Pascal Nasahl, BSc

Design and Implementation of a Trusted Execution Environment with Secure I/O for RISC-V

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Robert Schilling, Mario Werner

Assessor

Stefan Mangard

Institute of Applied Information Processing and Communication

Graz, October 2019

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

Acknowledgements

This thesis would not have been possible without the help of my supervisors Robert and Mario. Special thanks to you and many other colleagues at IAIK for the countless discussions inspiring me for my work.

Furthermore, I want to thank Stefan for introducing me into the world of information security and giving me innumerable opportunities which always allow me to develop my passion.

Many thanks to all my friends in Graz for the experiences we shared together. Thank you for joining and enriching my path throughout my studies.

To my parents for providing me with all opportunities in life and triggering my curiosity from an early age on. Last but foremost, I would like to thank my partner Theresa for her unconditional love, support, and patience during the last years.

Abstract

Computing systems, ranging from small gadgets like smartphones up to complex supercomputers, are getting omnipresent in our lives. By integrating such electronic devices into our daily routines and linking available information together, people's lives tend to get more comfortable. However, the technological progress does not exclusively offer advantages, but also disadvantages: when, for instance, placing such devices into sensitive places like our homes, especially privacy concerns arise. Furthermore, due to their massive distribution, electronic devices connected to the internet are a profitable target for malicious adversaries. For this reason, manufacturers are integrating countermeasures against various kinds of attacks into their systems.

Since the complexity of these computing architectures is growing steadily, providing an extensive protection against a broad variety of attack scenarios on the overall system level is hard to achieve. To address this challenge, many device producers are offering so-called trusted execution environments. These environments are isolated, safe spaces embedded into the potential insecure system. Due to isolation properties of these containers, secure execution of sensitive code can be guaranteed to some extent.

When developing the security concept of a system, also the protection of input and output interfaces have to be considered. A user entering secret information like passwords or banking credentials wants to keep this information secret. For this reason, the concept of secure input/output devices are introduced and they are usually combined with trusted execution environments.

In this thesis, we present a novel trusted execution environment scheme. We are enhancing the concept of trusted execution environments by combining them with trusted I/O paths and introducing a configurable security monitor. Our proposed design allows developers to flexibly use the secure environment and build powerful systems. Furthermore, we are offering a generic mechanism for creating a secure I/O path between the trusted execution environment and peripherals. By using dedicated hardware architectures with build-in countermeasures, our scheme provides protection against several physical attacks. To demonstrate the feasibility of our concept, we integrated the features into a RISC-V platform and provide an FPGA prototype. Furthermore, we demonstrate the practicability of our architecture using a secure boot scenario.

Keywords: information security, trusted execution environments, secure execution, secure I/O, RISC-V, FPGA, secure boot

Kurzfassung

Computersysteme, angefangen von kleinen, elektronischen Helfern wie Smartphones bis zu großen, komplexen Supercomputern, nehmen immer mehr Platz in unserem Leben ein. Durch die Integration von elektronischen Geräten in unseren Alltag sowie durch die Verknüpfung von Informationen können solche Systeme das Leben der Benutzer erleichtern. Allerdings werden die Gefahren, welche von Computersystemen ausgehen können, von vielen ignoriert oder einfach gebilligt. Insbesondere im Bereich des Datenschutzes bedarf es noch einigen Verbesserungen. Zudem stellen diese Geräte durch ihre massive Verbreitung ein lukratives Angriffsziel dar. Hersteller versuchen, dem entgegenzuwirken, indem sie Gegenmaßnahmen in die Systeme integrieren.

Da die Komplexität moderner Rechnerarchitekturen aber immer mehr zunimmt, ist es nahezu unmöglich, einen allumfassenden Schutz für das gesamte System zu gewährleisten. Deswegen bieten einige Hersteller von Computersystemen isolierte, vertrauenswürdige Laufzeitumgebungen an, welche in der potenziell unsicheren Umgebung integriert werden. Durch die Isolationseigenschaften dieser Container kann eine sichere Ausführung von sensiblem Code bis zu einem gewissen Grad garantiert werden.

Ein weiterer Aspekt der Absicherung eines Computersystems gegenüber Attacken ausgehend von einem böswilligen Angreifer sind die Ein- und Ausgaben eines Systems. Benutzer, welche geheime Informationen wie Passwörter und Bankdaten eingeben, möchten sichergehen, dass diese Daten auch geheim bleiben. Aus diesem Grund bieten verschiedene Hersteller Konzepte zur sicheren Handhabung von Ein-/Ausgabe Geräten an und kombinieren diese mit vertrauenswürdigen Laufzeitumgebungen.

In dieser Arbeit stellen wir ein neuartiges Konzept für eine vertrauenswürdige Laufzeitumgebung vor. Um bessere Sicherheitsgarantien zu bieten, kombinieren wir eine vertrauenswürdige Laufzeitumgebung mit einem sicheren E/A Pfad und einem Hardware-Sicherheitsmodul. Dieses Konzept erlaubt dem Entwickler eine flexible Benutzung dieses Systems. Weiters wird, durch eine geschickte Auswahl der Architektur, ein Schutz gegenüber physikalischen Angriffen geboten. Die Funktionsweise des Gesamtkonzepts wird anhand einer RISC-V Plattform und einem FPGA Prototyp demonstriert. Zudem zeigen wir die Praktikabilität unserer Architektur anhand eines sicheren Systemstarts auf.

Stichwörter: Informationssicherheit, Vertrauenswürdige Laufzeitumgebung, Sichere Ausführung, Sichere E/A, RISC-V, FPGA, Sicheres Hochfahren

Contents

List of Figures

Chapter 1

Introduction

Since the publication of Weiser's pioneer work [106] about ubiquitous computing in 1993, many of the proposed futuristic concepts were introduced in the last decades. Nowadays, more and more devices of our daily lives are linked together and form the Internet of Things. Previous years have shown that technology deeply integrated into everyday items had an impact on people's lives. Communicating with others scattered all over the globe and consuming different information channels is easier than never before. However, despite all advantages of connecting and integrating electronic devices into our daily routines, the technological progress also demands great challenges. One risk often ignored by people heavily using digital systems is the loss of privacy. Since these devices are deeply integrated into people's lives and users voluntarily share sensitive information with these systems, companies and governments easily can monitor dozens of people. Connecting devices and mounting them into safety-critical places like cars also raises the question of the overall security of these technological gadgets. As time-to-market and production costs are the dominating device manufacturing factors in our competitive world, system security is not always the top priority of many product manufacturers. This statement is supported by various, famous security breaches published recently in the news [31, 33, 67]. One remarkable example of a product with an insufficient security concept connected to the internet was the Jeep Cherokee. Security analysts in 2015 showed how a security issue easily can transfer to a safety issue by attacking the network interface of a device [67]. As such attacks could lead to life-threatening situations and the reputation of companies is in danger, corporations start to heavily invest in information security. However, providing protection against attacks from any adversary is a challenging task. Whereas development engineers need to close any potential security flaw, an attacker only needs to find one single weakness to exploit the system. As the complexity of such devices is steadily growing, this is a demanding problem.

Software running on a device usually is executed on top of an operating system providing an abstraction between hardware and software. Additionally, an operating system usually provides a rich set of features like system timers, drivers, and a network stack. However, having a rich set of features also have some drawbacks. Due to the immense codebase of such systems, protecting security-sensitive code even gets more challenging.

Downsizing the codebase by reducing the feature set of software often is not feasible, as users usually do not want to trade off functionality against security. To address this issue, one possible mitigation technique introduced in the past are trusted execution environment (TEE). In this concept, a safe, isolated space within the overall system is generated. Security critical code, secret information, or other assets are shifted into this

space and executed isolatedly. When using TEE solutions, still an operating system offering a rich set of features can be used. This approach is already widely used in many mobile devices, personal computers, and servers.
In the safe trusted execution environments, often secret information like banking credentials or passwords are processed. As these assets usually are entered by the user using some form of input device, a TEE solution should also offer the protection of peripherals. To solve this problem, trusted I/O paths between the peripherals and the execution environment are created by using hardware features of the system.
TEE schemes offered by vendors often only provide limited flexibility [28]. In many trusted execution environment solutions, developers cannot mount their own applications directly to the isolated space. Instead, small services are deployed into the environment by the device manufacturer and software executed by users can use these services. Furthermore, only few TEE schemes offer sophisticated solutions for building a secure path between a trusted execution environment and a peripheral. In one solution, which is introduced in Chapter 3, only legacy peripherals are supported natively. When analyzing commercial TEE implementations, different protection level against attacks can be guaranteed. All isolated execution environments from the big chip manufacturers exclude physical attacks from their threat model.

1.1 Contribution

In this thesis, we are proposing a novel trusted execution scheme with support for secure I/O interaction. To enhance the overall system security, we embed a dedicated security coprocessor into the system architecture and use this subsystem as a trusted execution environment. In our scheme, we use a security hardened architecture to protect against physical attacks like fault attacks. Furthermore, we create a trusted I/O path between the processing units and the peripherals by integrating hardware firewalls to the communication fabric. In our architecture, a security monitor module exclusively binds a peripheral to a processing unit. The security monitor allows a designated party to flexible claim and release specific peripherals. Moreover, the owner of the security monitor can transfer this privilege to any other party in the system. To demonstrate the feasibility of our novel TEE approach with trusted I/O paths, we integrate our scheme into a RISC-V computing platform and provide an FPGA prototype. Furthermore, we show the practicability of our system in a secure boot scenario.

1.2 Structure

The first chapter of this thesis provides an insight into information security and expresses various attack vectors. We explicitly focus on physical attacks like fault attacks and side-channel attacks and give a detailed explanation of the physical foundations of these attack scenarios. We introduce the used architecture in Section 2.3 and the specific platform in 2.3.1. Then, we complete Chapter 2 by presenting the Frankenstein architecture, which offers countermeasures against the attack scenarios introduced in this chapter. Chapter 3 presents TEE solutions offered by computing platform vendors like Apple, Intel, and ARM and visualizes their technical characteristics. In Chapter 4, we then summarize requirements of trusted execution environments and elaborate weaknesses of schemes proposed in the future. Chapter 5 presents our overall architecture, including our TEE scheme and the

handling of peripherals. The proof-of-concept implementation, which uses a RISC-V system as a base platform, is introduced in Chapter 6. To point out potential use cases, Chapter 7 systematically constructs a secure boot scenario using the proposed scheme integrated into the RISC-V platform. Furthermore, we illustrate the overhead produced by the TEE system and the secure I/O paths in Chapter 7. The last chapter then summarizes the contribution of this work and proposes possible future work.

Chapter 2

Background

This chapter provides background information to better understand the proposed security mechanisms. First, the term security is declared, then an overview of classical software attacks is given. Due to the rising popularity and the immense security threat for embedded devices, we also give an insight into hardware attacks, in particular fault attacks and side-channel attacks. Since the proof-of-concept implementation of this thesis uses the open-source core from lowRISC, the last section of this chapter introduces the RISC-V instruction set architecture (ISA), the lowRISC project, and the Frankenstein processing system.

2.1 Information Security

Information security deals with the protection of an asset from a potential, malicious adversary. In computing, an asset represents any piece of information with a value to the owner. To provide a certain level of security, systems are designed to prevent some attacks using software and hardware features. As more and more safety-critical applications, like autonomous driving and life-sustaining medical devices, are using computers, information security gets more important than ever before. Usually, computer security is defined using a model like the CIA triad [20].
The most apparent security property of this model is the confidentiality of an asset. A system providing information confidentiality ensures that an attacker is not able to gain insight into an asset, which can be achieved by restricting access to the secret information using encryption or similar approaches. The second property of the CIA model is integrity. Data integrity guarantees that the information cannot be tampered over its entire life cycle by an unauthorized user. Cryptographic checksums can be used to prove the integrity of data. Now, data only can be read and modified by the owner. Nevertheless, these properties are meaningless if the availability of the asset cannot be guaranteed. Therefore, ensuring availability by using redundant system or backup solutions is the last property of the CIA triad. An adversary can violate security guarantees in different ways.
In most cases, attackers try to gain remotely unauthorized access to an asset. As finding security weaknesses often requires thinking out-of-the-box, many creative attacking models can be found in the wild. In general, attack models can be categorized in logical, physical, and combined attacks, which will be explained as follows.

2.1.1 Logical Attacks

The first classification category of attacks is logical attacks. Attacks in this category are threatening the security of software running on a machine using the input communication channel and exploiting the inner logic of the software. As these attacks can often be performed remotely, the vast majority of attacks on computers fall into this category. Software offering a service to a user typically provides an input communication channel to process data. Examples could be a webserver processing incoming website requests or a banking application requiring a pin code. On a system level, other forms of inputs are application programming interface (API) calls, system event notifications, or interrupts. As all of these channels are available to a user, a potential, malicious user can use these input channels to attack a system. The attack is performed using a logical flaw in the implementation of the processing logic. Examples of these vulnerabilities are buffer overflows, validation errors, or a weak implementation of an encryption algorithm. [36, 49, 52, 88]

2.1.2 Physical Attacks

Despite the threat of an adversary challenging confidentiality, integrity, and availability of an asset by using logical attacks, physical attacks are another possibility for attacking a system. The key idea behind this attacking method is that the attacker uses physical properties of the system to perform an attack. In real-world attack scenarios, physical properties like power consumption, electromagnetic radiation, temperature, voltage, and clock supply are exploited in various ingenious ways [46, 63, 75]. These attacks often, but not always, require the attacker to have physical access to the device under attack.

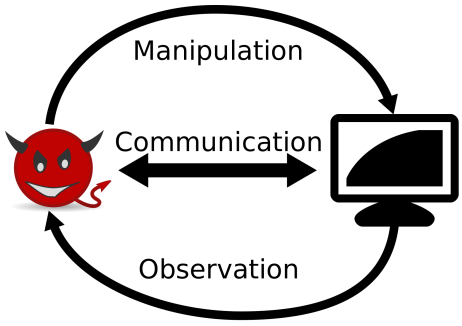


Figure 2.1: In a physical attack, an adversary can observe and manipulate physical parameters of the target device. Besides that, a communication channel is available between the two parties.

As seen in Figure 2.1, an adversary can attack a system by manipulation and observation. In an observation attack, which is also called a passive attack, the attacking party tries to gather information by some kind of side-channel to reveal secret information. Gathering information can be done by using timing, power consumption, or other similar side-channels. For an active attack, the adversary tries to manipulate the other party in such a way that an exploitable behavior is provoked. Faulting the device to bypass security checks or inducing faults into cryptographic calculations to exploit mathematical properties are some examples for active attacks. Furthermore, physical attacks can further be categorized into the degree of invasiveness needed for the attack. Attacks which do not alter the device

under attack at all are called non-invasive attacks; semi-invasive and invasive attacks do alter a device to a certain extent.[62, 102]

Fault Attacks

A prominent example of a physical attack manipulating the device under attack is the fault injection. In a fault injection (FI) attack, an adversary manipulates the environment conditions of a device to challenge the security of the system. An adversary has several possibilities for manipulating the system environment. One example are non-invasive techniques like voltage, and clock supply glitching or injecting EM disturbances to the chip. Applying high or low temperatures to the surface of the device is another possibility, but sensitive devices tend to take damage [17]. In using more invasive and expensive attacks like decapsulating the chip and directly shooting with a laser to the chip die, the likelihood of inducing a targeted fault is higher [17].

Figure 2.2: A glitch can be induced by tampering the system clock.

Figure 2.2 depicts a non-invasive and relatively cheap fault injection scenario. In this attack, the system reference clock is manipulated by an attacker. As a steady clock synchronizes most of the inner logic of the system, increasing or decreasing the clock period could have various side effects. When, for example, decreasing the length of a single clock pulse used by a processor, the processors' time to execute the instruction could be too short and therefore the instruction is skipped [71].

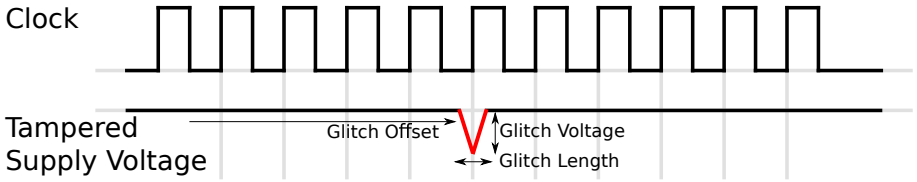


Figure 2.3: In a fault injection scenario using a voltage glitch, the supply voltage is manipulated by the attacker.

Another way to induce a glitch is by tampering the supply voltage. In a natural environment, the supply voltage powering the circuit is stable. However, when injecting a short positive or negative pulse into the supply line, malfunction, like a malformed data read or write, can happen [39].

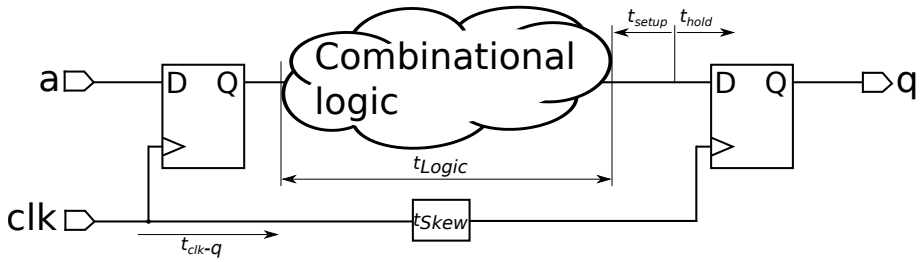


Figure 2.4: Concept of synchronous hardware design and relevant timing parameters.

Most of today's digital circuits use the concept of synchronous hardware design. As depicted in Figure 2.4, these designs consist of two registers, namely D-flip-flops and a combinational logic in between [85]. If the first D-flip-flop detects a rising clock edge, data is released and processed by the combinational logic. On the next rising clock edge, the processed data is again stored in the second register [111]. The time it takes the data to process through the combinational logic is called combinational propagation delay t_{Logic}. This time coheres with the data being processed. Besides delays in the combinational path, a D-flip-flop also has timing dependencies. Data must remain stable at the input pin and the output pin for a clearly defined period. This time is called setup time t_{setup} for the input and hold time t_{hold} for the output. The clock-to-q delay t_{clk-q} represents the delay from the input of the flip-flop to the output [64]. When now considering a voltage fault injection attack where the supply voltage is decreased for a short period of time, the setup time for some parts of the design can be violated and faulty data can be captured. Due to physical properties, decreasing the supply voltage increases the combinational propagation delay and a setup time violation can occur [111].

As timing is critical in most systems, the success rate of injecting an exploitable fault depends on various parameters. Figure 2.3 gives an idea of the parameter space of a voltage glitch attack. For a targeted attack, the timing of the inserted glitch, as well as the glitch length and the glitch voltage, is essential. The latter two properties together are also called induced glitch energy. Finding the correct parameters for the attack is often done by a parameter sweep; however, also more clever ways of finding a possible parameter combination exist [26].

The effects and the way how to exploit faults can vary. In one of the first known intentional fault attack, the RSA Bellcore attack, the authors proposed an attack against the RSA cryptosystem. In this attack, the fault manipulated results of the computation. Through using mathematic properties of the scheme, a key recovery attack could be performed [21].

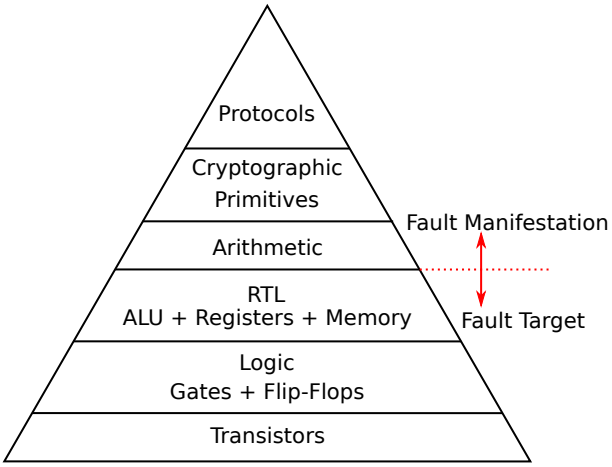


Figure 2.5: Security pyramid. Shows the difference between a fault target and the fault manifestation [101].

In Illustration 2.5, all steps required for designing a cryptographic protocol are depicted. Interestingly, the fault induced into the lower, physical level of the design influences the higher abstraction levels [101]. The effects a fault can provoke as well as the fault characteristics are summarized in the so-called fault model. In the fault model, the bit granularity, the fault type, the fault location, and the duration of the fault are captured [16]. The goal of the induced glitch can vary, but in most cases, an adversary tries to influence the control-flow or the data-flow of a target [3]. Attacks targeting the control-flow of a program executed on the victim system either try to corrupt or skip instructions [95]. These techniques allow an attacker to bypass security checks, corrupt Linux superuser privilege checks, or even gain remote-code execution on automotive platforms [66, 70, 95].

Countermeasures

Faults are not only a threat when injected by an attacker intentionally, but they also can occur through physical interference from the environment and harm the correct execution of microcontrollers in satellites for example [73]. For this reason, countermeasures against faults are a well-studied field in computer science. In literature, the most common ways to provide certain protection against these attacks are detection-based, infection-based countermeasures, and fault space transformation-based countermeasures [84].

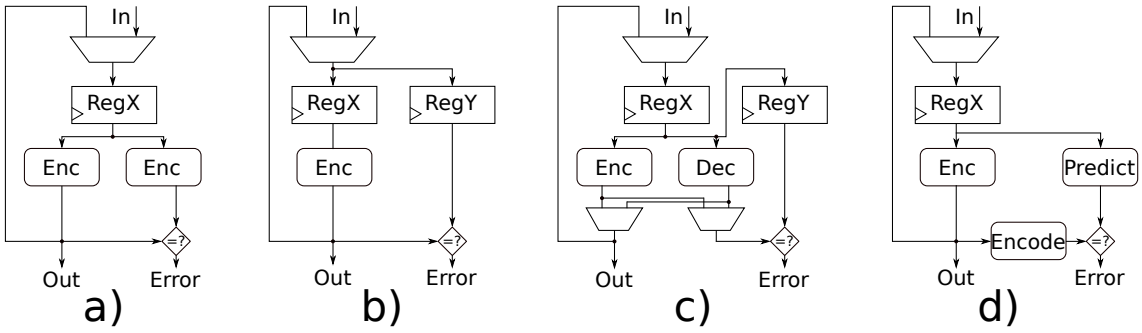


Figure 2.6: Four different concurrent error detection schemes. **a:** Hardware redundancy. **b:** Time redundancy. **c:** Hybrid redundancy. **d:** Information redundancy. [44]

The first method, concurrent error detection (CED), uses different forms of redundancy to detect a fault and a check for fault nullification [76]. As seen in Figure 2.6, different forms of using redundancy are possible: hardware redundancy, time redundancy, hybrid redundancy, and information redundancy. In hardware redundancy based systems, the sensitive circuit is copied and both instances are executed. By using a comparison check, a fault can be detected. This provides reasonable protection against random faults. However, an attacker capable of injecting a fault twice can bypass this countermeasure. Time redundancy based schemes perform the same operation twice on the same hardware and throw an error if there is a mismatch in the result. Again, this scheme is not secure against second-order fault attacks, where two faults are injected. In hybrid redundancy schemes, the inverse of the performed calculation is computed and compared to the input. Information redundancy based countermeasures are using error detection codes to generate parity bits from the input and a prediction circuit to predict the result. These bits are then compared with the calculated parity bits from the output [44].

The weak spot of detection based countermeasures is the comparison step. If an adversary is able to bypass this check, all the redundancy introduced before is futile. Infection-based countermeasures try to bypass this limitation by amplifying the effect of an induced fault [58]. No additional detection step is required as the infection of information should destroy any leakage an attacker could exploit. Another possible countermeasure is the fault space transformation [76]. In this scheme, the attacker is prevented to use the fault bias and to induce the same fault in the redundant part again.

Side-Channel Attacks

Side-channel analysis is a category of an attack where the adversary collects information related to a secret and uses this meta-data to learn about the secret value. Compared to fault attacks, side-channel attacks are completely invasive. Information is gathered by observing physical parameters like power consumption, electromagnetic radiation (EMR), or calculation time.

Timing Side-Channel

One example of a side-channel used to attack a system is the timing side-channel. The basic idea of this attack is to measure the time needed to operate on the sensitive data.

```
1 int memeq(const uint8_t* i1, const uint8_t* i2, size_t s)
2 {
3   for (; s; --s, ++i1, ++i2)
4   {
5     if (*i1 != *i2)
6     return 1;
7   }
8   return 0;
9 }
```

Listing 2.1: Memory compare function vulnerable against timing side-channel attacks [80].

Listing 2.1 shows a simple C-function comparing two memory regions. If both regions are equal, 0 is returned. Since the value 1 is returned as soon as the first mismatch occurs, an attacker can learn how many bytes are equal by measuring the timing differences. One example of a timing attack with a significant impact in the past is the Lucky Thirteen attack [2]. This attack is a full plaintext recovery attack and uses a timing side-channel attack in the message authentication code (MAC) comparison check in the TLS algorithm. To mitigate timing attacks, algorithms handling sensitive data need to be designed to consume constant time regardless of the processed data.

Power Analysis Attacks

Another side-channel attack is the power analysis attack. Here, the attacker uses the information leakage caused by the power consumption of the device to gain secret information.

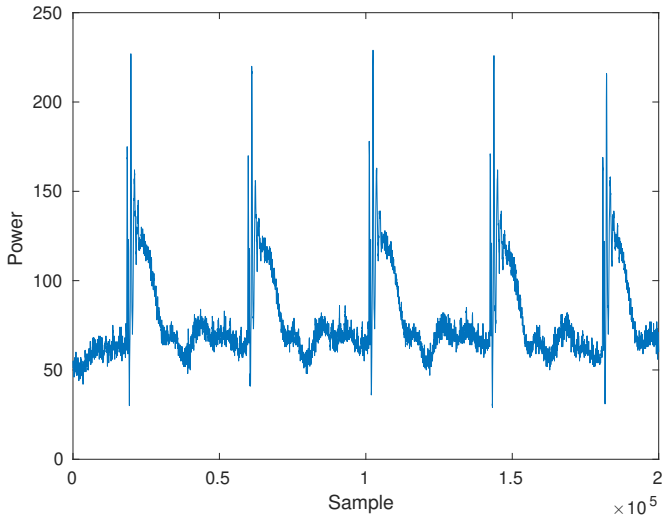
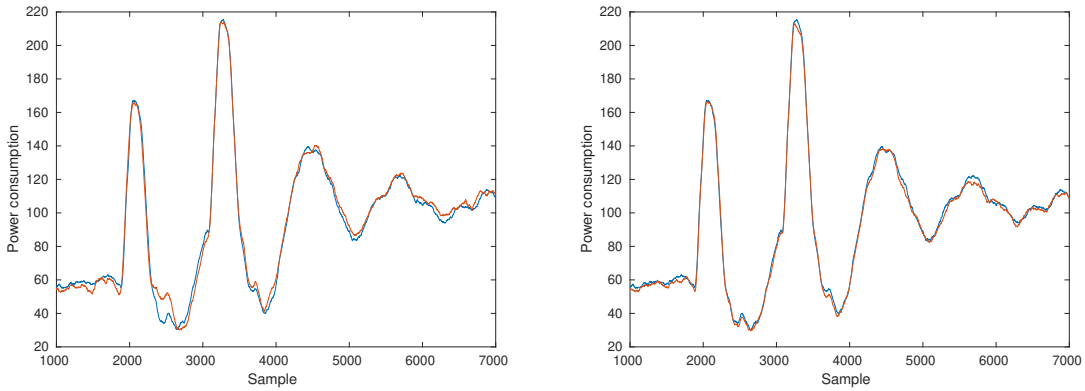


Figure 2.7: Power consumption measured during an AES encryption [69].

In Figure 2.7, the measured power consumption of a microcontroller performing an Advanced Encryption Standard (AES) encryption is shown. Internally, the microcontroller uses a cryptographic accelerator performing two AES rounds in one step. As one could see, the 10 rounds of the AES encryption scheme can easily be identified using this power trace.



(a) Different plaintexts.

(b) Identical plaintexts.

Figure 2.8: Power traces of two AES rounds performed on a microcontroller unit (MCU) [69].

When analyzing one AES primitive mode operation consisting of two AES rounds, the behavior illustrated in Figure 2.8 can be observed. Figure 2.8a depicts two primitive mode operations with two different input plaintexts. Compared to Graph 2.8b, where the same plaintext is encrypted twice, a small variance in the power consumption when encrypting different plaintext values can be detected. In general, a power analysis attack uses the data dependency of the power consumption during a cryptographic operation. By exploiting mathematical properties of the encryption scheme, the secret key can be retrieved.
In literature, hiding and masking are suggested countermeasures against these kinds of attacks [79]. When using hiding as countermeasure, the data dependency of the power consumption is reduced to a minimum. For masking based countermeasure schemes, a random number is added to the intermediate result of the algorithm. When using a proper random number generator, the power consumption measured during the execution of the algorithm now is independent of the processed data up to a certain order.

Microarchitectural Attacks
Microarchitectural attacks are targeting the information leakage of the underlying hardware and exploit this leakage by using side-channels [43]. One example of such an attack are cache-based attacks. In an usual computing architecture, caches are used to improve the performance of memory accesses. As on-chip memory is expensive, these caches are usually shared among the CPU cores. By exploiting timing differences when accessing data in the cache, powerful attacking tools can be crafted [18, 19, 81]. In 2018, a new attack category based on transient executions was presented by Kocher et al.. Almost all modern CPUs nowadays are using out-of-order execution and speculative execution to speed up computations. Computing architectures implementing these techniques are re-ordering instructions to gain high utilization of the pipeline and all execution units. Moreover, the CPU tries to predict the result of a branch and executes the code after this branch. When the CPU detects a wrong branch prediction, the results of the executed instructions need to be revoked. These instructions are called transient instructions. Attacks like Spectre [51], Meltdown [57], and Foreshadow [100, 107] are using these instructions to perform powerful attacks, like extracting keys from an Intel SGX enclave.

2.2 Secure Boot

Protecting a security sensitive system already starts directly after power is applied to the computing unit. The bootloader, which is usually located on the motherboard of a computer, loads the operating system image and executes the OS. However, the Edward Snowden leaks showed that the NSA actively was using a malware capable of modifying the system BIOS and injecting backdoors [74]. Even when reinstalling the operating system, this malicious system stays active as the BIOS firmware is persistent. An adversary with less attacking capabilities still can threaten the system by directly modifying the operating system and embed malicious software to it. Primarily, this attack scenario targets ordinary users and tries to steal assets like banking login data and other secrets. Nevertheless, vendors offering computing systems are also interested in protecting their bootloader and operating system. Despite of protecting a user from malicious tampering attempts, another reason for vendors of protecting the overall system using secure boot is to protect the authenticity of the software executed on the device. For example, Apple uses secure boot to prevent users from installing an unofficial version of the system software. Moreover, manufacturers of gaming consoles also try to keep their devices genuine in order to thwart users installing pirated games. To prevent such tampering attempts, the concept of secure boot was introduced. In a system supporting secure boot, a chain-of-trust is generated allowing only authenticated software to boot [30]. This is achieved by embedding a root-of-trust element to the system [53]. This root-of-trust element, which is often embedded in the system read-only memory, consists of a first-stage bootloader loading the second-stage bootloader. When loading this image, the first-stage bootloader (FSBL) verifies the cryptographic hash of the image by comparing it to the hash stored in the root-of-trust element. After a successful check, the system fully trusts the second-stage bootloader. Now, this bootloader loads the remaining part of the operating system and again compares the signature of the image. Finally, the operating system boots and the user can use the authenticated system. If one verification step fails, the system traps into a fallback mode. Due to the mechanism of loading and verifying images step by step, a chain-of-trust system is generated. This scheme now protects the integrity of the software images loaded during the boot stage. However, a company developing software deployed on a certain device also wants to protect the confidentiality of their intellectual property (IP). For this reason, secure boot often is combined using a cryptographic encryption and decryption scheme. The system software image, which usually is stored on an external memory, e.g. an SD card or a hard drive, is encrypted. During the secure boot procedure, the FSBL decrypts the image using the keys stored in the root-of-trust. Using encryption and signature checks during the boot processor, the confidentiality, and integrity of the system software can be guranteed.

2.2.1 Attacks against Secure Boot

Most attacks against secure boot try to bypass the write protection of the root-of-trust element and replace the signatures of accepted images [25]. An attacker having hardware access to the device under attack also has several other possibilitites to threaten the security of the system. In most systems, the root-of-trust is stored on an external SPI flash chip on the motherboard [110]. Using an SPI programmer, an adversary easily can overwrite the firmware stored in it. However, also physical attacks are a serious attack vector. One of the most famous attacks against a system using fault injection is the XBOX

360 glitch hack [39]. This gaming console is using a chain-of-trust system to securely load the operating system software. When injecting a precise glitch during the signature check, this check can be bypassed and arbitrary software can be executed on the gaming console.

2.3 RISC-V

Open-source software, like the Linux kernel, Git, and others, gained immense popularity in the past. These projects are not only mostly free-to-use, but others can also learn from the ideas and improve their code. However, open-source hardware projects are rare. Not even the Raspberry Pi platform is fully open-source as it uses a proprietary ARM system-on-chip (SoC) provided by Broadcom [37]. The RISC-V foundation tries to tackle this problem by providing a fully open-source hardware ISA [38]. As the ISA defines the interface between hardware and software, system engineers can design their own processor following this specification and are still supporting software written for this ISA [77]. Having such an open-source instruction set architecture could lead to more innovation and cheaper chips through competition and also a shorter time to market through shared open-core designs [12]. Due to these advantages, companies are already offering various RISC-V-based processors, even capable of running Linux [92]. The RISC-V ISA offers specifications for standard extensions like integer base instructions, bit manipulation instructions, vector operations instructions, and many more. Moreover, RISC-V even allows customized instructions for specialized applications. As the ISA is flexible, simple, single-core microcontrollers, as well as huge node clusters, can be built [103].

2.3.1 lowRISC

The lowRISC project aims to provide an open-source RISC-V SoC with the capability of running Linux on the chip [68]. In the last release of the project, the chip is able to boot an up-to-date version of the Debian operating system and even support for an Artix-7 field programmable gate array (FPGA) is given. In addition, hardware IP for peripherals like UART, SD card, Ethernet, and VGA is provided [59]. With these properties, the lowRISC project can be considered as a full and rich RISC-V ecosystem. In the past, several novel concepts like tagged-memory were integrated into the core. One remarkable approach introduced into the SoC was the concept of so-called minion cores. A minion core is a small, basic processor embedded to the SoC which is running in parallel to the main core [22]. Example usages of these cores could be outsourcing the computing power needed for handling I/O devices or preprocessing data. The lowRISC designers also suggest that the main core could delegate tasks to the small cores for performance reasons or creating isolation for security reasons [22]. However, in the latest version of the chip, the tagged-memory feature and the minion cores were abolished.

The lowRISC Computing Architecture

As the lowRISC project offers a standalone SoC capable of booting Linux on an FPGA, several components are required. Starting from the core itself to the peripheral controller and a Xilinx DDR3 controller, the chip already provides all of these components.

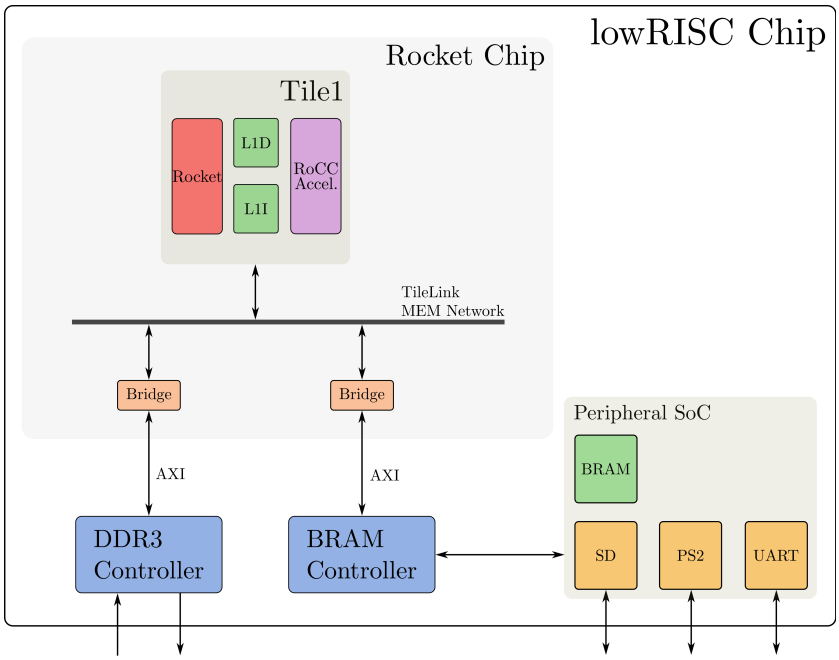


Figure 2.9: Block diagram of the lowRISC chip [59].

A systematic overview of the current lowRISC chip release is depicted in Figure 2.9. lowRISC uses the Rocket chip [13] as a processing unit. This chip is actively developed and maintained by UC Berkeley and SiFive and it can be instantiated using the Rocket chip generator. By using this generator, the number of cores, the cache size, and many other parameters can easily be changed and this allows the chip designer to easily create flexible designs [54]. As the Rocket chip generator is written in the high-level language Chisel [15], object-orientated and functional programming is possible. The Rocket chip consists of one or multiple so-called Tiles. A single Tile contains the Rocket core, the RoCC accelerator, and an L1 data and instruction cache. The Rocket core itself is constructed as an in-order, scalar, 64-bit processor with a 5-stage pipeline [59]. The Tiles are connected to each other and to the AXI bus using the TileLink network. In the default lowRISC configuration, one Tile with an L1 data and instruction cache is used. By using the TileLink network and TileLink to AXI bridges, the Tiles have access to the attached Xilinx DDR3 controller IP and to the BRAM controller, which is also offered by Xilinx. The peripheral SoC, which contains additional IP for the SD card controller, UART controller, and many more, is connected to the BRAM controller by using a simple address remapping. In addition to supporting various hardware peripherals, the lowRISC chip is also capable of booting a RISC-V Linux port. As the internal BRAM has a limited storage capacity, a small bootloader placed there copies the larger Berkeley bootloader (BBL) from SD card to DDR3 memory. This second-stage bootloader configures the environment and fetches the Linux kernel from the SD card. Another possibility to initialize the Linux kernel is by using a server serving as a remote boot unit.

2.3.2 PULPino

PULPino is an open-source microcontroller system designed by the University of Bologna and ETH Zurich [34]. The PULP platform offers a variety of RISC-V compliant cores,

starting from a small 32-bit 2-stage core up to a 64-bit 6-stage core with Linux support [35]. For the PULPino processing system, the designer can choose between the RI5CY core and the smaller ZERO-RISCY core [96]. Both cores fully support the RISC-V base integer instruction set (RV32I) and extensions like the floating-point instruction set, hardware loops, and many more are available [34]. By supporting various communication peripherals, PULPino can be used in real-world applications and the system even had been taped-out in 2016 [34].

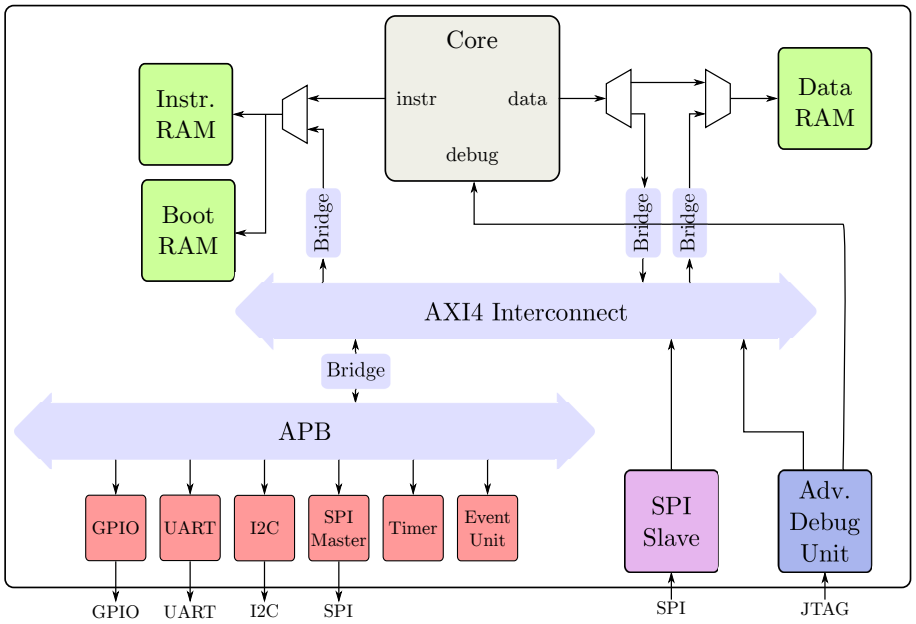


Figure 2.10: Schematic overview of the PULPino platform [97].

Figure 2.10 depicts the inner structure of PULPino. As already mentioned, PULPino is either available with a RI5CY core or a smaller ZERO-RISCY core. To simplify the hardware design, PULPino uses separate data and instruction memory [83]. The boot ROM contains a small bootloader, which loads a program from an external device connected over SPI [96]. The core itself is connected to the AXI4 bus using bridges, access to the peripherals connected to the APB bus is granted by using an AXI4 to APB bridge. For convenient debugging, the debug unit enables access to the core registers and the two memory instances over JTAG [96].

RI5CY Core

The RI5CY processor is the heart of the PULPino. RI5CY is a 32-bit, in-order core with a 4-stage pipeline supporting the base integer instruction set (RV32I), the extension for compressed instructions (RV32C), the integer multiplication and division instruction set extension (RV32M), and also the single-precision floating-point extension (RV32F) [98]. Furthermore, several specific extensions, like hardware loops and arithmetic logic unit (ALU) extensions, are offered. RI5CY can be used for real-world applications as the core offers support for application-specific integrated circuit (ASIC) synthesis as well as for FPGA synthesis when using the flip-flop based register file.

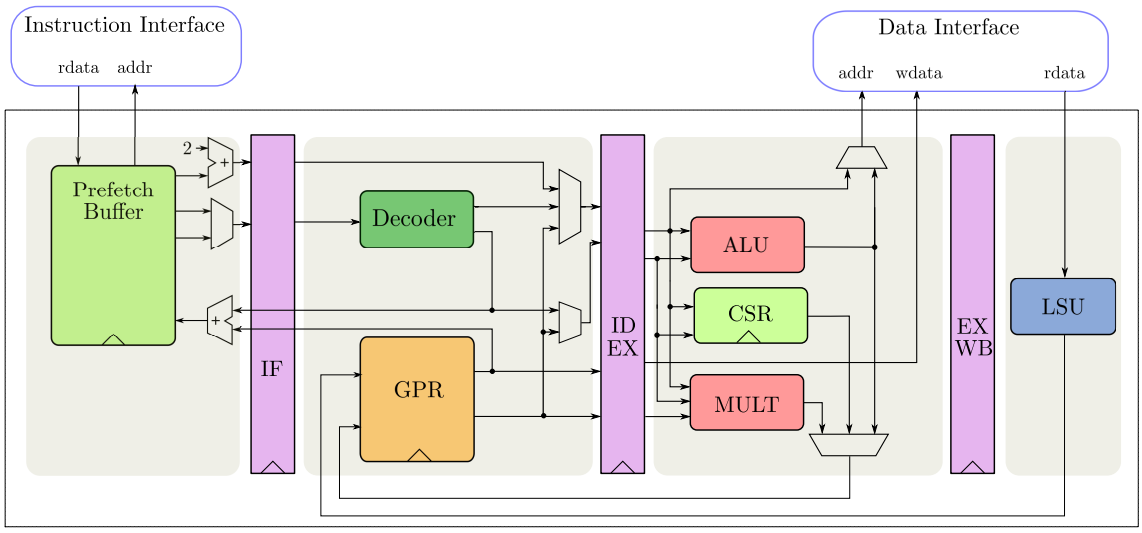


Figure 2.11: RI5CY processor pipeline [98].

Figure 2.11 depicts all pipeline stages of the RI5CY core pipeline. For a given instruction address, the instruction fetch stage loads an instruction from the instruction cache or instruction memory. For performance reasons, the prefetch buffer is able to preload instructions and save them in the internal FIFO. In RI5CY, a simple protocol, similar to the protocol used by the load-store unit (LSU), enables access to the instruction storage. As the instruction fetch unit only needs read access to the storage, only a few signals are needed in the hardware design. After the instruction fetch (IF) stage, the instruction is passed to the instruction decoder (ID) unit, which now analyses the raw instruction and sets registers and control signals of the datapath according to the instruction type. Finally, the actual computation takes place in the instruction execute stage using dedicated modules. The ALU is capable of performing arithmetic operations, bit-shifting, and comparisons efficiently. Additionally, modules like a multiplier or hardware loops, are used to perform operations at a reasonable speed. When the executed instruction manipulates data, the write-back stage is used to write data to memory using the LSU.

2.3.3 Frankenstein Core

Frankenstein [87] is a RISC-V-based processor embedding several security features in its hardware and software design. The processing unit is based on the previous introduced RI5CY core and is extended to 64-bit. Compared to RI5CY, Frankenstein consists of a control-flow integrity (CFI) unit [108] protecting the control-flow of programs, encoded pointers, and extended an LSU unit to protect memory accesses.

Control-Flow Integrity

In one possible attack scenario, an adversary tries to find a vulnerability in the software stack and uses this vulnerability to either inject own code or reuse existing code to build an exploit. In either way, the attacker tries to manipulate the control-flow of the target to compromise the security of the device. When including physical attacks in the threat model, the attacker is additionally able to inject malicious code directly to the external memory or use FI to redirect the control-flow by skipping or manipulating instructions.

CFI schemes try to tackle a wide range of these problems by ensuring that the control-flow of the program cannot escape a predefined path of a control-flow graph (CFG) [1]. A control-flow graph represents all valid paths through the given program and is usually determined by using source code or binary analysis. The control-flow integrity scheme integrated into Frankenstein encrypts the program to be executed during compile-time and decrypts and authenticates the instructions during runtime in the decoder stage [108]. Internally, the CFI scheme uses a cryptographic state which gets updated after each instruction. As manipulating the control-flow by using some kind of attack would destroy the state irreversible, this then returns invalid instructions which can be detected by an invalid trap. As the control-flow of software is not linear, the scheme presented by Werner et al. uses patching of branches and other CFI instructions to ensure that the same state is produced in all valid paths. These properties of the proposed countermeasure ensure the authenticity of the software executed on the device. Furthermore, encrypting the software provides an effective countermeasure against IP theft.

Memory Access Protection

CFI schemes ensure the correct execution of a program by protecting the code executed on a device. However, in common CFI schemes, conditional branches are not protected from fault injection attacks targeting the comparision step [86].
Furthermore, an attacker capable of modifying arbitrary memory regions is still able to influence the control-flow by manipulating data used for decision making. To address this problem, a well-studied countermeasure against these attacks is data encoding. By transforming data into a redundant representation, data tampering can be detected to a certain level [87]. Nevertheless, even when combining data encoding and control-flow integrity protection, a powerful attacker having physical access to the device under the attack still has attacking possibilities. The first attack targets the pointer addresses itself. An attacker injecting a glitch using fault injection could manipulate the value of the address pointer and data from the wrong memory location is fetched. When again considering fault injection attacks, a glitch directly introduced into the memory bus line could also manipulate the address transmitted on the bus and again data from a wrong memory location is fetched. These two suggested attacks cannot be detected by CFI and data encoding schemes as no data gets manipulated and the control-flow graph gets not violated.

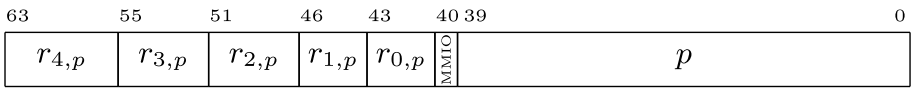


Figure 2.12: Pointer encoding scheme [87].

The Frankenstein core implements a memory access protection scheme to close this attack vector. To mitigate the first attack, the value of the pointer is encoded by using residue codes [87]. As residue codes are arithmetic code, frequently used pointer arithmetic operations can be performed efficiently in the encoded domain by having dedicated instructions. As seen in Figure 2.12, in the lower 40-bits the pointer information is stored and in the upper 24-bits the redundancy information. By using the upper 24-bits, no additional storage for the encoded representation is needed and still up to one terabyte of memory can be addressed [87]. Furthermore, Frankenstein links data with addresses to provide protection against the second attack category. Before data gets written into the memory,

Frankenstein destructively overlays data with the address information. When an attacker tampers the memory address during the bus request, unlinking the data with the address fails on the next read operation as a different address was used. This attack attempt then is detectable in software.

Overall Architecture

The scheme proposed by Schilling et al. is directly embedded into the RI5CY architecture after expanding the system to 64-bit. Thanks to the extensive hardware and compiler support, the runtime overhead amounts to roughly 7% and the code size overhead is at around 10%.

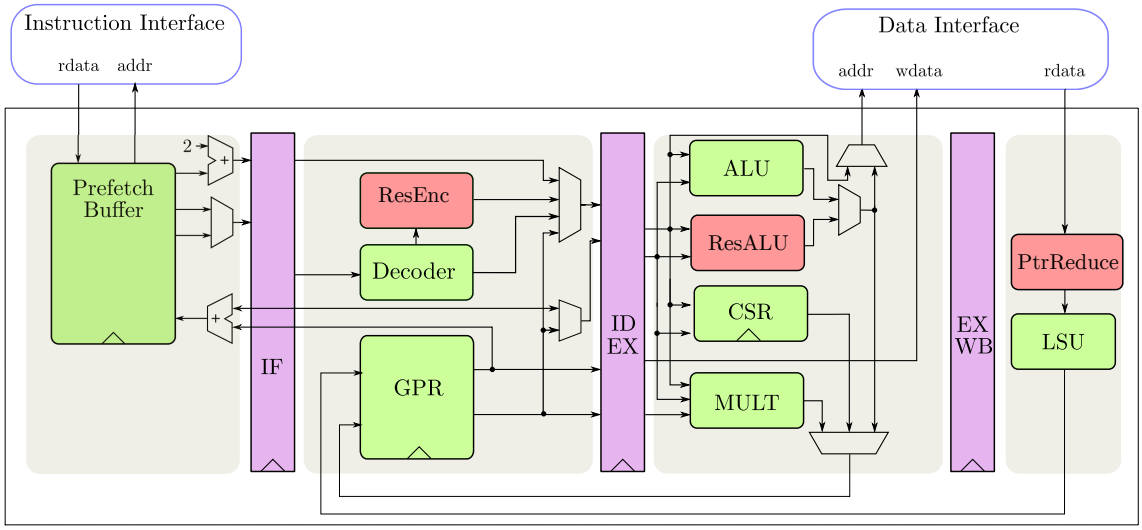


Figure 2.13: Extended pipeline of Frankenstein [87].

Similar to the RI5CY pipeline overview, Figure 2.13 depicts the extended pipeline of Frankenstein. Since accessing memory is a frequently performed task, the pointer encoding and data linking procedure is realized in hardware. For pointer encoding, decoding, and arithmetic, dedicated new instructions are part of the Frankenstein architecture. These instructions are part of the enhanced decoder shown in Figure 2.13. Since performing arithmetic operations is natively supported by arithmetic codes, the residue ALU performs the computations in this domain. For the data linking part of the countermeasure, each data word gets linked with the address before leaving the processor. Therefore, the load-store unit of Frankenstein automatically performs the linking and unlinking operation. The control-flow integrity unit, which extends the 4-stage pipeline to 5-stages, is not shown in Illustration 2.13.

Chapter 3

Related Work

Almost all major commercial vendors in the system-on-chip market are offering various trusted execution environments. This chapter introduces the need and the concept for this technique and compares solutions offered by different vendors.

3.1 Secure Enclaves

An application executed by a user is mostly running on top of an operating system (OS). The main purpose of the OS is to provide a level of abstraction between hardware and software. As these operating systems provide countless features, the code base of such systems is growing on a yearly rate. A modern operating system, like the Linux kernel, consists of roughly 25.3 of million lines of code (LOC) and increases by more than $200,000$ lines each year [45]. This is problematic as research expects around $1-25$ bugs per $1,000$ lines of code written in industry on average [65]. When analyzing recently discovered vulnerabilities with high impact, it shows that almost all of them are exploiting a bug in the underlying software. The Heartbleed bug [31], which affected approximately 25% of all https servers worldwide, allowed an attacker to read chunks of memory by using a bug in the OpenSSL library. Another example of a vulnerability exploiting a bug is Shellshock [33], which targets the Unix-Shell Bash and enables an adversary to remotely execute commands. Certainly, not all of the $1-25$ bugs found in $1,000$ lines code can be exploited. Nevertheless, having a larger and more complex codebase increases the likelihood of introducing an exploitable bug. As the features offered by today's general-purpose computers steadily increases, decreasing the codebase of the operating system kernel and drivers seems not to be reasonable. To mitigate this problem, the concept of trusted execution environment was introduced.
A trusted execution environment (TEE) is a decapsulated area located in the application processor with a clearly defined interface to the outside [41]. As the application processor only has access to the trusted environment using this interface, a secure, and isolated space is generated. The use cases of enclaves are diverse. Developers could execute security-critical applications in the enclave, move a frequently used security library to this area, or store sensitive information like keys in there. However, in most trusted execution environments, software with a reasonable small codebase compared to the rich operating system is deployed into the TEE. When deploying a small operating system kernel like the seL4 kernel [50] to the enclave, it is even possible to formally verify the correctness of the system. Note that state-of-the-art formal verification only works for smaller codebases, verifying the Linux

kernel with dozens of LOC is not yet possible [29].

3.1.1 ARM TrustZone

ARM TrustZone [109] is a hardware security extension integrated into ARM processors to enhance the overall system security. Due to the high market share of ARM in the mobile sector and the immense security threat for mobile applications, a widespread use case of ARM TrustZone is Android, where sensitive cryptographic keys can be securely stored in the TEE [27]. To achieve an isolated execution environment, ARM TrustZone partitions the system in a so-called, secure and non-secure world [8]. By applying this concept to all hardware and software resources, a strict separation between security-critical and non-security critical applications can be created. Typically, a common operating, e.g., Android, is used in the normal world to provide a rich execution environment (REE). In the secure world, a small, secure kernel is deployed to generate a trusted computing base (TCB) [55].

Hardware Architecture

The security protection mechanisms of ARM are achieved by directly embedding the TrustZone technology to the system architecture [8]. Therefore, ARM TrustZone is a part of the overall hardware architecture. As mentioned before, security is provided by dividing the system in a secure and a non-secure world. Instead of having two separate systems in one chip, a built-in mechanism allows almost all system components ,e.g., memory or a peripheral, to operate in both domains. Splitting up resources into a secure and a non-secure world is even done for the ARM processor cores itselfs, which allows the processing unit to operate in both security domains. As this mechanism deeply is embedded to the core itself, no additional security coprocessor is needed and therefore expensive die area and the power consumption can be saved.

System Bus Architecture

In an ARM system-on-chip (SoC) a bus protocol following the ARM Advanced Microcontroller Bus Architecture (AMBA) specification is used to connect all available building blocks of the system [11]. As TrustZone divides the whole system into two domains, all intellectual property (IP) inside the system have to distinguish between requests from the secure and non-secure world. ARM is doing this by introducing the two new control signals *ARPROT* and *AWPROT* into the Advanced eXtensble Interface (AXI) bus protocol. These two signals allow the participant to distinguish between a privileged request from the secure domain or unprivileged request from the non-secure domain. Then, the master or slave can grant or deny this request. Privilege violations are reported by raising a bus error. For peripherals, often the low-bandwidth peripheral bus Advanced Peripheral Bus (APB) is used. As this bus protocol does not support the non-secure (NS) indicator bit directly, the AXI-to-APB bridge has to verify the privileges. Providing a mechanism integrated into the system and peripheral bus to create access policies is extremely powerful as a trusted path from the user I/O peripheral to the system core can be created. Two security domains also require separate memory regions. Therefore, ARM TrustZone extends 32-bit physical addresses to 33-bit, where the highest bit represents the NS-bit [8].

Processor Architecture

Each physical processor core in the SoC consists of a non-secure and a secure virtual core.

When accessing peripherals over the system bus, the NS-bit stored in the virtual core is used to distinguish between the secure and non-secure core.

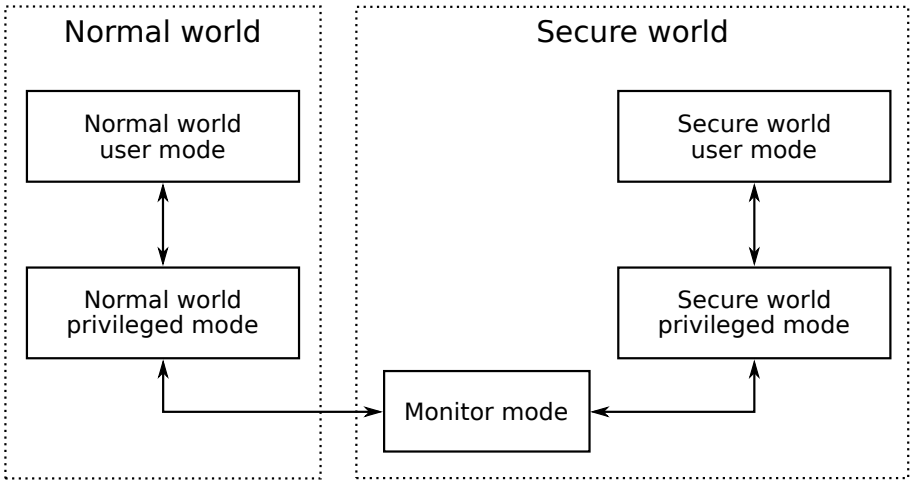


Figure 3.1: Different CPU modes in ARM TrustZone [8].

In Figure 3.1, a physical ARM core with its different modes is depicted. The secure and the non-secure domain both support user mode and privileged mode. Both virtual cores are scheduled in time slots; the context switch is performed using an additionally introduced monitor mode. A context switch is initiated by using the secure monitor call (SMC) instruction. The software running in the monitor mode saves and restores the states of both worlds, the secure configuration register (SCR) indicates the current security domain of the virtual core.
Since accessing memory is one of the most frequent operation a processor is performing, protecting memory from a potential adversary is crucial. For this reason, memory attached to the system bus is guarded by the additional NS-bit in the physical address. As the high-speed L1 cache is shared between the virtual cores, this memory also needs to be separated into the two security domains. ARM, therefore, uses additional metadata in the cache to differentiate between secure and non-secure content. Since both worlds can load data to the same cache, a cache flush on a context switch is not necessary. Translating virtual addresses to physical addresses is performed by two virtual memory management unit (MMU) belonging either to the secure or non-secure world.

Interrupts
Internal, as well as external peripherals, are frequently generating interrupts. In TrustZone, two different interrupt lines are provided. The non-secure world makes use of the IRQ and the secure world of the FIQ interrupt source. When an interrupt happens while beeing in the corresponding domain, no context switch to the security monitor is needed. In the other case, the software running in the monitor mode is responsible for routing the interrupt correspondingly and switching to the other domain.

Multicore Support
As many ARM processors are deployed in larger processors, e.g., smartphones, and notebooks, a security feature only available for single-core processors are not sufficient for

today's demand. Therefore, ARM TrustZone is also available in multicore systems.

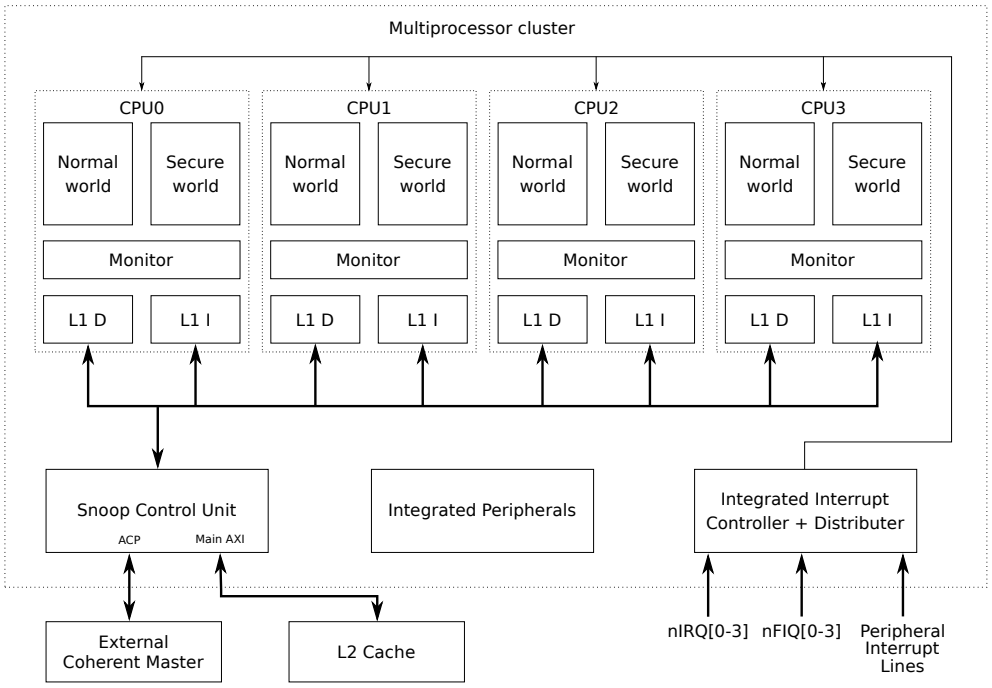


Figure 3.2: Multicore ARM processor with TrustZone support [8].

Figure 3.2 shows TrustZone deeply embedded in a quad-core processing system. As one physical core consists of two virtual cores, in total, eight virtual processors are available. Each of the physical cores consists of the secure monitor and an L1 data and instruction cache. This allows each core to either operate in the secure or non-secure world, independently of the other cores. In this system, a dedicated interrupt controller supporting TrustZone is available.

Software Architecture

Having two virtual cores running in parallel allows the system to execute two pieces of software almost concurrently. In most cases, a conventional operating system like Linux is executed in the REE and the secure environment provides additional security features. As suggested by ARM, a simple library or a complex, security-hardened operating system can be placed in the secure domain [8].

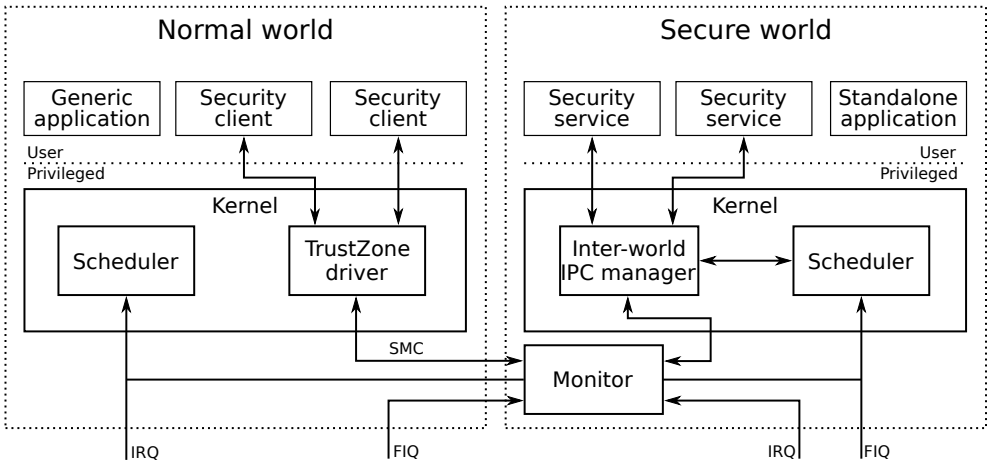


Figure 3.3: Physical ARM core with one OS running in the REE and the other in the secure domain [8].

Figure 3.3 depicts one possible application of ARM TrustZone, where a rich operating system is deployed in the normal world comprising several independent tasks. A small operating system executed in the secure world offers security services and also consists of standalone applications. Using a TrustZone driver embedded into the normal world kernel allows applications to use these services. Communication between the two worlds is done using the security monitor. As both virtual cores are still executed on a single physical core, only one of them can run at a certain point in time. The security monitor triggers the scheduler of one of the virtual cores by using interrupts.

Use Cases

One use case of ARM TrustZone is securely booting an operating system in the REE. The secure world cryptographically verifies each step during the boot procedure of the OS. If all steps have successfully been verified, the user of the device can be sure that a verified OS is loaded an running. Communication between the operating system and the secure world again is done using the monitor kernel mode. [72]
Android uses ARM TrustZone as a TEE in its middle to high-end phones with an ARM processor [32]. Inside the enclave, a small secure operating system is running. The device manufacturer is able to deploy its own code, e.g., features for secure payment, to the TEE. The Android operating system then communicates through an application programming interface (API) specified by GlobalPlatform [40] with the TEE to use the provided services.

3.1.2 Intel SGX

Intel's trusted execution environment, called Software Guard Extensions (SGX), offers protection from various attack attempts initiated by the operating system bios, firmware, and drivers [14]. Additionally, the assets stored in the enclave are even secured against attacks from higher privileges like the System Management Mode (SMM) and Intel Management Engine (ME) [28]. Compared to ARM TrustZone, Intel SGX is built on a set of security extensions integrated into Intel's CPU architecture [28].

Hardware Architecture

By introducing the concepts of processor reserved memory (PRM), enclave page cache (EPC), and enclave page cache map (EPCM), Intel SGX isolates sensitive code and data from the hostile environment.

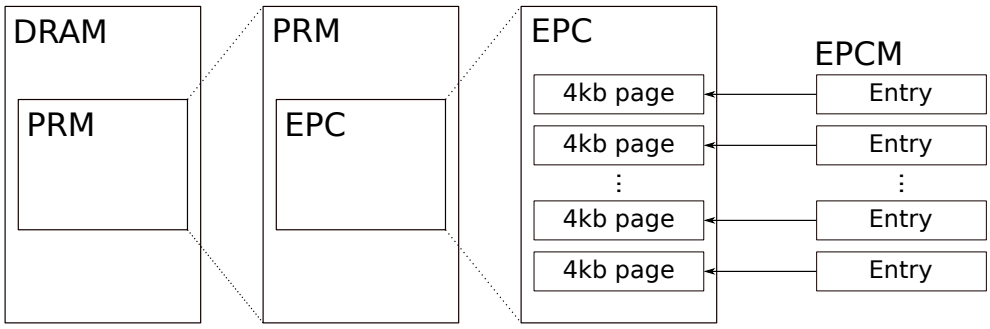


Figure 3.4: Schematic illustration of PRM, EPC, and EPCM [28].

As seen in Figure 3.4, the PRM is a subset region of the DRAM accessible only by the system software. All data and metadata related to an enclave is stored in the EPC, which is further divided into 4kb pages. By having several 4kb pages, multiple enclaves can be used. Control information belonging to a single enclave is stored in the SGX enclave control structure (SECS), which is part of a enclave page cache. Using a hardware memory encryption engine (MEE) protects against unprivileged EPC accesses [104]. As the system software has to assign EPC pages to the enclaves and the system software cannot be fully trusted, the EPCM performs several checks [28].
Intel SGX introduces new instructions for creating, managing, and loading data to the EPC. These instructions are only accessible for the operating system kernel.

SGX Software Architecture

In a typical system, multiple enclaves can be started and managed by the system software.

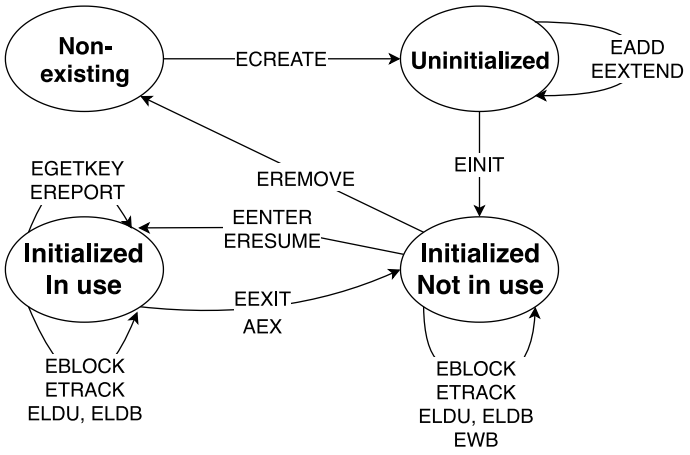


Figure 3.5: Life-cycle of an SGX enclave [28].

In Figure 3.5, the life-cycle of an SGX enclave is shown. Creating an enclave can be initiated by the system software using the *ECREATE* instruction. This instruction copies the starting information provided by the system software to the EPC page which is then used as an enclave control structure. Now, the enclave is in the uninitialized state, where the system can load data and code using the *EADD* instruction. After this step, an *EINIT* token has to be requested by starting the launch enclave provided by Intel. All enclaves not created by Intel have to use this special enclave to request the initial token. Only if a valid token is given to the *EINIT* instruction, the enclave is marked as initialized. Now, software deployed into the enclave can be executed. After finishing the task, the *EREMOVE* instruction deallocates EPC pages and the enclave gets destroyed.

3.1.3 Apple Secure Enclave Processor

With the launch of the iPhone 5S, Apple introduced the Secure Enclave Processor (SEP) in late 2013 [4]. Compared to Intel SGX and ARM TrustZone, Apple's SEP is not part of the main processor. Instead, it is a dedicated security coprocessor placed into the main SoC. In the iPhone, the secure processor is used to store sensitive data like fingerprint identifiers and cryptographic keys. As the coprocessor is an independent ARM coprocessor, even code executed with the highest privilege on the main core has no access to this enclave.

Hardware Architecture

As described in [7], the SEP runs independently of the application processor (AP). This allows the secure enclave processor to run independently from the rest of the system. The only possibility for the AP to communicate and interact with this core is through a clearly defined mailbox system. To submit a message to either the AP or the SEP, data is written to a specific memory region and an interrupt is triggered. To avoid denial-of-service (DoS) attacks, a hardware filter detects anomalies and blocks malicious interrupts [61]. To provide a broad range of security features, the SEP has access to dedicated peripherals like security fuses, cryptographic engines, and random number generators. To provide a high level of security, the AP cannot access these peripherals directly, a request to the secure processor has to be sent through the mailboxes. In addition to peripherals directly embedded into the SoC, dedicated I/O lines for off-chip peripherals are available. However, the security co-processor also shares hardware peripherals like the memory controller and the power manager with the application processor.

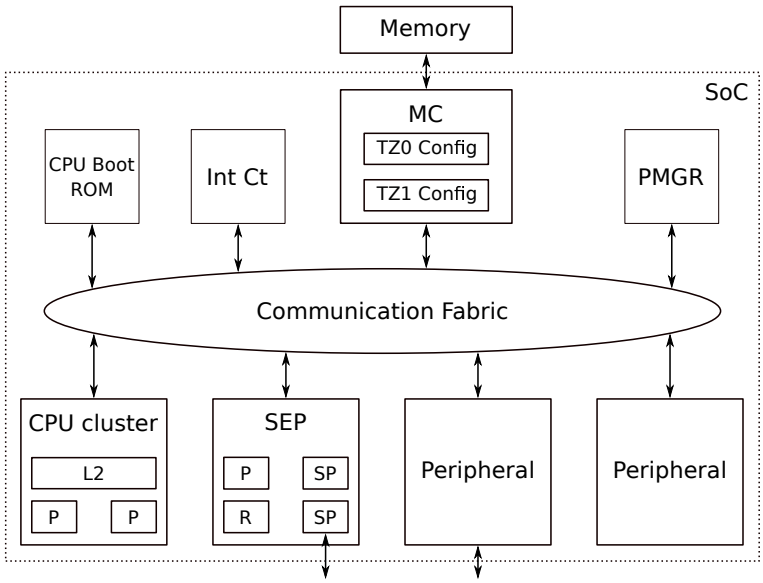


Figure 3.6: Secure enclave processor integrated into an Apple SoC [7].

In Figure 3.6, the overall architecture of an Apple SoC using a secure enclave processor is shown. The application processor cluster consists of one or more processors and an L2 cache. Using the communication fabric, which is some form of bus interconnect, access to the memory controller (MC), peripherals, and the power manager (PMGR) is given. The small ARM processor (P), which is placed in the SEP, has exclusive access to the security peripheral (SP) and a secure boot ROM (R). The memory controller allows the secure processor to configure trust zone regions. For some devices, Apple even encrypts the external RAM using AES [61].

Software Architecture

To prevent an attacker from booting his own, tampered operating system, Apple uses the SEP and AP for a secure boot procedure. Secure boot ensures that only a verified operating system is executed on the application processor and the secure processor. After the device is turned on, first the boot ROM of the application processor gets executed. This code releases the secure processor from reset and configures the two memory zones *TZ0* and *TZ1*. As the SEP cannot trust the main processor, the processor checks the configuration of its secure memory region *TZ0* by polling the hardware registers of the memory controller. Now, only the secure memory processor has access to the memory region configured in *TZ0*. However, an adversary with physical access to the device still could tamper the data. To mitigate this issue, Apple uses on-the-fly AES memory encryption to protect memory. After memory setup, the AP copies the SEP firmware to the secure processor. Now, the signature of the image and several security fuses get verified and finally, the so-called, SEPOS gets executed. This operating system is based on the L4 microkernel, which is a kernel optimized for embedded systems [61].
The SEPOS offers various drivers for accessing a true random number generator (TRNG), the AES engine, and other security peripherals. As stated by Apple, the secure processor is responsible for handling device unlocking using FaceTime and TouchID, secure boot, and data encryption and protection [5].

3.1.4 Sanctum

The last TEE concept providing strong software isolation introduced is Sanctum [29]. Compared to ARM TrustZone, Intel SGX, and Apple's SEP, this trusted execution environment is fully open-source. The proof-of-concept design is integrated into the Rocket RISC-V core and can be freely downloaded and assessed. Furthermore, Sanctum includes some software-based side-channel attacks in its threat model.

Hardware Architecture

Similar to TrustZone, Sanctum introduces a security monitor running on the highest privilege mode. The enclave memory, which is configured by the operating system and checked by the security monitor, is used as private enclave storage. As Sanctum provides support for multiple enclaves running in parallel, metadata for each enclave is stored in specific memory regions and maintained by the security monitor. In contrast to Intel SGX, Sanctum flushes the L1 cache and the TLB when switching between the enclave and non-enclave world [29].

Software Architecture

The programming model of Sanctum is similar to the SGX model. Sensitive code and data is shifted to the enclave and communication is achieved by using well-defined interfaces.

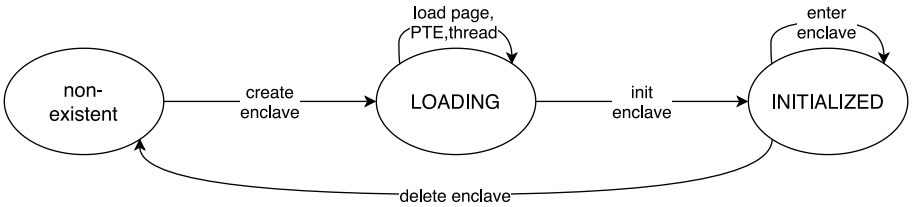


Figure 3.7: Life-cycle of a Sanctum enclave [29].

Again, dedicated instructions are available to create an enclave and to initialize its metadata structure. As seen in Figure 3.7, in the *LOADING* state the memory regions, page table entries and enclave threads are established by the security monitor. In the *INITIALIZED* state, the measurement hash is generated and the enclave threads are started. Similar to the Apple secure enclave processor system, the communication between enclaves and host operating system is done using a mailbox system [29].

Chapter 4

Trusted Execution Environment Features

This chapter summarizes the requirements and desired features of trusted execution environments embedded into processors. For this reason, we are comparing isolated enclave solutions from different manufacturers and identify potential weaknesses.

4.1 Isolation

The fundamental property of any trusted execution environment (TEE) is the isolation between the trusted environment and the remaining part of the processor. As already mentioned in Chapter 3.1, the vast codebase executed in the application processor offers a broad attack vector. In this rich execution environment, the security of sensitive code can be threatened in different ways. Even when the application is programmed in a secure way, the environment in which the program is executed still can be attacked, and information can leak. To mitigate this attack scenario, the concept of trusted execution environments was introduced. By creating an isolated, separated area, the security-sensitive application can be executed without any interference of the potential insecure operating system. In practice, security-sensitive code, shared libraries, or a small, secure operating system are placed into the enclave and executed in the protected area. The different enclave approaches introduced in Chapter 3 are offering isolation in different ways. In ARM TrustZone, isolation between the secure and non-secure world is guaranteed using the non-secure (NS)-bit. This bit indicates, in which security domain the operation is performed. Since all internal components are supporting the NS-bit, two virtual environments are created on a single, physical core. ARM processors with TrustZone support even provide two MMUs for each domain to securely protect memory.
Compared to ARM, Apple creates the isolation between the two security domains by placing a dedicated, small ARM coprocessor to the system. Both, the fast application processor offering a rich set of features and the small secure processor offering security services are entirely independent of each other and are therefore running simultaneously. The shared physical memory is divided into two trusted zones for each processor by the memory controller. The isolation property of the secure enclave processor is guaranteed by using a second, dedicated security processor and a clearly defined interface between the main application processor and the Secure Enclave Processor (SEP).
In Intel's Software Guard Extension concept, isolation between the untrusted operating

system and the secure enclave is solved differently. Using memory encryption and access control, an isolated region for sensitive operations is created [91]. This isolation is achieved by using hardware features directly embedded into the processor and even restricts access to the enclave from privileged software like the kernel itself [89].

To summarize the isolation property of trusted execution environments, all enclave solutions provided by different vendors are preserving the confidentiality of data used and computed in the enclave. Since the operating system only can access the data using a predefined communication channel valuable information like encryption keys, user passwords, and other sensitive information can be stored securely in an enclave. The second characteristic of the isolation feature of trusted execution environments is the integrity of the code executed in the enclave. Since the program is executed independently of the application processor, no direct or indirect interference is possible.

4.2 Programming Model

Vendors providing TEE features are offering different programming models for their enclaves. On Android-based devices with hardware support for ARM TrustZone, a trusted operating system usually is deployed into the enclave by the device manufacturer. As stated by Google [32], the trusty TEE operating system has full access to the device and is completely isolated from the other operating system and the applications executed there. Secure applications are directly integrated into the OS by the device manufacturer and are considered to be trustworthy. Using a strictly defined application programming interface (API), applications from the normal world communicate with secure services running in the TEE. Secure storage of cryptographic keys and mobile payment are two example use cases for TrustZone on Android. However, as secure applications can only be mounted by the manufacturer, developers and users cannot deploy their binaries into the secure world.
A similar approach is taken with the secure coprocessor used in recent Apple devices. This dedicated coprocessor again runs an own, small operating system called SEPOS in the secure domain and services are directly deployed by Apple [61]. This has the disadvantage that users and developers again cannot mount their own, trusted applications in the secure world. Compared to ARM TrustZone and Apple SEP, Intel SGX allows the application developer to run their code in a secure enclave. By offering a software development kit (SDK) for SGX, developers can move the security-sensitive part of their code in a so-called enclave image [90]. This enclave image is then loaded to the EPC pages using an Intel signed enclave. Therefore, this approach allows programs to flexible outsource critical computations to a secure SGX enclave. However, this flexibility only is given theoretically. When developing SGX applications, the code is first executed in the debug mode. In this mode, debug possibilities are given using a debug interface. When releasing the application, the code should be executed in the release mode. However, in order to execute code in the release mode, Intel requires the developer to acquire a production license for SGX [47].
Most security weaknesses discovered recently in the TrustZone architecture are targeting the communication channel between the secure and non-secure world [78]. In the BOOMERANG attack, a user application executed in the non-secure world can trick a trusted application to modify and read arbitrary memory by sending inputs, which are not correctly validated by the TEE [60]. In 2014, an attack again targeting the interface between the TEE implementation of Qualcomm and the rich execution environment (REE)

operating system was presented [82]. By sending malicious secure monitor call (SMC) requests and exploiting an erroneous bound check, even arbitrary code execution in the trusted execution environment was possible.

4.3 Measurement Hash

In addition to the isolation property of a TEE, the integrity of code and data loaded into an enclave has to be guaranteed. For Intel SGX, an application executed in the insecure world is able to load sensitive code and data to the enclave. Since the operating system cannot be trusted, a tampering attempt initiated by a malicious kernel thread is hard to detect. However, the host application has to fully trust the enclave as sensitive operations, like signing, encrypting, and decrypting sensitive information can be handled by the enclave. For this reason, SGX introduces the measurement hash. This cryptographic hash automatically gets computed over the code and data loaded into the enclave region by the signed Intel loader enclave. In an SGX system, the measurement hash is securely stored in the SGX enclave control structure (SECS). When initiating a communication channel with the enclave, the third party can verify the measurement hash by comparing it to the expected hash. Iff the hash matches, the host application starts to trust the enclave application. This procedure is known as attestation process and can be done locally or remotely by a third party like a server [28]. ARM TrustZone does not natively support attestation, so applications executed in the secure world have to implement their own mechanism if desired [53]. Since hardly any information about the Apple SEP is publicly available, no information about an build-in attestation method can be found.

4.4 Trusted I/0

Computing almost always includes some kind of interaction with the outside world by using peripherals. Not only prominent examples like a keyboard attached to a personal computer or a smartphone with a touchscreen, also complex systems like server's with a network interfaces, heavily rely on information transferred to or from another party. When now considering security aspects of a system using I/O interaction, new attack vectors are revealed. In an usual environment, peripherals are shared between processes using a common software stack. A keyboard connected to a computer can be used to write text in a word processor software, writing electronic mails, or entering the password in a banking application. This flexibility does not only have advantages; an attacker might be able to mount a keylogger on the infected system and obtain all sensitive input entered by the victim. Another possible attack scenario are man-in-the-middle (MITM) attacks. Secrets like personal messages, contracts, and other critical information transferred by the user using, e.g., a network interface are threatened. A malicious MITM attack software deployed by an attacker can intercept this information, tamper it, and forward it to the recipient. This weakness is not solved when executing code in a protected enclave. Undoubtedly, the secure code execution is guaranteed by the enclave, but securely accessing peripherals outside of the enclave is not assured. A malicious adversary can trick the trusted execution environment into using a faked peripheral and obtaining or tampering all secrets. To tackle this weakness, ARM TrustZone natively propagates the NS-bit to all supported peripherals. This is possible because in an ARM system, I/O devices, like a USB keyboard, are usually connected to a hardware driver with AMBA AXI bus support. As explained

in Section 3.1.1, the separation between the secure and non-secure world is achieved by directly supporting the NS-bit in the AXI protocol. An example of a secure mobile payment application using I/O devices is described by ARM in the TrustZone whitepaper [8]. In this scenario, the user input, which consists of a secret PIN code and transaction details, is secured by using a PS/2 controller with AXI support. To hide sensitive information from a malicious adversary, even the framebuffer of the display controller can be divided into a secure and non-secure world. Furthermore, the payment process initiated by the near-field communication (NFC) reader is secured by TrustZone. Apple uses a similar approach with the dedicated secure processor. The secure enclave directly processes sensitive information, like authentication data captured by the fingerprint reader or the camera [6]. Unfortunately, the Intel SGX solution does not natively offer peripheral support. However, as I/O interaction is a frequently used and security-critical task, recent research in this area focuses on enabling trusted I/O paths for Intel SGX. SGXIO [105] enables support for secure I/O communication. In this scheme, a trusted hypervisor and secure I/O drivers are used to establish a secure channel between user applications and peripheral drivers.

4.5 Resilience against Physical Attacks

By creating a safe and isolated environment, security-critical applications can be executed independently from other applications or the operating system. This concept protects code from external logical attacks, as only the communication interface is exposed to untrusted software. However, when evaluating the security of a system and developing the threat model, physical attacks need to be considered. All TEE solutions introduced in Chapter 3 offer a different level of resilience against physical attacks. In the TrustZone whitepaper, ARM categorizes attacks threatening the security of the system into hack attacks, shack attacks, and lab attacks [8]. Hack attacks are defined as classical logical attacks exploiting bugs and scenarios, where an incautious user installs untrusted software. In shack attacks, a semi-professional adversary having physical access to the device under attack uses low-cost hardware to break security primitives. This category covers attacks using debug ports, passively observing physical connections to external modules, or actively manipulates external bus lines. Professionals in a laboratory environment can conduct the most powerful attack. Here, the adversary almost has unlimited possibilities to break the system. One possible scenario includes decapsulating the chip and observing or manipulating internal signals of the chip. As stated by ARM, lab attacks decapsulating the chip are out of scope in the TrustZone threat model [8]. Furthermore, ARM TrustZone does not natively offer countermeasures against any physical attacks. However, when moving all sensitive information into memory integrated into the processor chip and when the chip package is considered to be secure, many attacks can be mitigated [28]. Nevertheless, fault injection attacks as well as side-channel attacks like differential power analysis (DPA) and template attacks are still an immense threat [24]. A recent publication showed that threatening the security of TrustZone with fault attacks even is possible using the energy management unit [93]. Similar to ARMs solution, the attacking model of SGX also does not cover physical attacks using invasive methods and side-channel attacks. However, data stored in external memory by an SGX enclave is encrypted using the memory encryption engine (MEE). Intel also explicitly states that cache timing attacks are out of scope in the threat model. The resilience of the Apple secure enclave processor depends on the dedicated processor used for the enclave. Since SEP is proprietary, only little information about its security is publicly known. However, Apple states in their security manual that

some kind of physical tamper detection is integrated [6].

Chapter 5

Trusted Execution Environment with Secure I/O

In this chapter, we introduce our novel trusted execution environment scheme. First, we give an overall overview of the design, including its properties. Then, in the threat model, we summarize the security guarantees of the proposed trusted execution environment (TEE). In the last section, we explain the programming model to use the isolated enclave solution.

5.1 Concept

Our TEE solution borrows concepts from enclave designs introduced in the past and proposes new features to enhance the overall security and to provide stronger security guarantees. In general, an enclave can be built by using a dedicated coprocessor like the Apple Secure Enclave Processor (SEP) or by integrating the enclave in the main processor using different isolation techniques and hardware features. In an integrated enclave approach, only a small overhead in terms of physical chip area is generated compared to a processor without any enclave. When analyzing the ARM TrustZone concept, the area difference results from the non-secure (NS) bit mechanism, the second MMU for memory isolation, and the integrated security monitor. Apparently, a second, dedicated security chip generates much more overhead as all components of a processor have to be physically available twice. For our enclave solution, we reintroduce the concept of the minon cores established in Section 2.3.1. Similar to Apple's approach with the secure enclave processor, one or multiple minon cores are used in our scheme as enclaves. Since these dedicated coprocessors are fully fledged, independent cores and directly embedded to the main system-on-chip (SoC), the isolation property needed for a TEE automatically is given. Security always requires some form of trade-off. Therefore we argue that the additional chip area required for these enclaves is reasonable given to the improved security features. Furthermore, having two independent core architectures in one system-on-chip allows the computer architect to design the cores with a different focus. Whereas the main applications processor usually provides a rich set of features, the secure coprocessor is designed to meet security goals. As stated in Section 3.1, secure enclaves are mostly used to execute security-critical code, sensitive libraries, or a small, secure operating system. Due to these requirements and to keep the physical chip area overhead small, we propose to use small cores as enclaves. When integrating dedicated countermeasures against physical attacks into a processor architecture, usually a performance and area penalty is added. Since we are using small

enclave systems and high performance is not the main goal of these subsystems, system designers can embed countermeasures against physical and logical attacks directly to the processor.

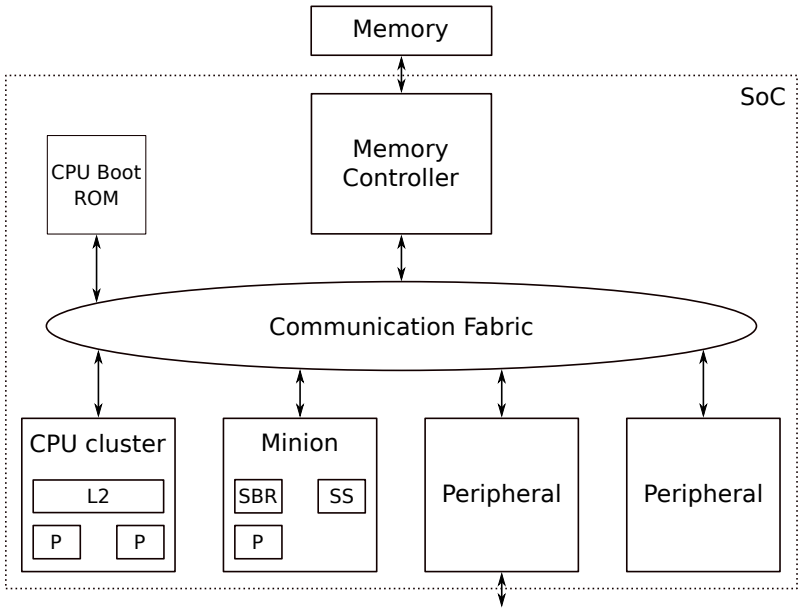


Figure 5.1: Proposed secure enclave system integrated into a SoC.

Figure 5.1 depicts the proposed trusted execution environment solution integrated into a system-on-chip design. The CPU cluster, which consists of one or multiple cores, can be designed to provide low-performance capabilities like a microcontroller or high-performance speed using out-of-order execution and Linux support. To start the application processor, a CPU boot-ROM is usually integrated into the system-on-chip design. Using a communication fabric, the CPU cluster is able to access the external memory and other peripherals. In our proposed setting, the minion block represents a fully fledged, independent computing core with all necessary components. In addition to the core consisting of the CPU pipeline, a load-store unit (LSU), data and instruction cache, a secure boot-ROM and a secure storage exclusively accessible by the minion is available.

5.1.1 Trusted I/O

Creating an isolated execution environment for safe code execution is the first step in building a secure enclave scheme. An enclave using an attached keyboard to process secret information needs to be sure that the incoming message stream is generated by the correct peripheral and is protected from other parties. For this reason, we introduce the concept of trusted I/O paths in our design.

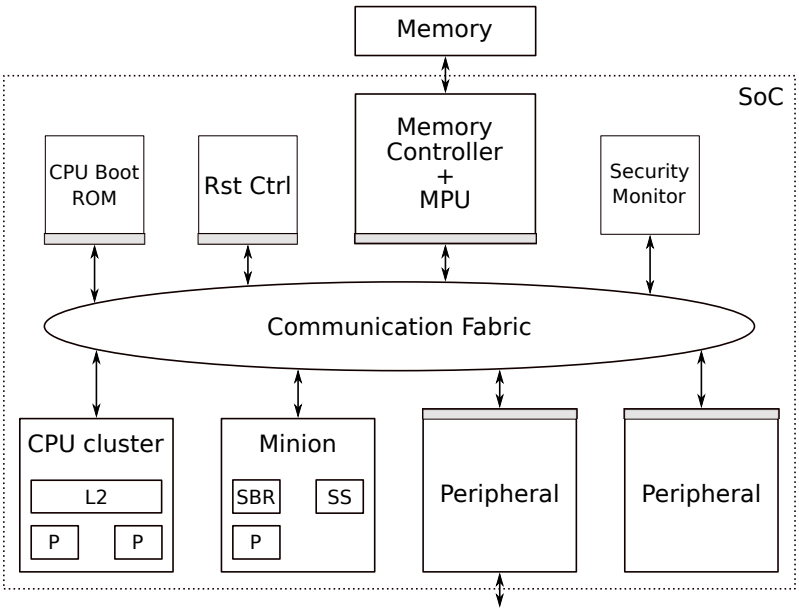


Figure 5.2: Proposed secure enclave system with trusted I/O integrated into a SoC. The hardware filters are highlighted in grey.

A trusted I/O path generates a secure communication channel between the hardware peripheral driver and the initiating party, e.g., the processing core. Following the idea of minions, one or multiple minions can be mounted to act as secure enclaves. In our design, we introduce a mechanism to share trusted I/O paths between the different parties. Furthermore, we do not distinguish between an enclave and the main processor. To enhance the overall security, even the main processor can use these trusted I/O paths. To realize this architecture, each peripheral consists of a lightweight protection mechanism wrapper. This hardware wrapper, which is marked in grey in Figure 5.2, implements a filter functionality similar to a firewall directly in hardware. Each of these hardware firewalls internally contains a small memory element indicating the current owner of the corresponding peripheral. To identify an enclave or the application processor, we assign each party of the system a unique identifier. This identifier directly is embedded in the design stage of the SoC and cannot be changed once the system is tapped out. Whenever a request is sent from one party to a peripheral, the identifier is transmitted over the communication fabric. The I/O wrapper then analyzes the identifier integrated into the request and grants or denies access to the peripheral driver.
In Chapter 6, an example on how to integrate the identifier with only a small performance and area penalty using the AXI4 bus is shown. Furthermore, our trusted I/O path mechanism integrates a security monitor in hardware to the SoC used as the root of trust. This module manages the access privileges of all computing units interacting with the peripherals.

Table 5.1: Access control structure integrated into the security monitor.

Peripheral	**Permission [IDs]**	**Claimed [ID]**
Peripheral 0	0,1,2	2
....	...	...
Peripheral n-1	1	-

Internally, the structure shown in Table 5.1 is stored in the security monitor. Each row of the table represents one out of n peripherals available in the SoC. The first column contains a list of identifiers with access privileges to this peripheral. Before using a particular peripheral, an enclave or the application processor has to send a claim request to the security monitor (SM). The security monitor checks the access privileges and grants the request by setting the peripheral as occupied in the second column. Moreover, the security monitor writes the identifier to the memory of the corresponding peripheral wrapper. When one enclave claims, e.g., the network interface controller (NIC), this enclave now has exclusive access to this peripheral. In our scheme, only a single identifier can be stored in the peripheral firewall mechanism. A peripheral claim request by another party even registered in the first column of the corresponding peripheral fails because the peripheral is already claimed. When another party tries to read or write to this device, the peripheral wrapper notices a security violation and physically terminates the request on bus protocol level. In order to claim a peripheral, the requesting party has to be registered as a legitimate participant by storing the identifier in the permission column of the peripheral. As the decision, which party can access which peripherals is critical, only a designated party is allowed to modify this column. From now on, we call this party the SM master, all other participants are called SM slaves. Technically, this concept is constructed using a dedicated master identifier register stored in the security monitor. This register is initialized during the design phase of the system-on-chip. To provide flexibility, the SM master can modify this register and nominate a other party to inherit the master privilege. Once a slave finishes using a specific claimed peripheral, a request to the security monitor is sent. The security monitor deletes the table entry and also the identifier from the security monitor; thus, making the peripheral available again. In this scheme, a malicious slave can easily occupy one or multiple peripherals and creates a powerful denial-of-service (DoS) attack. To prevent this attack vector, the security monitor master can withdraw a claimed peripheral from a slave. When the master plans to revoke access to a device, the corresponding slave gets notified using dedicated interrupt lines. After a certain timeout, the security monitor deletes the identifier entry in the table structure as well as in the peripheral wrapper itself. Now, the peripheral is available again.

5.1.2 Memory Protection

The scheme introduced in Figure 5.2 only contains a single external memory interface. For small data storage, various block RAM modules integrated to the chip fabric are available in the system architecture. Since these modules are connected using the communication fabric, a block RAM module is also protected using the same access control wrapper like any other peripheral. Therefore, parties can claim and release these modules like any other peripherals. However, for storing larger amount of data or communication between the parties, an external memory is needed. Since all parties can access the shared, external

memory using the system bus, a protection scheme needs to be integrated. For this reason, we are deploying a memory protection unit (MPU) directly to the system. The memory protection unit divides the physical memory into several memory regions, which can be accessed by one party exclusively or can be shared among one or multiple parties. To provide a consistent scheme across the entire system, the MPU configuration interface can be claimed like any other peripheral as discussed before. In most cases, the memory allocation is initiated by the security monitor master entity.

5.1.3 Communication

One key element of each trusted execution environment scheme is the communication channel between the secure and non-secure world. When using the TEE to outsource the encryption or decryption of a message with a secret key stored in the enclave, information needs to be exchanged between both security domains. In a more complex use case of an enclave, a security-sensitive library offering different services to other parties is placed in the secure domain. Here, excessive information exchange is performed between all parties operating in different security levels. For this reason, a shared memory region is established using the MPU. Due to the flexibility of our scheme, the enclave system can be used in different use case scenarios. Therefore, the software executed in the application processor and the enclave agree on a communication interface using the shared memory region themselves.

5.2 Threat Model

The features introduced in Section 5.1 are creating a powerful trusted execution environment by improving several weaknesses of other TEE solutions. Our proposed system consists of one security monitor master entity and one or multiple slave parties. The threat model assumes that all participants are not trustworthy, only the security monitor master can be trusted to a certain extend. Software, which is executed on one entity, is entirely isolated from the other parties. The only possibility to interact with an entity is by using the dedicated communication interface. Once the master establishes a slave entity, e.g. loading security-sensitive code to an enclave, the code is executed securely. This strong isolation is garantueed by using dedicated processing units directly embedded into the SoC.
Furthermore, our threat model also considers attacks based on transient executions. Our proposed scheme consists of one application processor and one or multiple hardware enclaves. As stated in Section 5.1, usually security-sensitive code pieces, libraries, or a small operating system are executed in these enclaves. Since these tasks only need limited computing resources, we propose to use a simple system as an enclave. In Chapter 6, we are constructing our proof-of-concept design with the small RI5CY core introduced in Section 2.3.2. This core is based on a simple in-order, 4-stage pipeline. Hence, attacks exploiting building blocks of out-of-order execution and speculative execution schemes cannot be performed on these enclaves. Nevertheless, the main application processor still is vulnerable against attacks based on transient executions when using a high-performance CPU with speculative execution support.
The threat model of Intel's SGX TEE solution explicitly excludes side-channel attacks, even side-channel attacks which can be performed using software only [29]. However, research in the past showed that this attack category still is an immense threat against the overall security of a system. One example of a side-channel attack using software only are cache

attacks. In 2017, Götzfried et al. showed an attack targeting an AES encryption running in an SGX enclave. However, not only Intel's enclave solution is vulnerable against cache based attacks, also in ARM TrustZone based enclaves cache attacks were already conducted [56]. Our threat model covers these attacks as secrets easily could be extracted using cache timing attacks [23]. In our design, only a single task is executed on the dedicated minion processing units. Since each independent enclave system consists of its own data and instruction cache, our secure enclave scheme automatically mitigates this attack vector.
In contrast to other trusted execution environments, our threat model also covers secure handling of peripherals. By creating a secure channel between an entity and the peripheral itself, a trusted path between the two parties is created. Our threat model considers a malicious piece of code, like an operating system, running on the application processor or an enclave. This entity tries to threaten the confidentiality and integrity of an asset stored or handled by a peripheral. One possible attack scenario can be a keylogger executed as an application in the operating system trying to obtain secret information entered to the keyboard and processed by an enclave. The system architecture mitigates these attacks since all peripherals are exclusively assigned to one entity. A party trying to access a certain peripheral, first has to claim this device by sending a request to the central security monitor. When the requesting party possesses the access rights and the peripheral is not claimed already, exclusive access to the device is granted. Any other party trying to read or write to this device fails, because the build-in security mechanism of the peripheral automatically terminates the bus session. The filtering mechanism is based on unique identifiers. Forging these identifiers is not possible, because each enclave and the main processor gets assigned an identifier during the design phase of the system. The identifier directly is integrated into the hardware of the bus interface of each entity and gets transmitted automatically during a bus request. Furthermore, our attack model also considers a malicious slave entity trying to threaten the availability of the system by occupying one or multiple peripherals permanently. Our scheme is designed to be a cooperative system, an entity finished using a peripheral should release it voluntary. Nevertheless, a bug in the enclave software or an attacker intentionally not releasing a peripheral could be a threat. For this reason, the security monitor master is able to withdraw access to any peripheral by sending a request to the SM. However, this feature can also be exploited by the master. Our threat model considers a master revoking access to a peripheral currently used by a slave and claiming this peripheral. As still sensitive information can be stored on this device the master would have full access to these information. To mitigate this security issue, the security monitor triggers the interrupt line of the entity currently owning the peripheral. When the interrupt is received, a clean-up functionality implemented by the entity can be executed. Due to this mechanism, security-critical memory regions or similar assets can be erased by the peripheral owner. After a timeout, which is defined during the design step of the system, the access to the peripheral is withdrawn by the security monitor.
Furthermore, due to the concept of ownership transfer, our system can be used in a flexible way. When specifying the system architecture, the designer designates one entity as the default security monitor master. During runtime, the master can initiate an ownership transfer and nominates a new security monitor master. An example demonstrating the flexibility of this technique is shown in Chapter 7.

5.3 Programming Model

The programming models of TEE solutions presented by other vendors offer only a certain amount of flexibility. In ARM TrustZone, the device manufacturer only can deploy code into the secure enclave and user application can use these services to some extent. A similar approach is taken with the secure enclave processor used by Apple. In Intel SGX, the enclave system is designed to execute code deployed by developers. However, there is one major restriction: developing and mounting secure code into the enclave is only possible using the debug mode. In this mode, ring 0 processes still can access the enclaves using the debug interface. Therefore, only limited protection against attacks can be guaranteed. To deploy code in the release mode, a license has to be requested from Intel.
Our secure enclave concept provides flexible use cases by using the SM concept. As stated in Section 5.1.1, one party is the dedicated security monitor master. This master can claim and release any peripheral within the system. Additionally, the SM master can manage access permissions of each peripheral device. Using this approach, the following two scenarios are possible.

5.3.1 Minion as Security Monitor Master

When assigning the security monitor master privilege to the minion, this core can run entirely independently from the rest of the system. Since the subsystem can fully control the security monitor, arbitrary peripherals can be claimed by this entity. Code is either executed from the internal storage integrated into the minion or from memory attached to the communication fabric. This setup allows the system engineer to develop various security schemes.
One example use case is the handling of the user authentication procedure. Since the processing of sensitive information like user passwords, or biometric data is usually a security-critical task, protection from tampering attempts is needed. By exclusively claiming the peripherals responsible for entering the secret, software executed on the main application cannot intercept this information. Additionally, processing of the information is done in the isolated space. For this reason, the secret never leaves the secure domain. Software running in the non-secure world can use the common communication channel to obtain if the user is authenticated and authorized for a certain task.

5.3.2 Application Processor as Security Monitor Master

In the second scenario, the security monitor master privilege is given to the main application processor and the minion subsystem can be used as an enclave. Here, the application processor first grants the minion access permissions for specific peripherals. Then, the security-critical code is moved to a memory, which is connected to the communication fabric. By claiming the minion interface, the application processor can configure the code address and start the subsystem. Now, code securely is executed in the enclave and data can be exchanged using the communication interface.

Chapter 6

Design of a TEE

To demonstrate the feasibility of the scheme proposed in Chapter 5, we integrate our enclave system, the central security monitor module, and the peripheral wrappers into a RISC-V-based system-on-chip design. First, this chapter introduces step for step our trusted execution environment scheme with secure I/O based on the lowRISC architecture presented in Section 2.3.1. Then, we demonstrate how to integrate the access permission scheme into the bus architecture.

6.1 System Architecture

In this thesis, we integrate our novel trusted execution environment (TEE) scheme and the secure I/O mechanism into an existing RISC-V-based platform. The lowRISC system is used as a basement for our implementation since the lowRISC platform is well documented and actively maintained. Having a working proof-of-concept design of our proposed countermeasure allows us to study the security guarantees and the resulting overhead in terms of area and speed. In the end, our prototype RISC-V system is capable of booting Linux on a Xilinx Kintex 7 FPGA development board.

6.1.1 Communication Fabric

The core element of each system-on-chip (SoC) design is the communication fabric enabling communication between the computing units and other modules like peripherals. For our proof-of-concept design, we are using the popular AMBA AXI bus protocol.

AMBA AXI Protocol

The Advanced eXtensble Interface (AXI) [10] protocol, which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification, is a high-speed bus used as communication fabric in many SoC systems. As stated in the AXI protocol specification manual, the bus system is offered in three different versions. The default version, which is called AXI4 in the latest version, combines a rich set of features with a high-bandwidth data interface. For data streams, the AXI4-stream interface is available. When designing a lightweight system, the AXI4 subset AXI4-lite is recommended to use. ARM released the AMBA AXI specification royalty-free. For this reason, system designers freely can implement their own version of the bus protocol to build powerful systems. Due to these features, AXI is a de-facto standard in many SoC architectures [9].

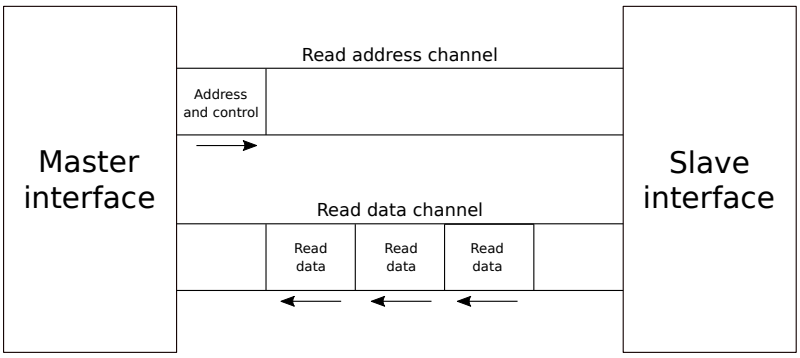


Figure 6.1: AXI read access initiated by the master [9].

AXI is a classical point-to-point protocol consisting of one or multiple masters and slaves. In this bus system, only the master interface can initiate a write or read transfer. The bus protocol consists of the read address, read data, write address, write data and write response channel. In Figure 6.1, a read request and in Figure 6.2, a write request initiated by the master is shown.

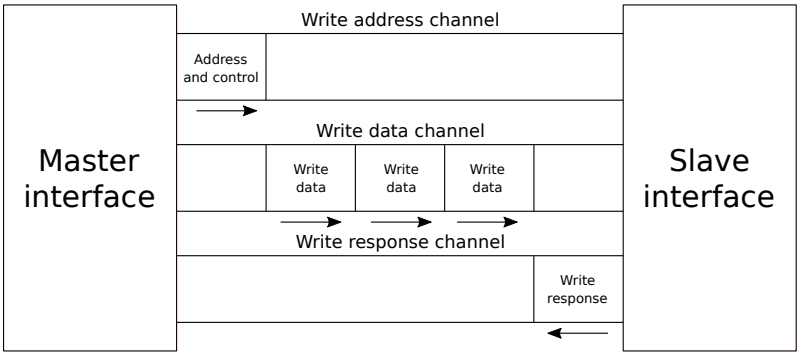


Figure 6.2: AXI write access initiated by the master [9].

The address and other control information is transmitted from the master to the slave over the read or write address channel. When the slave detects a read access request, the corresponding data is sent to the master using the read data channel. In a write request, the data is written to the slave using the write data channel. The slave acknowledges the written data using the write response channel.

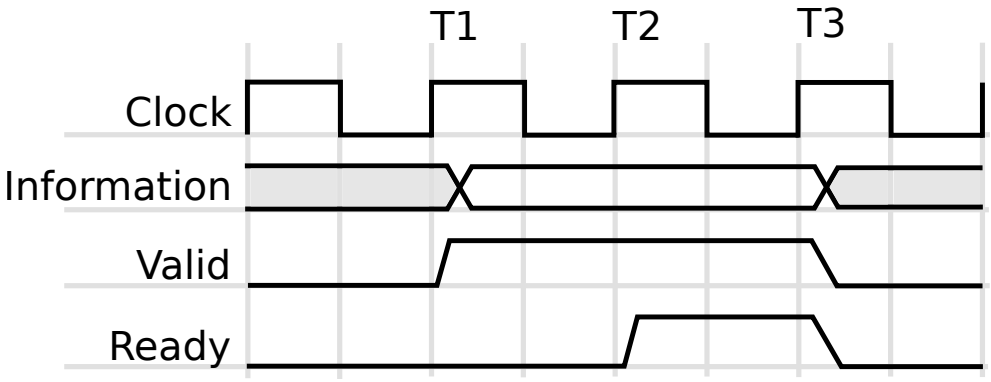


Figure 6.3: Handshake procedure in AXI [9].

The AXI bus protocol specification uses five channels to transmit address, control information, and data. When transmitting data from the master or the slave, all channels perform a handshake between the sender and the receiver. An overview of the handshake timing is given in Figure 6.3. Each of the five channels consists of a *VALID* and *READY* signal and the information channel. The party initiating the handshake procedure first assures that valid information is applied to the bus. Then the initiating participant sets the *VALID* signal to high. When the receiving party is ready to receive data, the *READY* signal is asserted by this participant. Now, information is transmitted. In a write request initiated by the master, the memory address is transmitted using the *AWADDR* signal and control information, like the burst length, is transmitted over additional control signals. Then, the master asserts *AWVALID*. The slave accepts this address by asserting the *AWREADY* signal. The actual data transfer is conducted using the write data channel using the same handshake procedure as described before. By using the *WLAST* signal, the last data transfer is indicated. The slave uses the write response channel to signal if the write request was successful. Using the handshake procedure allows the master as well as the slave to have control over the rate data is transmitted [9].

AXI Interconnect

As seen in the previous section, AXI is a point-to-point based bus system. To connect multiple masters with multiple slaves, the AXI bus system specification defines a crossbar interconnect mechanism. The crossbar receives a transfer request initiated by the master and routes it to the receiving party. Using the address information, the interconnect determines the receiving party and transmits the request to the slave. Furthermore, AXI supports out-of-order transactions. Due to performance reasons, the interconnect re-orders these incoming requests by using the transaction identifier and then forwards them to the receiving party.

AXI4 as Communication Fabric

Due to the flexibility of the AXI4 bus protocol, we are using AXI4 as communication fabric in our system-on-chip design.

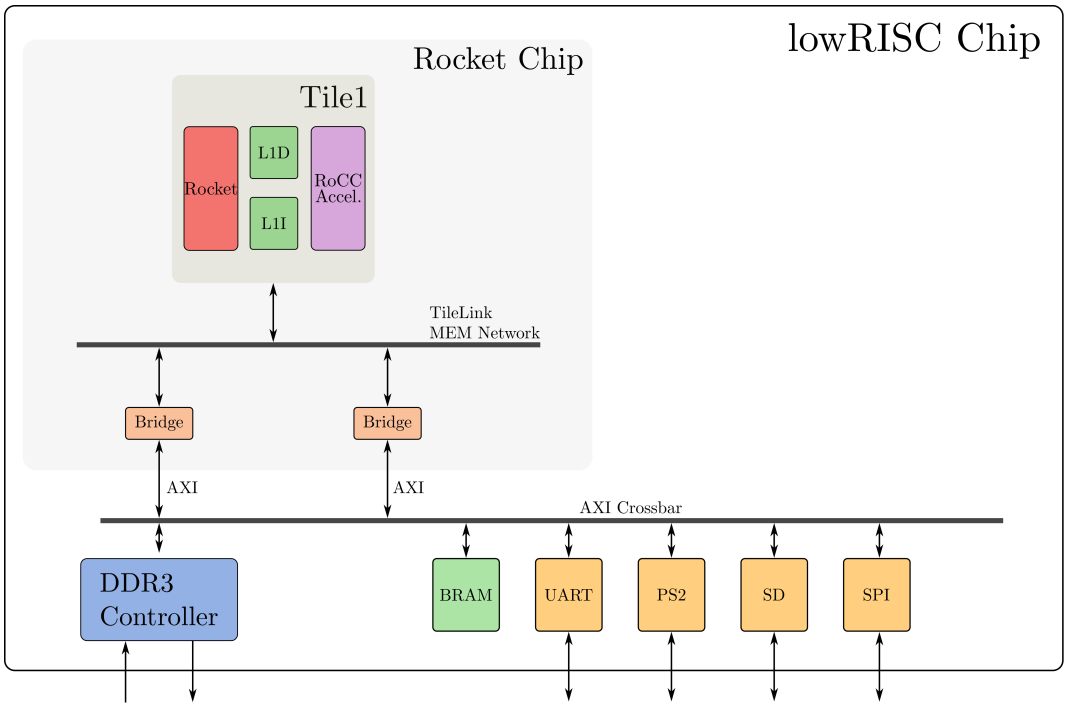


Figure 6.4: Initial system-on-chip design with extended bus architecture.

In the initial lowRISC design, as is depicted in Figure 2.9, an independent AXI interface for the DDR3 controller and the peripheral controller was used. Instead of having two separate busses, we combine these two systems by using a single AXI interconnect module. Furthermore, we depreciate the peripheral SoC and connect each peripheral module individually to the system bus. Now, all peripherals, including the DDR3 controller, have access to the bus using a dedicated AXI interface. The original processing system, which is based on the 64-bit RISC-V Rocket Core, is not modified at all. Figure 6.4 depicts the first changes made to the overall architecture.

6.1.2 Secure Enclaves

As stated in Chapter 5.1, for our enclave design, we reintroduce the concept of minions. In our design, one minion core is attached to the system using the AXI bus. With this approach, the minion, as well as the application processor, can access all other participants in the design. Due to the simple design of a minon and the flexibility of the AXI protocol, multiple minon cores can easily be deployed to the SoC. However, in this prototype implementation, we only integrate one minion to the chip.

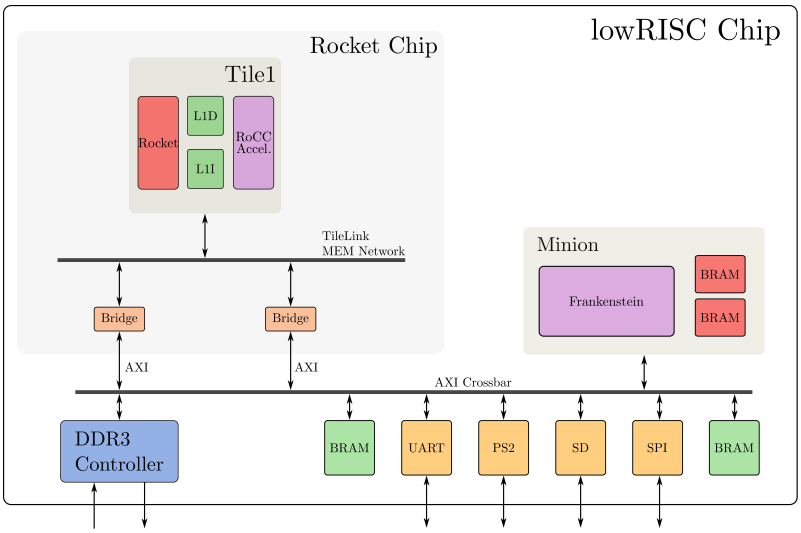


Figure 6.5: Initial SoC design with extended bus architecture and Frankenstein as minion core.

In Figure 6.5, the overall architecture containing the minon core attached to the AXI network is shown. In this system, both computing units have access to the two internal BRAMs and the external memory.

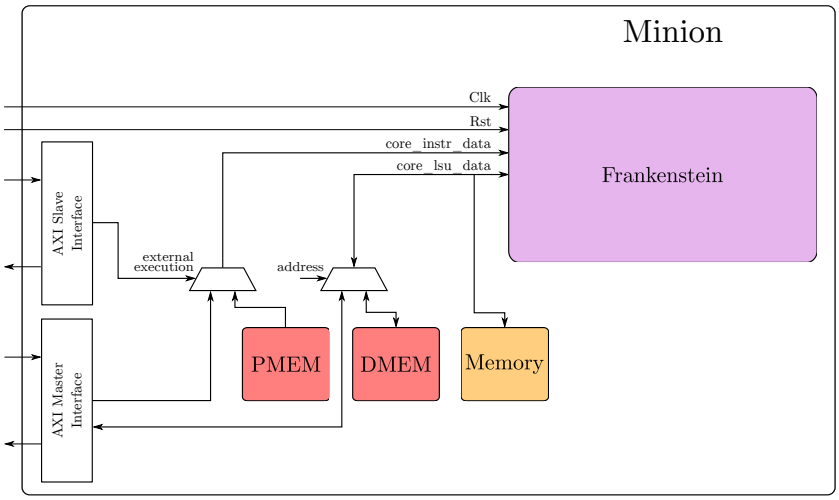


Figure 6.6: Design of the minion subsystem

In Figure 6.6, the internal components of a minion subsystem are depicted. The Frankenstein CPU, as introduced in Section 2.3.3, is the heart of the subsystem. Since this core consists of all necessary components a CPU needs, Frankenstein is capable of operating fully standalone. Furthermore, the Frankenstein core provides several hardware countermeasures to mitigate fault attacks. In the default setup, the subsystem can access the program memory (PMEM) with read-only permissions using the core instruction data interface. By using the core's load-store unit (LSU) data interface, read and write access to the attached data memory (DMEM) is given. As already stated, the minion subsystem

can access the AXI bus by using a dedicated AXI master interface. Again, a program running in the minon can use the core's LSU data interface and a specific address region to interact with peripherals attached to the bus. In our design, the minion core also offers the possibility to act as a secure enclave. The application processor can transfer security-critical code to one of the two BRAMs shown in Figure 6.5. By resetting and triggering the external execution mode of the minion, the code from the external BRAM then is executed. Note that only Frankenstein has access to its internal program and data memory. Furthermore, using the LSU's data interface the minion can access a internal attached BRAM.

6.1.3 Secure I/O using the AXI Protocol

The secure I/O approach introduced in Section 5.1.1 requires hardware support from the communication fabric to provide minimal overhead. To identify the parties, the communication fabric needs to offer some kind of authentication. Due to the missing authentication mechanism in AXI, this chapter introduces our solution to identify parties in the system. Furthermore, we show how the access control procedure is integrated into the peripheral modules.

AXI Identification

Our scheme requires clear identification between all participating parties in the bus system. However, the AXI bus scheme does not natively support identification between two parties, because the address is used to determine the slave interface only. When having a system-on-chip setup consisting of multiple master and slave interfaces and a crossbar interconnect, the slave cannot determine the origin of the incoming request. To introduce unique identification of masters, we are using user-defined signals in the bus protocol. According to the AMBA AXI specification, these signals are part of the AXI bus protocol and are available in all five channels [9]. Since the AXI specification allows designer to use these signals freely, we are introducing identifiers using the **USER* signals. With a bit width of 4-bits, up to 16 masters can be uniquely identified. By integrating the identifier directly to the hardware logic of each bus interface, modifying the unique identifier is not possible.

AXI Peripheral Wrapper

Building a secure path from one entity to the peripheral module requires hardware features integrated to the overall architecture. The first building block of our proposed scheme is the central security monitor module managing access permissions. However, the actual access control check is performed directly by the peripherals. To keep the development of peripherals simple, our scheme provides a wrapper module performing the access control verification. This wrapper module is located between the AXI bus and the peripheral AXI interface. On each incoming handshake request, the identifier transmitted over the bus is compared to the identifier stored in the peripheral wrapper module. When both identifiers are equal, access is granted and the wrapper module connects the peripheral AXI interface physically to the AXI system bus. An access permission violation is detected when the identifier stored in the wrapper differs from the identifier transmitted during the handshake. Then, the wrapper module terminates the AXI request according to the bus specification.

Table 6.1: AXI RRESP and BRESP status encoding [9].

RRESP[1:0] **BRESP[1:0]**	**Response**
0b00	*OKAY*
0b01	*EXOKAY*
0b10	*SLVERR*
0b11	*DECERR*

In Table 6.1 the RRESP and BRESP status encoding is shown. On each read or write request initiated by the master, the slave sends back a response indicating the status of the request. The *OKAY* and *EXOKAY* status are used when the request was fully or partially accepted. When the interconnect cannot determine the corresponding slave interface with the given address transmitted during the handshake procedure, *DECERR* is transmitted using the RRESP or BRESP signal. According to the AXI specification, a slave error *SLVERR* is thrown, when the communication between master and slave was successfully, but the slave wants to return an error condition [9]. Using this error code to terminate a session violating access permissions allows the initiating master to react on this slave AXI error.

6.1.4 Security Monitor

The security monitor is the central access permission management system. As stated in Section 5.1.1, internally a table is used to keep record of the access permissions for each peripheral. Furthermore, the security monitor also stores the current owner of a specific peripheral.

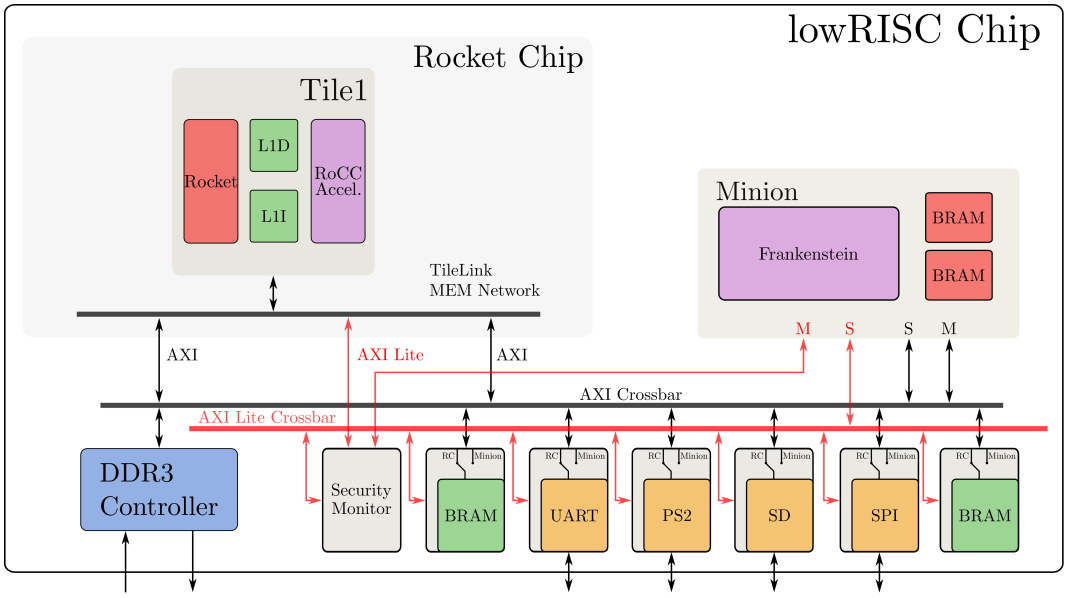


Figure 6.7: Overall system architecture including minion enclave, security monitor and secure I/O paths.

Figure 6.7 depicts the overall architecture, including the central security monitor mod-

ule. This module solely can be configured by the dedicated security monitor master, which is in our design the secure minion enclave. As in the configuration procedure of the security monitor and the peripheral wrappers only a small amount of data needs to be exchanged, we are using the lightweight AXI4-lite protocol. To identify the security monitor master, each participating party offers a dedicated AXI4-lite master interface. This is necessary because the reduced version of the AXI specification does not support user-defined signals. The security monitor itself configures the current owner of a certain peripheral by using a AXI4-lite crossbar.
As illustrated in Figure 6.7, the minion consists of a dedicated AXI4 master and slave interface. The AXI4 master interface is used to access peripherals like the on-chip BRAM. Moreover, the minion also acts as a AXI4 slave. Other participants can configure the minion to boot from the original PMEM or an external memory connected to the AXI4 bus. Since configuring the boot address is security-sensitive, this functionality can be claimed by other parties like an ordinary peripheral. For this reason, a build-in AXI4-lite slave interface is available. To send claim requests to the peripheral manager, the minion uses a AXI4-lite master interface.
In our cooperative system, a entity finished using a peripheral has to release this specific device by sending a release request to the security monitor over the AXI4-lite bus. The security monitor processes this request and checks, if the request transmitter is the legitimate owner of this peripheral. Since each participating party has its own, dedicated AXI4-lite connection, identifying the request transmitter easily can be done. To release a claimed peripheral, the security monitor first updates the internal table by erasing the corresponding entry. Secondly, the security monitor resets the peripheral wrapper register by sending a corresponding message using the AXI4-lite crossbar. Now, other parties with the corresponding access privileges again can claim this peripheral. When transferring ownership from one party to another party, the current security monitor master can use the release all peripheral functionality of the security monitor to again establish a clean system state.
However, when considering a malicious or erroneous enclave occupying a peripheral permanently, the security monitor can be forced by the security monitor master to release this peripheral.

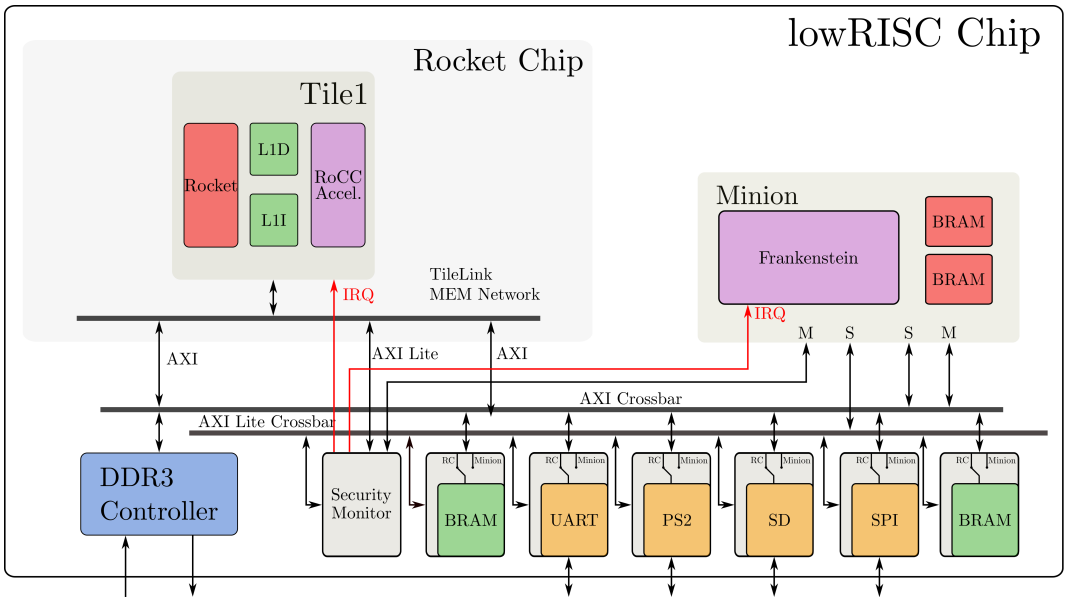


Figure 6.8: Dedicated interrupt lines are used to notify the peripheral user of a pending peripheral release request initiated by the security monitor.

To mitigate attack scenarios explained in Section 5.1.1, the security monitor first triggers a peripheral release request by using dedicated interrupt lines. Now, the peripheral currently claiming the peripheral can call a dedicated clean-up function to release assets safely. After a fixed pre-defined, the security monitor releases the peripheral by updating the internal table and sending a release command to the peripheral wrapper. This timeout is permanently programmed into the security monitor hardware logic during design time of the system.

6.1.5 Initial System State

After a device reset of our system-on-chip design, all peripherals are in the unclaimed state and no access permissions are stored in the access permission table. However, the security-monitor is aware of all peripherals connected to the AXI bus and a unique identifier is directly embedded to the master interface of each participating party. Moreover, the identifier of the master entity is set in the security monitor master register. Now, this master can start granting other parties access permissions to peripherals by setting the corresponding table entries. Additionally, the security monitor also allows the master to claim peripherals for other parties.

6.1.6 Reset Unit

After powering up the chip and the initialization of the main components, the reset mechanism of the computing unit is released. As our concept consists of multiple computing units, we also improved the reset procedure to define the order.

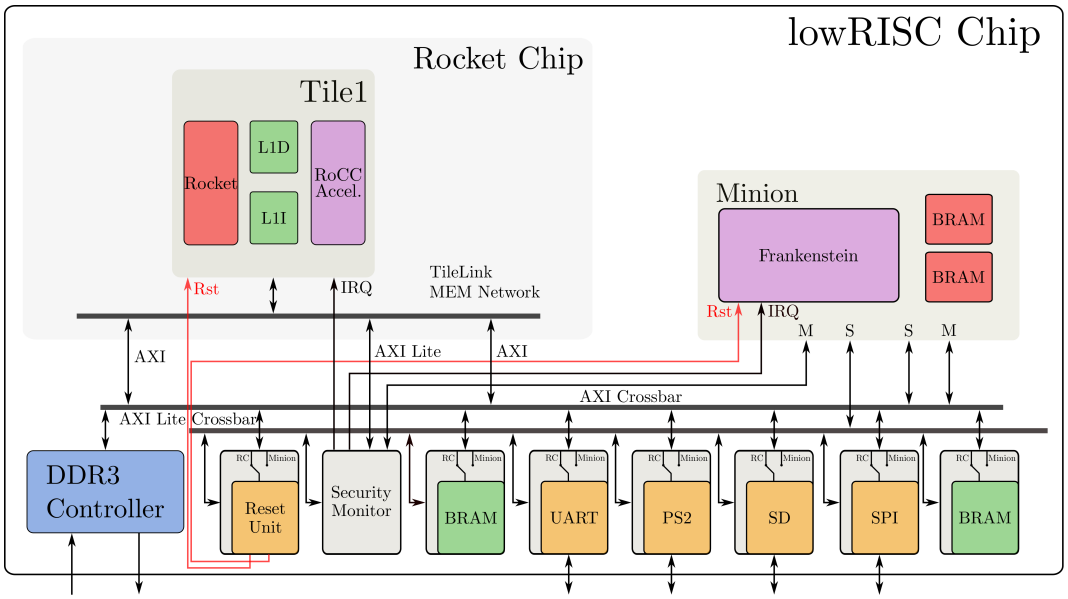


Figure 6.9: The reset unit is used to reset specific entities.

The reset lines of each entity embedded into the system are managed by the central reset unit as shown in Figure 6.9. This unit can be used like any other peripheral device available in the overall system and therefore has to be claimed first. However, resetting other parties is a potential security-critical operation. Therefore, we propose that only the security monitor master can claim this mechanism. Internally, the reset unit automatically releases the reset mechanism of the master after the system is started. Then, the security monitor entity can claim this peripheral unit and is able to start or halt other system parties.

6.1.7 Memory Protection Unit

In our architecture, the application processor and one or multiple minion subsystems have access to the external memory using the AXI communication fabric. To protect the integrity and confidentiality of assets stored in this memory, we embed a memory protection unit (MPU) into our design.

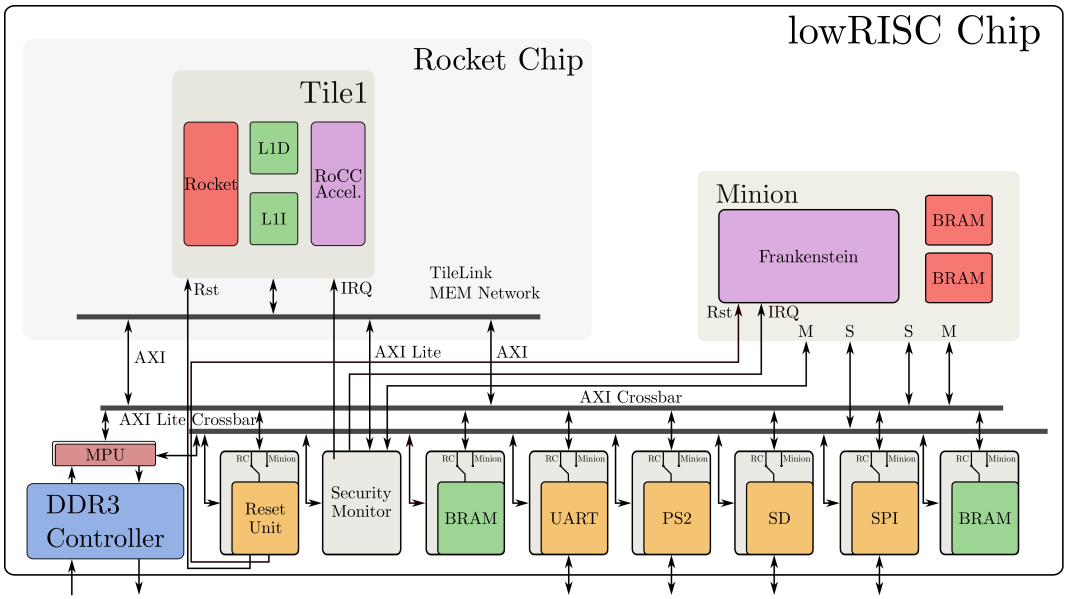


Figure 6.10: Overall system architecture including the memory protection unit.

Figure 6.10 illustrates the placement of the memory protection mechanism. Read and write requests are transmitted from the initiating party to the DDR3 controller using the AXI bus protocol and are filtered by the MPU. The memory controller then translates the request and transmits it to the external attached memory. To keep our design consistent, our MPU implementation can be claimed like any other peripheral in the system using the security monitor master and the AXI4-lite interface. The entity currently claiming the MPU can configure up to 16 individual physical memory regions. By using the entity identifiers integrated into the system bus architecture, read or read/write permissions to certain participants are granted.

6.1.8 Secure Storage

The current design allows each participating party with the corresponding access permission privileges to access all peripherals. However, some scenarios require a secure storage element which can be accessed solely by a particular entity. For this reason, we introduce a secure storage element only accessible by one entity.

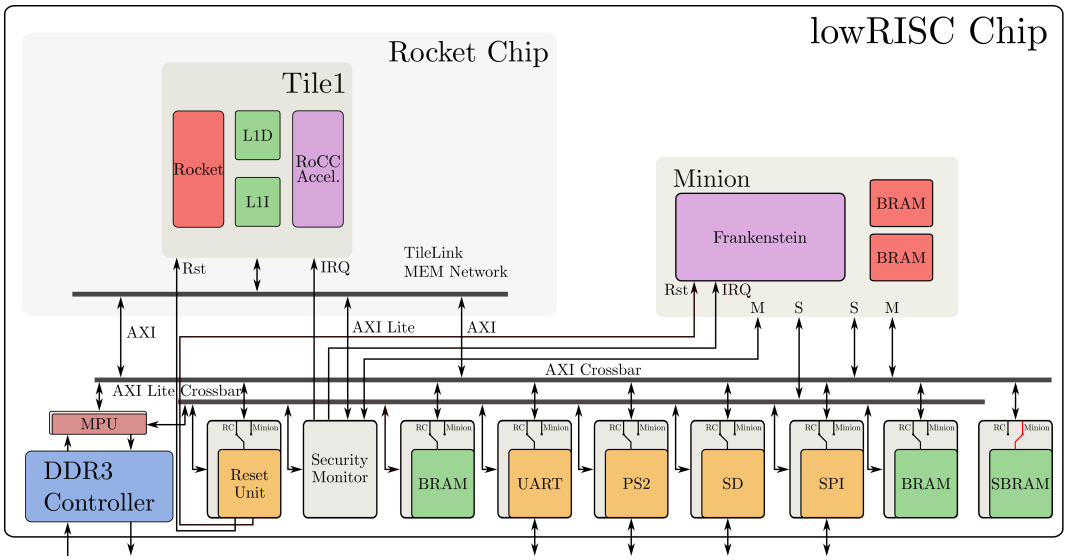


Figure 6.11: Exclusive access to a secure BRAM element by the minion subsystem.

In Figure 6.11, the secure memory element (SBRAM) attached to the AXI4 system bus is shown. In contrast to all other peripherals integrated into the overall system architecture, this element cannot be configured using the AXI4-lite interface. During the system design phase, one identifier is directly programmed to the peripheral wrapper allowing only the entity with the corresponding identifier access to the memory. However, the minion subsystem can operate in two domains. By using the external execution mode, the subsystem can either execute code stored in the internal PMEM or in an external memory attached to the AXI communication fabric. To differentiate between the two execution modes, we assign each domain of the minion an unique identifier.

6.2 Software Architecture

To easily use the proposed hardware features, our system also provides software support. The overall system architecture consists of a Linux operating system executed on the application processor and a minion subsystem. On the minion, either the code stored in the internal PMEM or a user defined code stored in an external storage is executed.

6.2.1 Linux Software Support

User applications running in the Linux environment can interact with the security monitor using a dedicated kernel module. This kernel module provides support for all necessary tasks needed to configure and access the security monitor and the minion.

Table 6.2: Security monitor (SM) kernel interface.

Command	Param. 0	Param. 1	Param. 2	Priv. Level
claim_peripheral	claim/release	ID of entity	ID of peripheral	Master/Slave
set_peripheral	set/release	ID of entity	ID of peripheral	Master
get_status	ID of entity	ID of peripheral	-	Master/Slave
ownership_transfer	ID of entity	-	-	Master
release_all	-	-	-	Master

Table 6.2 depicts the commands offered by the kernel module to interact with the security monitor (SM). Internally, the security monitor validates the access permissions by comparing the identifier of the request with the identifier stored in the security monitor master register. On a valid request, the SM updates the internal table and forwards the request to the corresponding peripheral. Applications executed on the main application processor can use this interface to claim or release peripherals and configure access permission privileges for other entities. Furthermore, we expanded a set of Linux peripheral drivers to automatically claim peripherals and verify access privileges.
As stated in Section 5.1.1, the master can withdraw access to a peripheral currently claimed by an entity by sending a request to the SM. The security monitor notifies the entity by raising an interrupt and the peripheral is expected to implement its own clean-up functionality. In our design, we added additional interrupts and implemented a interrupt handler for each sensitive peripheral. The interrupt handler then destroys potential sensitive assets individually for each peripheral.
When using the minion system as a secure enclave, four steps are required to start the enclave. First, the application initiating the start of the enclave needs to copy the application binary to a memory attached to the communication fabric. Then, access permission privileges for peripherals needed by the minion enclave are programmed to the security monitor. Furthermore, the boot mode of the enclave is set to external. In the last step, the user application starts the minion by using the reset unit. Now, a small loader program gets executed by the minion subsystem. This software establishes a shared, private, and read-only memory region using the memory controller. Moreover, the loader program computes the SHA3 hash of the binary and stores it to the read-only memory region. Finally, control is passed from the loader to the enclave program.

6.2.2 Minion Software Support

In the default execution mode, the minion runs code stored in its internal program memory. During the design phase of our overall architecture, the system engineer freely can place an application dependent software to this internal memory. In one possible use case scenario, a secure boot scheme, a small bootloader is placed into the minion subsystem. This bootloader loads and verifies a system image stored on an external memory and then starts the main application processor.
When the minion operates in the external execution mode, a small loader program fetches code stored in an external memory and executes this code. By writing a program binary to this external memory, the application processor can use the minion as an enclave to execute arbitrary software. Using this scheme, several use case scenarios are possible. To ease the development of applications executed in the minion subsystem environment, we provide a small library to communicate with the security monitor. This library consists of

functions similar to the interface shown in Table 6.2.

Chapter 7

Results

In the previous chapters, we introduced trusted execution environment solutions offered by different vendors, compared them and analyzed potential security weaknesses. Then, we presented our novel enclave system with support for trusted I/O paths and introduced our RISC-V-based proof-of-concept design and our FPGA implementation. In this chapter, we now analyze in detail the hardware overhead of our proposed security features. Furthermore, we introduce a secure boot use case scenario using our proposed scheme.

7.1 Hardware Overhead

As our proposed system includes a fully fledged, independent RISC-V core additionally integrated into the overall architecture, the chip area of the system clearly grows. To highlight the main sources for the additional chip area overhead, we compare the final architecture with the lowRISC base platform. For the base platform, we first extended the lowRISC base project to support the Kintex-7 FPGA and then attached all peripheral modules individually to the AXI system bus. Our overall design consists of the Rocket chip, a minion subsystem, the security monitor, and several peripheral wrappers for various peripherals.

Table 7.1: Number of lookup tables for the overall architecture.

Component	**Area [LUTs]**	**Area [%]**
Rocket chip	33,341	52.38
Minion RI5CY	5,780	9.08
Security Monitor	446	0.70
Peripheral Wrapper	43	0.07
AXI4 Crossbar	3,052	4.79
AXI4-lite Crossbar	93	0.14
Overall	63,648	100

In Table 7.1, detailed numbers for the main parts of the overall architecture are given. The system utilizes 31.28 % of the lookup tables (LUTs) available on the xc7k325tffg900-2 FPGA. In total, 63,648 lookup tables are occupied, the main processor roughly uses 52.38 % of them. The minion subsystem, which consists of a fully fledged RISC-V core, uses 9.08 % of the overall number of lookup tables. Due to the lightweight design of the

security monitor module and the peripheral wrapper, these two modules together are occupying less than 1 % of the overall LUTs of the FPGA. When replacing the RI5CY system with the Frankenstein core, the minion subsystem requires 10,878 LUTs. Since the Frankenstein adds a control-flow integrity (CFI) protection unit, a memory management unit (MMU), and a memory protection scheme, the area overhead compared to the base RI5CY is reasonable.

Table 7.2: Overhead of the overall architecture compared to the lowRISC base project.

Component	**Area [LUTs]**	**Area [%]**
lowRISC base	55,443	100
Overall architecture RI5CY	63,648	114.8
Overall architecture Frankenstein	68,746	123.99

Table 7.2 compares the lowRISC base platform with our overall architecture including the security monitor master, the minion subsystem, the peripheral wrappers, and the memory protection unit (MPU). When using the proposed scheme with the RI5CY core, an overhead of 14.8 % is produced. The chip area increases by 24.99 % when using the Frankenstein subsystem.

7.2 Secure Boot

In this scenario, we are using our overall architecture to securely boot the Debian 9 operating system on the RISC-V proof-of-concept design. First, we explain how the operating system usually is started on the lowRISC platform. Then we demonstrate our secure boot approach integrated into our trusted execution environment (TEE) approach. After the boot procedure of the OS, we then show how the minion can be used as enclave.

7.2.1 Booting Linux on the lowRISC SoC

Natively, the lowRISC system-on-chip (SoC) supports booting the Debian 9 operating system. As the on-chip memory size is too small to store the bootloader for Linux, a first-stage bootloader is placed in the internal BRAM. After a device reset, the reset vector address points to the start of the first-stage bootloader and this bootloader gets executed. The first-stage bootloader (FSBL) simply mounts the SD card and copies the second-stage bootloader to the external memory. Then, control is passed from the FSBL to the second-stage bootloader, which is the Berkeley bootloader (BBL). During compile time of the system software, the BBL is linked against the Linux kernel [94]. When executing the second-stage bootloader, first all hardware threads (HART) except HART0 are disabled. Now, the bootloader parses and filters the device tree blob (DTB) and sets up the memory. Finally, the other hardware threads are started and control is passed to the Linux kernel.

7.2.2 Securely Booting Linux

In our secure boot approach, we are using the minion core to securely boot the Linux kernel on the application processor. When applying power to the system, the minion is the designated security monitor master and the reset unit only releases the reset line of the minion. Therefore, the main application processor still is in the halted state. Since

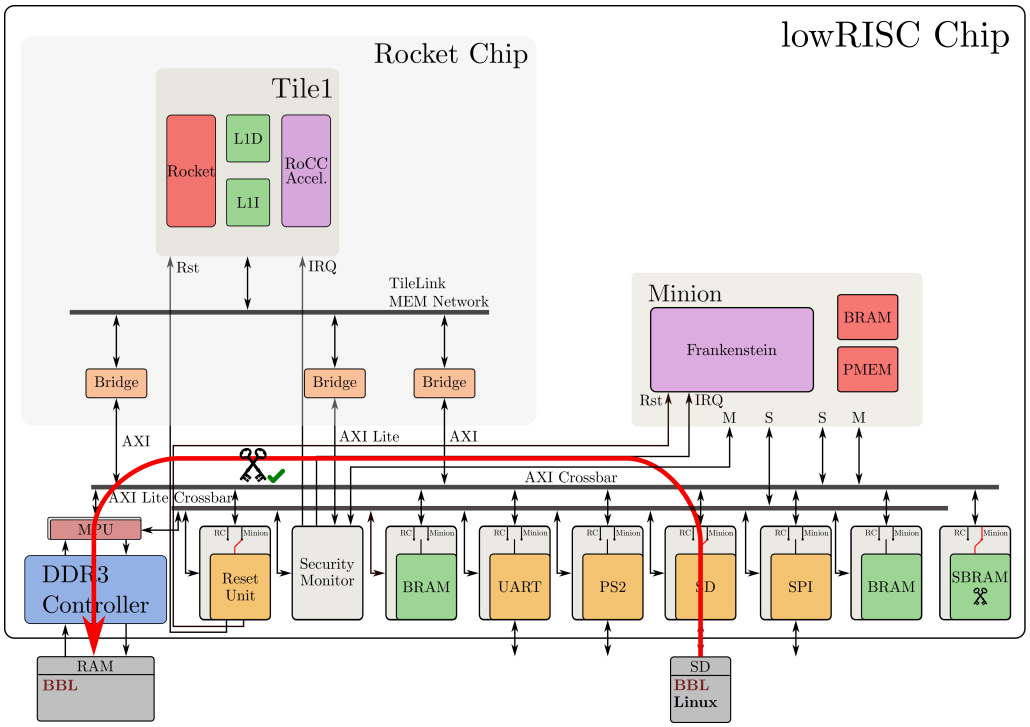


Figure 7.1: The FSBL copies the BBL from the SD card to the memory. Additionally, the signature of the image is computed and compared to the signature stored in the secure memory element of the minion.

the reset vector of the minion points to the program memory, the software in the PMEM gets executed first. As stated in Section 6.1.2, the minon can only access this memory using a read-only interface. Therefore, the code stored in the program memory acts as root-of-trust. The first-stage bootloader, which is placed in the program memory (PMEM), is extended to support claiming peripherals. First, the bootloader claims the SD card, the DDR3 memory controller, and the reset unit. Then, the minion starts fetching the BBL from the SD card and copies it to the external DDR memory. When the copy process is finished, the subsystem computes the signature of the loaded image and compares it to the signature stored in the secure BRAM element. If both signatures are equal, the integrity of the FSBL is ensured and the minion releases the SD card peripheral to start the Rocket chip by using the reset unit.

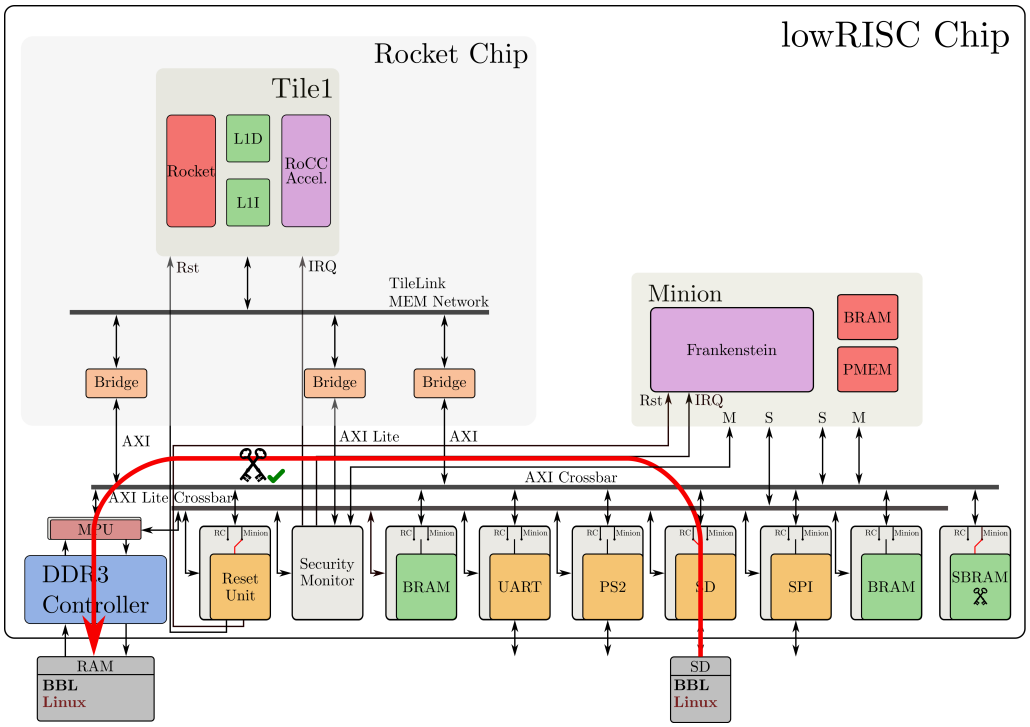


Figure 7.2: Last step of the secure boot procedure. The BBL copies the Linux kernel to the external memory and starts setting up the Linux environment. Additionally, the signature of the loaded image is computed.

The application processor starts executing the BBL from memory and first claims the SD card controller. Then, the Linux kernel is loaded from the external storage and again the signature of the loaded image is computed and compared by the minon. After the execution of the BBL, a fully authenticated Linux system is running on the main application processor.

Using our proposed TEE scheme to implement secure boot mitigates potential attack vectors shown in Section 2.2.1. Since the FSBL is stored in the read-only PMEM, the FSBL is protected from tampering attempts. Furthermore, our system also provides a secure storage element embedded into the overall architecture. As stated in Section 6.1.8, only the entity with the identifier stored in the secure element can access the memory. Due to this strong protection, the signatures of the BBL and the Linux kernel are placed into this element and the minion exclusively can read or modify these assets. By integrating the bootload sequence and the signatures keys directly to the internal chip memory, modifying attempts using an SPI programmer also fails. Since the minion provides countermeasures against fault attacks, skipping the signature checks by inducing a fault can be detected by the system.

7.2.3 Secure Enclaves

After the secure boot procedure, the system is in an authenticated state. However, using a dedicated subsystem to securely boot the operating system for the application processor adds a huge overhead to the overall chip area. For this reason, our approach allows the system to use the minion subsystem in different ways. First, in the secure boot scenario,

the dedicated coprocessor is used to start the operating system and verify the signatures. Then, the minion can initiate a ownership transfer by assigning the security monitor master privilege to the Rocket core. Now, the application processor has full control over the overall system and can use the minion as secure enclave for arbitrary code execution.

Chapter 8

Conclusion

In this thesis, we compared trusted execution environment (TEE) solutions offered by various vendors and analyzed security guarantees and implemented features. By combining the concepts of secure enclaves, trusted I/O paths, and introducing a flexible security monitor, we created a powerful TEE solution. In our design, we are using a dedicated secure coprocessor as an enclave. Following the concept of the minions introduced by the lowRISC project, one or multiple of these subsystems can be integrated into the overall architecture. As these subsystems are fully fledged cores, the minion enclave system is executed independently from the remaining part of the system. Due to these properties, we created a strong isolation between the secure and non-secure domain. Furthermore, we combined our TEE concept with secure I/O paths. By creating a path between a peripheral device and the enclave or the application processor, a trusted communication channel between the two parties is generated. Moreover, the minion core includes countermeasures against physical attacks. Compared to other TEE solutions, our approach offers a more flexible programming model. Using the security monitor, either the application processor or the enclaves can manage access permission privileges. Additionally, the owner of the security monitor entity can transfer this privilege to any other participant in the system. To demonstrate the feasibility of our proposed scheme, we integrated our novel approach into a RISC-V-based platform and implemented a secure boot example. Our analysis showed that the overall chip area increases by 14.8 % compared to the base project. When replacing the RI5CY core with the security-hardened Frankenstein core, we measured an overhead of 24.99 %. Finally, our design is capable of booting the Debian operating system and offers software support for interacting with the security monitor and the enclave system. The flexibility of our scheme is then shown by creating a secure boot scenario.

8.1 Future Work

The proof-of-concept design using a RISC-V platform can serve as a base platform for several follow-up projects. Currently, software is executed using a bare-metal program inside the minion subsystem. The main advantage of this concept is that only one security-sensitive assets reside in the enclave. However, reserving a whole dedicated coprocessor for just a single task produces huge overhead. To improve the efficiency, we propose to enable multitasking support by integrating a small operating system into the enclave. Since this allows multiple security-sensitive assets to be executed concurrently in the subsystem, the operating system needs to provide strong process isolation features as well. Hence, we

suggest to use an operating system like the seL4 kernel to the secure domain.
Another possible future work would be to develop a common interface for the communication channel. In our design approach, the applications executed on the main processor and the enclave systems need to implement their own communication channel using a shared memory region. This approach allows the developer to flexibly implement the communication interface based on the application requirements. However, as most attacks on TEE solutions are targeting the communication API, developers need to carefully implement the communication interface. For this reason, we further suggest to extend our design with a common interface. One possible solution would be a secure mailbox approach. In this scheme, a first-in first-out (FIFO) buffer is installed for each enclave instance. When exchanging messages through this system, an interrupt is raised and the receiving party can process this data. To protect the participants from denial-of-service (DoS) attacks, the secure mailbox system implements a filter system to detect anomalies. By claiming this system like a peripheral, the owner can configure the filtering mechanism.
In TEE schemes, the concept of remote attestation often is used to authenticate the host software or hardware to an external party [48]. To further extend our work, such a mechanism could be implemented to authenticate software executed in the secure encalve.
The case study shown in Chapter 7 demonstrates our overall architecture by implementing a secure boot scenario. Here, we guaranteed the integrity of a loaded image by creating a chain-of-trust. However, to also protect the confidentiality of the image stored on an external device like a SD card, data encryption on the SD card is required. We propose to integrate a transparent memory encryption scheme, such as [99], to automatically decrypt the image when loading from the SD card.

Appendix A

Abbreviations

AES	Advanced Encryption Standard
ALU	arithmetic logic unit
AMBA	ARM Advanced Microcontroller Bus Architecture
AP	application processor
APB	Advanced Peripheral Bus
API	application programming interface
ASIC	application-specific integrated circuit
AXI	Advanced eXtensble Interface
BBL	Berkeley bootloader
CED	concurrent error detection
CFG	control-flow graph
CFI	control-flow integrity
DMEM	data memory
DoS	denial-of-service
DPA	differential power analysis
DTB	device tree blob
EMR	electromagnetic radiation
EPC	enclave page cache
EPCM	enclave page cache map
FI	fault injection
FIFO	first-in first-out
FPGA	field programmable gate array
FSBL	first-stage bootloader
HART	hardware threads
ID	instruction decoder
IF	instruction fetch
ISA	instruction set architecture
IP	intellectual property
LOC	lines of code
LSU	load-store unit
LUTs	lookup tables
MAC	message authentication code
MCU	microcontroller unit
ME	Management Engine
MEE	memory encryption engine

MITM	man-in-the-middle
MMU	memory management unit
MPU	memory protection unit
NFC	near-field communication
NIC	network interface controller
NS	non-secure
OS	operating system
PMEM	program memory
PRM	processor reserved memory
REE	rich execution environment
SCR	secure configuration register
SDK	software development kit
SECS	SGX enclave control structure
SEP	Secure Enclave Processor
SGX	Software Guard Extensions
SM	security monitor
SMC	secure monitor call
SMM	System Management Mode
SoC	system-on-chip
TCB	trusted computing base
TEE	trusted execution environment
TRNG	true random number generator

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009. (cited on p. 24)

[2] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013. (cited on p. 17)

[3] N. Alimi, Y. LAHBIB, M. MACHHOUT, and R. Tourki. An rtos-based fault injection simulator for embedded processors. *International Journal of Advanced Computer Science and Applications*, 8(5):300–306, 2017. (cited on p. 15)

[4] Apple. Apple kündigt iphone 5s an – das fortschrittlichste smartphone der welt, 2013. URL `https://www.apple.com/de/newsroom/2013/09/10Apple-Announces-iPhone-5s-The-Most-Forward-Thinking-Smartphone-in-the-World/`. (cited on p. 32)

[5] Apple. Product security certifications, validations, and guidance for sep: Secure key store, 2019. URL `https://github.com/pulp-platform/pulpino`. (cited on p. 33)

[6] Apple. ios security. *Apple Inc.*, 2019. (cited on p. 38, 39)

[7] Apple. Efficient texture comparison, U.S. Patent 2013/0308838 A1, Nov. 2013. (cited on p. 6, 32, 33)

[8] ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009. (cited on p. 6, 27, 28, 29, 30, 38)

[9] ARM. Amba axi and ace protocol specification. 2011. (cited on p. 6, 47, 48, 49, 52, 53)

[10] ARM. Introduction to axi protocol: Understanding the axi interface, 2016. URL `https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface`. (cited on p. 47)

[11] ARM. Amba bus architecture, 2019. URL `http://infocenter.arm.com`. (cited on p. 27)

[12] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014. (cited on p. 20)

[13] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016. (cited on p. 21)

[14] J. Aumassion and L. Merino. Sgx secure enclaves in practice security and crypto review. *Blackhat US*, 2016. (cited on p. 30)

[15] J. Bachrach. Risc-v rocket chip soc generator in chisel. In *Chisel – Accelerating Hardware Design*, 2015. (cited on p. 21)

[16] J. Balasch. Introduction to fault attacks. *IACR Summer School 2015*, 2015. (cited on p. 15)

[17] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012. (cited on p. 13)

[18] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014. (cited on p. 18)

[19] D. J. Bernstein. Cache-timing attacks on aes, 2005. (cited on p. 18)

[20] M. Bishop. Computer security: Art and science. 2003. *Westford, MA: Addison Wesley Professional*, pages 4–12, 2003. (cited on p. 11)

[21] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997. (cited on p. 14)

[22] A. Bradbury, G. Ferris, and R. Mullins. Tagged memory and minion cores in the lowrisc soc. *Memo, University of Cambridge*, 2014. (cited on p. 20)

[23] F. Brasser, U. Mueller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure:sgx cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017. (cited on p. 45)

[24] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *IFIP International Conference on Information Security Theory and Practice*, pages 93–109. Springer, 2017. (cited on p. 38)

[25] Y. Bulygin, J. Loucaides, A. Furtak, O. Bazhaniuk, and A. Matrosov. Summary of attacks against bios and secure boot. *Proceedings of the DefCon*, 2014. (cited on p. 19)

[26] R. B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, and M. Golub. Glitch it if you can: parameter search strategies for successful fault injection. In *International Conference on Smart Card Research and Advanced Applications*, pages 236–252. Springer, 2013. (cited on p. 14)

[27] T. Cooijmans, J. de Ruiter, and E. Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20. ACM, 2014. (cited on p. 27)

[28] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016. (cited on p. 6, 9, 30, 31, 37, 38)

[29] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium USENIX Security 16)*, pages 857–874, 2016. (cited on p. 6, 27, 34, 44)

[30] A. Dent. Secure boot and image authentication v2.0. *Qualcomm Technologies, Inc.*, 2019. (cited on p. 19)

[31] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014. (cited on p. 8, 26)

[32] J.-E. Ekberg. Trusted execution environments (and android). *Black Hat Las Vegas*, 2016. (cited on p. 30, 36)

[33] T. Enache. Shellshock vulnerability. *The Open Web Application Security Project*, 2015. (cited on p. 8, 26)

[34] ETHZ. Pulpino github repository, 2019. URL `https://github.com/pulp-platform/pulpino`. (cited on p. 21, 22)

[35] ETHZ. Pulp platform, 2019. URL `https://pulp-platform.org/`. (cited on p. 22)

[36] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer, 2001. (cited on p. 12)

[37] R. P. Foundation. Raspberry pi documentation, 2019. URL `https://www.raspberrypi.org/documentation/`. (cited on p. 20)

[38] R.-V. Foundation. Risc-v: The free and open risc instruction set architecture, 2019. URL `https://riscv.org/`. (cited on p. 20)

[39] B. Giller. Implementing practical electrical glitching attacks. *Black Hat Europe*, 2015. (cited on p. 13, 20)

[40] GlobalPlatform. Tee client api specification, 2010. (cited on p. 30)

[41] GlobalPlatform. Introduction to trusted execution environments. 2018. (cited on p. 26)

[42] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017. (cited on p. 45)

[43] D. Gruss. Microarchitectural attacks and beyond. *Graz University of Technology*, 2019. (cited on p. 18)

[44] X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri. Security analysis of concurrent error detection against differential fault analysis. *Journal of Cryptographic Engineering*, 5 (3):153–169, 2015. (cited on p. 6, 16)

[45] Heise. Ausblick auf 4.16, 2018. URL `https://www.heise.de/ct/artikel/Die-Neuerungen-von-Linux-4-15-3900646.html?seite=9`. (cited on p. 26)

[46] M. Hutter and J.-M. Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013. (cited on p. 12)

[47] Intel. Intel sgx: Debug, production, pre-release –what's the difference?, 2016. URL `https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference`. (cited on p. 36)

[48] L. Jain and J. Vyas. Security analysis of remote attestation. *Stanford*. (cited on p. 67)

[49] N. Kaur and P. Kaur. Input validation vulnerabilities in web applications. *Journal of Software Engineering*, 8(3):116–126, 2014. (cited on p. 12)

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. (cited on p. 26)

[51] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018. (cited on p. 18)

[52] H. Langweg and E. Snekkenes. A classification of malicious software attacks. In *IEEE International Conference on Performance, Computing, and Communications, 2004*, pages 827–832. IEEE, 2004. (cited on p. 12)

[53] I. Lebedev, K. Hogan, and S. Devadas. Secure boot and remote attestation in the sanctum processor. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 46–60. IEEE, 2018. (cited on p. 19, 37)

[54] Y. Lee. Risc-v rocket chip soc generator in chisel. In *Online slides:" https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015. pdf*, 2015. (cited on p. 21)

[55] W. Li, Y. Xia, and H. Chen. Research on arm trustzone. *GetMobile: Mobile Comp. and Comm.*, 22(3):17–22, Jan. 2019. ISSN 2375-0529. DOI 10.1145/3308755.3308761. URL `http://doi.acm.org/10.1145/3308755.3308761`. (cited on p. 27)

[56] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016. (cited on p. 45)

[57] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018. (cited on p. 18)

[58] V. Lomne, T. Roche, and A. Thillard. On the need of randomness in fault attack countermeasures-application to aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 85–94. IEEE, 2012. (cited on p. 16)

[59] lowRISC. lowrisc webpage, 2019. URL `https://www.lowrisc.org`. (cited on p. 6, 20, 21)

[60] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017. (cited on p. 36)

[61] T. Mandt, M. Solnik, and D. Wang. Demystifying the secure enclave processor. *TRUSTONIC*, 2015. (cited on p. 32, 33, 36)

[62] S. Mangard. Chapter 6 – physical attacks and countermeasures. *Introduction to Information Security*, 2015. (cited on p. 13)

[63] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008. (cited on p. 12)

[64] D. Markovic, B. Nikolic, and R. W. Brodersen. Analysis and design of low-energy flip-flops. In *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No. 01TH8581)*, pages 52–55. IEEE, 2001. (cited on p. 14)

[65] S. McConnell. *Code complete second edition*. Microsoft Press, 2004. (cited on p. 26)

[66] A. Milburn, N. Timmers, N. Wiersma, R. Pareja, and S. Cordoba. There will be glitches: Extracting and analyzing automotive firmware efficiently. (cited on p. 15)

[67] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015. (cited on p. 8)

[68] R. Mullins. The lowrisc project, 2017. URL `https://fossi-foundation.org/assets/lowRISC-Munich.pdf`. (cited on p. 20)

[69] P. Nasahl. Power analysis attacks on the fulmine chip. Graz University of Technology, 2017. (cited on p. 6, 17, 18)

[70] P. Nasahl and N. Timmers. Attacking autosar using software and hardware attacks. In *escar USA*, 2019. (cited on p. 15)

[71] NewAE. Tutorial a2 introduction to glitch attacks (including glitch explorer), 2018. URL `https://wiki.newae.com/Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer)`. (cited on p. 13)

[72] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016. (cited on p. 30)

[73] A. M. Normann. Hardware review of an on board controller for a cubesat. *Norwegian University of Science and Technology Trondheim*, 2015. (cited on p. 15)

[74] NSA. Deitybounce, ant product data. *US Government*, 2008. (cited on p. 19)

[75] C. O'Flynn and A. Dewar. On-device power analysis across hardware security domains. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 126–153, 2019. (cited on p. 12)

[76] S. Patranabis, A. Chakraborty, D. Mukhopadhyay, and P. P. Chakrabarti. Fault space transformation: A generic approach to counter differential fault analysis and differential fault intensity analysis on aes-like block ciphers. *IEEE Transactions on Information Forensics and Security*, 12(5):1092–1102, 2016. (cited on p. 16)

[77] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017. ISBN 0128122757, 9780128122754. (cited on p. 20)

[78] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):130, 2019. (cited on p. 36)

[79] T. Popp, S. Mangard, and E. Oswald. Power analysis attacks and countermeasures. *IEEE Design & test of Computers*, 24(6):535–543, 2007. (cited on p. 18)

[80] S. Ramacher, V. Hadzic, D. Lindenbauer, and M. Mandl. It security ku - implementing and attacking cryptographic primitives and protocols. *Graz University of Technology*, 2018. (cited on p. 17)

[81] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009. (cited on p. 18)

[82] D. Rosenberg. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*, page 26, 2014. (cited on p. 37)

[83] D. Rossi. Pulp project update. *ORCONF 2018*, 2018. (cited on p. 22)

[84] S. Saha, D. Jap, J. Breier, S. Bhasin, D. Mukhopadhyay, and P. Dasgupta. Breaking redundancy-based countermeasures with random faults and power side channel. In *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 15–22. IEEE, 2018. (cited on p. 15)

[85] E. Salman and Q. Qi. Path specific register design to reduce standby power consumption. *Journal of Low Power Electronics and Applications*, 1(1):131–149, 2011. (cited on p. 14)

[86] R. Schilling, M. Werner, and S. Mangard. Securing conditional branches in the presence of fault attacks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1586–1591. IEEE, 2018. (cited on p. 24)

[87] R. Schilling, M. Werner, P. Nasahl, and S. Mangard. Pointing in the right direction - securing memory accesses in a faulty world. *Annual Computer Security Applications Conference (ACSAC) 2018*, 2018. (cited on p. 6, 23, 24, 25)

[88] E. Schultz, J. Mellander, and D. Peterson. The ms-sql slammer worm. *Network Security*, 2003(3):10–14, 2003. (cited on p. 12)

[89] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with intel sgx. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2019. (cited on p. 36)

[90] K. M. Severinsen. Secure programming with intel sgx and novel applications. Master's thesis, 2017. (cited on p. 36)

[91] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan. To isolate, or to share?: That is a question for intel sgx. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, page 4. ACM, 2018. (cited on p. 36)

[92] SiFive. Hifive unleashed, 2019. URL `https://www.sifive.com/boards/hifive-unleashed`. (cited on p. 20)

[93] A. Tang, S. Sethumadhavan, and S. Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium USENIX Security 17)*, pages 1057–1074, 2017. (cited on p. 38)

[94] A. Teixeira and D. Nehab. The core of cartesi. (cited on p. 62)

[95] N. Timmers, A. Spruyt, and M. Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016. (cited on p. 15)

[96] A. Traber and M. Gautschi. Pulpino: Datasheet. *ETHZ*, 2017. (cited on p. 22)

[97] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. Gürkayank, and L. Benini. Pulpino: A small single-core risc-v soc. In *3rd RISCV Workshop*, 2016. (cited on p. 6, 22)

[98] A. Traber, M. Gautschi, and P. D. Schiavone. Ri5cy: User manual. *ETHZ*, 2019. (cited on p. 6, 22, 23)

[99] T. Unterluggauer, M. Werner, and S. Mangard. Meas: memory encryption and authentication secure against side-channel attacks. *Journal of cryptographic engineering*, 9(2):137–158, 2019. (cited on p. 67)

[100] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018. (cited on p. 18)

[101] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt. The fault attack jungle-a classification model to guide you. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–8. IEEE, 2011. (cited on p. 6, 15)

[102] E. Vetillard and A. Ferrari. Combined attacks and countermeasures. In *International Conference on Smart Card Research and Advanced Applications*, pages 133–147. Springer, 2010. (cited on p. 13)

[103] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011. (cited on p. 20)

[104] S. Weiser. Secure i/o with intel sgx. *Master's thesis, Graz University of Technology*, 2016. (cited on p. 31)

[105] S. Weiser and M. Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268. ACM, 2017. (cited on p. 38)

[106] M. Weiser's. Ubiquitous computing. *Computer*, (10):71–72, 1993. (cited on p. 8)

[107] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018. (cited on p. 18)

[108] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard. Sponge-based control-flow protection for iot devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 214–226. IEEE, 2018. (cited on p. 23, 24)

[109] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008. (cited on p. 27)

[110] R. Wojtczuk and C. Kallenberg. Attacks on uefi security. In *Proc. 15th Annu. CanSecWest Conf.(CanSecWest)*, 2015. (cited on p. 19)

[111] L. Zussa, J.-M. Dutertre, J. Clediere, and B. Robisson. Analysis of the fault injection mechanism related to negative and positive power supply glitches using an on-chip voltmeter. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 130–135. IEEE, 2014. (cited on p. 14)