $h = H(x) = H(x')$. Due to the fact that the hash value computed on a large input is of limited size, usually only a few hundred bits, there will always exist collisions. This problem is known as birthday paradox. The requirement for the hash function is to hinder the calculation of collisions, which is known as *collision resistance*.

If it is possible to reverse the hash function $H(x) = h$, such that an image x can be computed from a given hash h, it is possible to compute preimages. For example, hash functions are used as a one-way function to store passwords one-way encrypted into a database. If an attacker gets access to the password database, computing a preimage from the hash value allows recovering passwords. The *preimage resistance* prevents from recovering the passwords.

Computing a second preimage means to compute a second image x' given the orignal image x, where $x' \neq x$, such that $H(x') = H(x)$. A hash function that is vulnerable to second preimage attacks is useless for many applications of cryptography. *Second preimage resistance* is important for signature schemes, where hash functions are used within a proof that states the origin of the image. For example, a document containing a contract is cryptographically signed. If an attacker can modify the document such that the hash value of the document stays unchanged, by computing a second preimage, it is possible to forge the contract while the signature stays valid.

As hash functions are used to guarantee integrity, they are also used in encrypt-then-hash schemes. A plaintext is encrypted first, and the resulting ciphertext is hashed. In this scheme, the hash function operates with no secret values. Therefore, the hash function is not valuable for side-channel attacks in this scheme.

3.3.4 Message Authentication Code

A Message Authentication Code (MAC) is used to verify the origin of a message. This approach ensures the integrity of messages sent between two parties. The two parties share a secret, also called password or a key, that is used to compute the MAC value. This MAC is sent along with the message, and the other party can verify it, using the shared secret. A MAC protects against transmission errors and illegal modifications.

HMAC Definition. A Hash-based Message Authentication Code (HMAC) is a MAC based on hash functions, which takes a message and the shared secret key as an input. As proposed by Bellare et al. [BCK96] the HMAC function is defined as shown in Equation 3.3 using a hash function (H), a secret key (k) and input data (x). The key length is variable, thus the security relies on this length. An attacker can apply a brute-force attack to find the secret used for the HMAC calculation. The key complexity as well as the size of the hash determines the computation time an attacker has to spend to find the secret. Of course, an attacker can rent a massive amount of cloud instances to reduce the real time to success, but the key complexity and hash size influences the financial effort required.

$$HMAC(x,\ k) = H(k \oplus \text{opad},\ H(k \oplus \text{ipad},\ x)) \tag{3.3}$$

HMAC Computation. The constants `ipad` and `opad` are applied with the XOR function to the key. Therefore, the key is padded with zero bytes to 64 bytes length. The constants are byte vectors with the length of 64 bytes where every byte of a vector holds the same value. Bellare et al. [BCK97] define $0x36$ for `ipad` and $0x5C$ for the `opad`. After computing the inner term ($k \oplus$ ipad) the result is concatenated to the input data (x). The result is hashed with the function (H) and the result of this operation is appended to the result of the outer term ($k \oplus$ opad). Again the result is hashed with the hash function (H) and the result is the output HMAC value.

An HMAC implementation can be vulnerable to side-channel attacks. For instance, Lemke et al. [LSP04] show a DPA on the HMAC algorithm, that extracts the involved secret. If an attacker can reveal the secret, the application building on the HMAC is threatened. For example, an attacker can create authenticated telegrams in an HMAC protected communication, which breaks the authenticity of the communication.

4 Building Blocks

Physical attacks are a significant problem for Industrial Internet of Things (IIoT) devices. One possible attack is to read out or exchange storage, for example, an SD-card. To protect the storage, a suitable mechanism is secure boot. Secure boot checks the authenticity of software before decryption and execution. However, as already covered in Section 3.2, secure boot is vulnerable to replay attacks. To mitigate these attacks, we use a Trusted Platform Module (TPM) to provide secure storage. An authenticated communication channel is used to prevent physical attacks on the communication to the TPM. This channel requires good random numbers.

Section 4.1 covers details about secure boot. Section 4.3 explains general properties and function of TPMs. Section 4.2 describes different approaches for random number generation.

4.1 Secure Boot

Unprotected IIoT devices do not require a highly educated attacker to steal data. With physical access, an attacker has easy access to the storage, e.g., an SD-card, of the IIoT device. Consider the following example: An attacker removes the SD-card from the IIoT device and puts it into a Personal Computer (PC). Doing so, the attacker has full read and write access to software and data stored on the SD-card. All secrets of the IIoT device are visible to the attacker. Further, the attacker can modify the software and for example, introduce backdoors and malware.

Encryption is used to prevent an attacker from reading storage. The use of encrypted storage requires a key for decryption. This key needs to be stored somewhere on the device. If this key is also saved in an unprotected

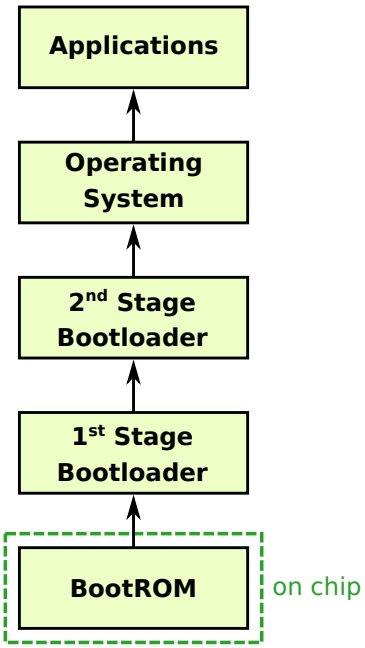

Figure 4.1: Chain of Trust.

location, the attacker can steal the key and decrypt the storage. To protect against this attack, the key needs to be stored securely. Secure boot is an established mechanism to protect the executed firmware. As ground truth, keys are burnt into a once writeable memory inside a System on Chip (SoC). The ground truth is also known as the root of trust, a physically protected storage. An attacker can not read out the root keys. In secure boot, the execution starts with the zero-stage bootloader, the so-called BootROM, which is located in a Read-Only Memory (ROM) and must not be changed. The BootROM, implemented by the SoC vendor, allows the verification and decryption of the executed software, that for example, is stored on an SD-card. Usually, several boot stages, as shown in Figure 4.1, are used in an IIoT device. Every stage verifies and decrypts the next one. This series is commonly known as the chain of trust. Hereby, secure boot ensures integrity and confidentiality for all software stages running on the device. Therefore, secure boot is an excellent feature to secure IIoT devices.

However, secure boot is vulnerable to replay attacks, as discussed in Section 3.2. Further, side-channel attacks, as discussed in Section 3.3, can break the chain of trust. We will revisit these problems later in this thesis.

4.2 Random Number Generation

True random numbers are a fundamental need for computer security. Random Number Generators (RNGs) are, as indicated by the name, used to generate random numbers. To generate cryptographic keys, for example, symmetric encryption keys, an RNG is used. An RNG is encountered as weak if the entropy of the output is too low, which means, that there is not enough randomness included. This problem leads to a limited set of possible outputs even if the output may be larger. If the key generation only creates a few thousand different keys, an attacker can try all possibilities and find the correct key. If the RNG always creates the same sequence of numbers, keys generated from these numbers can be predicted. These examples show that good random numbers are essential. Two different approaches are discussed in the following paragraphs.

Pseudo-Random Number Generator (PRNG). One approach is to use a PRNG, that uses a deterministic algorithm to generate new random numbers from a random initial value, the so-called Initialization Vector (IV). There exist different ways how to build a PRNG. One possible design is the use of cryptographic sponges, as proposed by Bertoni et al. [Ber+10]. Another way is to use the Advanced Encryption Standard (AES) as a Pseudo-Random Generator (PRG) for the generation of a random stream. Figure 4.2 shows an AES based PRNG that is similar to the leakage-resilient cryptography scheme described in Section 3.3.2. The IV, also known as the seed of the PRNG, is composed out of different sources that is expected to include randomness. Typical sources used to compose the IV are uninitialized memory, network traffic or the noise in analog inputs. The structure of the PRNG ensures, that when the output r_i is known, it is not possible to calculate back to the state s_i. Further, it is not possible to compute a previous output r_{i-1} or a future output r_{i+1}. These properties are fundamental in

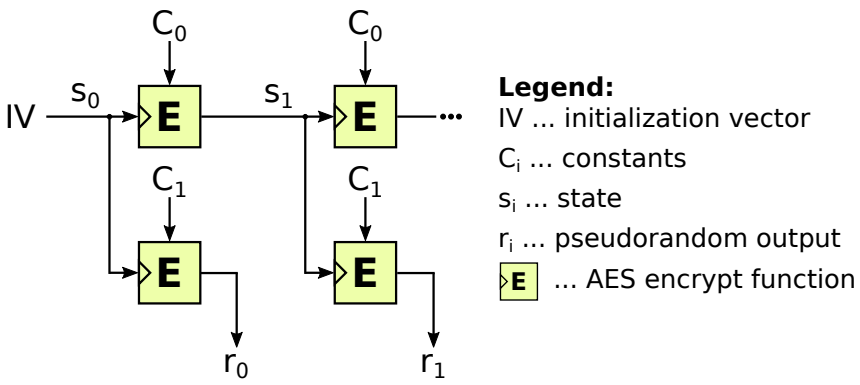

Figure 4.2: Advanced Encryption Standard (AES) based 2PRG scheme used as Pseudo-Random Number Generator (PRNG).

the design of PRNGs. Important for selecting the parameters is that the constants C_0 and C_1 are not equal. In this case, the output r_i and the next state s_{i+1} are equal, and a calculation of all future outputs is possible.
It is possible to store the state of the PRNG on an encrypted filesystem. In this way, the system can continue with the same state at the next power cycle. However, this behavior is vulnerable to replay attacks, as described in Section 3.2. If an attacker can restore old images of the filesystem the PRNG is threatened. Using this attack, the generated random numbers will be equal to the one from the last power cycle. Since PRNGs are used to generate numbers only used once (nonces), communication protocols are in danger in this case. For example, if the device under attack sends a request to a server, the request can contain a nonce to ensure freshness. More precisely, the nonce protects against a replay of the server response. If the nonce stays constant, an attacker can replay a server response and therefore trick the device under attack.
Another problem of PRNGs using this approach is that the device might get not enough randomness for the IV. Heninger et al. [Hen+12] demonstrated that weak randomness could cause serious problems. For example, on the first start of a device, Secure Shell (SSH) keys might be generated. If there is not enough randomness in this phase, the security of communications is compromised. This problem is caused by missing entropy in the IV, which propagates to the output of the PRNG. Missing entropy on an RNG output is a problem, as initially mentioned.

Side-channel attacks are also a threat for PRNGs. The output of a PRNG is often security relevant, for example, used as a cryptographic key. Since the attacker must not learn the key, the PRNG design needs to be hardened against side-channel attacks as well.

True Random Number Generator (TRNG). A TRNG is a hardware implementation of an RNG that generates random numbers out of a physical process. For example, a lottery machine is a TRNG, which uses physical randomness to draw random numbers. In electronic devices, sources for randomness are, e.g., the noise of analog measurements or unsteady electric circuits. A TRNG claims to produce true random numbers, which means that the output is from high-quality randomness. However, testing randomness is not trivial. When encountering a binary output signal with the states 0 and 1, one test is to check for the occurrence of both symbols in a very long sequence. An alternating sequence "010101..." will perfectly master this test, but is not random at all. More sophisticated tests for randomness are standardized by the National Institute of Standards and Technology (NIST) [Bas+10].
Different SoCs, like the AM3358 of Texas Instruments, have a hardware RNG integrated. However, not all SoC vendors integrate a TRNG to their designs. If the SoC has no TRNG, but a Field Programmable Gate Array (FPGA) included, a Phase-Locked Loop (PLL) based TRNG can be realized. For example, Allini et al. [All+18] developed a TRNG to be integrated into an FPGA design. The TRNG is shipped as a so-called Intellectual Property (IP) block. This IP block can then be added to an FPGA design. On system startup, the TRNG is loaded with a bitstream into the FPGA. This process can also be protected by the secure boot. During operation, the TRNG emits random numbers that can be processed in the system. In this way, a system can use strong random numbers that are required for cryptographic operations. TRNGs are also used to seed a PRNG if the output bandwidth of the TRNG is too low.

4.3 Trusted Platform Module (TPM)

A Trusted Platform Module (TPM) is a security module in today's computers, which offers a lot of different possibilities. A TPM has features like a TRNG, Platform Configuration Registers (PCR), non-volatile storage, sealing, authentication and cryptographic primitives. Therefore a TPM is a universal component for all kinds of different usecases, like Digital Rights Management (DRM), attestation of secure boot, the so-called measured boot, or as security coprocessor.

There exist different types of TPMs. First there is a software TPM, which is mostly used for testing purposes as a TPM simulator. If the software implementation runs inside a trusted execution environment, it can also be a secure TPM. Another option is the firmware TPM, where the TPM implementation is integrated into a security coprocessor that may fulfill other tasks as well. For example, the Intel Management Engine (ME) is a security coprocessor on the PC mainboards, where a TPM is integrated. The last option is a hardware TPM, an external chip that implements the TPM specification. The hardware TPM is the most secure option. Vendors like Atmel, Intel, Infineon, and STMicroelectronics integrate physical protections to the TPM chips.

The Trusted Computing Group (TCG) specifies the functionality and interfaces of a TPM in the TPM specification. Earlier TPMs, following the outdated TPM specification version 1.2 had multiple problems. First, these TPMs had vulnerabilities to physical attacks as shown by Winter and Dietrich [WD11] and Kauer [Kau07]. Second, the specification was restricted to a fixed set of cryptographic algorithms. For example, the hash algorithm MD5 was broken in the last years, which lead to the requirement of a revision of the TPM specification. The current version of the TPM specification, version 2.0 [Gro14], improved this property and is now flexible regarding the used cryptographic algorithms. Trusted Platform Module version 2 (TPM2) is not backward compatible to old TPM specifications. Section 4.3.1 covers fundamental features of TPM2.

4.3.1 TPM Features

Cryptographic Primitives. The TPM implements different cryptographic primitives including hash algorithms, Hash-based Message Authentication Code (HMAC), asymmetric and symmetric algorithms. The algorithms are used internally, for example, in PCRs described in the next paragraph, but also are available to a caller as commands. In this way, a TPM can be used as a cryptographic coprocessor and provide cryptographic primitives for platforms with little computational power. The advantage of this approach is that the cryptographic implementations are reviewed and should be error free. However, using the TPM as cryptographic coprocessor does not allow to build a secure channel between a caller and the TPM. Cryptography needs to be implemented on the caller's side, to build a secure channel. As a result, this approach is only suitable for settings without physical attackers.

Platform Configuration Registers. Platform Configuration Registers (PCR) are an attractive technique suitable for attestation of secure boot. It is based on the principle of a hash chain, in terms of TPMs called hash extend. The hash of the previous input is concatenated to the next input and hashed together. The resulting hash is concatenated with the next input value and again a hash computed. The mechanism of PCRs is used for the so-called measured boot. Measured boot is used additionally to secure boot and is used to monitor the process. While the secure boot progress is executed, hashes over every boot stage are written to the PCR and therefore appended to the hash chain. After the boot progress, the TPM can use the measurement to prove the performed secure boot. The result of the PCR is a guarantee for the executed chain of trust and can be used for attestation to prove a systems state.

Non-volatile storage. Chips implementing the TPM2 specification offer storage of non-volatile data in different variants, like unstructured data, counters, bitfields, and hash extend fields. The most straightforward case is unstructured data, that can be read and written with arbitrary data. Next, there are counters that only can be incremented. One use case for these

counters would be a version information to protect against replay attacks. TPM2 also supports bitfields, that can be configured so that bits can only be set. This is like a hardware fusebox, with the exception, that a privileged user can reset it. Non-volatile storage can also be configured as an hash extend field. The hash extend field works equivalent to the PCR. This feature offers the possibility to use hash chains for custom tasks, for example, if all PCRs are already used.

Authorization. Different authentication techniques are supported to handle permissions for the TPM access. For example, the stored non-volatile data should only be read by authorized parties. The supported methods are password, Hash-based Message Authentication Code (HMAC) and policy. A password is the simplest method. The password is transferred in plaintext on the communication channel to the TPM. This type of authentication is vulnerable to a physical attacker since the attacker can eavesdrop the password. Once the password is known, the attacker can impersonate as an authorized user and read or write data. On the other hand, an HMAC session allows building an integer channel between a caller and TPM. The key used in the HMAC session is a shared secret between the caller and the TPM. Based on the protocol used for the HMAC session, the communication parties prove the knowledge of the shared secret. For all telegrams, requests, and responses, in an HMAC session, an HMAC value is calculated and verified. To ensure freshness, the HMAC authorization includes nonces. Therefore good random numbers are required on both sides, on the side of the caller and on the side of the TPM. Freshness ensures that old telegrams are not reused again in a replay attack on the TPM communication. The policy authorization is based on the HMAC authorization but requires additional conditions, like for example, a dedicated TPM state. Based on the methods of authentication, the caller can trust the values read from the TPM so that no one modified it. On the other hand, the caller proves the authorization to the TPM, so that the TPM can trust the caller. For this reason, the TPM can, for example, accept a write command to a secure non-volatile variable.

Random Number Generator. A True Random Number Generator (TRNG) is an essential component in modern cryptography. The TCG specifies that a TPM requires a TRNG. It is up to every TPM vendor to implement a proper TRNG into a hardware TPM. The TPM uses the TRNG internally to ensure freshness in HMAC sessions by generating nonces. A caller can request random numbers from the TPM. The required command is described in the next section.

4.3.2 TPM2 Commands

The TCG specifies an interface with several commands to communicate with a TPM. These commands are mapped to a binary representation that every TPM2 needs to support. A so-called TPM software stack implements the commands and offers an Application Programming Interface (API) to a programmer. This section lists a few of them and hereby focuses on commands used in this work.

TPM2_Startup. The `TPM2_Startup` command is used to bring the TPM into the operational state after power-on or a reset. If the command is not sent, the TPM will respond with an error and refuse the execution of any other command. In case the `TPM2_Startup` command is sent twice, the TPM also responds with an error.

TPM2_GetRandom. Using the `TPM2_GetRandom` command, random data can be requested from the TPM TRNG. It is a simple command with only one free parameter, the number of requested bytes. The TPM responds with the number of random bytes requested.

TPM2_StartAuthSession. To start an authorized Hash-based Message Authentication Code (HMAC) session, the `TPM2_StartAuthSession` command is used. Using this command, the caller has to provide a nonce to the TPM. The TPM answers with a nonce and a session handle. Both nonces have to be used in the following command belonging to the authorized session.

TPM2_NV_DefineSpace. With `TPM2_NV_DefineSpace` a caller can define a non-volatile object in the TPM memory. Therefore the caller has to select a free index, where the object should be located. Several parameters like size and type of the non-volatile object, e.g., unstructured data, counter or bitfield, can be selected. Also, the authorization has to be configured, this means, the secret for password authorization or an HMAC session.

TPM2_Write. After defining a non-volatile object, the caller can write data. `TPM2_Write` is used to write objects with the type unstructured data. The command can be used with all types of authorizations including password-based authorization and HMAC sessions. The index specified while defining the object is used for addressing the variable. It is possible to write only parts of a variable by specifying the offset and sending data smaller than the whole variable.

TPM2_Read. `TPM2_Read` is used to read all types of non-volatile objects. It supports the same authorization techniques like `TPM2_Write`. If an object was never written before, the TPM flags an error. This behavior is essential to prevent uninitialized reads and Use After Free problems, like described in Section 2.1, inside the TPM.

TPM2_NV_UndefineSpace. `TPM2_NV_UndefineSpace` is used to undefine a non-volatile object in the TPM memory. The command requires the index as an argument, removes the data and releases the index for future usage. The index needs to be defined within the command. Otherwise, the TPM will return an error.

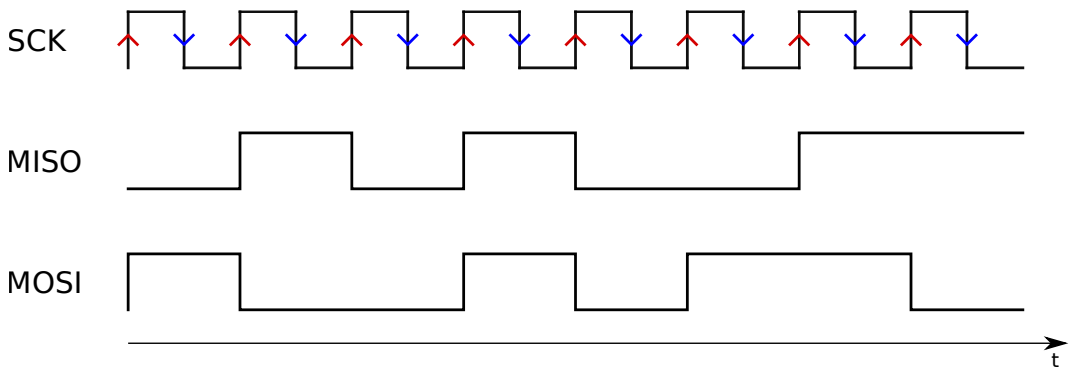

Figure 4.3: The timing diagram of an example SPI communication.

4.3.3 Hardware TPM / Low Level Protocol

The PC Client Platform TPM Profile (PTP) specification [Gro17] standardizes many properties a hardware TPM has to fulfill including chip packaging, interfaces, and the low-level protocol. For example, Serial Peripheral Interface (SPI) is one interface that can be used to connect a TPM to other components, like a SoC. The low-level protocol runs on the SPI and is used as a transport layer for the TPM commands.

SPI is a synchronous serial bus that is often used to connect different components in a range of up to a few meters, depending on the transmission speed. An SPI bus has one master and can have multiple slaves. The SPI master controller is a component that most SoC designs implement. This offers the possibility to extend the functionality of a SoC with external devices. Slaves could be, e.g., a memory, Input/Output (I/O) extensions, LED drivers, ethernet modules and many more. The bus consists of Master In Slave Out (MISO), Master Out Slave In (MOSI), Serial Clock (SCK) and Chip Select (CS) lines, where MISO and MOSI are data lines. Figure 4.3 shows an example timing diagram of communication on the SPI. The SCK line is used as a synchronous clock and driven by the SPI master. The master defines the transmission speed with the SCK line. With the CS line different members can be addressed on a bus, and the master can schedule the communications to the slaves. The SPI bus works with the principle of a shift register. Within one clock period, one bit is transmitted, which means that one bit is written and one bit is received. At the specified clock

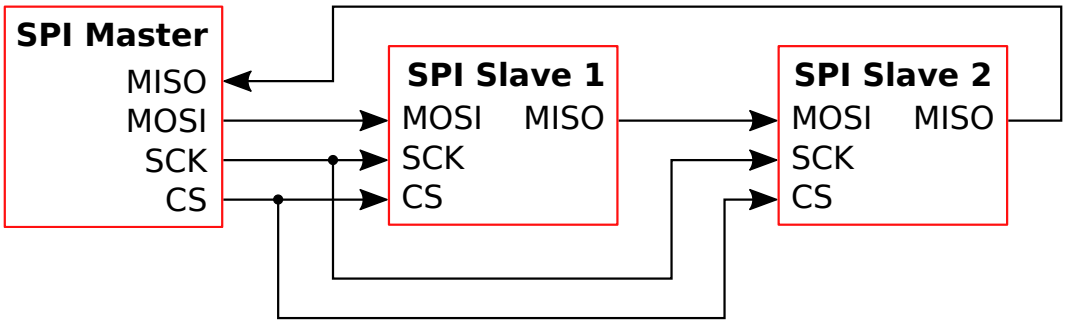

Figure 4.4: Example usage of the SPI bus with chained clients and a single CS.

edge, for example, the rising edge, the shift register reads the input. On the other clock edge, for example, the falling edge, the shift register writes its output. The parameter with the two possible options, positive or negative edge for reading from a data line, is called clock phase. Because SPI uses one clock edge for writing and the other clock edge for reading, SPI slaves can be chained together, as shown in Figure 4.4. The other free parameter is the clock polarity. Depending on the idle level of the SCK, the first edge is falling or rising. This parameter is called clock polarity and has two possible options. These two parameters, clock phase and clock polarity, have to match for all parties on an SPI bus. For the TPM, these properties are specified in the PTP. SPI also supports the use of multiple CSs, as shown in Figure 4.5, to use multiple clients on the same physical bus.

Because of the simple structure of SPI, there is no protection against physical attacks. To spy on the bus, devices like the Bus Pirate, are available. In any case, a physical attacker can apply a Denial of Service (DoS) attack, but that holds true for all Internet of Things (IoT) applications. The integrity and confidentiality of SPI based communication needs to be secured on a higher-level protocol.

The TPM uses a simple bit protocol on top of SPI as transport layer for TPM commands, described in the PTP [Gro17, Table 46]. It consists of one bit deciding whether it is a read or a write operation, address and data. A simple implementation of this protocol can be found in the tpm2_server [vbe16]. Using the *bit protocol* different registers can be accessed. These registers include a revision id and a vendor id, that is used to determine

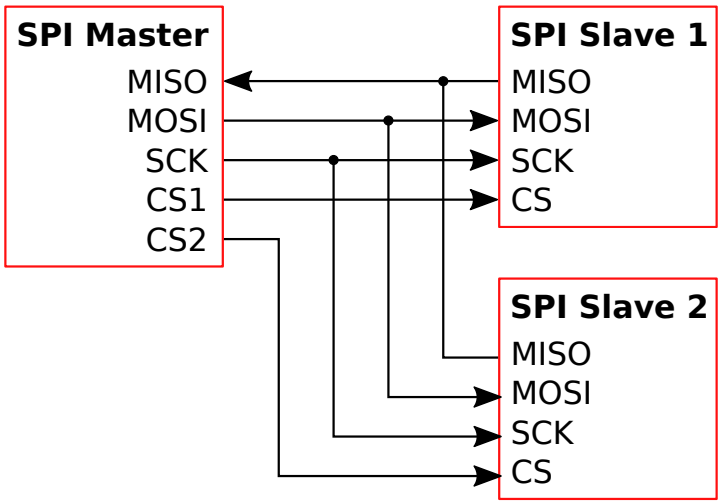

Figure 4.5: Example usage of the SPI bus with multiple CSs, where CS1 is the first CS and CS2 is the second CS.

the TPM origin. Further, there are status registers, access registers, and First In First Out (FIFO) registers. The FIFO registers are used to transmit TPM commands.

5 Prototype

The usage of secure boot for IoT devices is a good step towards device security. Nevertheless, secure boot is vulnerable to replay attacks. A significant requirement for Industrial Internet of Things (IIoT) devices is the resistance against these attacks. This chapter covers details about how the basic building blocks showed in Chapter 4 are used to reach the goals of replay resistance, mitigating side-channel attacks and excluding memory safety issues, and the respective implementation.

The central part of this work is the implementation of a bootloader, named *rs-zynq-boot*. We implemented the bootloader to prevent replay attacks on the firmware within secure boot. For the purpose, a version identifier of the firmware image is stored in secure storage that allows the bootloader to verify the version on every boot. The bootloader implementation is based on the modern programming language Rust to exclude memory safety issues. This is essential since the bootloader must not suffer from programming errors. Otherwise, the chain of trust and the replay resistance of the firmware would be threatened. In addition, we considered side-channel attacks in the implementation and integrated different side-channel countermeasures. We run *rs-zynq-boot* on the Zybo platform. However, the prototype platform being used does not offer secure storage, so we use a Trusted Platform Module (TPM) to save a version identifier.

Section 5.1 describes the concept and how the building blocks together fulfill our security goals. Section 5.2 gives details about the Zybo platform, the used TPM, and the required tools. Section 5.3 explains the structure and the implemented methods to mitigate replay attacks in secure boot. Section 5.4 covers the actual implementation of the bootloader. Section 5.5 illustrates the provisioning and the steps that are required in production to build a secure IIoT device.

5.1 Concept

In this work, our goal is to build a secure Industrial Internet of Things (IIoT) device. Since IIoT devices are deeply integrated into the environment, they suffer from physical attacks. For this reason, we want a strong binding of the firmware to the device. The device should only execute firmware of the IIoT device vendor, and the firmware should be kept confidential. The device needs protection from cloning and the production of replicas. We also require protection against replay and side-channel attacks, as covered in Section 3.2 and Section 3.3.

Secure boot is the first block we use to ensure the authenticity and confidentiality of firmware. However, secure boot is vulnerable to replay attacks. To prevent replay attacks, we need to verify the version of the firmware image. The IIoT device hence requires a tamper-resistant memory to store the reference value of the firmware version. However many System on Chips (SoCs) do not offer such storage that is unmodifiable by an attacker. A TPM offers secure storage and is hence chosen to be the second building block in our design. To prevent an attacker from tampering with the connection between the TPM and the CPU inside the IIoT device, we use an authenticated communication protocol. This protocol requires random numbers to prevent replay attacks on the communication. To get good random numbers, we use a True Random Number Generator (TRNG), which is the third building block. This setup can successfully defeat replay attacks.

However, if *rs-zynq-boot*, the bootloader implementing this concept, suffers from memory safety issues or is vulnerable to side-channel attacks, the whole concept is broken. We hence use the programming language Rust to guarantee memory safety for the implementation of the bootloader. We further ensure resistance against side-channel attacks by applying the following approach. First, a hash, which is stored in the TPM, prevents an attacker from modifying the ciphertext to enhance the leaked information. Second, a leakage-resilient cryptographic scheme, described in Section 3.3.2, reduces the number of ciphertexts decrypted using the same key to prevent differential side-channel attacks. Third, a bit-sliced implementation of the AES encryption function hinders single-trace attacks.

The combination of all these mechanisms results in a strong shield to protect an IIoT device against physical attacks. We prove the practicality of our concept with an implementation of a prototype. Section 5.1.1 takes a closer look at the implemented side-channel mitigations.

5.1.1 Side Channel Mitigations

Side channel attacks, as discussed in Section 3.3, on secure encrypted boot are a severe threat to IIoT devices. The different boot stages of the chain-of-trust are decrypted during boot. Therefore, every stage uses cryptographic algorithms and processes confidential data that is threatened by side-channel attacks. The implemented algorithms hence need to be hardened against side-channel attacks.

rs-zynq-boot is in the scope of a side-channel attacker. The decryption of the next stage is the target for such an attack. An attacker is interested in the next stage plaintext, which contains confidential credentials and Intellectual Property (IP). Also, the cryptographic key needs to be protected, since it also offers access to the plaintext. To protect the confidentiality of the next stage, we implemented countermeasures against side-channel attacks.

The first countermeasure against side-channel attacks is the usage of an encrypt-then-hash scheme in the next stage encryption. While boot, the hash is verified before decrypting. The decryption is only performed if the integrity has been successfully verified. The correct hash over the encrypted next stage is stored in our secure storage, the TPM. In particular, the boot-loader computes the hash over the encrypted image on every boot and compares the result with the value stored in the TPM. Since we can trust the secure storage, we can ensure the integrity of the encrypted next stage. If the computed hash value does not match the hash stored in the TPM, because malicious data was introduced, the decryption is not performed. This mitigation prevents an attacker from creating a large number of side-channel traces by modifying the ciphertext. Therefore the attacker is unable to collect enough traces for a differential side-channel attack such as a Differential Power Analysis (DPA). Since there is no confidential data involved in the hash computation, this approach is not vulnerable to side-channel attacks.

However, as discussed in Section 3.3.2, several modes of operations are vulnerable to side-channel attacks. For example, Cipher Block Chaining (CBC) uses the same key in every block and therefore generates a large number of traces that can be used in a differential side-channel attack. For this reason, the leakage-resilient scheme, based on Pseudo-Random Generators (PRGs), described in Section 3.3.2, was implemented and integrated into *rs-zynq-boot*. The implementation is encapsulated in a dedicated Rust crate, named *rs-crypto*. This offers the possibility to use this crate in other projects, like for the encryption of the firmware images, described in Section 5.5. The crate uses unit tests to check the implementation against test vectors.

While the vulnerability to differential side-channel attacks is covered quite well, a bad Advanced Encryption Standard (AES) implementation can lead to single trace side-channel attacks. Our implementation uses an AES implementation from the RustCrypto project [Tea18b]. It is a bit-sliced implementation of the AES function [RSD06; Kön08]. In this thesis we assume the bit-sliced AES implementation to be safe against Simple Power Analysis (SPA). This crate is called *aes-soft* and can be found in the crates repository.

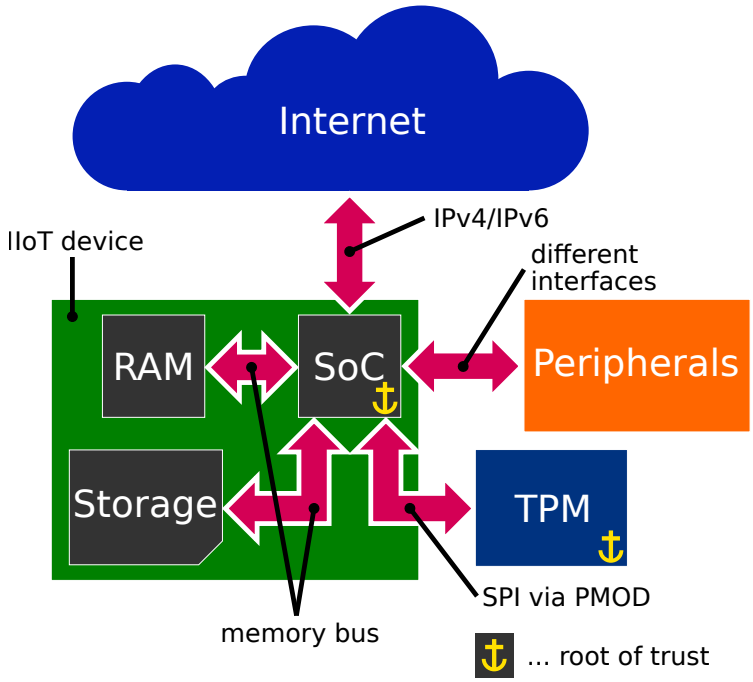

Figure 5.1: IIoT device with Trusted Platform Module (TPM).

5.2 Platform

This section covers the platform, which our implementation is based on. This platform supports the features described in Figure 5.1 and is used for building the prototype. Figure 5.2 shows the used hardware: a Zybo board, and a TPM module.

The Zybo board is a single-board computer, suitable for a wide range of tasks. The Zybo has different connectors, including an ethernet interface for a network connection and Universal Serial Bus (USB) connectors for all kinds of different peripherals. Important for this project are the so-called Peripheral Module (PMOD) connectors that the Zybo board includes, as shown in Figure 5.2 . Figure 5.3 shows the pin assignment of the PMOD connectors used on the Zybo boad. These connectors are 12 pin connectors with power supply and eight configurable signal lines. The signal lines are connected to the SoC, which is the core component of the Zybo board. It is

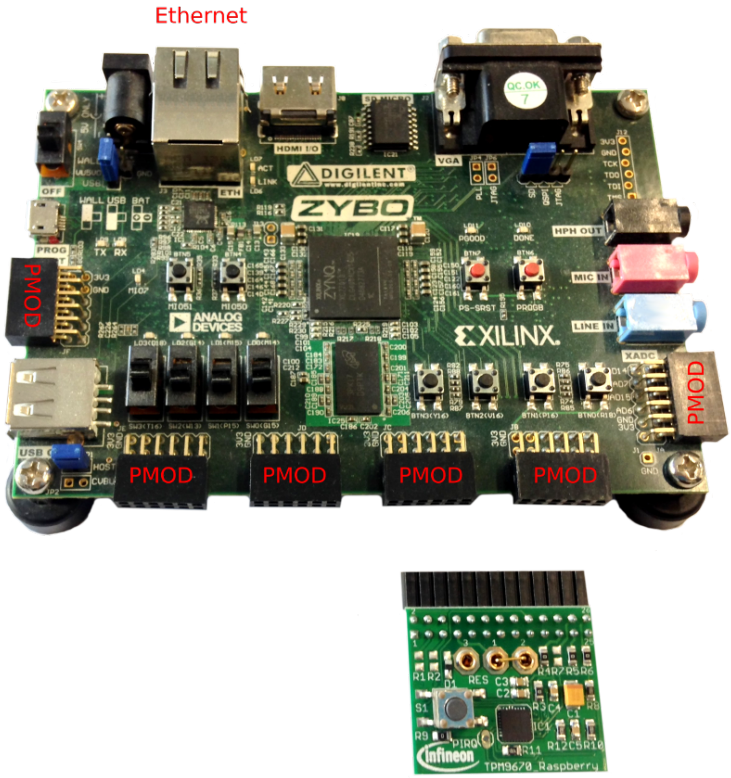

Figure 5.2: Zybo platform and TPM.

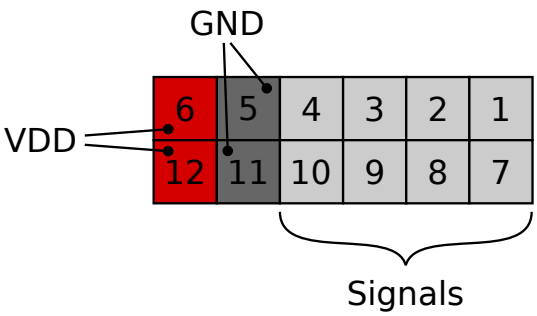

Figure 5.3: PMOD pin assignment, front view. VDD and GND are used to supply a PMOD module. The signals are directly connected to the FPGA and can be used for different use cases.

possible to use the signal lines as General Purpose Input Output (GPIO), where a digital signal can be read or written by software.

The SoC built into the Zybo board is a Xilinx Zynq 7010 SoC that comes with two ARM Cortex-A9 processor cores as the processing system. Further, the Zynq 7010 includes an FPGA of the Xilinx 7-series, that is internally connected to the processor cores with a memory bus. The functionality implemented on the FPGA can be accessed from the processing system with the concept of Memory Mapped Input/Output (MMIO). This means that a register of a component is mapped to a specific address the processor can reach. In software, this address is used to access the component. The SoC supports secure boot, as described in Section 4.1, to boot firmware to and load the bitstream file for FPGA configuration. Unfortunately, the Zynq 7010 SoC does not include secure storage to protect against replay attacks. For this reason, we use a TPM as a provider for secure storage.

The TPM being used is an Infineon *SLB*9760 Trusted Platform Module version 2 (TPM2) with a Serial Peripheral Interface (SPI). Distributors recently released a suitable TPM PMOD extension board as well. Since we realized the implementation earlier, we however used a TPM2 module for the Raspberry Pi, and built an adapter to fit the PMOD specifications. Figure 5.4 shows the wiring diagram realized inside this adapter. Through this adapter, a program executed on the Zynq SoC can communicate with the hardware TPM.

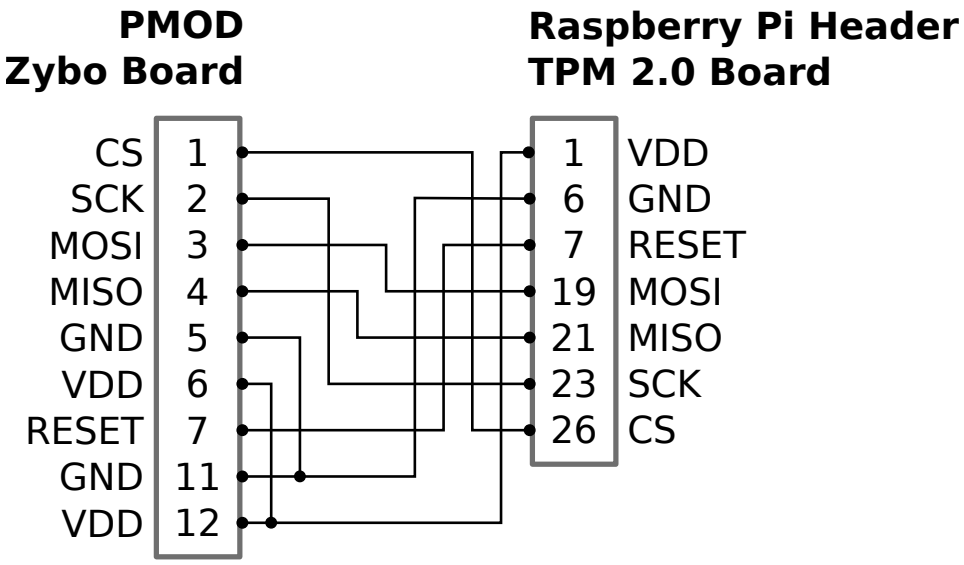

Figure 5.4: Wiring diagram between Zybo PMOD connector and the TPM board.

To establish a secure connection to the TPM, the SoC requires secure random numbers. Since the Zynq SoC does not offer a TRNG, an implementation from Viktor Fischer and Oto Petura, as described in Section 4.2, is used. The TRNG is an IP core for the FPGA and is accessible from the software running on the Central Processing Units (CPUs).

Software Tools. The Xilinx tools Vivado and Xilinx SDK are used in version 2017.2. Xilinx Vivado is used to program the FPGA and to configure hardware-related settings like clock frequencies for the processing system. Vivado creates a bitstream file and a hardware description file. The bitstream file is a binary file that is used for FPGA initialization. The hardware description file is reused in the Xilinx SDK. The Xilinx SDK is based on the Eclipse Integrated Development Environment (IDE) [Fou18] and used to program the processing system. Based on the hardware description file, a so-called board support package is created. With the board support package, drivers are generated to control basic components of the SoC and components integrated into the FPGA. The Xilinx SDK supports to also generate a First Stage Bootloader (FSBL). Since the Xilinx SDK is an IDE for the

languages C and C++, it is not suitable for programming Rust. We used the Rust package manager Cargo in version 0.25.0 for compiling and testing library crates. The bare-metal application is built with Xargo in version 0.3.10. Both build tools use the Rust compiler `rustc` underneath. The exact version information of the `rustc` used is shown in Listing 5.1.

Listing 5.1: `rustc` version.

```
$ rustc --version
rustc 1.24.0-nightly (73bca2b9f 2017-11-28)
```

5.3 Structure of Bootloader Implementation

Figure 5.5 shows the boot process of our bootloader. It is based on the secure boot feature the Xilinx Zynq SoC offers, but uses an additional root-of-trust. The original root-of-trust is located on-chip in the BootROM and the additional one is located in the TPM.

The execution starts in the BootROM, with the on-chip, non-modifiable zero-stage bootloader in the Zynq SoC. It loads the firmware image from the SD-card. This image includes the encrypted and signed FSBL. After verification and decryption, the BootROM hands over to the FSBL.

The FSBL is responsible for loading additional binaries and the bitstream file. The encrypted and signed bitstream file includes the TRNG, GPIO and the SPI wiring. Also, our bootloader, *rs-zynq-boot*, is included in the image located on the SD-card. It is encrypted and signed so that it is part of the secure boot process. The FSBL also loads the encrypted next stage firmware to the Random-Access Memory (RAM), where it is later processed by the *rs-zynq-boot*. As an alternative, *rs-zynq-boot* could be extended to access the SD-card and load the firmware directly.

When *rs-zynq-boot* takes over control, *rs-zynq-boot* first builds up an authenticated communication channel to the TPM and fetches the hash value stored in the TPM. Next, it computes the hash value over the encrypted next stage and compares it with the value received from the TPM. If both match, the integrity is proven and the next stage is decrypted. *rs-zynq-boot* hands over to the next stage.

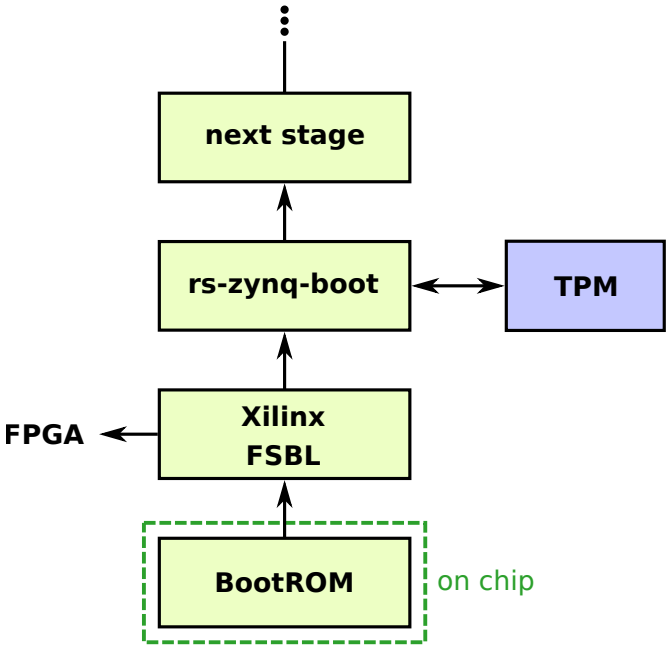

Figure 5.5: Chain of Trust implemented.

5.3.1 Crates

rs-zynq-boot is a bare-metal application for the Zynq processing system. An Executable and Linking Format (ELF) file is the result of a project build, that can be executed on the SoC. Figure 5.6 shows the dependency graph of the bootloader implementation. Every node is a Rust crate. *rs-zynq-boot* is the main crate, with the same name as the overall project, where the main function is located. The other crates are libraries included with Cargo, the package manager. Some of them are found locally since they live in the directory structure. Others are part of the official crates repository [Tea18a] and downloaded automatically with Cargo.

zynq. The *zynq* crate includes all Zynq related low-level implementations based on the principle of MMIO. It supports GPIO to control LEDs located on the Zybo board and to trigger a reset on the TPM module. The implemented UART allows input and output of text on the serial interface.

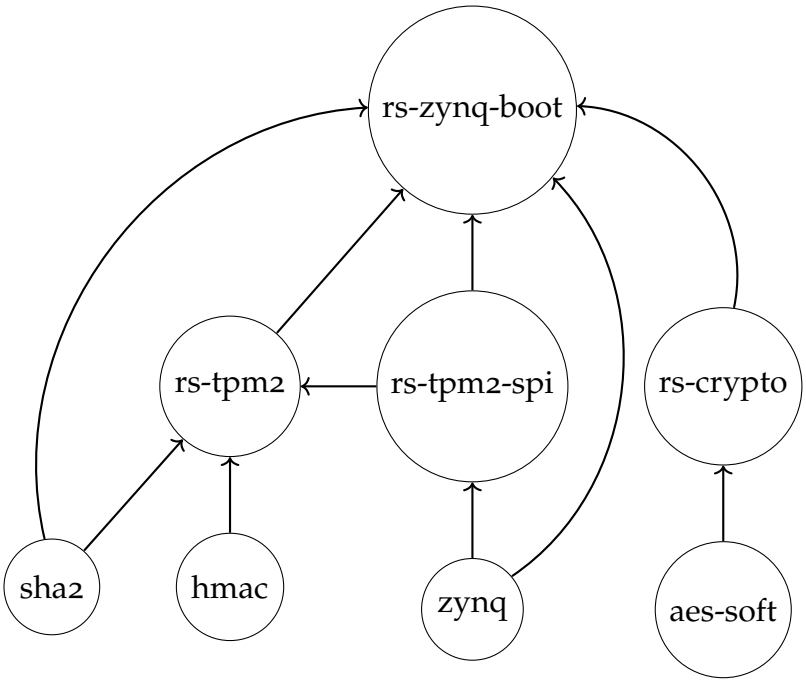

Figure 5.6: Dependency graph of *crates* used inside the bootloader.

This enables the possibility to track the program behavior and is used for logging of warnings and errors. Further, the SPI unit of the Zynq SoC can be controlled with the *zynq* crate. This is essential for the communication with the TPM. Since the TRNG is integrated into the FPGA design and reachable via MMIO, the *zynq* crate is responsible for fetching random values from the TRNG.

rs-tpm2. *rs-tpm2* is a light-weight implementation of a TPM software stack. It supports different commands and can use them with Hash-based Message Authentication Code (HMAC) authorization. To compute the HMAC value, a cryptographic hash primitive is required. Therefore the sha2 crate from the RustCrypto project is included. The *sha2* crate is used in version 0.7 and supports the SHA2 family of cryptographic hash functions. Out of these, the SHA256 function is used. The TPM also supports SHA256 for the HMAC session. To calculate the HMAC, the *hmac* crate from the RustCrypto project is used in version 0.5.

rs-tpm2-spi. *rs-tpm2-spi* is a Rust crate that handles the TPM low-level connection. It is responsible for starting up the TPM. The crate sets up the bit protocol to the TPM, described in Section 4.3.3. The implementation is based on the C implementation of tpm2_server, that we ported to Rust. On startup, the TPM vendor ID is checked to ensure that the TPM is responsive. Because of the available hardware, only the vendor ID of Infineon is accepted, but this check can be easily extended to other vendors. The *rs-tpm2-spi* crate offers a callback function to be used in the *rs-tpm2* crate. With this callback function bytes can be sent to the TPM, and the response is propagated back to the caller. Underneath, data is packed into the bit protocol and sent through the First In First Out (FIFO) register of the TPM to the TPM2 core. The response is fetched and written back to the caller.

rs-crypto. We implemented the leakage resilient AES mode of operation described in Section 3.3 in the crate *rs-crypto*. The crate includes the stream-cipher implementation and the re-keying function. Section 5.1.1 explains the motivation for the implementation.

rs-zynq-boot. *rs-zynq-boot* is the main crate of this project, which depends on the *sha2* crate since the SHA256 hash function is used to calculate the hash of the next stage. The calculated hash is compared to the value read from the TPM. It uses the *rs-tpm2* crate for assembling TPM2 commands that are transmitted via the callback function offered by the *rs-tpm2-spi* crate. *rs-zynq-boot* includes the *rs-crypto* crate for the leakage-resilient decryption

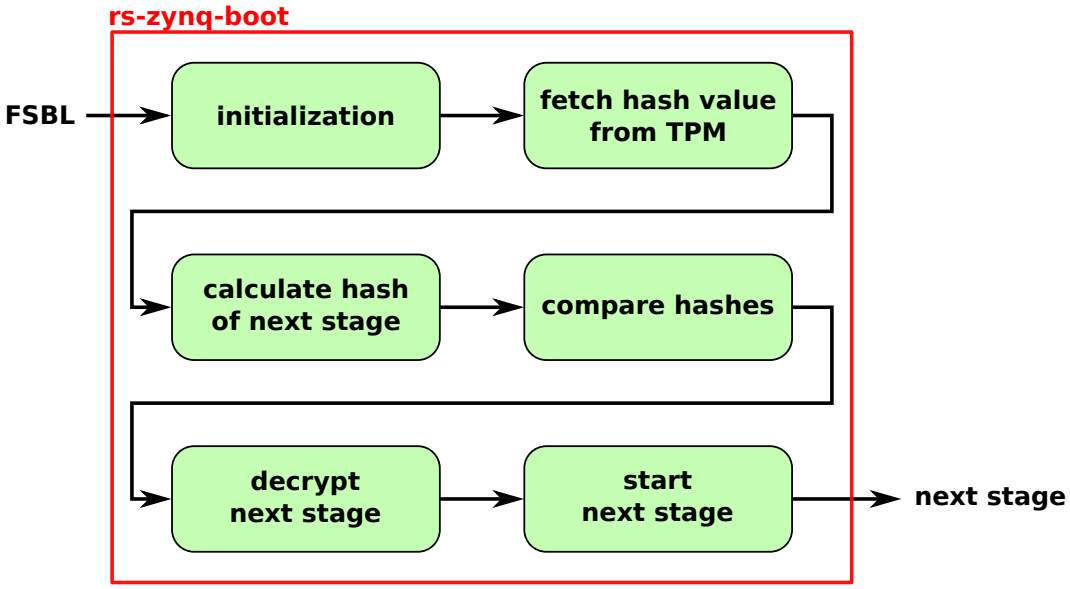

Figure 5.7: Processing sequence performed by *rs-zynq-boot*.

of the next stage. It also depends on the *zynq* crate to output debug messages and for logging. The exact procedure is described in the next section.

5.4 rs-zynq-boot

Figure 5.7 shows the execution flow that *rs-zynq-boot* follows. It includes six states that are executed sequentially until the next stage, possibly a bootloader or an operating system, is executed. This process guarantees the integrity and confidentiality of the next stage and prevents replay attacks on the next stage's firmware.

The initialization step sets up all hardware components. First, it performs a reset of the TPM module via GPIO. For this purpose, a one-bit output line is connected physically to the TPM module reset input. Using this line, the TPM can be reset, meaning the TPM is brought back to a clean state. Next, *rs-zynq-boot* sets up the SPI unit of the Zynq SoC. These settings contain the SPI baud rate, clock phase, and polarity that are set according to the PC

Client Platform TPM Profile (PTP). After initializing the SPI unit, *rs-zynq-boot* can start the communication with the TPM. The low-level initialization of the TPM bit protocol, implemented in *rs-tpm2-spi*, is performed.

A cryptographic hash of the next stage represents a specific version since every modification on the next stage can be detected. The TPM is used to store the cryptographic hash of the next stage. While booting the system, the hash of the next stage is read from the TPM through an authenticated channel, based on the protocol shown in Figure 5.8. The TPM enters the operational state after receiving the `TPM2_Startup` command. We use an HMAC authorized session to the TPM to ensure the integrity of the connection and further, the integrity of the secure storage.

5.4.1 Authorized TPM Session

To use an authorized session with the TPM, a session needs to be created first. The TPM responds with a session handle that is used to assign commands to the correct session. When sending a telegram, the caller and the TPM generate a fresh number only used once (nonce). The nonce of the other party is used within the next command in the respective session.

The command `TPM2_StartAuthSession(nonceCaller)` is used to start an authorized session on the TPM. As an argument, the `nonceCaller`, a fresh nonce, is sent. The TPM responds with the `sessionHandle` and the `nonceTPM`. The sessionHandle is an identifier for the current session and equals $0x2000000$ if only one session exists. The `nonceTPM` is required for the next command in the authorized session.

In *rs-zynq-boot* the command `TPM2_NV_Read(sessionHandle, nvIndex, size, offset, nonceCaller, nonceTPM, HMAC)` is used to read from the non-volatile variable. This command is already authenticated. As described before, the `sessionHandle` is used in every request to the TPM. The non-volatile storage is addressed with the arguments `nvIndex`. The size and offset refer to the memory inside the variable. For example, it is possible to read the last two bytes of a 32-byte variable with `size=2` and `offset=30`. The bootloader however reads from zero offset 32 bytes, according to the block size of the SHA256 hash function, that is used to compute the hash over the

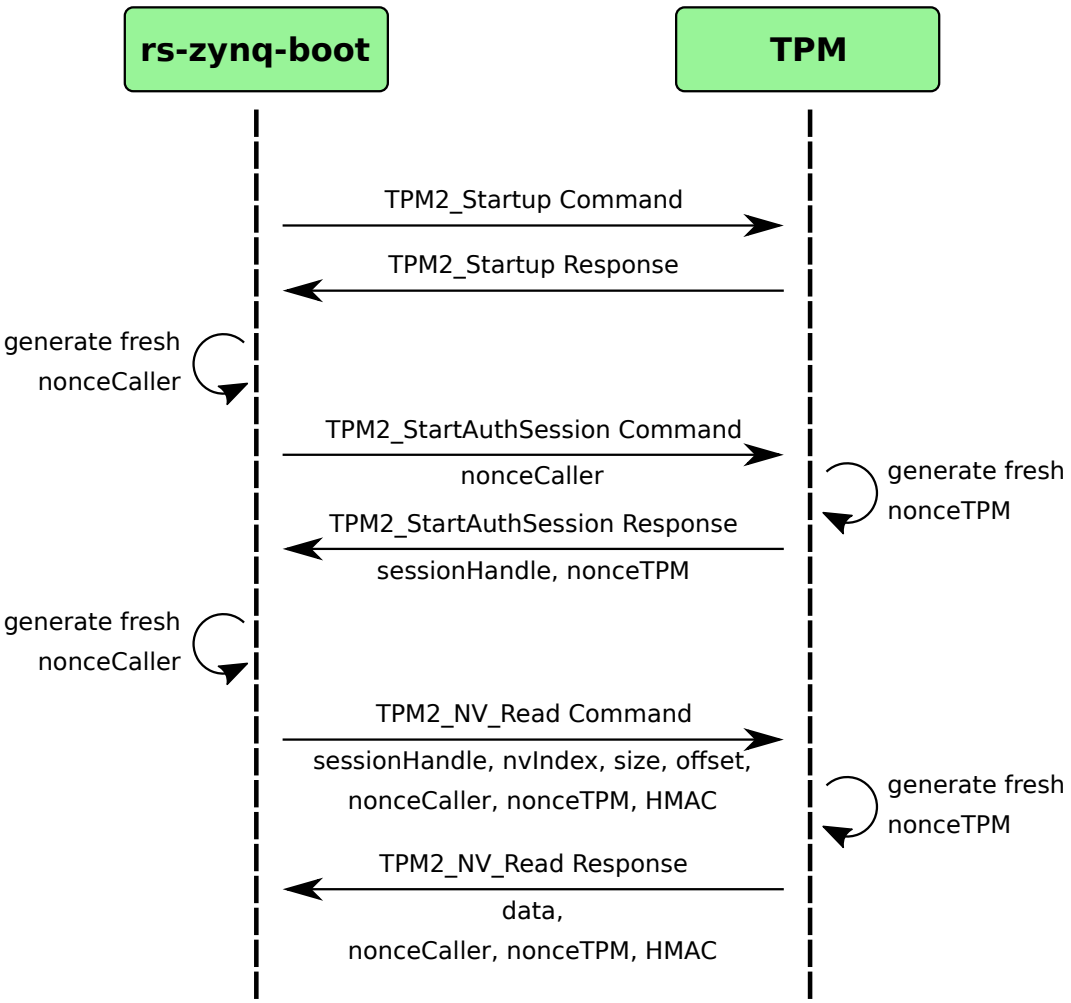

Figure 5.8: Sequence diagram showing the communication between the bootloader and TPM performed on every boot.

next stage. A fresh `nonceCaller` and the `nonceTPM` of the last command are included. The caller, *rs-zynq-boot*, calculates an HMAC over the command using the shared secret. The HMAC result is inserted into the command before sending.

After receiving the command, the TPM verifies the command HMAC with the help of the shared secret. On success, the TPM responds with the requested data. Again, a fresh nonce, the `nonceTPM`, is generated and packed into the response. The `nonceCaller` of the request is also part of the response. The TPM calculates an HMAC, over the response and inserts the result into the packet.

After transmitting the response back to the *rs-zynq-boot*, the HMAC is verified. If this is successful, the *rs-zynq-boot* accepts the answer. Otherwise, the boot process is terminated. In this way, the bootloader can ensure that the TPM knows the shared secret and therefore can trust the read data, representing the hash over the next stage.

5.4.2 Verification and Start of Next Stage

The third step in Figure 5.7 is the calculation of the cryptographic hash over the next stage. The next stage now already been loaded into the main memory at this point, done by the Xilinx FSBL. As a result, *rs-zynq-boot* simply calculates a SHA256 sum over this memory area. If the result equals the hash value received from the TPM, the execution can continue. Otherwise, the boot process aborts.

In the next step, the next stage is decrypted. The decryption is performed with the leakage-resilient scheme described in Section 5.1.1. The decrypted next stage is written back to the RAM, where it is ready for execution.

At this point, the integrity of the next stage is assured, the image is decrypted, and the next stage is ready to start. Since the processor cores inside the Zynq SoC have separated first-level instruction and data caches, the decryption of the firmware can cause unexpected behavior. While decryption, the firmware is handled as data, and therefore the data cache is used. For execution, the dedicated memory is however loaded to the instruction cache.

To cope with this situation, *rs-zynq-boot* flushes the caches before calling starting the next stage. In the last step, *rs-zynq-boot* finally executes the next stage with a branch instruction to the dedicated address. As a result, the replay-protected next stage runs and can fulfill the tasks of an Internet of Things (IoT) device.

5.4.3 Security Properties

The nonce used for the rekeying function is stored within the next stage image. The authenticity of the nonce is protected with the hash over the next stage. Therefore, a replay attack on the nonce is not possible. All cryptographic constants used for the leakage-resilient cryptography as shown in Figure 3.11, including the secret key and the constants C_0 and C_1, are stored inside *rs-zynq-boot* and therefore protected by the secure boot. Secure boot guarantees confidentiality and authenticity for these values.

As pointed out in Section 5.4.1, the generation of random numbers is required to get fresh nonces. A good random number generator is not only required inside the TPM. The SoC also requires a good random number generator. Otherwise, the secure communication between the SoC and the TPM is vulnerable. *rs-zynq-boot* fetches all required random number from the FPGA TRNG that was described in Section 4.2.

5.4.4 TPM2 Software Stack

A TPM software stack is a software library handling the interaction with a physical TPM. The TPM software stack provides an Application Programming Interface (API) to be integrated in arbitrary software implementations. Through API calls, different TPM commands, such as described in Section 4.3.2, can be executed.

There exist multiple open-source TPM software stack implementations, like from Tricca et al. [Tri+18] and Goldman [Gol17]. Neither of these is simply adaptable for bare-metal usage, the usage without an Operating System (OS), nor was written in Rust. For this reason, we implement a lightweight

TPM software stack using Rust. The implementation is encapsulated in the crate *rs-tpm2*, and we use it in the secure bootloader. The *rs-tpm2* crate is a library that is statically linked into *rs-zynq-boot*. In the end, the TPM stack is part of the final ELF file.

On the backend, the TPM software stack connects to the hardware TPM via SPI. To generalize the implementation for future usage, the SPI driver is not part of the TPM library crate. A callback function, following the footprint shown in Listing 5.2, is passed to the TPM software stack. This callback is responsible for sending and receiving raw commands, represented as byte sequences. Using this logic, the TPM software stack is used once inside the bare-metal bootloader with an SPI backend and second in Continuous Integration (CI) tests with a Transmission Control Protocol (TCP) backend for connecting to a TPM simulator.

Listing 5.2: Rust source code for the footprint of the callback function used in *rs-tpm2*.

```
pub fn transmit_handle(req: &CommandBuffer,
    resp: &mut CommandBuffer) -> Result<(), &'static str> {
    Err("not implemented")
}
```

The implemented TPM software stack *rs-tpm2* supports two types of authorization. It supports password-based and HMAC authorization as described in Section 4.3.1. While the boot progress, *rs-zynq-boot* only uses HMAC authorization, since password-based authorization is vulnerable to eavesdropping on the SPI bus.

The TPM software stack supports the following commands that allow an application to use the TPM as secure storage. The functionality of the different TPM commands is described in Section 4.3.2.

- `TPM2_Startup`
- `TPM2_StartAuthSession`
- `TPM2_Clear`
- `TPM2_GetRandom`
- `TPM2_NV_DefineSpace`
- `TPM2_NV_UndefineSpace`
- `TPM2_NV_Read`
- `TPM2_NV_Write`

Listing 5.3: Example using Rust TPM software stack.

```
1 let ts = Tpm2Stack::new(transmit_handle);
2 let mut data = [0u8; 32];
3 ts.tpm2_getrandom(&mut data).unwrap();
4 println!("data: {:?}", &data);
```

Example of Usage. Listing 5.3 shows the usage of the Rust TPM software stack on the example of the `TPM2_GetRandom` command. This command is used to fetch random numbers from the TRNG inside the TPM. Line 1 in Listing 5.3 creates a new TPM stack object with `transmit_handle`, the callback method for send and receive, specified. Line 2 creates a zero-initialized mutable array of 32 bytes. In line 3, `ts.tpm2_getrandom` detects the size of the argument and sends a request for that amount of bytes. In this example, 32 bytes are requested and line 4 prints the output to the console. When executing this code, the TPM stack triggers a request that is sent to the hardware TPM. Figure 5.9 shows the byte sequence sent to the TPM. The `tag`, `TPM_ST_NO_SESSIONS` in this case, indicates that no authorization session is used for this request. The field `commandSize` contains the number of bytes that are sent with this request. The remaining fields contain the command code and the number of requested bytes. Figure 5.10 shows one possible answer from the TPM. The TPM stack processes the response, then extracts the requested random bytes and in the end hands them back to the caller. The console output of the example is shown in Listing 5.4 and will alter for the next request since the response contains random bytes.

Listing 5.4: The decimal output of the example, produced by line 4 in Listing 5.3.

```
data: [228, 230, 125, 208, 127, 233, 30, 183,
49, 17, 86, 222, 116, 29, 181, 113, 17, 85, 149,
148, 175, 125, 219, 200, 68, 193, 142, 152,
79, 153, 219, 36]
```

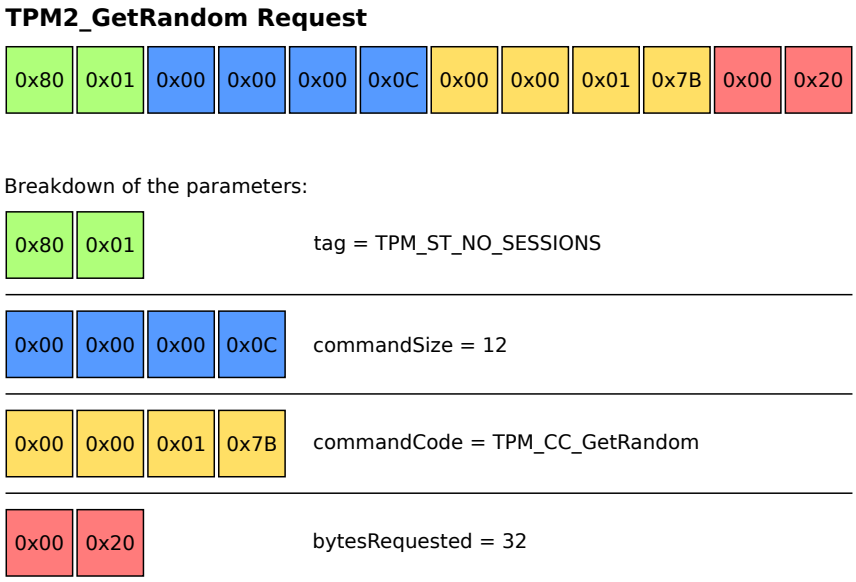

Figure 5.9: `TPM2_GetRandom` example request with description of all parameters, that are sent to the TPM. Every box represents one byte.

5.5 Provisioning

The provisioning process is an essential part of the production of IIoT devices. While provisioning, brand-new hardware is prepared for the tasks in a production environment, basic configurations are set, and software is installed. Both secure boot and our mechanisms to protect replay attacks add additional steps in the provisioning process. These steps require additional time in production, but these resources are well invested in security.

In the first step, secure boot has to be configured. Therefore, encryption and signature keys need to be generated. The decryption and verification keys are burned into so-called eFUSES on the device and are used later. This step is performed with tools offered by the SoC vendor.

The second step is to encrypt the firmware with the leakage-resilient scheme and compute the hash value. Therefore, a root key needs to be generated. We created a small Rust application, named *next-stage*, to encrypt-then-hash the firmware image. The dependency graph of all used crates is shown in Figure 5.11. This application uses the *rs-crypto* library crate, that is also used

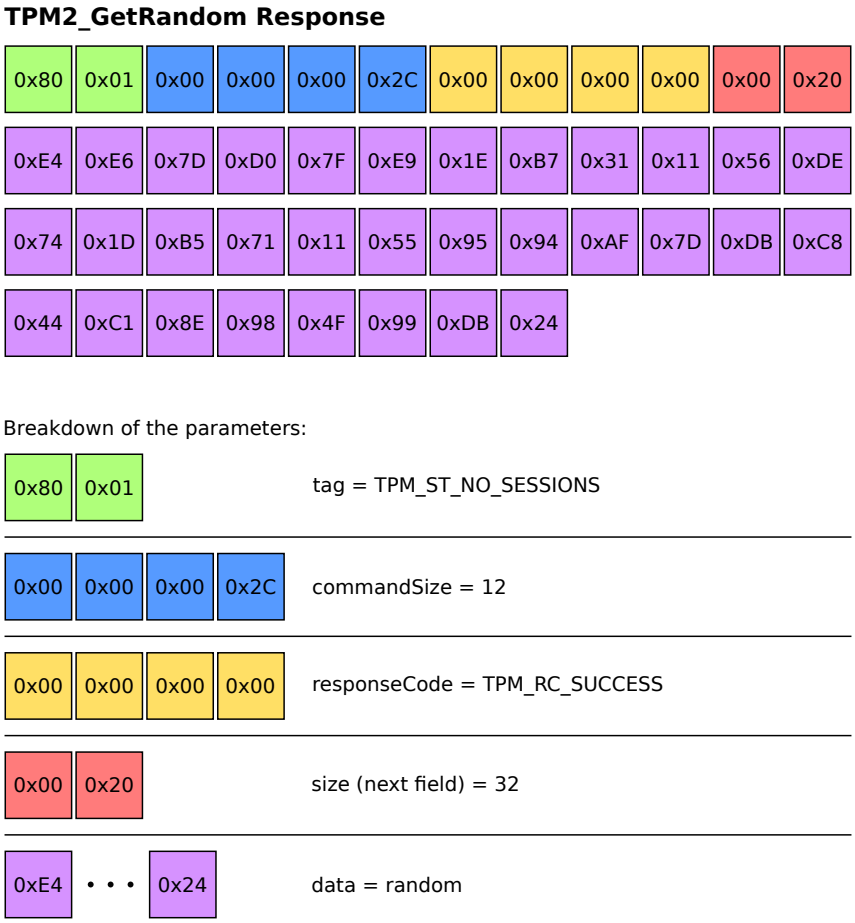

Figure 5.10: `TPM2_GetRandom` example response with description of all parameters, that are received from the TPM. Every box represents one byte.

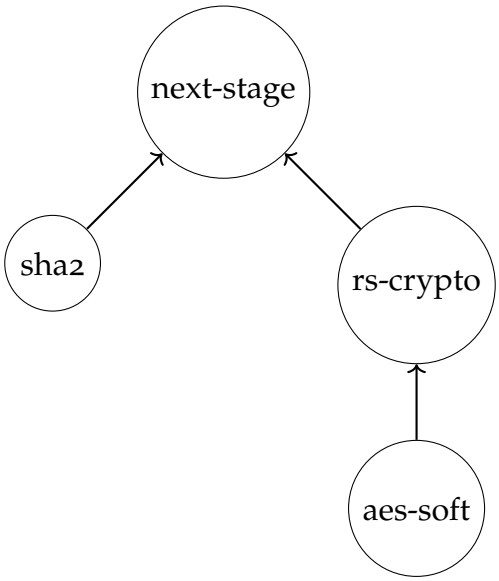

Figure 5.11: Dependency graph of *crates* used to prepare the next stage firmware.

within the bootloader. The application is designed for a host computer so that a developer can use it for shipping the firmware. *next-stage* reads a firmware binary file and processes it. It generates a fresh nonce and uses it for encrypting the plaintext firmware with the leakage-resilient scheme explained in Section 3.3.2. The SHA256 hash is computed over the encrypted firmware image and displayed to the user. The tool then saves nonce and the encrypted firmware image into a binary file. The generated root key is finally compiled into the *rs-zynq-boot*.

As the third step, the hash value of the encrypted firmware image needs to be stored inside the TPM. Therefore the *rs-zynq-boot* bootloader can be compiled for the provisioning usage. Hereby, the non-volatile variable needs to be defined. With this step, the shared secret is set, which will be later used for authorization. Parties that know the shared secret are permitted to read and write the non-volatile variable inside the TPM. Figure 5.12 shows the sequence diagram of the TPM communication in the provisioning step. This is only required once in the lifetime of the TPM used in the IoT device. In this sequence, the TPM is brought into the operational state with the *TPM2_Startup* command. Next, the non-volatile variable is defined with the

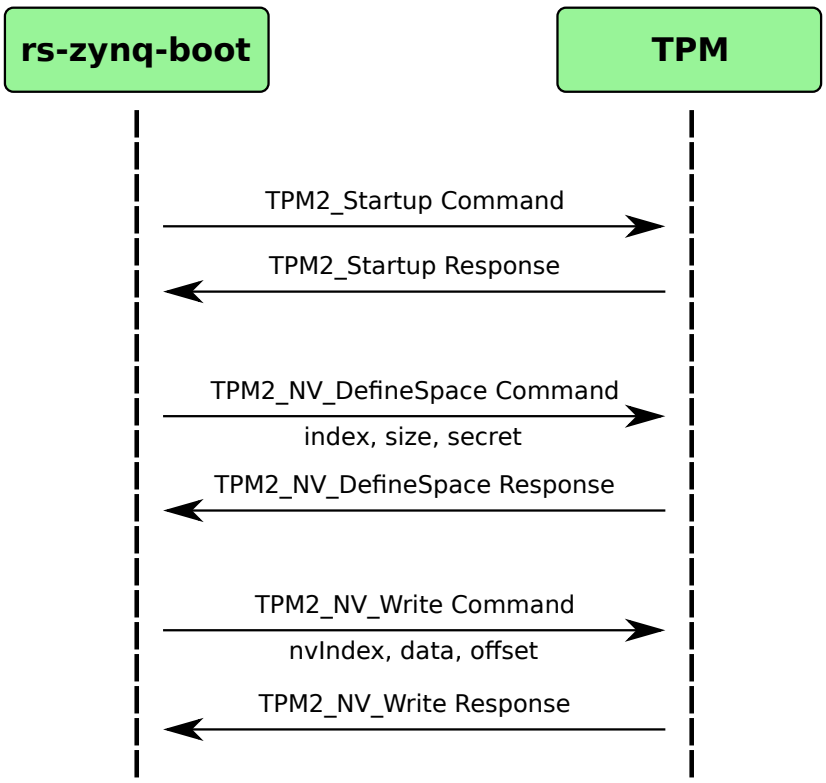

Figure 5.12: Sequence diagram showing the communication between the bootloader and TPM performed while provisioning.

TPM2_NV_DefineSpace command. With this command, the index, size, and secret are set. Moreover, the hash value over the encrypted firmware image is written to the non-volatile variable with the *TPM2_NV_write* command.

The fourth and last step is to prepare the storage of the device. We use a micro-SD-card in this project with the Zybo board. The bootloaders and bitstream file need to be encrypted and signed. Therefore, the SoC vendor, Xilinx in our case, offers a tool named `bootgen`. The tool receives all files and keys as input and creates an encrypted binary file. The resulting binary file can be stored on the SD-card. When putting all together, the IoT device starts with secure boot and our bootloader works with replay attack countermeasures.

5.6 Evaluation

We created a proof-of-concept firmware used as the next stage after *rs-zynq-boot*. The purpose of this implementation is to show the successful handover to the next stage. Based on this firmware, we perform different experiments trying to bypass the implemented security features.

Setup. The Zybo board is prepared as described in Section 5.5. The SD-card includes the packaged boot image, including Xilinx FSBL, the bitstream file, *rs-zynq-boot* and the firmware. The firmware is a small assembler implementation that prints out the string *"Next stage!"*. Listing 5.5 shows the source code of the implementation. After loading the register address of the Universal Asynchronous Receiver Transmitter (UART), the characters are written to the UART register. It is not necessary to poll on the transmit flag since the UART includes a FIFO. Because of that, it is possible to write up to 128 characters, which are buffered and transmitted afterwards. The proof-of-concept firmware is packed into the boot image.

System Boot. As already mentioned, the BootROM loads the Xilinx FSBL and further ensures authentication and decryption. After starting the FSBL, it first loads the FPGA configuration with the bitstream file. Second, it verifies, decrypts and loads the *rs-zynq-boot*. Third, the Xilinx FSBL loads the firmware into the main memory. *rs-zynq-boot* verifies and decrypts the firmware later. As the last step, the Xilinx FSBL hands over to the *rs-zynq-boot*. The *rs-zynq-boot* executes as described in Section 5.4. If the calculated hash over the encrypted firmware is correct, the next stage is decrypted and started. Otherwise, *rs-zynq-boot* reports an error and terminates the boot sequence. The following evaluation will cover both cases.

Listing 5.5: Source of the next stage prototype written in assembler.

```
1          movw r0, 0x1030
2          movt r0, 0xe000
3          mov r1, #0x4E
4          str r1, [r0]
5          mov r1, #0x65
6          str r1, [r0]
7          mov r1, #0x78
8          str r1, [r0]
9          mov r1, #0x74
10         str r1, [r0]
11         mov r1, #0x20
12         str r1, [r0]
13         mov r1, #0x73
14         str r1, [r0]
15         mov r1, #0x74
16         str r1, [r0]
17         mov r1, #0x61
18         str r1, [r0]
19         mov r1, #0x67
20         str r1, [r0]
21         mov r1, #0x65
22         str r1, [r0]
23         mov r1, #0x21
24         str r1, [r0]
25 EndL:   b EndL
```

Listing 5.6: Output of a successful boot process.

```
Welcome to RS-ZYNQ-BOOT!
Correct vendor ID
Connected to device, rid: 00 00 00 10
transmit_command: 80 01 00 00 00 0C 00 00 01 44 00 00
transmit_command resp: 80 01 00 00 00 0A 00 00 00 00
startup with rc: 00 00 00 00
trng random bytes:
    AD A9 94 52 0C DD F7 E0 69 B3 F0 06 1D ED A1 D9
    97 73 59 8B 03 8B 33 21 F3 B5 F1 D4 23 F6 D8 B0
transmit_command:
    80 01 00 00 00 3D 00 00 01 76 40 00 00 07 40 00
    00 07 00 20 AD A9 94 52 0C DD F7 E0 69 B3 F0 06
    1D ED A1 D9 97 73 59 8B 03 8B 33 21 F3 B5 F1 D4
    23 F6 D8 B0 00 00 00 00 0A 00 0B 00 0B
transmit_command resp:
    80 01 00 00 00 30 00 00 00 00 02 00 00 00 00 20
    06 7A 95 59 3B 36 7C CC E3 24 CA CC 8F 32 09 4F
    C4 34 EA D5 CD 11 18 7F 41 40 1E 36 B1 9A 85 C4
nonce_tpm:
    06 7A 95 59 3B 36 7C CC E3 24 CA CC 8F 32 09 4F
    C4 34 EA D5 CD 11 18 7F 41 40 1E 36 B1 9A 85 C4
trng random bytes:
    30 C0 1D 40 DA C6 5F 65 17 F0 FC 7F D2 3E F7 12
    6A 05 55 7D 3E EE 6A 28 2B 80 9F 74 5B 1C 2B 68
transmit_command:
    80 02 00 00 00 63 00 00 01 4E 01 50 00 15 01 50
    00 15 00 00 00 49 02 00 00 00 00 20 30 C0 1D 40
    DA C6 5F 65 17 F0 FC 7F D2 3E F7 12 6A 05 55 7D
    3E EE 6A 28 2B 80 9F 74 5B 1C 2B 68 01 00 20 2F
    BF CC AB 17 97 EF 8C 54 BC 0F 7E 3C 4C FD 31 19
    A6 FF 16 06 67 40 AA 28 E5 76 BE 72 21 86 B6 00
    20 00 00
transmit_command resp:
    80 02 00 00 00 75 00 00 00 00 00 00 00 22 00 20
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
```

```
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
    00 20 7D 98 F2 FE 73 F7 C6 49 3F DA 50 26 67 AE
    E4 08 D7 7C 68 80 60 70 75 EE B3 DF 51 26 99 6A
    00 B7 01 00 20 CA DC C2 81 55 F4 76 C9 A8 4F A9
    FB 12 2A E9 97 DC 1F 2B F0 C9 D1 EE 19 2C F4 95
    90 04 51 CD 7D
read_bytes:
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
size: 00 00 00 80
hash from TPM:
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
hash calc:
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
next stage ciphertext:
    5D BC 02 59 E7 A3 D4 3C A0 D3 ED B5 3F 2D 2F CC
    B5 4A 4A BF C8 9F F2 DD E2 0F 08 BA A6 9C CE A8
    A2 94 63 AB D5 7E C6 7A DF 6A C4 76 BF 3A FC 8A
    18 2D 41 14 AB 34 4B 30 AC 79 2F 77 0B 8D 99 0C
    9B C7 30 67 D7 AC 19 2D AD EB 99 AC B0 10 53 E4
    19 7F 2F BA 65 5B EC 21 C0 1F 8B BB 8C 0A 10 00
    A9 D7 57 19 E7 CC BC BE 6A 0C 5E C6 97 B2 55 D0
next stage plaintext:
    30 00 01 E3 00 00 4E E3 4E 10 A0 E3 00 10 80 E5
    65 10 A0 E3 00 10 80 E5 78 10 A0 E3 00 10 80 E5
    74 10 A0 E3 00 10 80 E5 20 10 A0 E3 00 10 80 E5
    73 10 A0 E3 00 10 80 E5 74 10 A0 E3 00 10 80 E5
    61 10 A0 E3 00 10 80 E5 67 10 A0 E3 00 10 80 E5
    65 10 A0 E3 00 10 80 E5 21 10 A0 E3 00 10 80 E5
    FE FF FF EA 00 00 00 00 00 00 00 00 00 00 00 00
handover to: 1C E0 00 14
Next stage!
```

Successful Boot. Listing 5.6 shows the output on the serial interface of a successful boot process. Line 4 shows the `TPM2_Startup` command sent to the TPM that is successfully executed with return code 0, as shown in line 6. Next, 32 random bytes are drawn from the hardware TRNG, shown in line 7, that are included into the `TPM2_StartAuthSession` command in line 8. The response, shown in line 9, includes the `nonceTPM`. The `nonceTPM` and a freshly drawn `nonceCaller` are used for the `TPM2_NV_Read` command in line 12. This command also includes a valid HMAC. The response, shown in line 13, includes an HMAC and the requested data. The bootloader successfully verifies the HMAC and proceeds with the boot progress. Next, the hash over the encrypted firmware image is calculated and displayed in line 17. The calculated hash equals to the hash read from the TPM, shown in line 16. Through this property, a replay attack is impossible. Line 18 shows the encrypted firmware image and line 19 the decrypted version. This output is only for debug purpose and has to be removed from a production device. Line 20 indicates the entry point memory address of the firmware, where the bootloader hands over to. Finally, line 21 shows the output of the next stage firmware.

Listing 5.7: Output of a boot process with a replay attack.

```
1  Welcome to RS-ZYNQ-BOOT!
2  Correct vendor ID
3  Connected to device, rid: 00 00 00 10
4  transmit_command: 80 01 00 00 00 0C 00 00 01 44 00 00
5  transmit_command resp: 80 01 00 00 00 0A 00 00 00 00
6  startup with rc: 00 00 00 00
7  trng random bytes:
8      7F 43 6E 27 C2 AB 01 33 26 6B 5A 4E E6 E0 4F 73
9      FC 97 3E 00 7B FD 69 E7 F6 18 40 71 E4 F8 1D 80
10 transmit_command:
11     80 01 00 00 00 3D 00 00 01 76 40 00 00 07 40 00
12     00 07 00 20 7F 43 6E 27 C2 AB 01 33 26 6B 5A 4E
13     E6 E0 4F 73 FC 97 3E 00 7B FD 69 E7 F6 18 40 71
14     E4 F8 1D 80 00 00 00 00 0A 00 0B 00 0B
15 transmit_command resp:
16     80 01 00 00 00 30 00 00 00 00 02 00 00 00 00 20
17     51 F7 99 0D 18 25 29 5E 2C 65 4E AD B8 D7 6E 81
```

```
    E9 67 D1 7E 19 9E ED 49 75 7E F7 BA D0 C9 46 18
nonce_tpm:
    51 F7 99 0D 18 25 29 5E 2C 65 4E AD B8 D7 6E 81
    E9 67 D1 7E 19 9E ED 49 75 7E F7 BA D0 C9 46 18
trng random bytes:
    6E 2C CA 3B 0F F0 2E 2C D0 7A DD 5E 6A 45 F7 AE
    BA 5B AA CD 57 1C 15 EE 1F 77 2B 62 01 4D 53 C5
transmit_command:
    80 02 00 00 00 63 00 00 01 4E 01 50 00 15 01 50
    00 15 00 00 00 49 02 00 00 00 00 20 6E 2C CA 3B
    0F F0 2E 2C D0 7A DD 5E 6A 45 F7 AE BA 5B AA CD
    57 1C 15 EE 1F 77 2B 62 01 4D 53 C5 01 00 20 F4
    48 18 DE AC 49 89 7F 88 B7 E3 16 E9 06 2B 3D 00
    9A 4A 88 1B 2A F6 DC EB 59 32 90 F0 15 6F DF 00
    20 00 00
transmit_command resp:
    80 02 00 00 00 75 00 00 00 00 00 00 00 22 00 20
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
    00 20 AA C0 A4 B3 C8 A5 CC 63 17 17 86 7F AB 03
    0A 23 64 E1 DC E2 D2 2B 95 10 23 58 E9 2C 24 51
    97 9A 01 00 20 44 7A 1D C3 D9 DC B2 1E 17 39 28
    AF DC 08 18 5F BE 0C CB 79 69 3E CE 72 2A 10 08
    2B 87 AA CA DA
read_bytes:
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
size: 00 00 00 80
hash from TPM:
    E6 1D CA 1E F4 D8 6F 40 33 61 5C 9A 1C 90 E9 03
    3B EB 3E B3 93 6E 6A BA D2 F3 90 0D 92 77 6F 78
hash calc:
    3D 47 15 B4 DC 94 04 E5 EF E6 AF AA D0 80 23 03
    71 95 C2 0A B7 0E 40 98 1A 11 E4 7C CB 76 29 2F
wrong hash
```

Replay Attack. Listing 5.7 shows a replay attack applied on the firmware, where an old firmware image is used. Listing 5.7 equals Listing 5.6 until line 16 with the exception of the random values. Next, the bootloader calculates the hash over the encrypted next stage firmware. The calculated hash, shown in line 17 of Listing 5.7, does not match the value from the secure storage, and therefore, a replay attack is detected. The bootloader terminates the boot progress and prompts an error.

5.7 Future Work

We point out that secure storage is essential for secure systems. We hence propose that future SoC designs include secure storage. Future projects would benefit from such storage as it would avoid the need for a TPM and a TRNG. Since secure network connections also rely on good random numbers, including a TRNG into a future SoC design is desirable as well. These features will however not be available by default in the next years, because the product life cycle of silicon chips is slow.

Moreover, we propose integrating side-channel protected cryptographic functions in upcoming SoC designs. With a protected hardware implementation of the AES function and the HMAC scheme, we can further improve the side-channel security of the implemented prototype. An evaluation of side-channel attacks on the current implementations is future work and will determine if protected implementations are necessary for the prototype.

6 Conclusion

The Industrial Internet of Things (IIoT) is getting increasingly popular today. However, these devices need to be secured against various attacks to protect the functionality of an IIoT ecosystem. For instance, today's computers often suffer from memory safety attacks. Hereby, programming errors introduce problems that are exploited by attackers. While security updates solve these problems, the rollback of a security update is a serious threat as well. This so-called replay attack is even possible for encrypted boot and is a severe problem for IIoT devices. Further, an attacker can use side-channel attacks to attack an IIoT device to reveal secrets stored inside the IIoT device. Consequently, designs of IIoT devices need to encounter attacks on memory safety, replay attacks, and side-channel attacks.

In this thesis, we developed a concept to secure IIoT devices against aforementioned attacks. With a practical implementation, we verified the feasibility of our concept. The proof-of-concept implementation used a Zybo board as a platform. The Zybo board comes with a Xilinx Zynq System on Chip (SoC), which supports secure boot. We used this feature as the basis for our work and enhanced it with protection against replay attacks of the firmware. For this reason, we implemented a secure bootloader. To protect against replay attacks, we integrated a method to verify the current firmware version into the bootloader. This method consists of calculating a cryptographic hash of the firmware image and comparing it with a reference value. The respective reference value is stored in a modifiable secure storage. Since the Zynq SoC does not offer such secure storage, we added another component, a Trusted Platform Module (TPM).

A TPM is a security element that offers, among other security features, secure storage. We used the secure storage of a TPM in our design. Because we face physical attackers, we also had to secure the communication between

the TPM and the SoC. In particular, we used the Hash-based Message Authentication Code (HMAC) authorization offered by the TPM. Based on a secret shared between both parties and freshness in the protocol, the integrity of the transferred data is ensured. The freshness is caused by fresh numbers only used once (nonces) used in every telegram sent between the SoC and the TPM. The generation of fresh nonces requires good random number generators for both communication parties. The TPM comes with a hardware True Random Number Generator (TRNG), but the Xilinx Zynq SoC does not offer such a feature. We hence integrated a hardware TRNG into the Field Programmable Gate Array (FPGA) inside the Xilinx Zynq SoC.

Since an IIoT device is exposed to physical attackers, it is also in danger of side-channel attacks. A bootloader can leak encryption keys and therefore break the chain of trust. In this way, the attacker can read the firmware plaintext or learn the cryptographic key to decrypt the firmware. We hardened our bootloader implementation to ensure the confidentiality of the executed firmware image in the presence of side-channel attacks. Namely, we used an encrypt-then-hash scheme for the firmware encryption. Using leakage-resilient cryptography, we encrypted the firmware that is started with the implemented bootloader. For this, we implemented a leakage-resilient mode of operation that uses frequent rekeying. Hereby, one key is never used more than twice, which limits the possibilities for a side-channel attacker. We saved the cryptographic hash of the encrypted firmware image in the secure storage. While the system boots, the hash of the ciphertext is calculated and compared with the stored hash before decryption. This step merged with the replay protection and prevented the application of differential side-channel attacks. These two methods together mitigate side-channel attacks on the implemented bootloader.

Further, memory safety issues are problems for a bootloader implementation. An attacker can exploit these issues and break the secure boot by using code injection or code reuse attacks. To ensure error-free memory handling in our implementation, we implemented the bootloader prototype in the programming language Rust. Rust is a modern system programming language that guarantees memory safety based on a so-called ownership model. It allows implementing bare-metal applications that execute on the bare hardware without an Operating System (OS) running. We implemented

the bootloader as a bare-metal application. Our implementation includes a TPM software stack, the low-level protocol to communicate with the hardware TPM, and leakage-resilient cryptography. We showed that all these components together follow our concept and successfully verify, decrypt and start the protected firmware.

With the prototype, we were able to reach the goals of this thesis. The implemented bootloader *rs-zynq-boot* prevents replay attacks, and is protected against memory safety and side-channel attacks. In the end, we showed that it is possible to increase the security of an IIoT device with off-the-shelf components. As a consequence, we want to see this concept in production to make the world more secure.

Appendix

Bibliography

[ABE05] Martin Abadi, Mihai Budiu, and Úlfar Erlingsson. "Control-Flow Integrity." In: *ACM Conference on Computer and Communication Security (CCS)*. Alexandria, VA: Nov. 2005, pp. 340–353. URL: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/ (cit. on p. 14).

[All+18] Elie Noumon Allini et al. "Optimization of the PLL configuration in a PLL-based TRNG design." In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. 2018, pp. 1265–1270. DOI: 10.23919/DATE.2018.8342209. URL: https://doi.org/10.23919/DATE.2018.8342209 (cit. on p. 42).

[Apa18a] Jorge Aparicio. *Embedded development on stable*. https://github.com/rust-lang-nursery/embedded-wg/issues/42. 2018 (cit. on p. 17).

[Apa18b] Jorge Aparicio. *[Request for Experiment] Sysroot building functionality*. https://github.com/rust-lang/cargo/issues/4959. 2018 (cit. on p. 17).

[Bas+10] Lawrence E. Bassham III et al. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. rep. Gaithersburg, MD, United States, 2010 (cit. on p. 42).

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication." In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1. URL: https://doi.org/10.1007/3-540-68697-5_1 (cit. on p. 36).

[BCK97] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. RFC Editor, Feb. 1997. URL: https://www.rfc-editor.org/rfc/rfc2104.txt (cit. on p. 37).

[Ber+10] Guido Bertoni et al. "Sponge-Based Pseudo-Random Number Generators." In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. 2010, pp. 33–47. DOI: 10.1007/978-3-642-15031-9_3. URL: https://doi.org/10.1007/978-3-642-15031-9_3 (cit. on p. 40).

[Ble+11] Tyler K. Bletsch et al. "Jump-oriented programming: a new class of code-reuse attack." In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*. 2011, pp. 30–40. DOI: 10.1145/1966913.1966919. URL: http://doi.acm.org/10.1145/1966913.1966919 (cit. on p. 15).

[Car+15] Nicholas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity." In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015, pp. 161–176. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini (cit. on p. 10).

[CFR10] Jean-Christophe Courrège, Benoit Feix, and Mylène Roussellet. "Simple Power Analysis on Exponentiation Revisited." In: *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*. 2010, pp. 65–79. DOI: 10.1007/978-3-642-12510-2_6. URL: https://doi.org/10.1007/978-3-642-12510-2_6 (cit. on p. 26).

[Cla17] Lin Clark. *Inside a super fast CSS engine: Quantum CSS*. https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/. 2017 (cit. on p. 16).

[Cor18] The MITRE Corporation. *Common Vulnerabilities and Exposures*. 2018. URL: https://cve.mitre.org/ (visited on 03/28/2018) (cit. on pp. 3, 6, 7).

[Dob+17] Christoph Dobraunig et al. "ISAP - Towards Side-Channel Secure Authenticated Encryption." In: *IACR Trans. Symmetric Cryptol.* 2017.1 (2017), pp. 80–105. DOI: 10.13154/tosc.v2017.i1.80-105. URL: https://doi.org/10.13154/tosc.v2017.i1.80-105 (cit. on p. 5).

[DR98] Joan Daemen and Vincent Rijmen. "The Block Cipher Rijndael." In: *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*. 1998, pp. 277–284. DOI: 10.1007/10721064_26. URL: https://doi.org/10.1007/10721064_26 (cit. on p. 29).

[Fou18] Eclipse Foundation. *Eclipse IDE*. https://www.eclipse.org/eclipse/. 2018 (cit. on p. 58).

[Gal18] Gallopsled. *Pwntools*. https://github.com/Gallopsled/pwntools. 2018 (cit. on p. 12).

[Gol17] Kenneth Goldman. *IBM's TPM 2.0 TSS*. https://sourceforge.net/projects/ibmtpm20tss/. 2017 (cit. on p. 67).

[Gro14] Trusted Computing Group. *Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.38*. https://trustedcomputinggroup.org/tpm-library-specification/. 2014 (cit. on p. 43).

[Gro17] Trusted Computing Group. *TCG PC Client Platform TPM Profile (PTP) Specification, Family "2.0", Level 00, Revision 01.03 v20*. https://trustedcomputinggroup.org/tpm-library-specification/. 2017 (cit. on pp. 48, 49).

[Hen+12] Nadia Heninger et al. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices." In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 2012, pp. 205–220. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger (cit. on p. 41).

[Inc18] NewAE Technology Inc. *ChipWhisperer*. https://http://newae.com/tools/chipwhisperer/. 2018 (cit. on p. 27).

[Jac+17] Nisha Jacob et al. "How to Break Secure Boot on FPGA SoCs Through Malicious Hardware." In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017, pp. 425–442. DOI: 10.1007/978-3-319-66787-4_21. URL: https://doi.org/10.1007/978-3-319-66787-4_21 (cit. on p. 21).

[Jeo+17] Yuseok Jeon et al. "HexType: Efficient Detection of Type Confusion Errors for C++." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017, pp. 2373–2387. DOI: 10.1145/3133956.3134062. URL: http://doi.acm.org/10.1145/3133956.3134062 (cit. on p. 10).

[Kau07] Bernhard Kauer. "OSLO: Improving the Security of Trusted Computing." In: *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. 2007. URL: https://www.usenix.org/conference/16th-usenix-security-symposium/oslo-improving-security-trusted-computing (cit. on p. 43).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. URL: https://doi.org/10.1007/3-540-48405-1_25 (cit. on p. 26).

[Kön08] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES." In: *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*. 2008, pp. 187–202. DOI: 10.1007/978-3-540-79263-5_12. URL: https://doi.org/10.1007/978-3-540-79263-5_12 (cit. on p. 54).

[Lan+15] Bingchen Lan et al. "Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses." In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. 2015, pp. 190–197. DOI: 10.1109/Trustcom.2015.374.

URL: https://doi.org/10.1109/Trustcom.2015.374 (cit. on p. 15).

[Lev98] Elias Levy. "Smashing the stack for fun and profit." In: *Phrack Magazine* 49(14) (1998) (cit. on p. 7).

[LSP04] Kerstin Lemke, Kai Schramm, and Christof Paar. "DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction." In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. 2004, pp. 205–219. DOI: 10.1007/978-3-540-28632-5_15. URL: https://doi.org/10.1007/978-3-540-28632-5_15 (cit. on p. 37).

[Mab16] Rémi Mabon. "Sigreturn oriented programming is a real threat." In: *46. Jahrestagung der Gesellschaft für Informatik, Informatik 2016, 26.-30. September 2016, Klagenfurt, Österreich*. 2016, pp. 2077–2088. URL: https://dl.gi.de/20.500.12116/1104 (cit. on p. 15).

[MK17] Markus Muellner and Markus Kammerstetter. *34C3 - Uncovering vulnerabilities in Hoermann BiSecur*. Youtube. 2017. URL: https://www.youtube.com/watch?v=vPhaTZK2e8w (cit. on p. 21).

[Moz17] Mozilla. *Servo Browser Engine*. https://servo.org/. 2017 (cit. on p. 16).

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. "Leakage-Resilient Authentication and Encryption from Symmetric Cryptographic Primitives." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015, pp. 96–108. DOI: 10.1145/2810103.2813626. URL: http://doi.acm.org/10.1145/2810103.2813626 (cit. on pp. 5, 33).

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342 (cit. on p. 26).

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. "Bitslice Implementation of AES." In: *Cryptology and Network Security, 5th International Conference, CANS 2006, Suzhou, China, December 8-10, 2006, Proceedings*. 2006, pp. 203–212. DOI: 10.1007/11935070_14. URL: https://doi.org/10.1007/11935070_14 (cit. on p. 54).

[Sal17] Jonathan Salwan. *ROPgadget*. https://github.com/JonathanSalwan/ROPgadget. 2017 (cit. on p. 12).

[Sch+15] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 745–762. DOI: 10.1109/SP.2015.51. URL: https://doi.org/10.1109/SP.2015.51 (cit. on p. 15).

[Sch18] Sascha Schirra. *Ropper*. https://github.com/sashs/Ropper. 2018 (cit. on p. 12).

[Sko16] Sergei Skorobogatov. "The bumpy road towards iPhone 5c NAND mirroring." In: *CoRR* abs/1609.04327 (2016). arXiv: 1609.04327. URL: http://arxiv.org/abs/1609.04327 (cit. on p. 22).

[Sta+10] François-Xavier Standaert et al. "Leakage Resilient Cryptography in Practice." In: *Towards Hardware-Intrinsic Security - Foundations and Practice*. 2010, pp. 99–134. DOI: 10.1007/978-3-642-14452-3_5. URL: https://doi.org/10.1007/978-3-642-14452-3_5 (cit. on p. 34).

[Tea18a] Rust Core Team. *The Rust community's crate registry*. 2018. URL: https://crates.io/ (cit. on p. 60).

[Tea18b] RustCrypto Team. *block-ciphers*. https://github.com/RustCrypto/block-ciphers. 2018 (cit. on p. 54).

[Tea18c] The Rust Core Team. *Rust's 2018 roadmap*. https://blog.rust-lang.org/2018/03/12/roadmap.html. 2018 (cit. on p. 17).

[Tra+11] Minh Tran et al. "On the Expressiveness of Return-into-libc Attacks." In: *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*. 2011, pp. 121–141. DOI: 10.1007/978-3-642-23644-0_7. URL: https://doi.org/10.1007/978-3-642-23644-0_7 (cit. on p. 12).

[Tri+18] Philip Tricca et al. *TCG TPM2 Software Stack (TSS2)*. https://github.com/tpm2-software/tpm2-tss. 2018 (cit. on p. 67).

[vbe16] vbendeb. *tpm2_server*. https://github.com/vbendeb/tpm2_server. 2016 (cit. on p. 49).

[WD11] Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus." In: *Public Key Infrastructures, Services and Applications - 8th European Workshop, EuroPKI 2011, Leuven, Belgium, September 15-16, 2011, Revised Selected Papers*. 2011, pp. 176–193. DOI: 10.1007/978-3-642-29804-2_12. URL: https://doi.org/10.1007/978-3-642-29804-2_12 (cit. on p. 43).

[Wer+17] Mario Werner et al. "Transparent memory encryption and authentication." In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–6. DOI: 10.23919/FPL.2017.8056797. URL: https://doi.org/10.23919/FPL.2017.8056797 (cit. on p. 21).

[Wil16] Huon Wilson. *Myths and Legends about Integer Overflow in Rust*. http://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/. 2016 (cit. on p. 16).