Masterarbeit

Analysis and Implementation of Cryptographic Hardware Solutions in Secure Embedded Systems

Tobias Rauter

Institut für Technische Informatik
Technische Universität Graz

Begutachter:	Dipl.-Ing. Dr.techn. Christian Kreiner
Betreuer:	Dipl.-Ing. Dr.techn. Christian Kreiner
	Dipl.-Ing. Christopher Preschern

Graz, im September 2013

Kurzfassung

Sicherheit in eingebetteten System ist ein aktuelles, sehr breites Forschungsthema. Vor allem im Bereich der Cyber-Physikalischen Systemen, bei denen Software für die Steuerung großer Maschinen verantwortlich ist, können Auswirkungen von Cyber-Angriffen fatal sein. Diese Arbeit analysiert und implementiert verschiedene Sicherheitsfeatures auf einem ARM-basierten System, das zur Steuerung von Wasserkraftwerken verwendet werden soll. Angreifer, die sich Zugang zu solchen System verschaffen, können nicht nur kritische Infrastruktur wie die Stromversorgung kontrollieren, sondern auch eine Gefahr für menschliches Leben darstellen. Um Mögliche Bedrohungen aufzufinden, wird eine Sicherheitsanalyse mithilfe von Threat-Modeling-Techniken durchgeführt. Um einige der gefundenen Schwachstellen abzuschwächen werden verschiedene Technologien analysiert und implementiert. Insbesondere wurde das System um Trusted Computing Technologien, die durch ein Trusted Platform Module unterstützt werden, erweitert. Außerdem wird eine Data-on-Rest Verschlüsselung für den nicht-volatilen Speicher eingeführt, welche durch ein integriertes Hardware-AES-Modul durchgeführt wird. Die hinzugefügten Erweiterungen werden bezüglich ihrer Auswirkungen auf die Systemsicherheit sowie der Auswirkungen auf die Leistung analysiert.

Abstract

Security in embedded systems is a wide research topic. Especially in Cyber-Physical Systems, where software controls large physical devices, the impact of attacks is potentially enormous. This work analyzes and implements security enforcing features in an ARM-based embedded system used for power plant automation. Adversaries which gain access to such control software may not only harm functionality of critical infrastructure but also human lifes.

In order to locate such threats, a security analysis with the help of threat modeling techniques is done. Different technologies are integrated to mitigate some of the found threats. In particular, trusted computing technologies assisted by a Trusted Platform Module are analyzed and implemented to ensure and attest systems' integrity. Furthermore Data-On-Rest encryption backed by an integrated hardware AES module and a key which is not readable by software is integrated. The new features are analyzed concerning their impact on systems' security and integrity, as well as their effects on performance.

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Credits

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.

At this point I want to thank my family for their mental and financial backup what at all enabled the possibility to finally write this thesis. Moreover I want to thank all my friends who stand by for all professional and personal questions due my whole study.

Moreover, I want to thank the staff of the Graz University of Technology, especially the team around this project for their help in technical and organizational issues.

Graz, 2013/09 Tobias Rauter

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Security in embedded systems is a wide research topic. Especially in Cyber Physical Systems (CPS), where software controls large physical devices, the impact of attacks is potentially enormous[CHL^{+}09]. Security problems in power plants may not only lead to a blackout of critical infrastructure but have also the potential to harm the integrity of human bodies.

The Stuxnet attacks showed that the simple prevention of physical access to automation systems is not sufficient to secure these systems [Lan11].

On the other hand, security enhancing features like trusted computing are implemented in embedded systems and show promising results [Win08]. Threat modeling techniques are introduced [HL09] in order to build a development process which is aware of possible security possible security problems.

This work focuses on a controller system used in hydro-electric power plants which is comparable to a Programmable Logic Controller (PLC). It provides a security analysis and enhancements for the system. The main contributions of this work are:

- *Security Analysis*: In order to locate possible threats, a security analysis based on threat modeling techniques is done.
- *Enabling Trusted Computing*: In order to mitigate a subset of the found threats, trusted computing capabilities assisted by a Trusted Platform Module (TPM) are introduced in the system.
- *Data On Rest (DOR) encryption*: Moreover, DOR encryption backed by an integrated hardware Advanced Encryption Standard (AES) module is implemented.
- *Security and Performance*: The impact on security, as well as on performance of the introduced features are analyzed and discussed.

1.2 Outline

In order to provide a short overview of the used techniques and tools, Chapter 2 provides a description of trusted computing and TPM features. Moreover, similar trusted computing designs and implementations which are used as base for this work are presented and discussed. The limitations of TPM based trusted computing are described by two exemplary attacks.

Chapter 3 illustrates the overall architecture of the implemented system. After a overview of the targeted system a comprehensive security analysis based on the STRIDE (Spoofing identity, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of Privilege) process is done. The security enhancements which are used to counter these threats are illustrated and alternatives are discussed.

In Chapter 4 these concepts are refined and the actual implementation on the real system is described.

Chapter 5 provides an analysis of the implemented features. Therefore, the effect on security is discussed and their impact on system's performance is analyzed.

Chapter 2

Related Work

This chapter starts with a short introduction of the basic principles and terms in the trusted computing field to provide the needed backgrounds for the following work. Moreover, a selection of trusted computing systems, especially in the embedded field, is presented as these systems represent the base if this work. To clarify the limitations of TPM based systems, two simple attacks are illustrated. The chapter ends with a short overview of existing software frameworks which can be used to enable trusted computing features on different platforms.

2.1 Trusted Computing Basics

2.1.1 Overview

Security and reliability are integral parts of modern information systems. The system's user, as well as a possible remote entity want to know if the software a system is running is trustworthy. Otherwise security or privacy critical information may be exposed to non-authorized entities.

The Trusted Computing Group (TCG) [TCG13] is a non-profit alliance of different hardware and software vendors with the aim to enhance security in computing systems. Therefore, the TCG specification defines different roots of trust, a Trusted Software Stack (TSS) and a TPM. The TCG defines a trusted system as follows:

```
A trusted system is one that behaves in the expected manner for a
particular purpose. [TCG13]
```

Thus, a trusted system has to be able to ensure that it is in a known state and has to provide possibilities to report this state to other entities in a tamper-resistant way. Otherwise, it can not be guaranteed that the system behaves as expected.

2.1.2 Trusted and Secure Boot

A common way to ensure a known system state is the "measure before execute" paradigm. The basic idea is that all executable code is measured by the loading entity before the system jumps to the entry point of the newly loaded executable. A measurement, in this case, is mostly a checksum or hash of the new executable which is compared to a pre-calculated value which is stored in the loader's binary. Consequently implementing this paradigm at all system levels leads to a Chain of Trust (COT) as shown in Figure 2.1.

In authenticated boot, all measured values are stored in a tamper-resistant way and the boot process is continued even if a module's measurement does not match the expected value. It is up to the software to read the measurement values and handle these cases. This behavior makes it important, that the saved values cannot be altered freely, as a malicious module could reset the saved measurements to pretend a trusted system state. An example how to achieve this behavior with the help of a TPM is shown in Section 2.1.4.

Sometimes, especially in mobile computing, it is desired to never run unknown code at all (above all in privileged mode, as the bootloader or the operating system does). This approach is called secure boot. The process is similar to authenticated boot in Figure 2.1 but the system stops execution before a tampered executable is loaded. Since malicious code can never be executed (at least at the boot process) a secure storage for any pre-measured values is not needed. The drawback of this approach is that the system does not run at all if something is altered, which is often not appreciate in systems with strong reliability constraints.

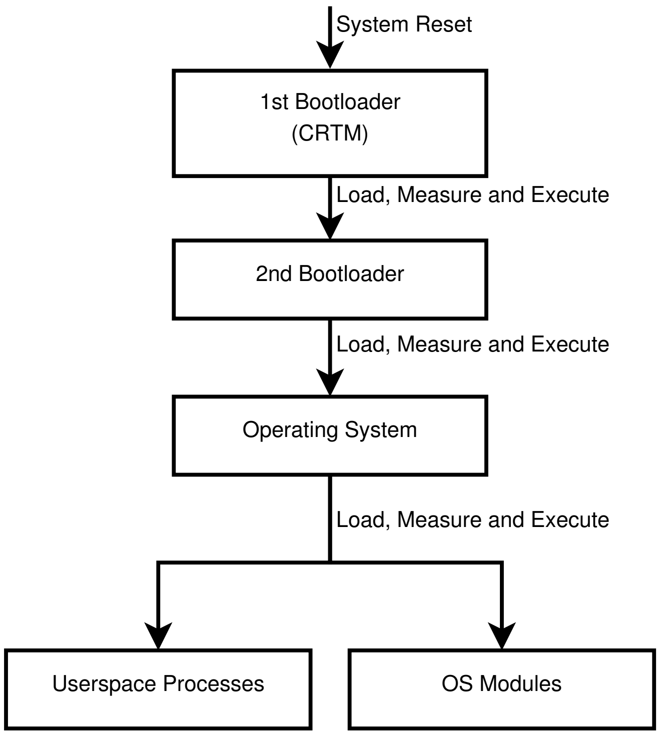

Figure 2.1: Chain of Trust: At each step in the boot process, the next module which has to be loaded is measured (e.g., a hash function) before it is executed.

2.1.3 Root of Trust

A trusted system needs some entities which are implicitly trusted at all. The TCG defines three root of trusts for measurement, reporting and storage.

Core Root of Trust for Measurement

The Core Root of Trust for Measurement (CRTM) is the first entity which performs a measurement operation. In Figure 2.1 the first level bootloader represents the CRTM. On a normal Personal Computer (PC), this is a part of the Basic Input Output System (BIOS). This entity has to be secured against tampering since an alteration of this part cannot be detected. Some approaches to achieve this in embedded systems are shown in Section 2.2.

Root of Trust for Reporting

The Root of Trust for Reporting (RTR) is used to securely report the system state acquired by the Root of Trust for Measurement (RTM) to a third party. In the TCG specification this is done by the TPM as described in Section 2.1.4 (remote attestation).

Root of Trust for Storage

The Root of Trust for Storage (RTS) includes functions to securely store keys or data. As described in Section 2.1.4 the keys are stored hierarchically encrypted in order to save keys in non-secure storage.

2.1.4 Trusted Platform Module

A TPM is basically a microcontroller with some cryptographic functions and secure storage. It can be seen as integrated smart card on a system's motherboard. The device is passive, thus it does not do anything on its own. However, there exist some software-only implementations too, as shown in Section 2.2.2.

Building Blocks

The basic blocks of a TPM are:

- A Non Volatatile Memory (NVM) which is used to store the Storage Root Key (SRK) and the Endoresement Key (EK) as well as user defined values. This memory is physically located in a shielded location where it is protected against interference from the outside and exposure.
- A Rivest/Shamir/Adleman (RSA) engine which is used for asymmetric encryption/decryption of keys/data and for creating and verifying digital signatures.
- A Secure Hash Algorithm (SHA-1) engine used for Hashed Message Authentication Code (HMAC).
- A True Random Number Generator (TRNG) which is used for key generation.

Moreover, it consists of a Central Processing Unit (CPU) running a firmware which handles the incoming requests. According to the TCG the supported I/O-interfaces are Low Pin Count (LPC) and Inter-IC (I2C).

Key Types and Structure

The TCG defines different keys for different purposes which are used by the TPM.

- *Endoresement Key (EK)*: This key is the unique platform identity key. Some manufacturers create this key at production time and sign it to certify that this key comes from a TPM. This key cannot leave the TPM and cannot be used for signing.
- *SRK*: The SRK is the root element of the key hierarchy and used to generate the next three keys.
- *Storage Key*: Used to encrypt other elements in the hierarchy.
- *Signature Key*: Used for signing operations. Have to be leafs in the hierarchy.
- *Binding Key*: Used to encrypt small amounts of data (like keys used for symmetric cryptography).
- *Attestation Identity Key (AIK)*: These keys are used as aliases for the EK and used to sign Platform Configuration Register (PCR) values for remote attestation as described later in this section.

Since the NVM of the TPM is very limited, only the EK and SRK are permanently stored. All other keys are managed in a tree structure shown in Figure 2.2. Since every private key is encrypted with the corresponding parent public key, all keys of the path have to be loaded in order to use a private key. Listing 1 shows an exemplary sign task with "Key2" of figure 2.2: With the help of the SRK, "Key1" is loaded and decrypted. This key enables the decryption of "Key2" which is used to sign the *tbs*-data-blob. Since the private parts can not leave the TPM unless they are explicitly marked migrateable, they are never accessible to software. The signing operation is done on the TPM and the software receives the result only.

```
TPM tpm;
tbs = "to be signed"
tpm.loadkey("srk");
tpm.loadkey("key1");
tpm.loadkey("key2");
signature = tpm.signData("key2", tbs);
```

Listing 1: Exemplary signing with the help of a TPM: The key chain has to be loaded in order to use "Key2" which is used to sign the string.

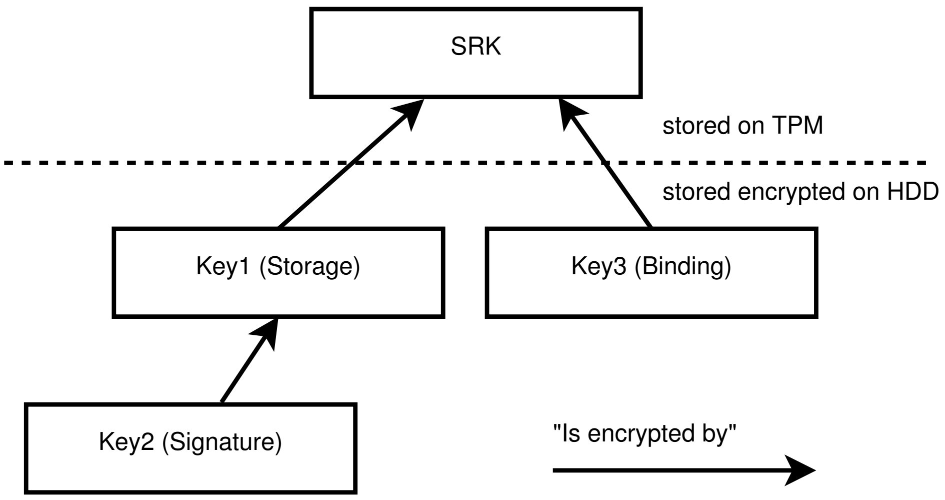

Figure 2.2: Key hierarchy on TPM: Since NVM is limited, only the SRK is permanently stored on the TPM. Other keys are encrypted and stored on the hard disk. In order to use a key in the hierarchy, the full path has to be loaded and decrypted.

Platform Configuration Register

The PCRs are used to save measurements on the TPM. As mentioned before it is necessary to prevent arbitrary write access to these registers. Otherwise, a malicious software with privileged access is able to write false measurement states and the system cannot detect the unauthorized software. However, a TPM only provides an ordinary read and an extend command. The extension of a PCR is defined in equation (2.1)[1].

$$PCR\_val(i+1) = SHA1(PCR\_val(i)||measurement) \tag{2.1}$$

Since the resulting value is concatenated and hashed with the given measurement, the complexity to generate a measurement which results in a given value is 2^{160} (first preimage).

Figure 2.3 show an exemplary content of the register on a ordinary PC. According to the specification[TCG05], current TPMs have to implement 24 of these registers, wherein the first 16 are only reset on platform reset and the remaining are reserved for Dynamic Root of Trust for Measurement (D-RTM) usage.

Binding and Sealing

Bind and unbind are TPM primitives used to encrypt/decrypt another key or data with a storage key. Sealing also uses given hash values which have to represent the TPM's state (i.e., the PCR content) at unseal time. Additionally the unsealing TPM has to be the same one which sealed the buffer. Thus, a defective TPM causes data loss of all data ever sealed on this device.

[1]The || sign represents a concatenation.

```
PCR-00: 7E 90 9A CA 58 42 A0 F1 EB 4B F1 10 34 60 3E C8 9E 91 F5 B2
PCR-01: 7C 41 68 85 BB 51 91 0C 32 06 FD A6 01 75 EC 53 19 B4 2B 22
PCR-02: 93 10 04 B1 81 BE 42 A6 D5 88 E7 FF FB 07 30 12 79 43 EE 27
PCR-03: B2 A8 3B 0E BF 2F 83 74 29 9A 5B 2B DF C3 1E A9 55 AD 72 36
PCR-04: 62 60 F2 46 38 33 4A 1E 8F FD FA BE 6E 70 33 EF 05 5B 9E B7
PCR-05: 9C 89 EB 4F BC D9 E8 36 3C 97 52 52 D8 9A 73 78 88 18 16 AC
PCR-06: 56 47 16 89 83 C8 94 E7 BB EF F1 AA 3D FB DE 84 8F 91 8F 06
PCR-07: B2 A8 3B 0E BF 2F 83 74 29 9A 5B 2B DF C3 1E A9 55 AD 72 36
...
PCR-17: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-18: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-19: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-20: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-21: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-22: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
PCR-23: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2.3: Exemplary content of the PCRs on an ordinary PC.

Remote Attestation

As mentioned before, a trusted system has to provide the functionality to prove its configuration to another entity. This process is called remote attestation. The basic progress is shown in figure 2.4.

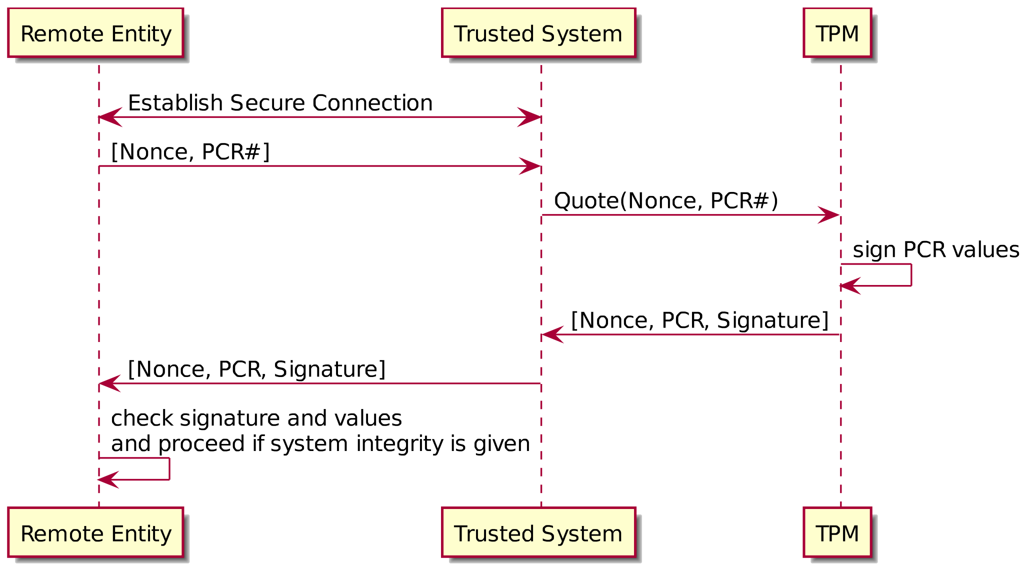

Figure 2.4: Attestation of a secure state: The remote entity challenges the trusted system with a random value to check system's integrity. The TPM signs the current PCR values and the nonce to attest its state.

The remote entity sends a random value, called nonce, and a quote request, which is basically a list of PCR numbers the entity is interested in. The trusted system performs a *TPM_QUOTE* operation on the TPM. The TPM signs the current PCR values and the nonce with an AIK. The remote entity is now able to compare the PCR values to stored references and check the signature with the public part of the AIK in order to ensure data integrity. In this case, the remote entity has to know the trusted system's public AIK and the key has to be distributed in a secure way.

Another possibility is the use of a Privacy Certification Authority (PrivacyCA) shown in figure 2.5. The trusted system generates an AIK and sends it, together with its vendor-signed EK certificate to the PrivacyCA. The PrivacyCA checks the EK-certificate, signs the AIK and encrypts the response with the public part of the EK.

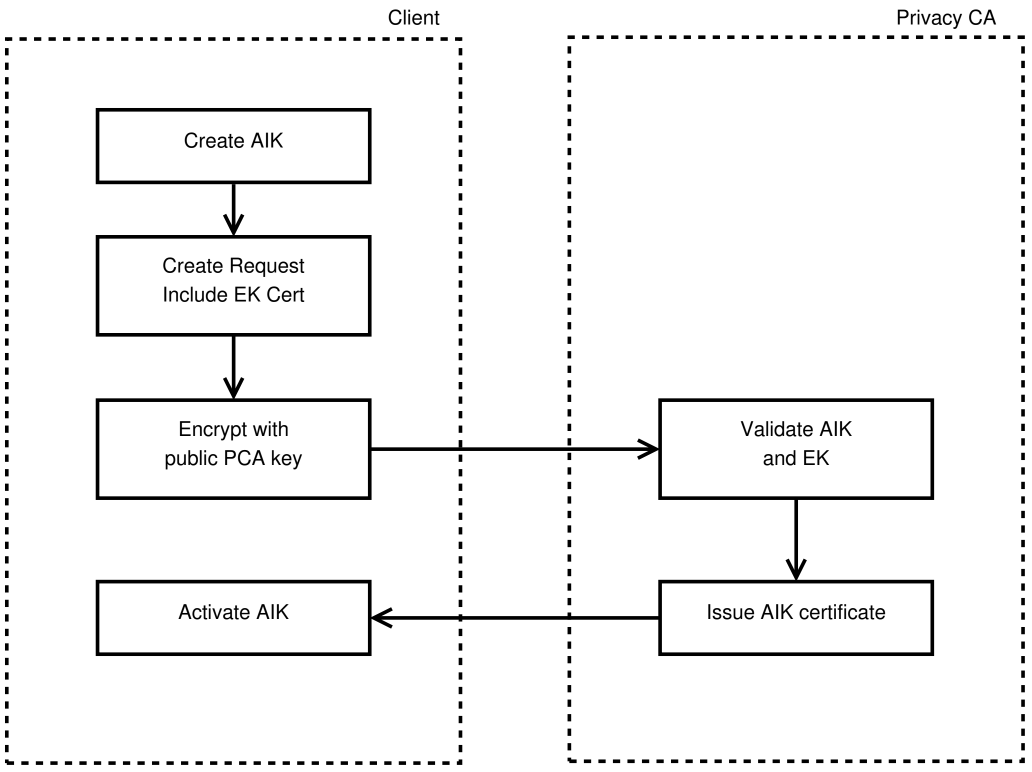

Figure 2.5: PrivacyCA: After a new AIK is created, the client sends a signing request including the vendor-signed EK certificate to the PrivacyCA. If everything is OK, the PrivacyCA signs the AIK and the key can be used for attestation.

In this case the entity which request an attestation can ask the PrivacyCA for the AIK certificate state and does not need to hold its own copy of the key.

2.2 Trusted Computing Technologies

The basic ideas behind the techniques described in section 2.1 were proposed in 1989 by Grasser et al.[GGKL89]. Yee et al. proposed the use of a cryptographic co-processor to verify the integrity of software[YT95]. Recent development focuses on the integration of trusted computing technologies in embedded systems to ensure secure and/or safe functionality. This section gives an overview of similar systems which are the base of this work.

2.2.1 TPM Based Trusted Boot Systems

Verification-Based Multi-Backup Firmware Architecture

Yin et al. used the measure-before-execute paradigm to increase reliability in embedded systems[YDJ11]. NAND flash is often used as NVM in embedded system. However, they stated that this technique has a high rate of bad blocks what may damage the firmware running on the system. Figure 2.6 shows the basic principle of their system. A minimal bootloader, called *PreBoot* is loaded and checks the integrity of the actual bootloader. If the bootloader's integrity check fails, a backup copy is checked and loaded. The system is able to update the damaged bootloader with the backup copy. An existing operating system is loaded the same way. *PreBoot* represents the CRTM, thus it is not checked anywhere. However, since this part can be implemented in few bytes, the chance of *PreBoot* to be damaged is very low. The footprints of the different modules implemented on an ARM-based system are shown in Table 2.1. The boot-time overhead introduced by this system is $400ns$ and $1.5ms$ in case of a valid or damaged bootloader. However, about 50% of this overhead is the execution time of the checksum function. Compared to the standard boot process, the VFMA approach uses 65% more time to boot in case of a valid bootloader.

Table 2.1: Size of PreBoot compared to the standard bootloader.

Module	Size
U-Boot (Standard)	219.7 kB
PreBoot (VFMA)	1.9 kB
U-Boot (VFMA)	219.5 kB

Reliable Trusted Boot

Another approach combining authenticated boot concepts with reliability was done by Li et al.[LZZ11]. They also proposed a CRTM which is integrated in a dedicated hardware. Thus, they do not rely on any software which has to be stored in a tamper-resistant way. First, they introduced the Extended TPM (E-TPM), a hardware module which covers basic TPM functionality with the following extensions:

- *Control Module*: The basic structure of the *Control Module* is shown in Figure 2.7. It is used to enforce E-TPM start-up before the rest of the platform gets access to peripherals. CPU and peripherals are set on hold while the *Control Module* checks the integrity of critical data.

- *Symmetric Cryptography*: Additionally the E-TPM contains a symmetric cryptography engine which can be used through the TSS. It implements Data Encryption Standard (DES), Triple Data Encryption Standard (3DES) and Encryption Algorithm for Wireless Network (SMS4) in hardware in order to provide a high computing speed.

Based on this technology, the boot process of an embedded system can be extended as shown in Figure 2.8. On system reset, the *Control Module* of the E-TPM takes over

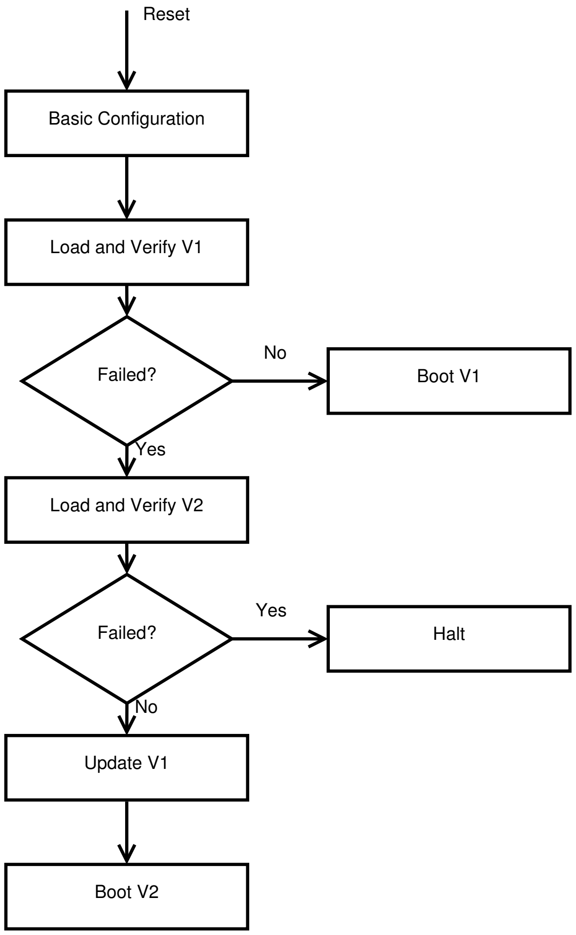

Figure 2.6: VFMA boot process: After minimal configuration done by *PreBoot*, the first version (V1) of the bootloader is loaded and verified. In case of integrity failure, the backup copy (V2) is loaded. If the backup copy can be verified, the original version is replaced with the backup and booted.

bus control and reads the bootloader from unprotected memory. If the signature of the executable can be verified, the bus is handed over to the CPU and the system can start up normally. Otherwise, the E-TPM tries to load and verify a recovery-copy of the bootloader from protected storage. When even this step fails, the system is halted. As mentioned before, this method provides two important advantages:

- The system does not rely on a bootstrap code that is securely stored (besides the recovery-bootloader) since the CRTM is build in hardware. Thus, updating the bootloader is simpler than in ordinary TPM-based systems.
- On the other hand, even if someone tampered with the NVM, the system is able to boot a minimal software and the user is able to detect these modifications.

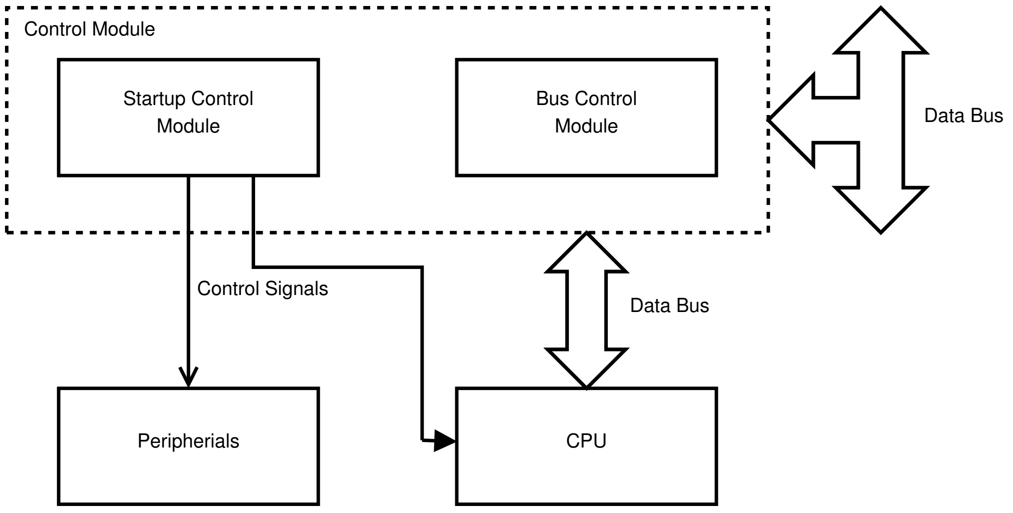

Figure 2.7: Control Module of ETPM: The *Control Module* is used to enforce E-TPM start-up while the rest of the platform is put on hold.

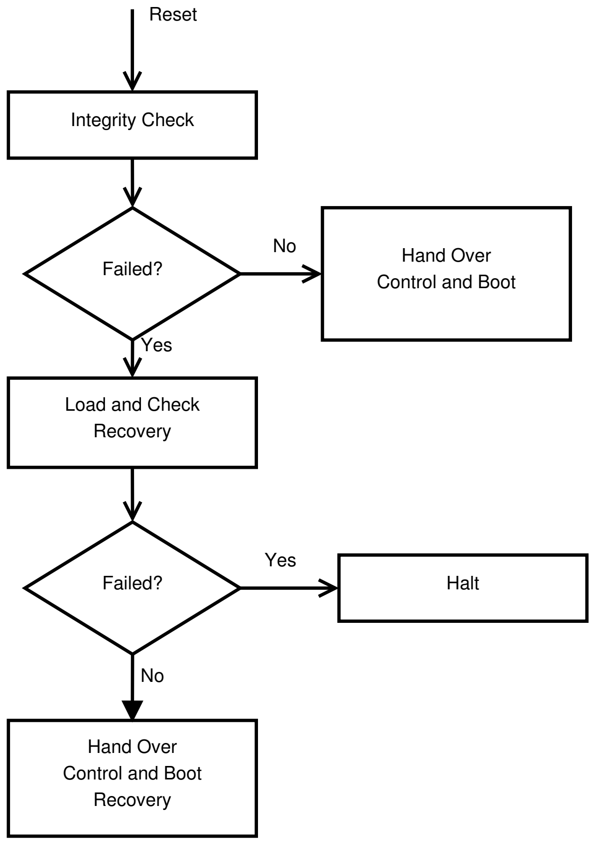

Figure 2.8: Boot process with E-TPM: The E-TPM loads and verifies the boot code and does not start the CPU without ensured integrity of the executable.

Integrity Measurement Architecture

The work presented so far focused on a measured boot process. The CRTM measures the bootloader, the bootloader the kernel and so on. In order to provide a completely measured system, account of the user-space processes has to be taken too.

As stated in Section 2.1 the extension of a PCR is non-commutative. However, in off-the-shelf operating systems, user-space applications and libraries load quite randomly. Some services only start if specific hardware is plugged in or on other external interrupts. Moreover, scheduling can lead to different behavior of late loaded libraries. If every application is measured and extended into the same PCR, such behavior leads to different cumulative measurements at each time. Additionally, it is not entirely clear, what is worth to measure. Measuring and extending every application binary at every execution potentially leads to a high measurement overhead.

Sailer et al. worked on this problem and introduced a TPM-based Integrity Measurement Architecture (IMA)[SZJvD04]. The basic idea is illustrated in figure 2.9. To initialize the IMA, the following tasks have to be performed.

- Ensure a chain of trust up to the kernel (i.e., verify the bootloader and the kernel before it is executed and extend the measurements to the TPM.
- Load the IMA module and the TPM driver before any code which is not integral part of the kernel has to be executed.
- Install hooks which are able to react on execution of user-space files or late load of kernel modules.

After the initialization, a measurement list is created. This list contains one line for each measured binary. Each of these lines involve the name of the measured binary and a hash value of the binary at execution time. The first entry contains the so called 'boot aggregate', what is the accumulated value of the pre-kernel measurements, taken from the appropriate PCRs. Before each execution of a user-space application the following tasks are done:

- Check if the binary is already in the measurement list and whether it was not altered since the last measurement of this binary has been taken. If yes, continue with execution.
- If the binary was not measured before, or have been altered since the last execution, hash it and append it to the measurement list.
- Extend the new measurement to the configured PCR.

Figure 2.10 shows an exemplary measurement list. If any other entity wants to check the integrity of the measured system it has to read the measurement list and a signed quote (as described in Section 2.1) of the PCR used by the IMA. With the quote, the remote entity is able to check the integrity of the measurement list and with the list it is able to verify all software which runs or have been running on the system.

The IMA has been implemented into Linux and is also able to measure other important, but not directly executed files like configurations or scripts. This can be adjusted with different policies and is discussed in Section 4.4.

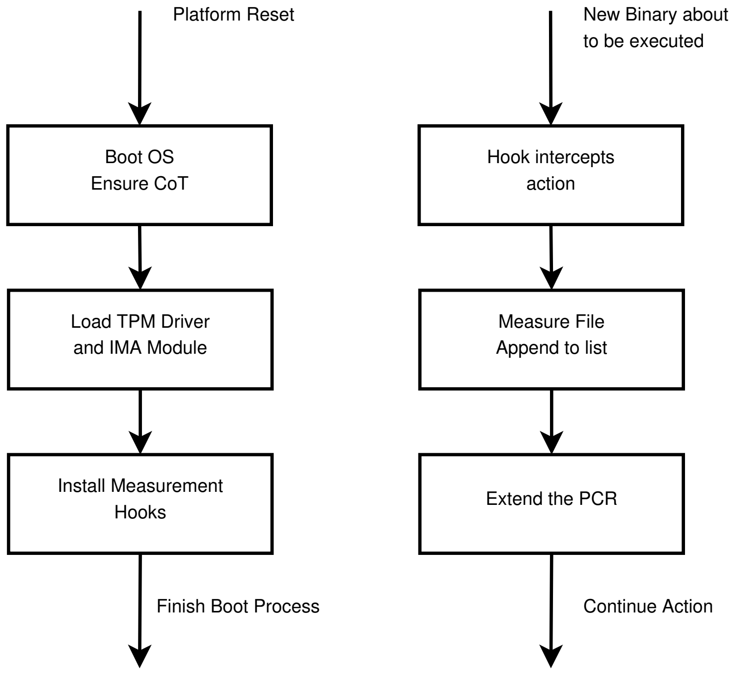

Figure 2.9: The basic work-flow of IMA: On system boot time, the TPM driver and IMA hooks are installed. Whenever a new file is about to be executed, the hash value is appended to the measurement list and the accumulation of the measurements are extended to the TPM.

```
10 d0bb59e83c371...ba6 ima 365a7adf8fa89...0b8 boot_aggregate
10 76188748450ab...a51 ima f39e77957b909...2ca /bin/sleep
10 df27e64596391...a8e ima 78a85b50138c4...a50 ld-2.15.so
... ...
10 30fa7707af01a...08b ima 72ebd589aa955...5fe parport.ko
```

Figure 2.10: Exemplary measurement list of the IMA in Linux.

DRTM for Secure Network

The conventional way to measure the integrity of a system is to build a chain of trust and measure all executables starting at system's boot time. In real computer systems this measurements grow quickly to a size where management and verification may be very hard. Different CPU vendors implement so called late launch technologies (e.g., Intel's Trusted Execution Technology (TXT)) to build a D-RTM. Hereby, the system stops executing normal code and switches to an isolated mode. The CPU starts a hypervisor and uses PCRs which are only usable in this special mode. This enables a starting point for a root of trust which does not need a real hardware reboot or a completely measured system.

Feng et. al proposed a Trusted Network Connection (TNC) based on this technology [FQmYF11]. The main goals of a TNC is to prevent insecure terminals from accessing restricted networks. Basically this is done by measuring a system and check the status before it is authorized to access the network. A problem of existing solutions is the Lying

Endpoint Problem (LEP). A compromised terminal can simply forge a valid measurement value. Technologies like authenticated boot and the IMA face this problem, but rely on a CRTM.

As shown in Figure 2.11, the proposed system uses a D-RTM based on Intel TxT:

- Whenever the client want to join the network, the Network Access Control Manager System (NACMS) sends a random nonce to the client.
- The client initializes the late-launch environment (called *NACVisor*) and extends the measurement of the hypervisor to a special PCR before executing it.
- Similar to IMA running processes are measured.
- When all measurements have been collected, the client switches back to its normal context and creates a quote of the PCRs.
- The NACMS checks the quote and the measurement list and allows network access if the integrity can be ensured.

The system has been exemplary implemented on Windows and Linux and is able to attest a trusted system state without the need of a CRTM.

2.2.2 Secure Boot Processes without TPM

Another possibility to ensure the integrity of the bootloader is to store it on tamper resistant memory which is located in the System on Chip (SoC) itself.

An exemplary technology which enables these kind of systems is ARM's TrustZone. This system introduces a secure and non-secure world in hardware: Only the secure world has unrestricted access to the systems hardware, memory and the on-chip Read Only Memory (ROM). A special command switches the SoC to non-secure mode while the secure world code is able to mask non-secure access to the different subsystems. However, it is possible to jump back to secure world with so called Secure Monitor Call (SMC) which can be seen similar to system calls in operation systems.

Based on this technology, a secure boot process using the on-chip ROM is presented in [AHYK11]. Moreover, these subsystems enable the use of a software-based TPM.

Software Based Trusted Platform Module

Instead of using a dedicated hardware TPM, sometimes it might be sufficient to simply use a software based implementation. Especially in development phase, it is advantageous to use a TPM which can be re-initialized without a platform reset. Moreover, software based approaches are generally faster and easier to debug. The drawback of such implementations is that the TPM is running in the same context as other software, which enables simple attacks. However, in virtualized environments this problem can be mitigated if the host and guest contexts are separated properly.

There are two different implementations which are used widely. The *IBM TPM Emulator* [Cor13] uses a network socket for communication. The TSS is able to connect to this socket and user software can use the software TPM transparently.

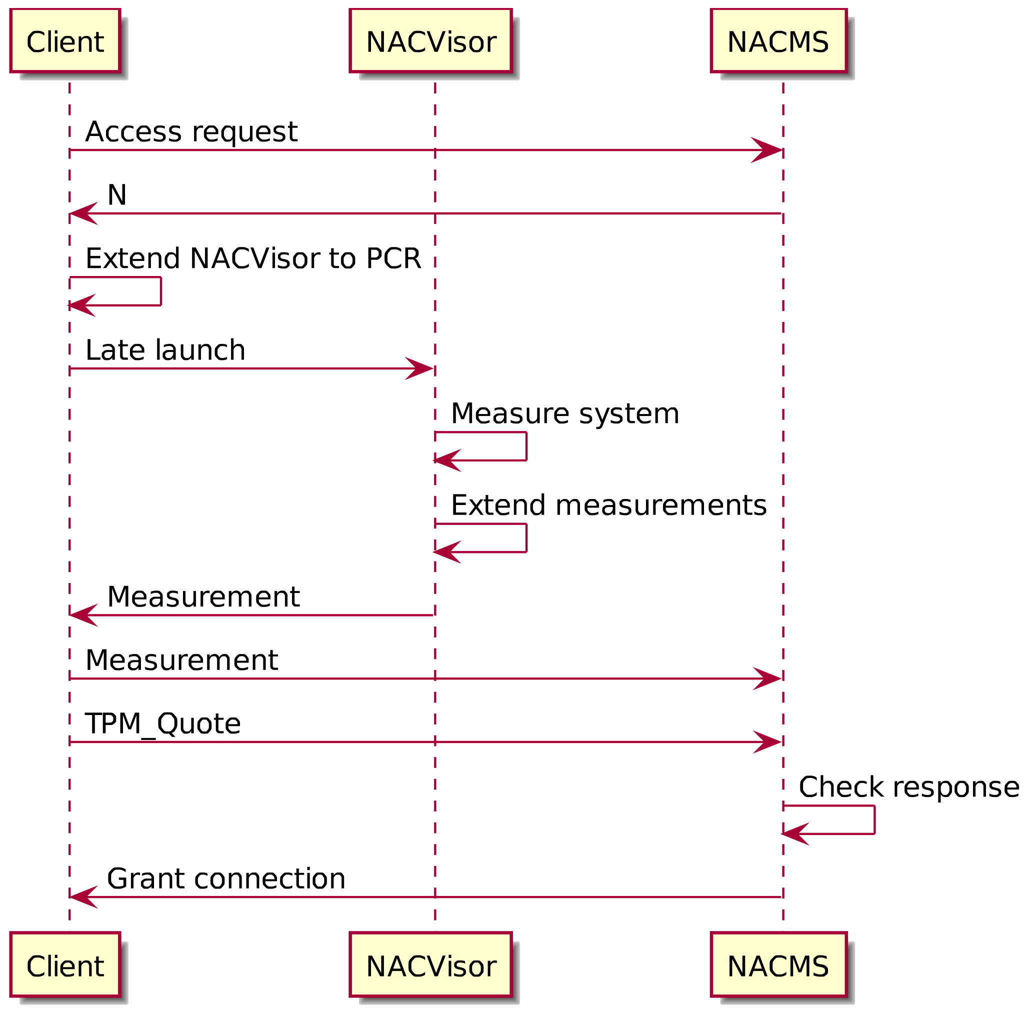

Figure 2.11: DRTM for trusted network connection: In order to get access to the network, the client has to prove its state to the NACMS. This is done with the help of the NACVisor, that is late-launched by the CPU, measures the system state and extends the results to the TPM. This approach removes the need of a chain of trust built up at boot time.

The *Berlios TPM Emulator* [SS13] implements a TPM device driver for Linux which communicates with a user-space-daemon which implements the actual TPM functionality. This approach enables the possibility to use the software TPM in kernel-space.

Virtualization

Based on the technologies described before, a system with software TPM in secure world virtualizing a non-secure operating system has been introduced in [Win08]. Figure 2.12 shows the basic building blocks.

In the secure world, a minimal Linux with a driver used to manage the privileges and callbacks of non-secure world runs a software based TPM and a Virtual Machine (VM) supervisor. The supervisor defines platform resources which may be accessed by the guest

VM and reacts on its secure monitor calls.

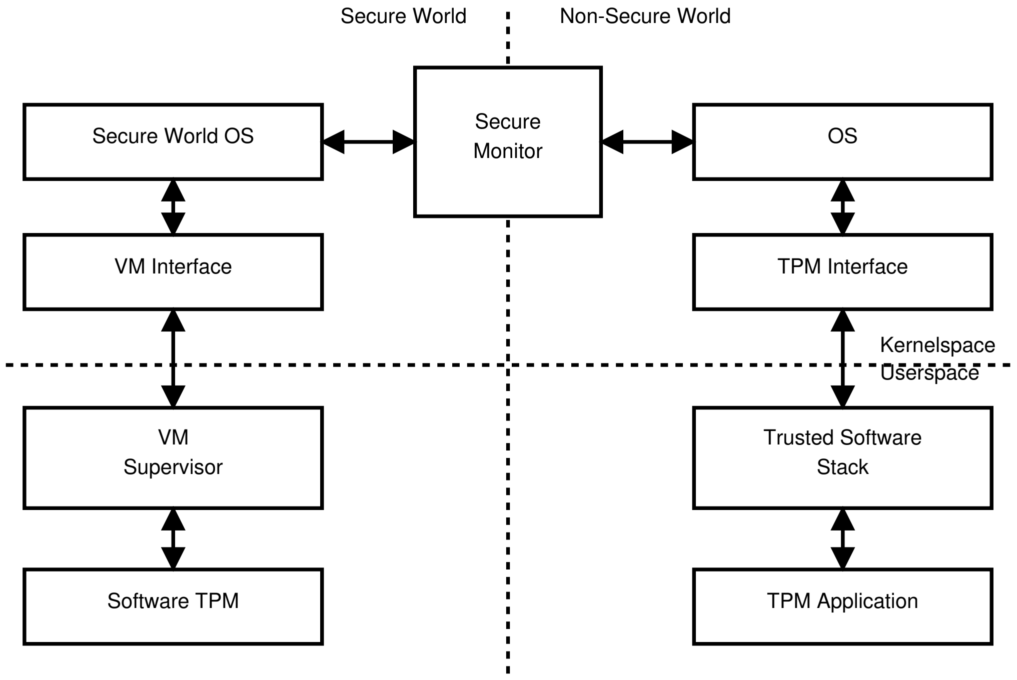

Figure 2.12: Virtualization with TrustZone: The supervisor is managing virtual machines in Non-Secure world.

It has been shown that the non-privileged user-space supervisor introduces a set of advantages compared to fully privileged VM management code:

- The amount of secure privileged code is minimized, what reduces the attack surface of the secure world.
- Debugging of secure user-space applications has turned out to be more straightforward compared to kernel-space code.

Furthermore, the authors implemented the system on real hardware and provides a framework for mobile trusted systems.

2.3 Limitations and Weaknesses

As mentioned before, physical access to the TPM potentially makes it futile. This section covers two hardware attacks on these systems.

TPM Reset Attack

As described in Section 2.1 the integrity measurement of the system relies on the integrity of the PCRs. As illustrated in Figure 2.13 the chain of trust can be compromised if the attacker is able to reset the PCR and extend a bogus value into the TPM. This value

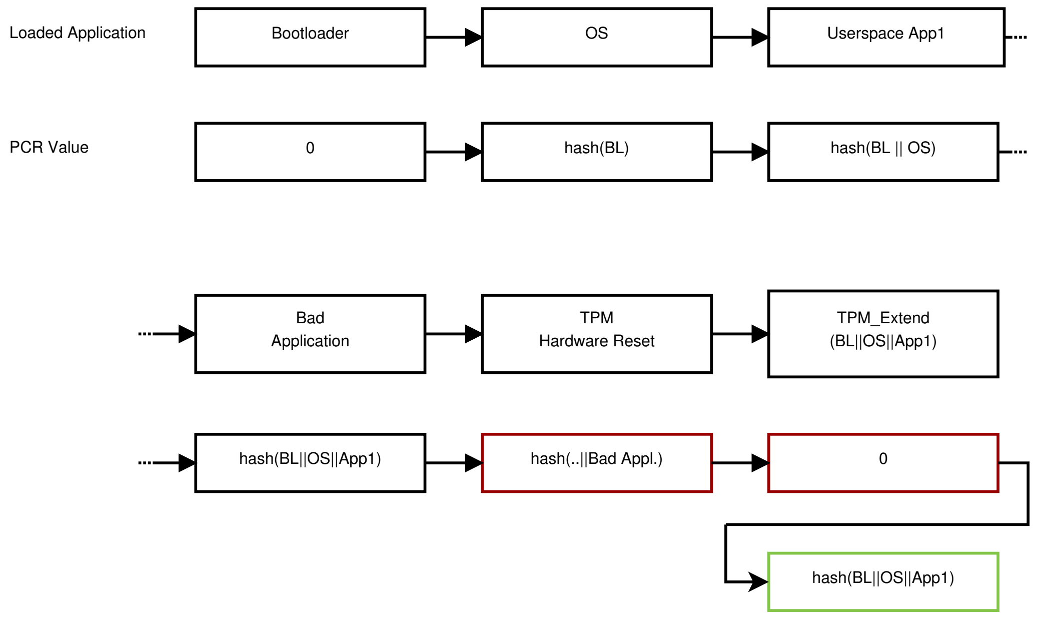

Figure 2.13: TPM reset attack: If an adversary is able to reset the PCR values he or she is also able to mask the fact that unknown software is running on the system.

can be an accumulation of the real values, which would be extended into the PCR if the system runs normally. In this case the system would not recognize the changed state.

Therefore, the TPM is only re-initialized on system reset. However, if an adversary has physical access, it is possible to achieve this behavior by grounding the LPC reset line with the help of a simple wire. Since the startup command of the TPM does not need a locality check, the attacker can get control over the TPM anytime.

These kind of attacks can be mitigated with the use of a D-RTM as in OSLO [Kau07], since the needed commands to initiate a D-RTM are hard to fake in software. However, it does not provide real security against hardware attacks as the next presented work shows. Other approaches might be an encrypted communication between the TPM and the CPU or the integration of TPM like features into the CPU-die as mentioned before.

Attacking a DRTM

As described in Section 2.2.1 some CPUs provide a 'Late-Launch' feature which enables the possibility to start a measurement chain anytime after system startup. Therefore the CPU needs an extra privilege level called *locality*. In normal PCs this locality level is checked by the southbridge. Thus, an attack between the CPU and the southbridge might be hard. However, the last mile to the TPM is done with the LPC bus which has been successfully attacked in [WD12]. Figure 2.14 shows the basic idea behind this attack. The upper LPC-frame describes an ordinary memory write action on the LPC bus which can be done by any software which gains root access. The lower frame shows a TPM write cycle with the locality bits used for authentication. As shown in the figure, it is possible to put a complete TPM command into the ordinary write frame when the adversary is able to mask the Start of Frame (SOF) line. The author of the attack used a Field Programmable

Gate Array (FPGA) to delay the SOF signal and wrote highly privileged TPM commands which were not detectable by the southbridge.

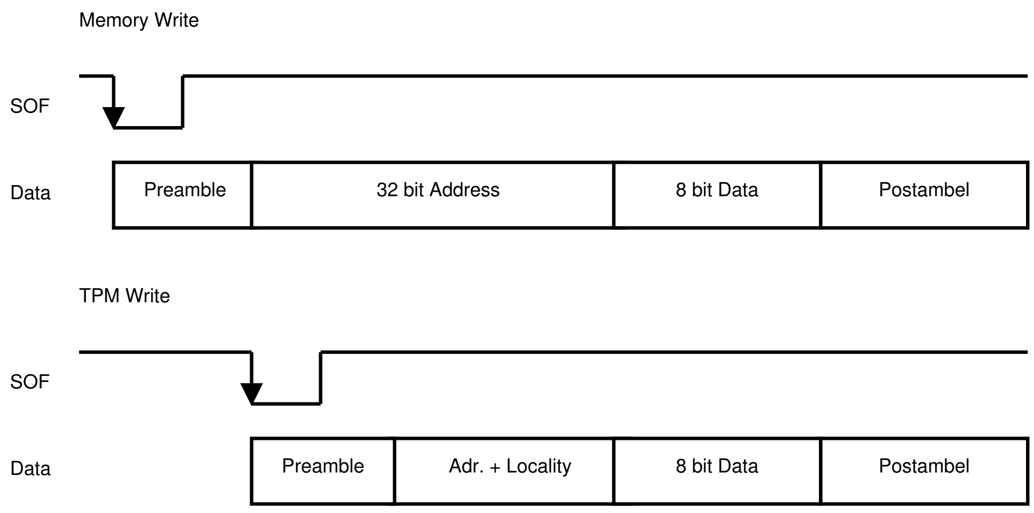

Figure 2.14: DRTM Attack on LPC bus: By delaying the SOF line of the LPC bus it is possible to circumvent TPM locality checks within ordinary memory write cycles

In summary, it has to be said that a TPM is a device which can be used to defend against remote software attacks but completely fails if someone has direct access to the hardware. Thus, to prevent malicious users from attacking the system (e.g., a user who wants to disable Digital Rights Management (DRM) features), the system has to use other technologies.

2.4 Trusted Software Stack

2.4.1 Overview

The TCG defined the TSS software stack specification with three parts: The Trusted Device Driver Library (TDDL), the Control Software (TCS), and the TSS Service Provider (TSP). The architectural overview is given in Figure 2.15.

The TDDL provides an Application Programming Interface (API) to interface with the actual TPM device driver used on the system. It offers low level functionality to open and close a device, as well as basic communication.

The TCS synchronizes the access to a TPM and handles resources like authorization sessions and key context swapping [CYC+07]. In case of TrouSerS, this module is implemented as a single service daemon which is also capable to interface over a network interface.

Every application interfaces with the TSP, what is mostly implemented as shared library and provides the API to communicate with the TCS.

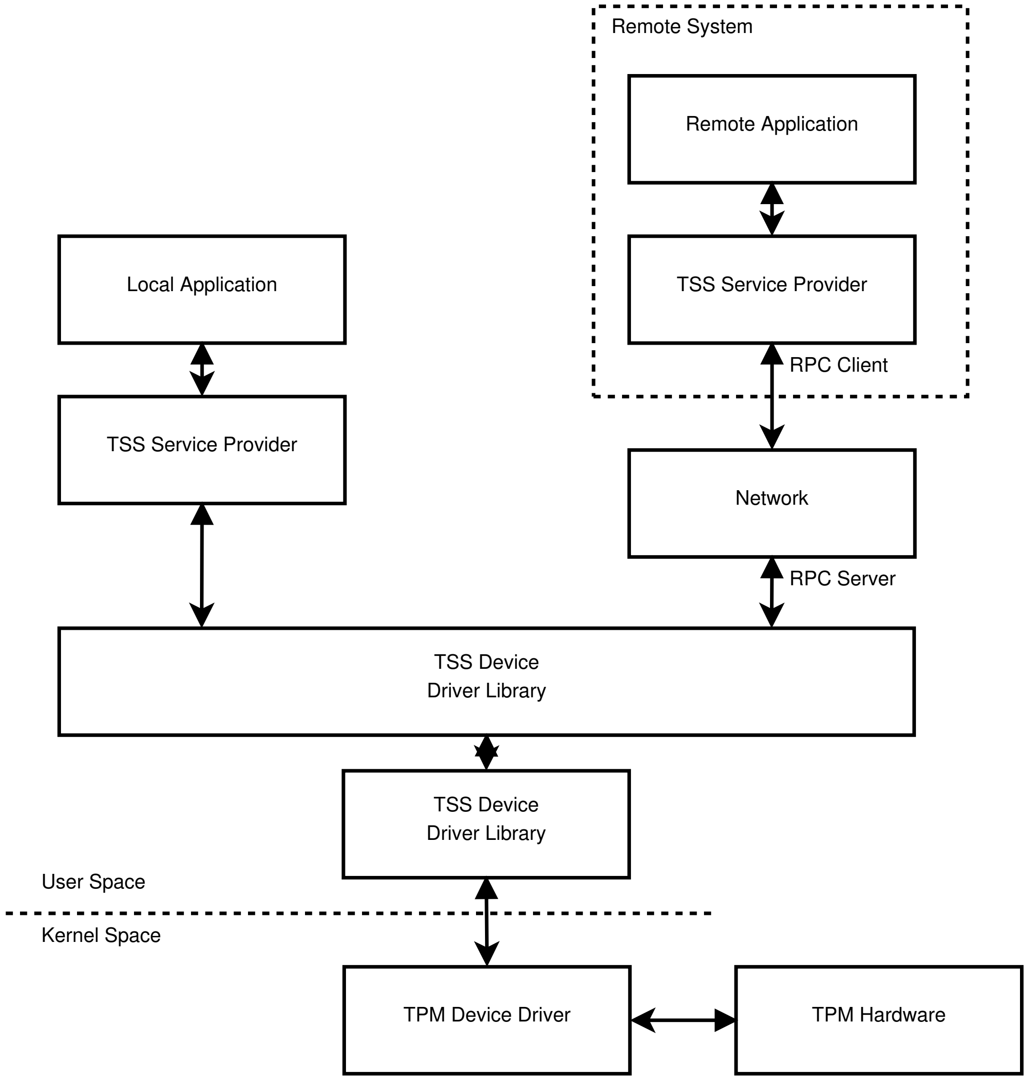

Figure 2.15: Basic Architecture of the TSS[CYC+07].

2.4.2 Implementations

The TSS has been implemented on different underlying techniques. TrouSerS was one of the first implementations and is written in *C*. However, there are version for *Java* and *Mono/.Net* too, as well as reduced complexity designs focused on embedded systems. An overview of the different implementations is given in table 2.2.

Table 2.2: Comparison of different TSS implementations.

	Trousers	jTSS	uTSS	doTSS
Language	C	Java	C++/Qt	.Net/Mono
Licence	CPL	Dual, GPLv2 for FOSS	Proprietary	Apache-2
Platforms	Linux/Windows Embedded	Linux/Windows/ Embedded	Linux/Embedded	Windows/Linux/ Embedded

Trousers

TrouSerS [TRO13] is IBM's version of the TSS. It is Free and Open Source Software (FOSS) and ships with additional tools to manage the TPM (*TPMTools*), as well as an extension for open Secure Socket Layer (openSSL) to use the TPM for asymmetric cryptography. The interface is very complex so it requires in-depth knowledge of the specification to use it safely.

jTSS

jTSS [IAI13] is an object-oriented *Java* implementation of the TSS. The object-oriented approach reduces the complexity but it does not reduce the effort of reading relevant parts of the TCG specification[RNK+11].

uTSS

μ*TSS* is an implementation by Sirrix [SZ10]. The design goals of this TSS targeted especially embedded systems and simplicity. However, there is no free implementation available.

doTSS

doTSS is an implementation with the goal to provide a developer friendly TSS [RNK+11]. It is written in C# and can be used with Microsoft's .Net and Mono frameworks.

Chapter 3

Architecture and Concept

This chapter provides an overview of the security enhancing features in a platform-independent way. Therefore, a comprehensive threat analysis with the help of STRIDE is done. Subsequently, the concepts of the threat mitigation technologies which are implemented in this work are presented. These mitigation technologies contain an authenticated boot method backed by a TPM for embedded systems and a way to attest the system's state over an insecure network connection. Moreover, an object-oriented wrapper for the TrouSerS TSS is presented in order to simplify the usage of the TPM capabilities and make the process less error prone. Additionally, the encryption of the filesystem is discussed and an approach to potentially speed up this process by the use of hardware accelerated AES encryption is presented.

3.1 System Overview

3.1.1 Basic Architecture

Figure 3.1 shows the basic environment.

The heart of the system is the communication controller, which is an embedded system including a CPU which is capable of running a 'full' operating system (i.e., provide a Memory Management Unit (MMU)). The communication controller is controlled with the client computer over a TCP-IP network. The network interface of the communication controller has to be considered untrusted as it may be connected to the internet. The communication controller communicates via Serial Periphial Interface (SPI) with $n >= 1$ application controller which are running critical controlling tasks defined in loadable software modules. These modules are defined by the user and loaded with help of the communication controller which also gathers information from all application controller and presents it over the network connection.

3.1.2 Attacker Model

Before it is possible to define the threats for the system, the targeted adversary has to be defined. First of all, it has to be assumed that an attacker is not able to get physical access to the communication controller or any application controller. As stated in section 2.3 physical access potentially breaks trusted computing approaches at little cost. None of

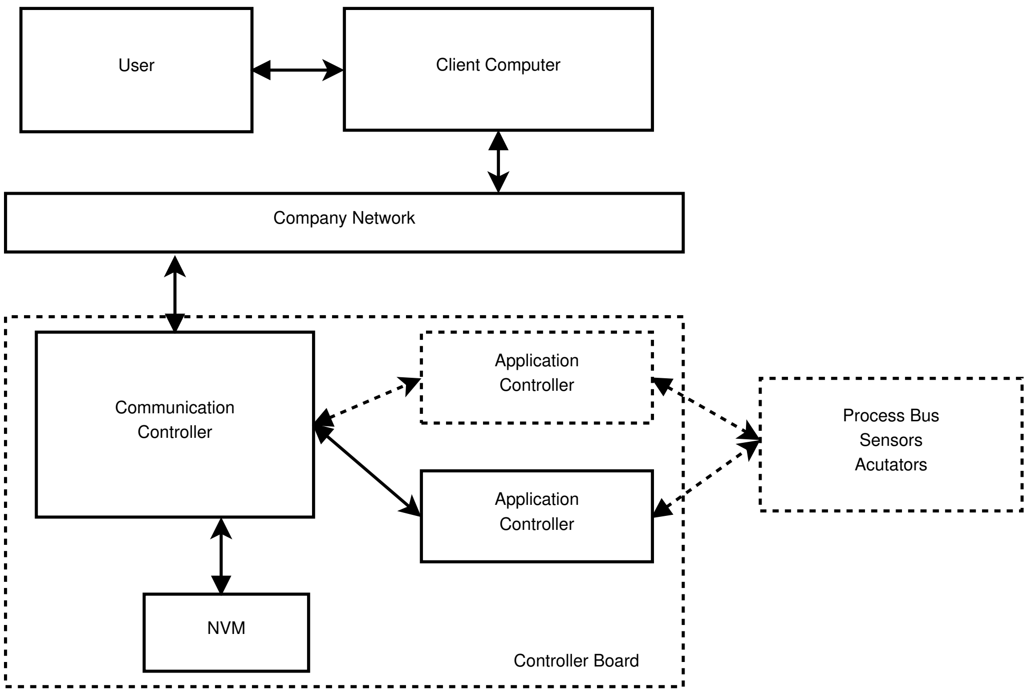

Figure 3.1: The basic system architecture

the described techniques is able to counter a malicious user with the aim to break systems security completely. The aim of such users may be breaking a DRM system in order to run unlicensed software. However, some techniques indeed increase the difficulty of such processes.

The attacker which has to be countered has the aim to affect the functionality of the system. This can be achieved by:

- Spoofing a user or a client computer in order to control the communication controller: Whenever an attacker is able to pretend it is a privileged user, it is possible to simply exchange the software running on the application controller.
- Get access to the communication controller[2] and send malicious messages to the application controller or intercept the communication between client and communication controller. In the former case it is possible to change the behavior of the application controller. The latter case opens the possibility to reveal user credentials or falsify information in order to force the user to force the user to perform tasks which may decrease system's functionality.

3.2 Threat Model

This section describes the threats to the system according to the risk analysis of the Security Development Lifecycle (SDL) [HL09]. In order to achieve this, some assumptions

[2]Since this device is connected to an open network and running a standard operating system it has to be assumed that privileged logical access to the attacker is available at least temporary.

about the system's security have to be done. Subsequently, the involved users, processes, data flows and the threats on these entities have to be defined.

3.2.1 Security Assumptions

Assumption 1

As discussed in Section 3.1.2, the main objective of the threat mitigation techniques described here are implemented to counter an adversary who wants to manipulate the system in order to decrease the functionality. Based on this claim, it can be assumed that no attacker has physical access to the communication controller and application controller at system's lifetime. However, it can not be guaranteed that the system is safe from access after disposal. Thus, a physical access is not allowed to disclose information that can be used to attack other systems.

Assumption 2

In order to simplify the model, it is assumed that a logical access on the communication controller of an adversary implies fully privileged access. There are some technologies to mitigate this elevation of privilege but it can not be countered completely.

Assumption 3

The third assumption states that the company's process bus is not accessible by the attacker since it is physically shielded. This assumption can be done safely because an adversary who is able to access this bus is also able to forge messages to and from sensors or actuators. In other words, it is possible to reduce the system borders and take out the process bus side completely from this security analysis.

3.2.2 Data Flow Analysis

In order to find possible attack vectors, the whole system has to be investigated under a data flow point of view. Figure 3.2 shows the data flow diagram of the system. The user, who may be a valid user or an adversary, communicates via a client process with the communication controller which itself talks to the application controller over SPI. The dotted lines represents trust boundaries. In this case, data flows from lowly to highly trusted entities and vice versa. Thus, these data flows have to be checked for validity.

In Figure 3.2, the software on the communication controller is shown as one big set of processes. Figure 3.3 shows a more detailed view on these processes. Only two of them are allowed to communicate over the network interface: The Secure Shell Daemon (sshd) for administration tasks and the TCS for user interaction. The sshd runs in a higher privileged world as the TCS and the kernel is the most trusted process. The NVM is modeled as data storage which is communicating with the kernel. Since all processes on the application controller are equally security relevant, a deeper view on this subsystem is not necessary.

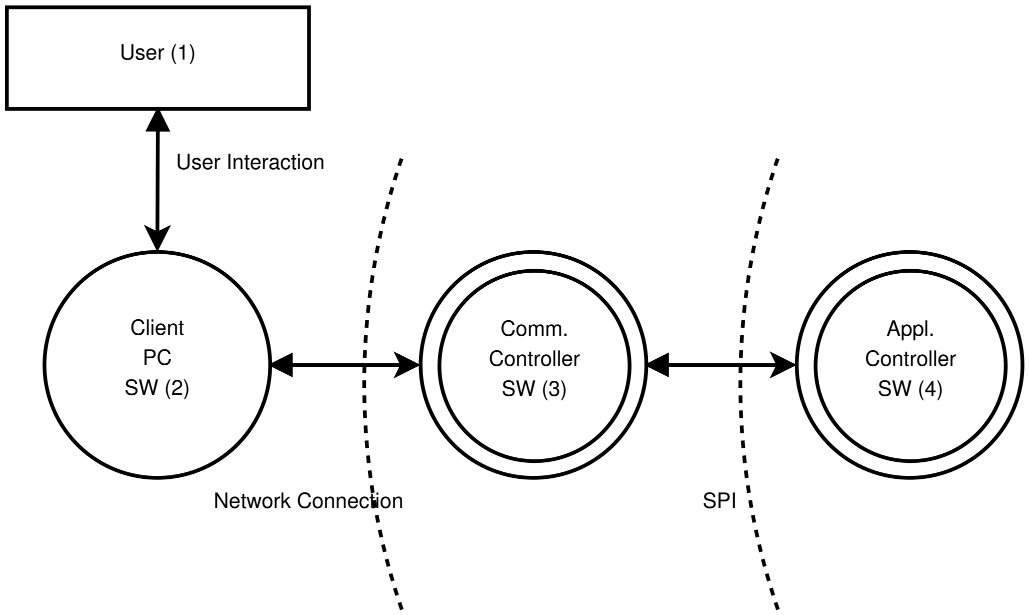

Figure 3.2: The basic data flow of the system: The user communicates with the communication controller via the client PC. The application controller is controlled by the communication controller.

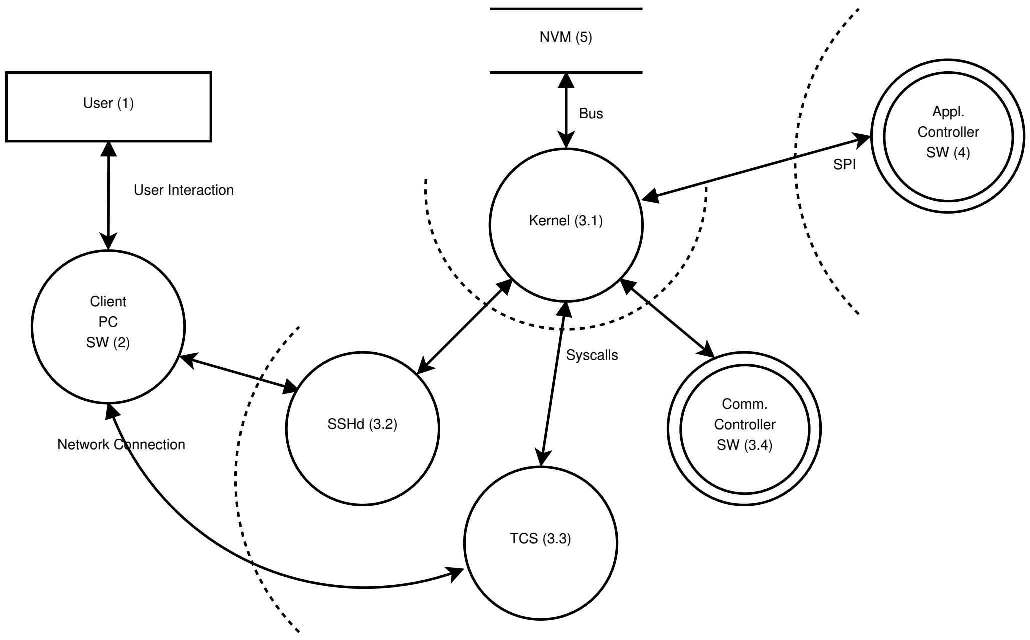

Figure 3.3: The data flow of the system with in-detail view into the communication controller.

In order to simplify the threat model, it is possible to merge some processes as shown in Figure 3.4:

- According to Assumption 2 all processes on the communication controller can be seen at the same trust level as an elevation of privilege on the system can not be excepted.
- The client process can be seen as a simple interface of the communication controller from security point of view. It does not matter if the user communicates with the help of the delivered software or uses other communication tools, so it can be discarded. This assumption does not hold when it is possible to use mutual attestation and the software running on the client PC is measurable.

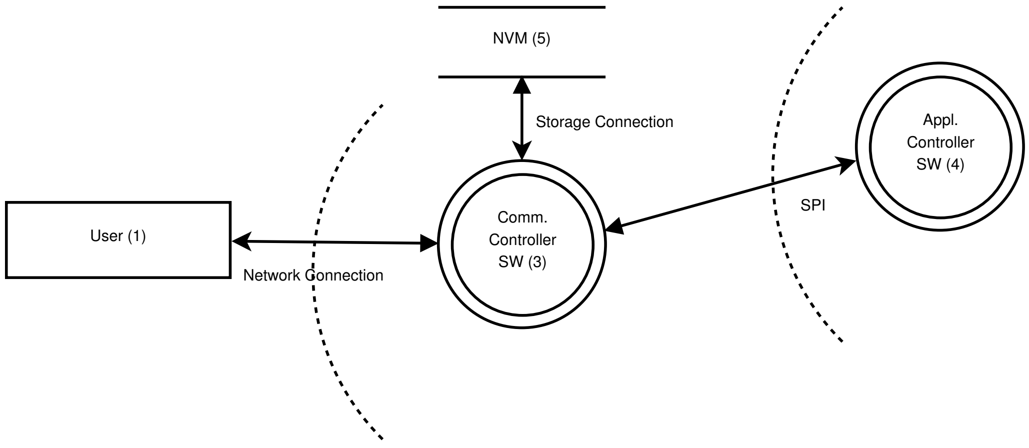

Figure 3.4: The data flow of the system with reduced complexity based on Assumption 2 and transparent client software.

Based on the simplified model, the modules which have to be analyzed can be summarized as in Table 3.1. This reduction may have been done on the basic data flow (Figure 3.2) too. However, a detailed view into important process groups may be necessary in order to determine all different trust levels to examine possible threats completely.

Table 3.1: The types of the different modules

Module Type	Modules
External Identity	User (1)
Data Store	NVM (5)
Process	C-Software (3), A-Software(4)
Data Flow	Network (1-3), Storage Connection (3-5) SPI(3-4)

3.2.3 Threat Definition

The modules defined in Table 3.1 are analyzed on possible threats with the STRIDE model of the SDL. For this purpose the threat tree patterns presented in [HL09] are used. Table 3.2 shows the basic attack types for the different kind of modules examined in this section.

Table 3.2: Possible threat types for the different modules

Module	Threat Types
User (1)	Identity Spoofing, Credentials disclosure, Replay Attack, Repudiation
NVM(5)	Data Confidentiality, Integrity of Executables
Controller Software (3)	Spoofing Controller, Tamper with Processes
Network Connection (1-3)	Link Confidentiality/Integrity

User (1)

The important threat types affecting the user are identity spoofing and repudiation. The authentication process has to provide the following properties:

- Credentials used by the user to authenticate have to be strong (i.e., not simply guessable) and there has to be a proper update policy.
- The credentials have to be stored on the communication controller. This storage has to be protected against unauthorized access.
- The transmission of the credentials has to be secure against eavesdropping.
- If key-based authentication is used, the key distribution process has to be secure.
- Commands and configurations placed by a user have to be logged in a tamper resistant way in order to ensure non-repudiation.
- Moreover, it has to be guaranteed that these commands cannot be placed by another person or replayed.

NVM (5)

The NVM has to be secured against tampering and information disclosure.

- Stored data like keys or user credentials may not be readable by unauthorized entities.
- Moreover, an altered binary or configuration file has to be detected in order to provide a secure system state.

Communication Controller Software (3)

The software running on the communication controller is the most critical part as it is the system's connection to the open network.

- First, it should not be possible for an adversary to spoof a communication controller because a user may present his credentials to the spoofed system.
- Moreover it should not be possible to tamper with the processes on the system, at least without notifying all communication partners of the altered state.

Application Controller Software (4)

Since the software running on this controller is safety-critical, it has to be ensured that only known and authorized software is running on this module. Since the only communication channel to the outside world is the SPI interface it has to be ensured that the communication partner is authorized and does not run software which is not trusted.

On-Board Data Flows

The *Storage Connection* (3-5) and the SPI(3-4) are not physically accessible for any adverse entity (Assumption 1). For this reason, they have not be secured against information disclosure or altering of data.

Network Connection (1-3)

Communication between the user (or the client software) and the communication controller is done via a relatively open network connection (the company's internal network).

- Since physical access can not be prohibited it has to be ensured that the link does not reveal information and that it is resistant against tampering.
- Critical data like encryption keys or credentials have to be encrypted the whole way between the two endpoints.
- Moreover, an altered message should be detected and discarded.

3.2.4 Security Enhancements

The main focus of the security enhancements lies on the communication controller since it is the system's gateway to outer world. Figure 3.5 shows the changes made in order to improve the system's resistance against attackers described in Section 3.1.2. These changes contain:

- Authenticated boot with the help of a TPM on the communication controller[3].
- Attestation of the communication controller's software state for the client software and the application controller.
- Encryption of parts of the communication controller's NVM.
- Acceleration of symmetric cryptography by enabling the on-chip cryptographic co-processor.

Authenticated Boot

The authenticated boot functionality on the communication controller is necessary to enable the possibility of remote attestation. Because a TPM is used, even a person with fully privileged access to the communication controller is not able to falsify measurements if the process is done properly as described in Section 3.3.1.

[3]The TPM can be simply exchanged with other similar devices as discussed in Section 3.3.1.

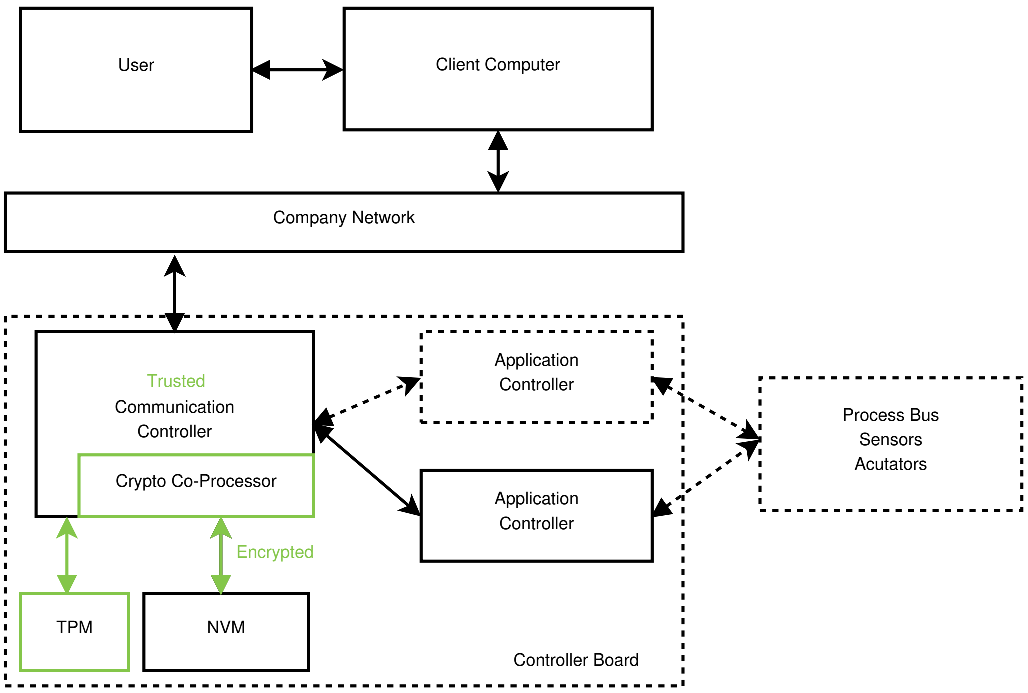

Figure 3.5: Overview of security enhancing features applied to the system.

Attestation

The attestation of the communication controller's state enables two important possibilities:

- A client software running on a system which is not affected by the attacker is able to detect the changes in the communication controller, rejects communication, and informs the user.
- An application controller simply rejects any communication with an affected communication controller. This may affect functionality, but enables the possibility to ensure at least the operation of some critical parts which are in charge for ensuring functional safety.

Disk Encryption

Disk encryption prevents data leakage on discharged systems. Providing that the key or the secret which is used for encryption is destroyed it is not necessary to wipe the disk with random data. Section 3.3.4 describes an approach with a non-software-readable key by using a cryptographic co-processor.

AES Hardware Acceleration

Both, the disk encryption and the network connections (Secure Sockets Layer (SSL)) use symmetric cipher algorithms. Especially on embedded systems with limited resources such algorithms need much CPU time. Section 3.3.5 shows an approach to drastically reduce CPU time by using a co-processor which is capable processing AES ciphers.

3.3 Subsystem Description

3.3.1 Authenticated Boot

The heart of the security enhancements represents the authenticated boot mechanism of the communication controller. Figure 3.6 contrasts the standard boot sequence with the authenticated boot backed by a TPM. Basically, the module which is about to be loaded is hashed and the results are saved into the PCRs at every stage.

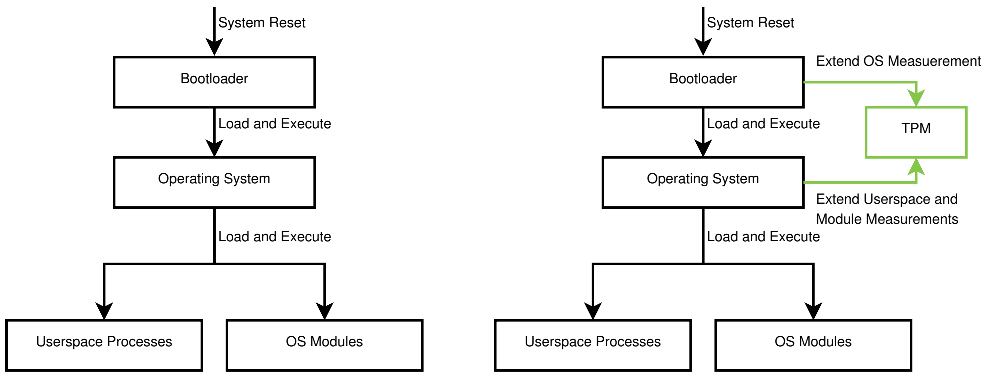

Figure 3.6: Overview of the authenticated boot mechanism compared to ordinary system initialization.

Bootloader and SRTM

The bootloader is the first code loaded on a platform startup or reset. Usually there exists some kind of reset-pointer on a specific address in memory which points to the start address of the bootloader.

The left side of Figure 3.7 shows a common way of its functionality. After some low-level initialization like CPU, timers, and memory a configuration file is loaded. The configuration is stored in some kind of NVM or compiled into the binary itself. Based on this information the operating system is loaded into the memory and the bootloader handles over the control by jumping to the load address.

The right side of Figure 3.7 shows the authenticated boot extensions of the bootloader. After the initialization, the TPM is started up. If the TPM does not respond properly, the boot process is stopped since system's integrity cannot be ensured at all. When the TPM has been enabled correctly, the configuration is measured and loaded. After the kernel is loaded into memory, the sections are extended too and the boot sequence is continued.

It should be noticed that this approach postulates a trusted bootloader. In conventional PC systems the CRTM is part of the BIOS boot block which is normally stored on ROM or, at least, only signed updates are possible[CPRS11]. On embedded systems this part is often fully exchangeable by any user who has privileged access so it has to be ensured that the bootloader cannot be altered. One approach may be the use of ROM or other types of One Time Programmable (OTP) memory for the first loaded binary. Other approaches based on hardware extensions like E-TPM or ARM TrustZone are described in Section 2.2. This work uses a hardware based secure boot function called High Assurance Boot (HAB)

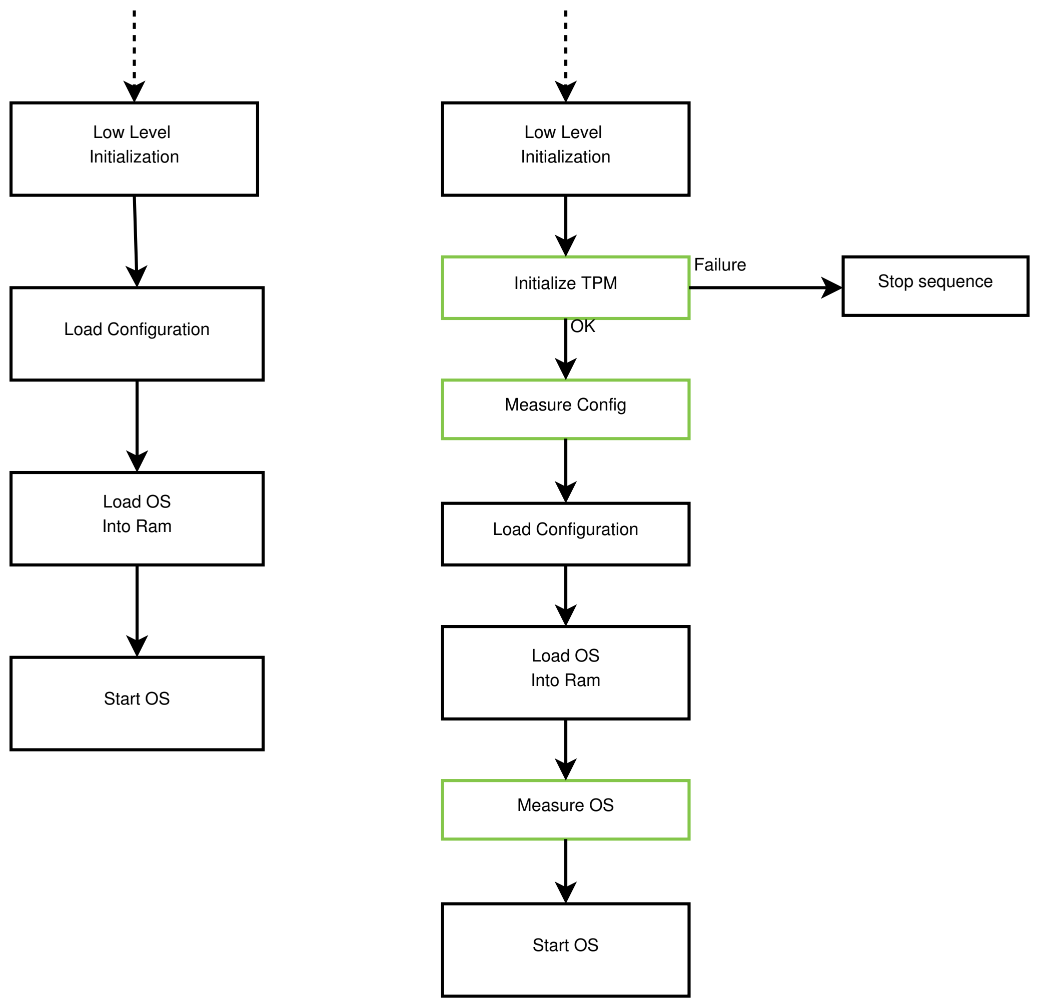

Figure 3.7: Bootloader extensions used for authenticated boot compared to ordinary setup.

which is a proprietary technology integrated in the used SoC as described in Section 4.2.

Operating System

After control is handed over, the operation system initializes itself and the used hardware as illustrated in Figure 3.8. Moreover, if present, additional late-loadable modules and drivers are loaded and filesystem is mounted. When the initialization phase is completed, the system starts the first user-space process (which is called the *init*-process here). Commonly this process is in charge to start all other processes.

The right side of Figure 3.8 shows the authenticated boot process of the operating system. The initialization of the TPM driver is done very early in the boot process in order to proceed with the measure and execute paradigm. Again, if the TPM does not react properly the boot process is halted. While the measurement of the late-loadable modules is essentially the same as the measurements taken so far, the measurement of user-space processes needs further investigation:

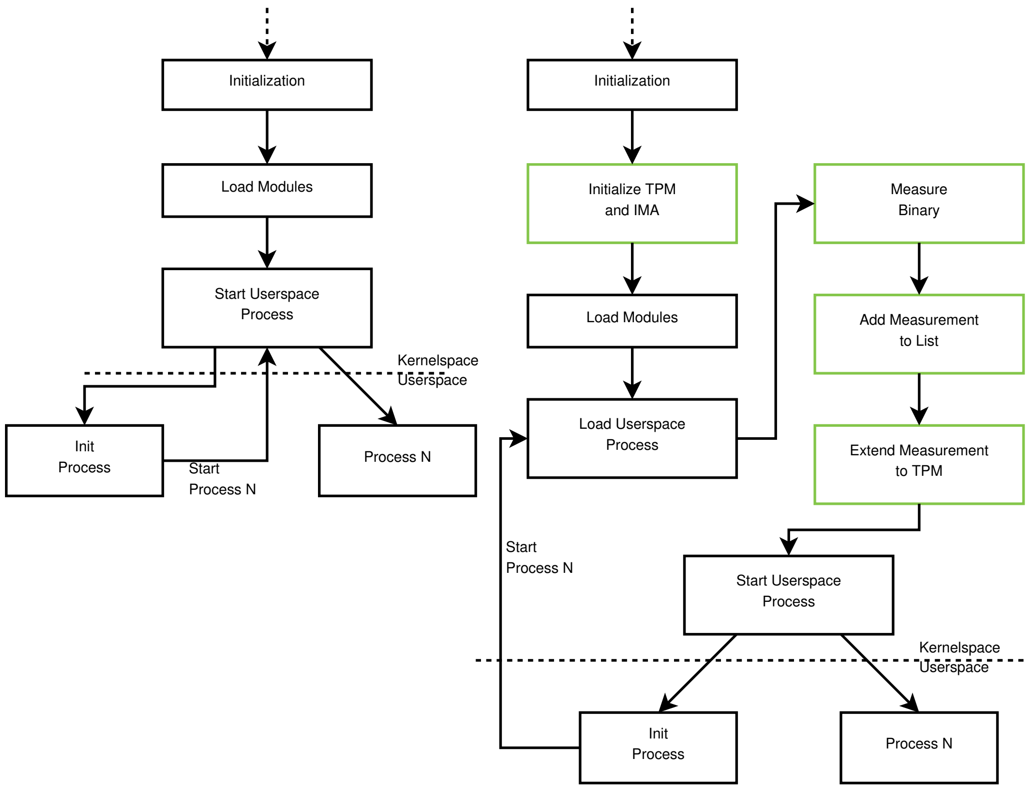

Figure 3.8: Extensions regarding the operating system used for authenticated boot.

First, as described in Section 2.1.4 an extension of a PCR is non-commutative. This means, that a different order of starting the same processes results in a different final measurement. On modern operating systems on off-the-shelf PCs this order is very hard to predict because modern initialization daemons execute start-up-programs simultaneously and their behavior strongly depend on the state of plug and play hardware. However, on embedded systems often only a small set of processes is started and the order may be maintainable. Additionally, these devices often use only hard wired external hardware. Anyway, the load time of late-loaded libraries or other files which have to be measured may depend on the scheduler and execution time of other processes. Thus it is not possible to rely on this behavior without accepting a number of false-negative measurements.

On the other hand, it is not entirely clear what to measure. Processes which are executed and their used libraries are the minimum. Furthermore configuration files of these processes may have a huge impact on system's integrity. Network traffic, user input and similar are also potentially malicious but a measurement of these values would make the approach impractical. However, starting a process or loading a configuration can be seen as the necessary consequence of a malicious user or network input.

The approach described in Section 2.2.1 has been implemented and solves these problems. The IMA is part of the operating system and performs measurements according to a configurable policy. Every measurement is extended to a PCR and the history of all measurement is saved in a list as tuple of result and input. As described in Section 3.3.3 this enables the possibility to ensure the integrity of the system by checking the measurement list wherein the integrity of the list itself is ensured with the PCR value. The standard policy measures every file a privileged user opens for read (or execute). This includes all executables, libraries and configuration files. Based on this information, the system is able to prove its integrity.

Alternatives to the Trusted Platform Module

The security enhancements described in this section depend on the use of a TPM. However, it is possible to use other, similar techniques with the same architecture. As described in Section 2.2 some architectures enable secure virtualization options. Moreover, if the constraints enable the possibility to just prevent execution of untrusted code, there is no need for attestation and the system simply rejects to load unknown modules.

3.3.2 Userspace Interface

As mentioned in Section 4.5.1 there are different implementations which enables user-space access of TPMs according to the TCG. However, the managed-code approaches may not be suitable for running on an embedded system if a complete runtime has to be installed only for this executable and the low-level implementation provided by the TCG is very complex.

Since complexity potentially leads to mistakes which may be harmful especially in security critical environments, a simple wrapper is introduced to hide the implementation details from TPM user.

Figure 3.9 shows the components used in this module:

- The *TPM* class provides methods to manage the actual device which may be a physically TPM interfaced within a driver in the operation system or a software emulator.
- The *TPMCommand* classes implement the logic to execute a specific command on a TPM. Hereby, it is notable that some commands like *AttestationRequest* can be run without an instance of *TPM* as they do not need it.
- The other classes are simple helpers which are used to store/load keys in files and contain methods to convert such structures between different formats.

This approach does not only hide complexity, it also simplifies error handling and container for error-prone structures like buffers in order to prevent common security problems. It is used as the base for the attestation process on both, the trusted and the remote side.

3.3.3 Attestation

In order to check the state of the communication controller and the software running on it, remote attestation is used. This section describes the technique on the client computer. However, the same approach can be adapted to the application controller.

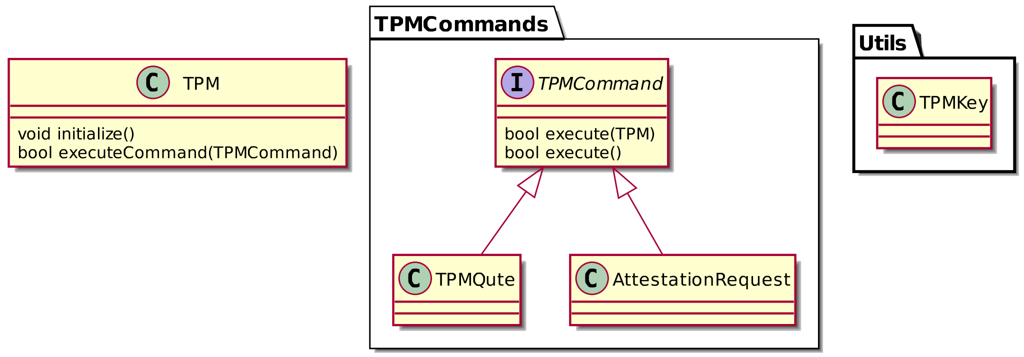

Figure 3.9: Interface for the TSS wrapper.

Attestation of the Communication Controller

Figure 3.10 describes the basic approach. It is assumed that the client software knows the public part of the AIK used by the TPM on the communication controller and the true measurement values of the different modules and applications.

- The user starts the communication software on the client computer.
- The software sends an attestation request with the interesting PCR numbers and a generated random to the communication controller.
- The communication controller loads the AIK on the TPM and generates a quote-request.
- The TPM processes the request by signing the PCR values with the private part of the AIK.
- The communication controller sends the quote and the measurement list of the IMA back to the client computer.
- The client computer checks the measurement list and verifies it with the quote. It also checks the quote itself and continues communication if everything is as expected.

Mutual Attestation

If the client computer also contains a TPM or some kind of D-RTM can be established, it is possible to enable a mutual attestation. This basically mirrors the process shown in Figure 3.10 after the communication controller attested it's state to the client computer. In this case, it is possible for the communication controller to prevent communication with a tampered communication partner. However, it does not protect against a malicious user which is able to authenticate on the system.

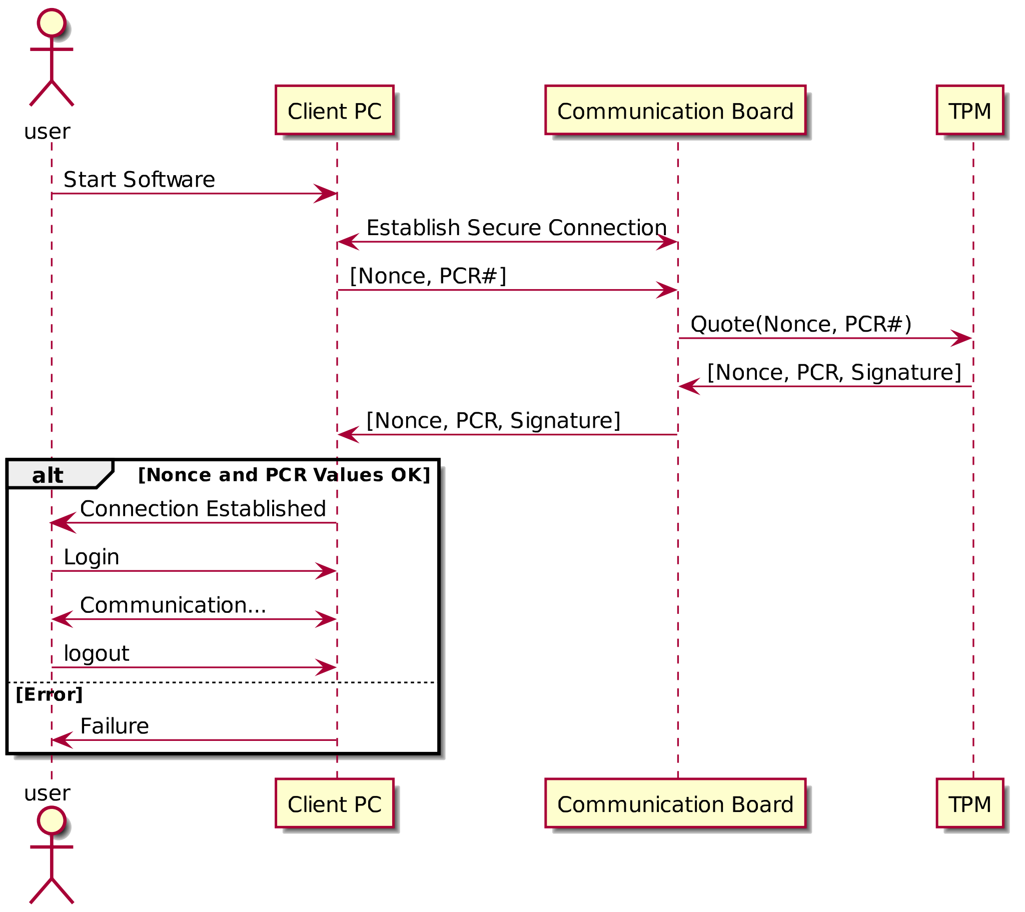

Figure 3.10: Attestation of the communication controller's state to the user.

Platform Configuration Register

Table 3.3 shows the PCRs and their content used for the attestation. With the information contained in these registers, the remote entity is able to check the bootloader configuration, the operation system and the processes started on the communication controller.

Table 3.3: The used PCR registers

PCR Number	Content
1	Bootloader Configuration
2	Operating System
10	Integrity Measurement Architecture

3.3.4 Encrypted Filesystem

Data On Rest (DOR) encryption is added to the system in order to prevent information exposure in case of unauthorized access or theft of the physical medium.

For performance reasons, the NVM is split up into two partitions. The root file system which contains the basic system what is not worth to be encrypted since it's build up of freely available components. The second encrypted partition holds data where an exposure should be prevented. Figure 3.11 describes the basic architecture of the encryption. While the root file system is mounted directly, the encrypted partition is mapped through a proxy which is in charge encryption and decryption of accessed data. This computation is transparent for user-space applications apart from some performance penalties. Moreover, the ciphering is done with a cryptographic co-processor which enables two important benefits:

- The dedicated hardware is generally faster than a software approach on most embedded systems.
- There exists a possibility to use a key which is only writable, but not readable by software.

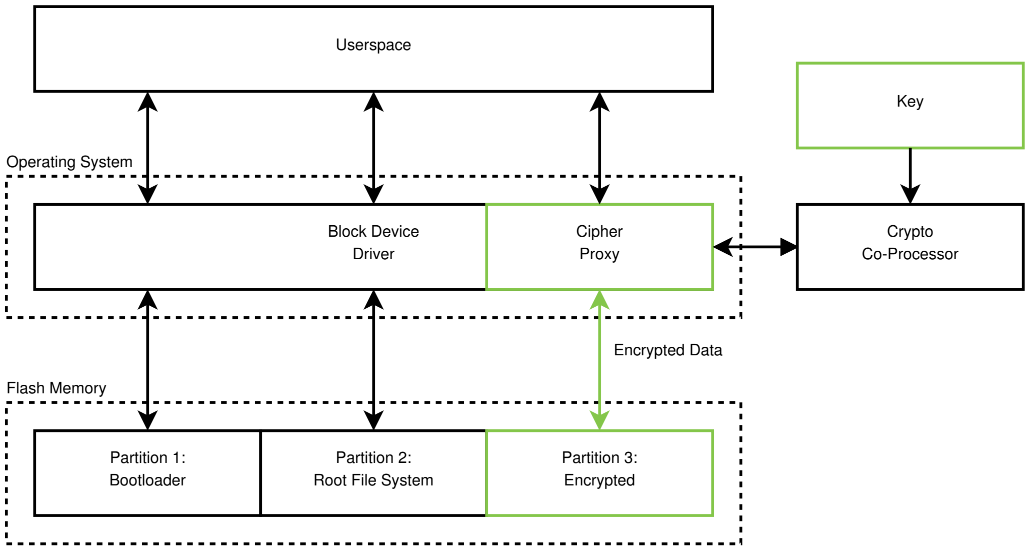

Figure 3.11: Partial encrypted filesystem with the help of a cipher proxy.

The key is stored on a special on-chip memory which can be fused to prevent the main CPU from reading. This approach binds the storage media to the current SoC. Another possibility to achieve this behavior would be the use of a key sealed by the TPM. However, this approach makes the key visible on the connection bus between the SoC and the TPM and in the system's Random Access Memory (RAM).

On end of life the non-readable key is destroyed by a random write what make a secure deletion of the physical media needless.

3.3.5 Hardware Accelerated AES

As mentioned before, on embedded systems ciphering potentially has a high impact on system's performance, especially when disk encryption is used. A cryptographic co-processor which does the actual cipher operations can mitigate these drawbacks. First, it is normally faster than a software implementation and on the other hand it gives the main CPU time for other operations.

This approach depends on the following features of the co-processor:

- Direct Memory Access (DMA) access is possible for the co-processor nearly randomly. [4]
- Additionally the co-processor is controlled with the help of description structures which contain the needed information like keys, source and destination addresses and similar.

Figure 3.12 illustrates the connection of the participating entities. An application wants to encrypt a plaintext into a ciphertext-buffer. Both buffers are virtually continuous from the application's point of view. In fact they may be scattered all over the physical memory by the operating system and the MMU. Somewhere in the memory exists a ring buffer of description structures mentioned before. The operating system fills these structures with the addresses of the scattered buffers and notifies the co-processor which itself throws interrupts whenever it finished a packet.

The basic workflow is shown in Figure 3.13. The application requests a cipher process with source and destination buffers and additional information like keys and mode of operation. After the request is placed, the application sleeps until its callback function is executed.

A thread inside the operating system continuously produces description structure with the physical addresses of the buffers until the ring is filled. After every produced description structure the co-processor is notified to compute the next work package.

The co-processor reads the description structure, performs the operation, and rises an interrupt to notify the CPU. If no other description structure is pending, it stops operation.

When the kernel thread receives an interrupt, the respective description slot is marked free. If not all data has been split up and send to the co-processor, producing is continued. Otherwise, if all data has been sent to the co-processor and all packets are finished, it executes the callback function of the application in order to report the finished process.

[4]This means that the co-processor is able to read and write from and to the whole RAM with little or less restrictions. These restrictions may be an address alignment as the co-processor is only able to address word-wise.

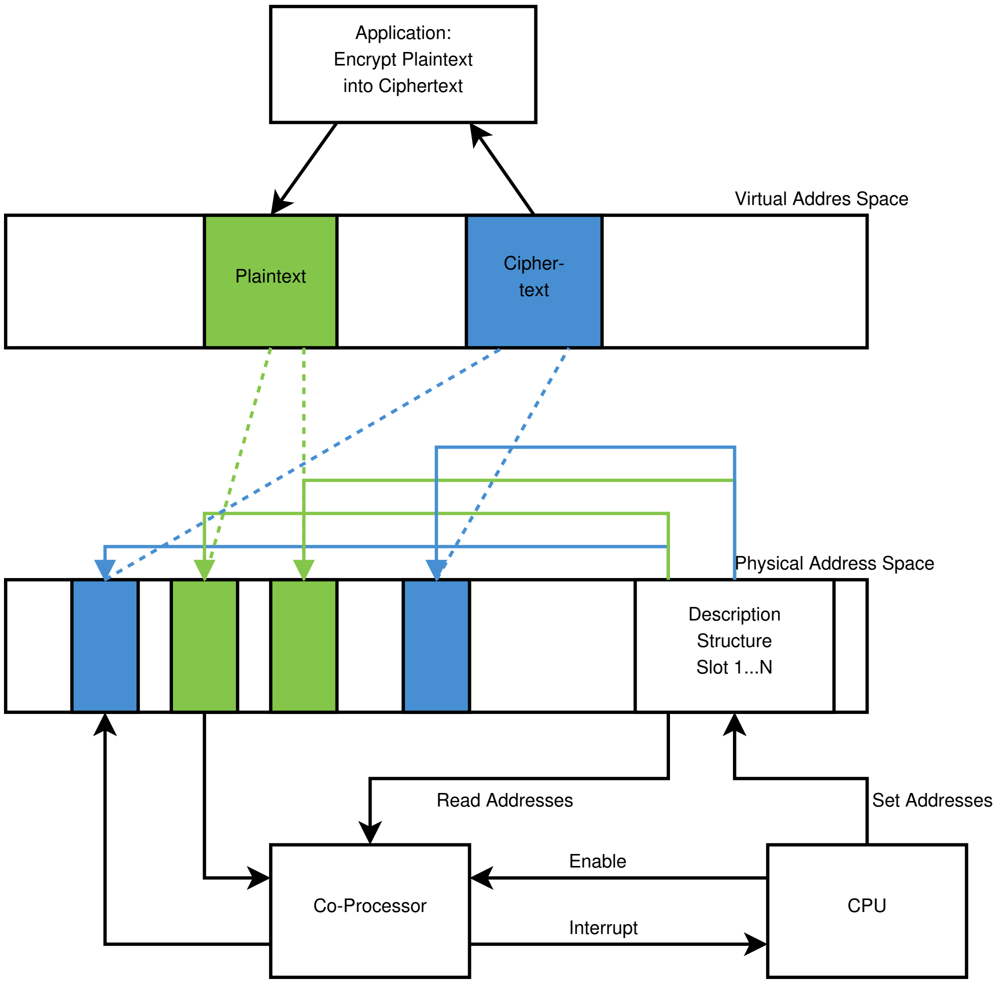

Figure 3.12: Memory handling: The main CPU is responsible to map the virtual addresses of the calling process to physical addresses which can be used by the co-processor.

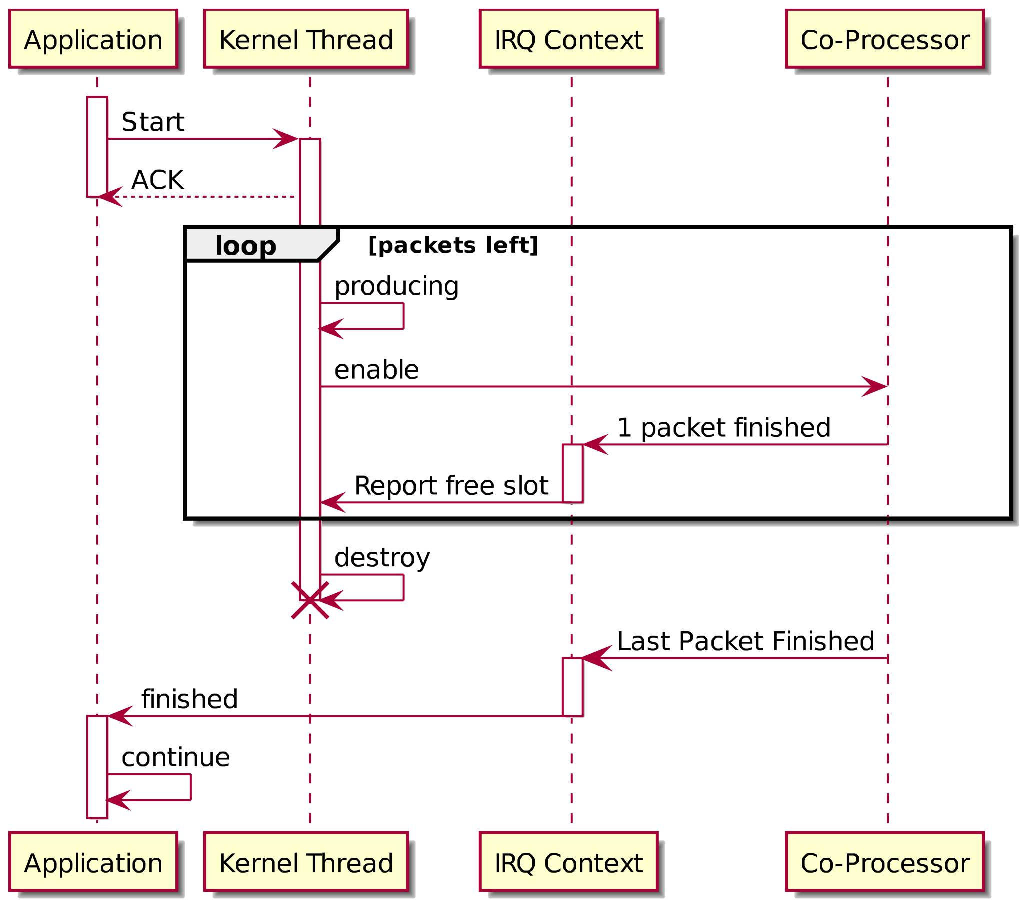

Figure 3.13: The workflow of the hardware-based AES encryption: The main CPU fills a ring-buffer which is processed by the co-processor.

Chapter 4

Design and Implementation

This Chapter refines the different concepts illustrated in chapter 3 and provides a detailed view on the actual implemented system. These descriptions also contain a short introduction of the used subsystems of the underlying software and technologies to clarify the design decisions which were made.

4.1 Target System

The target system system will be used in hydro-electronic power plants as control system, similar to PLCs. The system architecture has been presented in Section 3.1.1. This section provides an overview of the used hardware and software.

4.1.1 Hardware Components

The *Controller Board* represents a custom printed circuit board containing the communication controller, the application controller, the TPM and additional peripherals like flash and Electrically Erasable Programmable Read-Only Memory (EEPROM)s.

The communication controller is build with Freescale's i.MX28 SoC, what is basically an ARM9-core with additional building blocks for industrial and consumer applications. The most interesting additional block for this work is the Data Co-Processor (DCP). This co-processor can not only be used for simple memory operations like raw copying but also for AES ciphers and SHA-1 operations.

The application controller is built with the same SoC but is implemented twice in order to build redundancy for safety-critical operations on the process bus of the power plant.

As TPM, Atmel's A97SC3204T is used. It communicates via I2C and only depends on one external $33MHz$ clock source. Because of non-disclosure reasons, it was not possible to get a datasheet for this device. However, Atmel is selling an evaluation board for this TPM, which exposures the pin definition.

The client computer can be seen as ordinary PC running an off-the-shelf operating system.

4.1.2 Software

The system is still in development (as by July 2013), so the final versions of the used software may change. Table 4.1 shows the software modules used for the different controllers. They differ from the available source trees only in some smaller patches to adjust them to the custom hardware.

Table 4.1: Software modules used in the system.

Module Name	Used Software
communication controller Bootloader	U-Boot 2012.10
communication controller Operation System	Linux 3.7-rc5
application controller Bootloader	U-Boot 2012.10
application controller Operating System	SafeRTOS
Distribution Software Development Kit	ELDK 5.2

4.2 U-Boot and Trusted Boot

4.2.1 Overview

"Das U-Boot" (The Universal Boot Loader) [Den13a] is an open source boot loader maintained by DENX Software Engineering. It is mostly used in embedded systems since it is able to run on many different architectures (ARM, AVR32, PowerPC and MIPS[Den13b]). U-Boot is highly configurable and includes a simple Command Line Interface (CLI), which can be used to influence the boot process and perform I/O operations. It also implements device tree support [Cor11] which is used by Linux (since version 3.7) in order to remove hard coded board information in source code[5]. It also supports TCP/IP which enables the possibility to load a kernel image via the Trivial File Transfer Protocol (TFTP).

4.2.2 Basic Boot Process

The basic boot process for the communication controller is shown in figure 4.1. The controller jumps to U-Boot which is typically stored on some non-volatile memory like flash or EEPROM. U-Boot does some low-level initialization (especially CPU and RAM) and copies itself onto the top of the memory. The startup-configuration, which contains the address of the TFTP-server and the kernel boot arguments are parsed and the boot-image is loaded into memory. However, this part is only for debugging and development in order to simply exchange the system image. The loaded image is verified with a CRC32 checksum and the program counter is set to the start address of the kernel. At this moment U-Boot gives up control over the system. The kernel does not know where the bootloader is located and treats the whole memory as empty.

[5]Until Linux version 3.7, each board (as in the physical printed circuit board with a CPU and external hardware) had to have its own board file, a source file which sets up the resources and devices. The device tree is a hierarchical structure describing the different hardware modules and their used resources. The kernel parses this file and loads the needed drivers. This leads to the advantage that the kernel has not to be recompiled after adding/removing an external module.

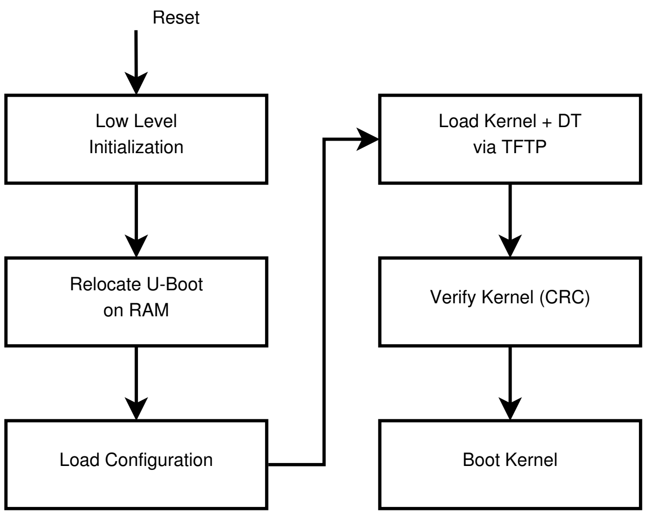

Figure 4.1: Basic boot process of U-Boot on i.MX28: After some low level initialization, the kernel image is loaded via TFTP. The received image is checked against a CRC value and booted.

4.2.3 Trusted Boot Process

In order to enable trusted computing capabilities, the bootloader has to initialize the TPM and extend the states of the kernel and its configuration to the PCRs. Figure 4.2 shows the extended boot process where the TPM is initialized and tested after relocation. If the TPM does not react properly, the boot process is stopped since the reporting capabilities cannot be guaranteed. The kernel image and the device tree are measured after they are loaded to RAM. This means that the binaries are hashed and the result is saved to the first PCR register of the TPM.

4.2.4 Trusted Boot with Root of Trust

As described in Section 2.1.2 it has to be ensured that the CRTM, which is U-Boot in this case, has to be in a trusted state at any cases. This is ensured by the HAB feature of the i.MX28. In the complete boot process U-Boot is only executed if the signature of its binary is valid. The use of this functionality binds the system to Freescale's architecture since it is proprietary. However, there are other possibilities to accomplish similar results like storing U-Boot on read-only memory[6] or other techniques discussed in Section 3.3.1.

4.2.5 U-Boot I2C Interface

U-Boot provides a basic interface for I2C transfers. Each controller with an I2C bus module has to implement a driver and register it in the board file. The interface is shown in Listing 2 and is defined as follow:

- *i2c_init* is used to initialize to module and to set the bus speed. This contains usually

[6]This approach demands on physical access to change the bootloader. Since physical access devastate TPM security as mentioned in Section 2.3, a read-only bootloader may be enough for a ROTFM.

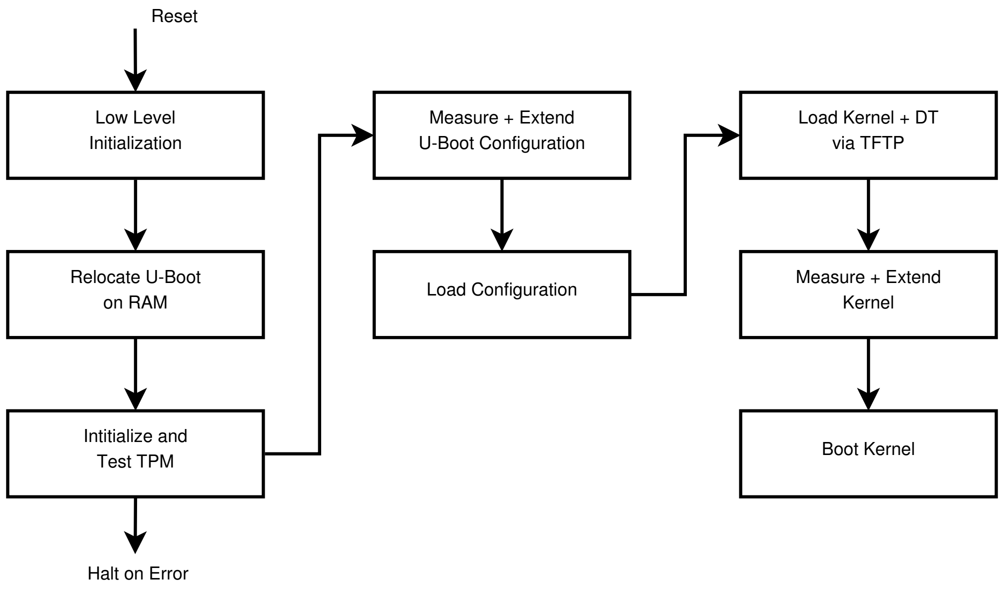

Figure 4.2: Trusted Boot Process in U-Boot: Before any additional content is loaded, the TPM is initialized and tested. The configuration, as well as the kernel are hashed and extended before execution.

a reset of the controller and some writes of configuration words to specific registers used by the I2C-module.

- *i2c_probe* is used to check whether a device with the given address is present on the bus. This can be done by a simple write call without any payload. The master sends the initialization byte and checks the presence of the acknowledge bit.
- *i2c_write* is used to write a buffer to the chip with a specific chip-address and *i2c_read* is the corresponding read function.

For some reason, U-Boot developers decided to define the read and write interfaces with additional register addresses for the slave device. However, since the implementation simply concatenates the *addr* and the *buffer* and sends the resulting buffer to the bus, it is possible to simply set $alen = 0$ in order to use the bus as serial interface. The bus driver used for the i.MX28 did not support this kind of operations so it had to be patched.

```
void i2c_init(int speed);
int i2c_probe(uchar chip);
int i2c_read(uchar chip, uint addr, int alen, uchar *buffer, int len);
int i2c_write(uchar chip, uint addr, int alen, uchar *buffer, int len);
```

Listing 2: Interface for I2C modules in U-Boot.

4.2.6 U-Boot TPM Interfaces

The TPM subsystem of U-Boot consists of three parts:

- The TPM device driver defined in *include/tis.h*
- The TPM commands defined in *include/tpm.h*
- The TPM command line interface defined in *common/cmd_tpm.c*

TPM Driver

As shown in Listing 3, a TPM driver has to implement the following interface in U-Boot:

- *tis_init* is used to initialize the TPM driver.
- *tis_open* and *tis_close* are used to open and close the connection to the TPM but they are not needed for this device.
- *tis_sendrecv* is used to send a raw command to the TPM.

```
int tis_init(void);
int tis_open(void);
int tis_close(void);
int tis_sendrecv(const uint8_t *sendbuf, size_t send_size, uint8_t *recvbuf,
                        size_t *recv_len);
```

Listing 3: Interface for TPM drivers in U-Boot.

TPM Library

The interface for the TPM usage is shown in Listing 4. The important functions for this work are:

- *tpm_startup* issues the startup command.
- *tpm_self_test_full* starts the built-in self-test of the TPM.
- *tpm_extend* and *tpm_pcr_read* execute the associated TPM commands for measurement handling.

```
uint32_t tpm_startup(enum tpm_startup_type mode);
uint32_t tpm_self_test_full(void);
uint32_t tpm_extend(uint32_t index, const void *in_digest,
                    void *out_digest);
uint32_t tpm_pcr_read(uint32_t index, void *data, size_t count);
```

Listing 4: TPM Interface in U-Boot.

As by U-Boot version 2013.01, the library does not support binding, sealing, or quoting. However, this is not necessary for authenticated boot but may be necessary in case of secure boot since it would be possible to seal the U-Boot and kernel binaries.

TPM Command Line Interface

The command line interface is used to provide TPM functionality to the user by enabling a set of functions on the command line.

4.2.7 Trusted Boot Integration

TPM Driver

The driver for the actual TPM in U-Boot mainly maps between the *tis_sendrecv* the corresponding *i2c_** calls. The architecture of the driver is very similar as the Linux version which is shown in Section 4.3.3.

Measuring the Kernel

There are some community projects running which target the integration of authenticated boot into U-Boot [Gla13]. Since official support can be expected soon, this work simply provides a proof-of-concept implementation.

First, the TPM has to be initialized with the startup command shown in Listing 5.

```
tpm startup TPM_ST_CLEAR
```

Listing 5: U-Boot TPM startup

In order to measure the kernel, U-Boot's TPM library is used in an extended version of the *bootm* command. This extended version simply calculates the hash of the binary and extends the value to PCR 1.

4.3 TPM in Linux

As mentioned before, the corresponding functions have to be implemented into the operating system in order to fully provide trusted computing capabilities on the communication controller. To achieve this, the following tasks have to be considered in the Linux kernel:

- In order to enable the TPM, a device driver has to be implemented.
- To actually use the TPM from user-space and enabling trusted computing technologies like attestation for applications, the device interface has to be enabled.
- The IMA implementation for Linux has to be enabled as described in Section 4.4.

Linux provides most of the functions needed for these tasks. The only part which has to be implemented (as by version 3.7) is the driver for the actual TPM. However, there are some subsystems which are involved in the communication as shown in figure 4.3:

- User-space applications call kernel functions via Input/Output Control (IOCTL) system calls on a virtual file, the */dev/tpmX* character device.

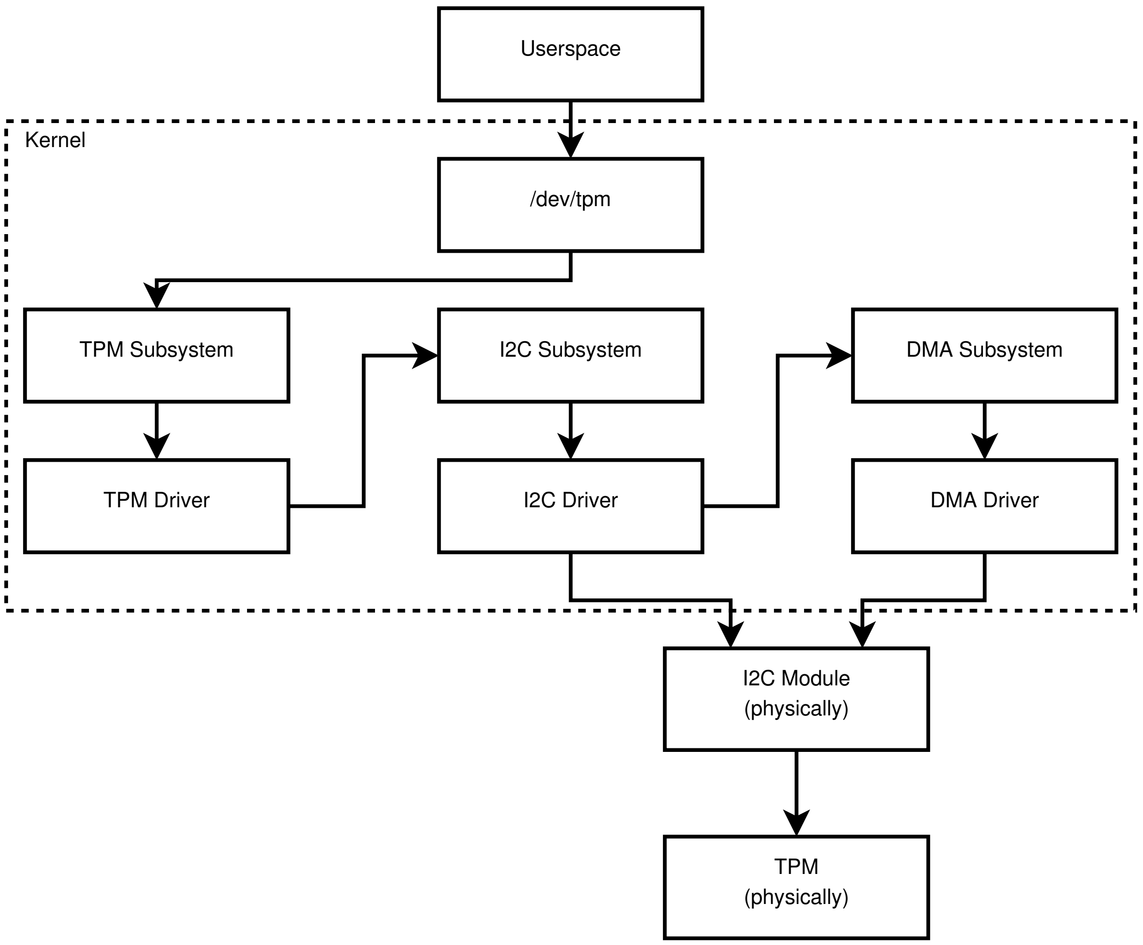

Figure 4.3: The subsystems involved in the TPM communication in Linux: The user-space applications interfaces via the character device with the *TPM Subsystem* which itself holds the *TPM Driver*. The TPM is connected on the I2C bus so the *I2C* subsystem is used for communication. On the i.MX28 the data transfer with the *i2c* module is done with DMA so this subsystem has to be considered too.

- The *TPM Subsystem* provides high-level functionality to communicate with TPMs and driver handling.
- To actually connect a specific TPM, the *TPM Driver* is used. It implements a simple interface to send and receive messages to or from the TPM.
- Since the actual TPM is connected via I2C, the *TPM Driver* calls the *I2C Subsystem* for message handling.
- The *I2C Module* on the i.MX28 is controlled by special function registers and DMA for data transfer, so the *I2C driver* uses this subsystem. As described later in this section, when using DMA transfers, some pitfalls regarding addressing have to be considered.

All implementation details in this section hold at least from Linux version 3.7 up to 3.10. However, it should be able to adopt this work for later versions with little or no effort. In order to enable the *TPM Driver* on Linux versions smaller than 3.7, little adoption regarding the device tree is necessary. The rest of this section describes the steps needed to implement the *TPM driver* for usage on the i.MX28.

4.3.1 I2C Subsystem

Overview

The Linux I2C subsystem provides an API for both, the actual I2C bus driver and client drivers. The subsystem architecture is illustrated in Figure 4.4.

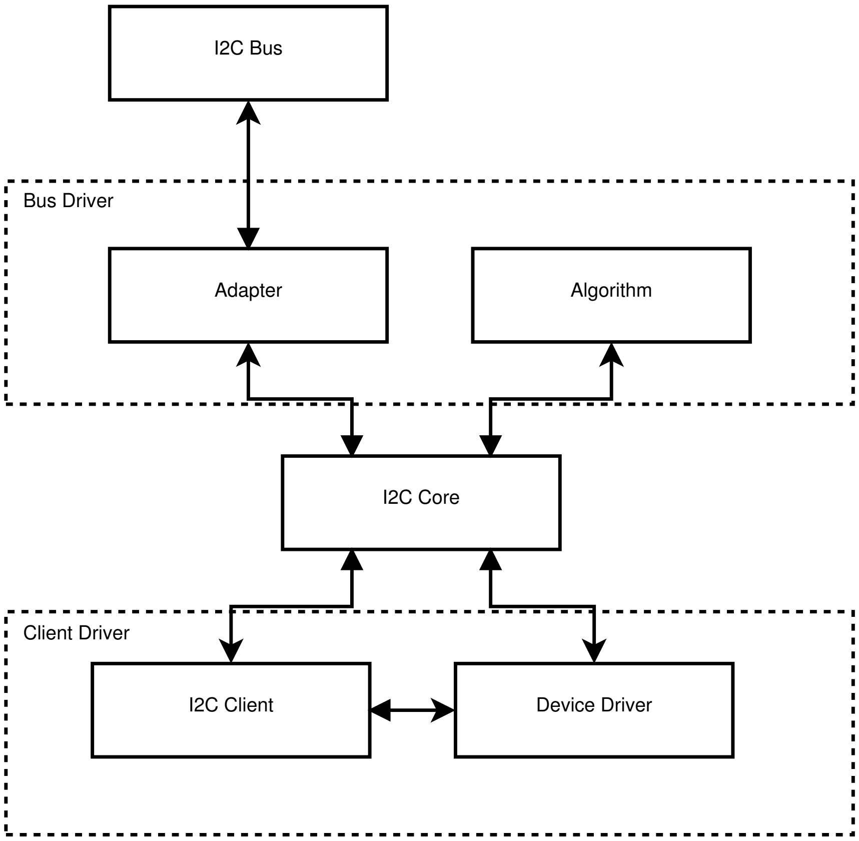

Figure 4.4: The Linux I2C subsystem: The actual I2C bus is controlled by the *Adapter* and *Algorithm* driver. To enable a new slave device, one has to implement a *Device Driver* which handles a new *I2C Client* structure.

On the bus side, the *Algorithm* driver is used for general functionality which is used over various systems and contains the data transfer function. The specific *Adapter* driver represents the code needed to communicate with the actual I2C module used in the system. In case of the i.MX28 (which uses the driver in *drivers/i2c/i2c-mxs.c*) the *Adapter* driver implements both, the *Adapter* and the *Algorithm* interface since it uses the DMA API.

On the client side, a *Device Driver* implements the functionality to communicate with an I2C device. For each connected device, the subsystem holds an *I2C Client* structure which itself contains information like the bus address.

I2C Client Driver Implementation

In order to implement a new device for the I2C bus, one has to implement an *i2c_driver* structure and register it in the subsystem. The important part of the structure definition is shown in Listing 6. The *i2c_client* structure is illustrated in Listing 7.

```
struct i2c_driver {
        int (*probe)(struct i2c_client *, const struct i2c_device_id *);
        int (*remove)(struct i2c_client *);


        void (*shutdown)(struct i2c_client *);
        int (*suspend)(struct i2c_client *, pm_message_t mesg);
        int (*resume)(struct i2c_client *);
};
```

Listing 6: I2C driver structure in Linux.

On initialization, the driver creates a new *i2c_driver* structure and set the function pointer for *probe()* and *remove()*. The other three functions can be used for power saving features but they are not mandatory and skipped in this implementation. After creation, the driver is registered in the *I2C Subsystem* and is ready to use.

```
struct i2c_client {
        unsigned short flags;
        unsigned short addr;

        char name[I2C_NAME_SIZE];
        struct i2c_adapter *adapter;
        struct i2c_driver *driver;

        int irq;
};
```

Listing 7: I2C client structure in Linux.

The *I2C Subsystem* creates a new *i2c_client* structure each time a new device is detected and a driver is registered for this kind of device. In the case of the TPM driver, the detection is done with the help of the devicetree subsystem. As shown in Listing 8, the devicetree holds the information, that a TPM is connected to bus 0 on address $0x29$. At registration time, the driver uses the same compatibility string as the devicetree (*tpm_i2c_atmel*). Thus the subsystem is able to identify the driver which has to be used for this device and calls the *probe()* function.

Within this function, the driver sets the remaining fields of the *i2c_client* structure and tries to communicate with the TPM. If the TPM is accessible (i.e., communication works), the function returns 0 and the device is set up.

There are other possibilities to report the existence of a device on the I2C bus beside the device tree. It can be done hardcoded or with a write operation on a file in *sysfs*. However, these methods are not very portable and considered for debugging purposes only.

```
[...]
i2c0: i2c@80058000 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins_a>;
        clock-frequency = <400000>;
        status = "okay";
        [...]
        tpm0: tpm@29 {
                compatible = "tpm_i2c_atmel";
                reg = <0x29>;
        };
};
[...]
```

Listing 8: Registering the TPM within the devicetree.

4.3.2 TPM Subsystem

The external TPM interface in Linux is shown in Listing 9. It provides functions to read and extend PCRs, to get a random and to send a message to the device. These functions are provided over the whole kernel whenever TPM support is activated.

```
extern int tpm_pcr_read(u32 chip_num, int pcr_idx, u8 *res_buf);
extern int tpm_pcr_extend(u32 chip_num, int pcr_idx, const u8 *hash);
extern int tpm_send(u32 chip_num, void *cmd, size_t buflen);
extern int tpm_get_random(u32 chip_num, u8 *data, size_t max);
```

Listing 9: Linux TPM interface.

TPM device drivers have to implement a *tpm_vendor_specific* structure which is shown in Listing 10. This structure contains the *send()* and *recv()* functions in order to enable the communication with the TPM. A *tpm_chip* structure is created by the subsystem when the driver calls *tpm_register_hardware()* with the *tpm_vendor_specific* structure as parameter.

```c
struct tpm_vendor_specific {
[...]
        int (*recv) (struct tpm_chip *, u8 *, size_t);
        int (*send) (struct tpm_chip *, u8 *, size_t);
        void (*cancel) (struct tpm_chip *);
        u8 (*status) (struct tpm_chip *);
[...]
};
```

Listing 10: Important part of the *tpm_vendor_specific* structure used to describe a TPM device driver

The subsystem generates a character device on */dev/tpmX* for each registered TPM to enable user-space access. Moreover, there are some files created on *sysfs* where status parameters, as well as a list of PCR values are exported.

4.3.3 TPM Device Driver

Figure 4.5 illustrates the combination of the different parts mentioned in this section in order to implement the driver for the used TPM:

- At initialization time, the module simply creates an *i2c_driver* structure and registers it in the I2C subsystem.
- When the *probe()* function is called, the driver generates the *tpm_vendor_specific* structure and a bounce buffer for the TPM and registers it in the TPM subsystem. The bounce buffer is needed because the I2C driver uses DMA: The DMA module on i.MX28 depends on addresses aligned by 4 bytes ($addr\&0x03 = 0$). Moreover, there are some other constraints for memory to use it for DMA. Since neither the I2C nor the DMA driver checks these conditions, it has to be done in the TPM driver. After the *tpm_chip* structure is created and registered, the *tpm_selftest* command is executed in order to check presence and functionality. If everything works as expected, the TPM can be used.
- The actual transfer is to and from the bounce buffer, so data has to be copied before and after communication to the proper buffers.

4.4 Integrity Measurement Architecture

4.4.1 Overview

The IMA described in Section 2.2.1 has been added to the Linux kernel by IBM in 2005 [Cor05]. The main goals of this implementation was to add an integrity measurement mechanism which is complementary to Mandatory Access Control (MAC) implementations like Security Enhanced Linux (SeLinux) with the following functions[IMA13]:

- *Collect*: Measure a file before it is accessed.

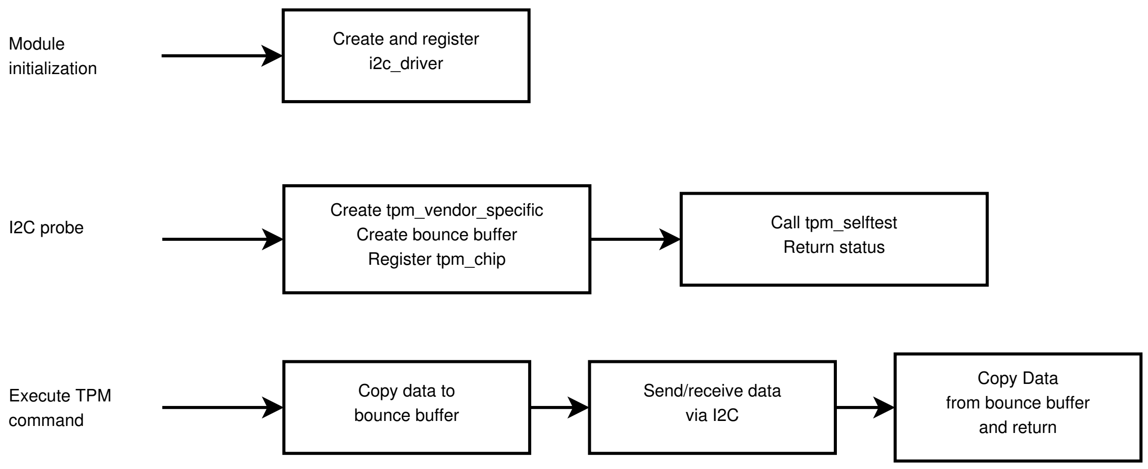
Module initialization
Create and register i2c_driver
I2C probe
Create tpm_vendor_specific
Create bounce buffer
Register tpm_chip
Call tpm_selftest
Return status
Execute TPM command
Copy data to bounce buffer
Send/receive data via I2C
Copy Data from bounce buffer and return

Figure 4.5: TPM driver overview

- *Store*: Store the measurement list and, if present, extend the cumulative measurement to a PCR on a hardware TPM.
- *Attest*: If a TPM is present, it is used to sign the IMA PCR in order to attest system's integrity to other entities.
- *Appraise*: Provide the possibility to validate measurements against known values locally.
- *Protect*: Securely protect the security critical file attributes (like the appraisal hash) against offline attacks.

The latter two functions are not implemented in this work, since local appraisal is not forced in this work. However, the appraise functionality can be enabled similar to the first IMA functions by the IMA-policies.

The basic data flow of an integrity measurement starts with an event like binary execution and follows the following data flow:

- If enabled, depending on the event, one of the hooks is called.
- After some preliminary checks, the *process_measurement* function is called.
- Based on the policy, the *ima_get_action* function decides the actions which have to be performed (measure and/or appraise).
- If a policy matches, the measurement is taken by *ima_collect_measurement*.
- At last, the measurement is stored to the measurement list (and the cumulative hash to the PCR by *ima_store_measurement*).
- If appraising is activated, the measurement is checked against a predefined value and the hook returns with nonzero in case of mismatch.

4.4.2 Collect Measurements

In order to collect the measurements, the hooks shown in Listing 11 are implemented and controlled by a policy file:

```
extern int ima_bprm_check(struct Linux_binprm *bprm);
extern int ima_file_check(struct file *file, int mask);
extern int ima_file_mmap(struct file *file, unsigned long prot);
extern int ima_module_check(struct file *file);
```

Listing 11: Measurement hooks of IMA in Linux

- *ima_bprm_check* is used to check binaries before they are executed. The *Linux_binprm* structure is created by the *do_execve_common* function in *fs/exec.c* and stores information like name and parameters of an executed file.
- *ima_file_check* measures files based on the policy, where the *mask* parameter contains access information like *MAY_READ*, *MAY_WRITE* and *MAY_EXECUTE*
- *ima_file_mmap* measures all files which are memory-mapped executable.
- *ima_module_check* measures all loaded modules. Since the kernel in this work is monolithic (i.e., all modules are compiled into a single kernel-binary), this function is not used.

4.4.3 IMA-Policies

The IMA-policies define what have to be measured. The standard policy is shown in Listing 12. Basically it disables measurement for the virtual file systems (*/proc*, */sys* and so on) and enables measurement for all files opened or mmap'd executable, all files read by the root user (uid = 0) and all kernel-modules.

```
static struct ima_rule_entry default_rules[] = {
        {.action = DONT_MEASURE,.fsmagic = PROC_SUPER_MAGIC,
                                                   .flags = IMA_FSMAGIC},
        {.action = DONT_MEASURE,.fsmagic = SYSFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = DEBUGFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = TMPFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = RAMFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = DEVPTS_SUPER_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = BINFMTFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = SECURITYFS_MAGIC},
        {.action = DONT_MEASURE,.fsmagic = SELINUX_MAGIC},
        {.action = MEASURE,.func = MMAP_CHECK,.mask = MAY_EXEC,
         .flags = IMA_FUNC | IMA_MASK},
        {.action = MEASURE,.func = BPRM_CHECK,.mask = MAY_EXEC,
         .flags = IMA_FUNC | IMA_MASK},
        {.action = MEASURE,.func = FILE_CHECK,.mask = MAY_READ,
                                              .uid = GLOBAL_ROOT_UID,
         .flags = IMA_FUNC | IMA_MASK | IMA_UID},
        {.action = MEASURE,.func = MODULE_CHECK, .flags = IMA_FUNC},
};
```

Listing 12: Pre-defined policy of IMA in Linux.

The policy can be changed at source level or by writing to a virtual file in *securityfs*. Since user management is not yet defined for the targeted system, it is assumed that all applications are ran by the root user and therefore the default policy is suitable.

4.4.4 IMA Activation and Usage

In order to enable the IMA in Linux, some tasks have to be done:

First, the kernel has to be compiled with IMA- and *securityfs*-support as shown in Listing 13. Moreover, the PCR number used by the IMA is set by *CONFIG_IMA_MEASURE_PCR_IDX*.

```
CONFIG_IMA=y
CONFIG_IMA_MEASURE_PCR_IDX=10
CONFIG_SECURITYFS=y
```

Listing 13: Enabling the IMA in the kernel configuration.

Secondly, the kernel command line parameter has to be extended with the *ima_tcb* flag. After recompiling and restart of the system, the IMA is set up and ready. By mounting the securityfs, the measurement list (which was shown exemplary in Figure 2.10) is read as shown in Listing 14.

```
# Mount securityfs
$ su -c 'mkdir /sys/kernel/security'
$ su -c 'mount -t securityfs securityfs /sys/kernel/security'

# Read IMA Measurement List
$ su -c 'cat /sys/kernel/security/ima/ascii_runtime_measurements'
```

Listing 14: Mounting the *securityfs* and reading the IMA measurement list.

4.5 TPM Usage in User-space

4.5.1 Overview

The authenticated boot chain is set up and the kernel provides access to the TPM functions with the */dev/tpm* interface. In order to complete the trusted computing capabilities of the system, the user-space part is implemented. First, the TSS has to be set up to provide the following functionality in user-space:

- Installing *TPMTools* which are used to set up and debug the TPM.
- Implementing the TSS-Wrapper described in Section 3.3.2 in order to simplify attestation.
- Implementation of an exemplary attested network connection.
- Enabling the usage of keys which are stored on the TPM by openSSL.
- Fill the kernel's entropy pool with randoms generated by the Hardware-based Random Number Generator (HRNG) of the TPM in order to increase quality and quantity of */dev/random*.

4.5.2 QTSSWrapper

The interface described in Section 3.3.2 has been implemented for the system. Figure 4.6 shows the overall architecture.

The TrouSerS TCS daemon runs on the system and handles the communication with the device driver. The trusted server application (TSA) links against the *QTSSWrapper* library. This library wraps the basic TSP interface for key handling and attestation in Qt(a framework for C++) [Dig13] objects. This object-oriented approach reduces the complexity in many ways:

- The context and session handling is hidden from the application, what reduces the Line of Code (LOC) count for TPM applications drastically.
- Error handling is done centralized.
- Applications do not have to care about memory handling, since basic Qt primitives for buffers are used.

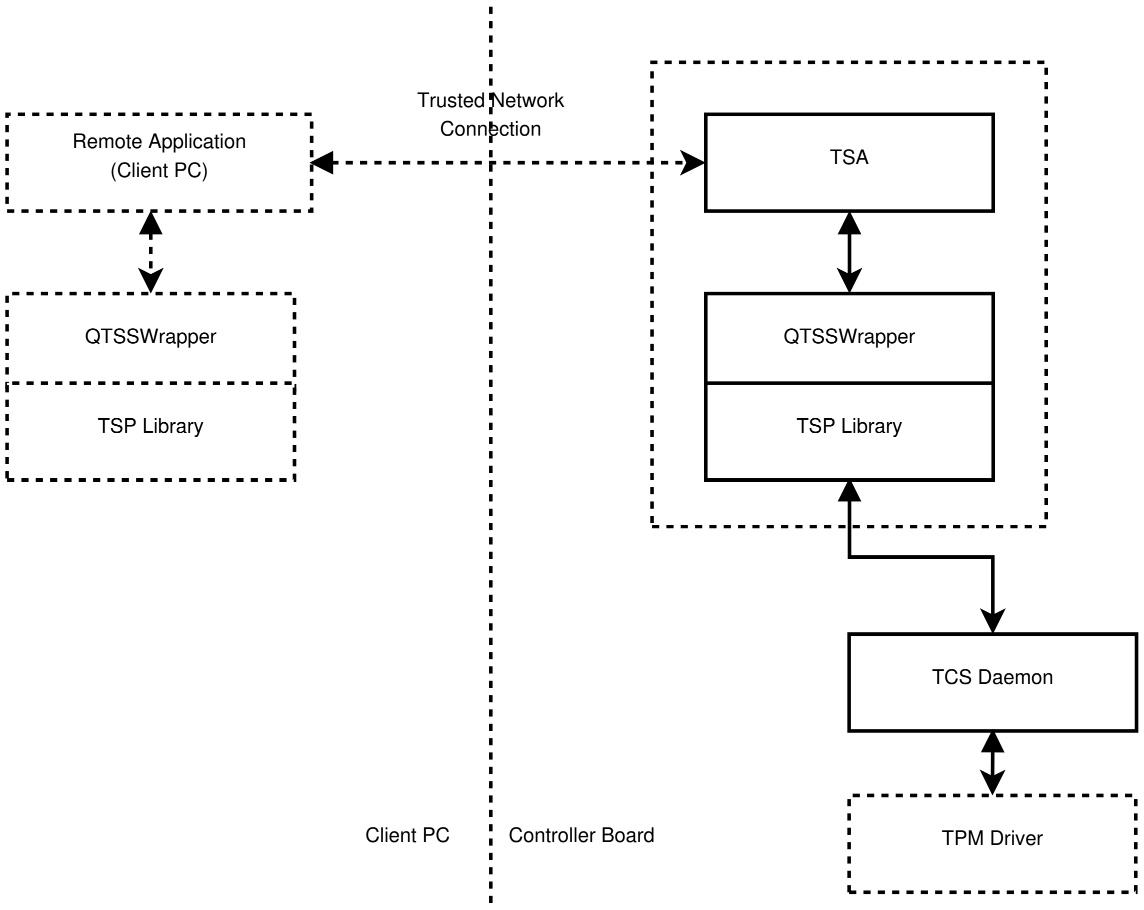

Figure 4.6: The *QTSSWrapper* library wraps the TSS Service Provider in order to reduce the complexity of attestation and *TPM_Quote* commands.

- Keys are serializable in a simple way.
- Since the used Qt libraries (namely *QtCore* and *QtNetwork*) are used by other applications on the system, no additional libraries have to be installed.

On the client side, a stripped-down version of this library is used to generate the nonce and check the *TPM_Quote* answer for integrity. The overall reduction in LOC for an attestation is about 90% compared to direct use of TrouSerS' TSP.

4.5.3 TPM Keys in OpenSSL

RSA keys managed by the TPM have the fundamental advantage, that the private part cannot be read out by software. In order to use these kind of keys without changing the networking software, *OpenSSL* is extended.

OpenSSL supports libraries which contain engines for different ciphers. Thus, it is possible to load a library containing the interface for RSA which does the actual operations on the TPM. TrouSerS ships with the *openSSL-tpm-engine* which exactly implements the desired behavior. A tool, called *create_tpm_key*, which creates a file containing a new key encrypted with the SRK of the TPM. By using this engine, every application is able to use the encrypted key to sign or encrypt data on the TPM.

4.5.4 TPM Random Generator

In order to fill the entropy pool and generate 'good' random numbers, the Random Number Generator (RNG) unit of the TPM is used. As described in [MWK+13], the ability to provide randomness is a fundamental property of a secure system. On ordinary PCs, user input like mouse or keyboard is used to fill the entropy-pool of the kernel. However, embedded systems lack of these possibilities and gathering this randomness is even more challenging.

In order to fill the entropy pool, the *rng-tools* [RNG13] daemon is used. This daemon gathers randomness from installed hardware random generators and supply it to the kernels entropy pool.

To enable *rng-tools* support for TPM RNGs, it has to be configured in */etc/default/rng-tools* as shown in Listing 15.

```
HRNGDEVICE=/dev/null
RNGDOPTIONS="-hrng=tpm -fill-watermark=90% -feed-interval=1"
```

Listing 15: Enabling TPM support in *rng-tools*

4.6 Hardware AES

To implement the DCP driver described in section 3.3.5 in Linux, the cryptographic API is used. The first version of the driver had some performance issues (what will be shown in Chapter 5). Thus, a second version is provided and discussed. Moreover, user-space access for different applications like network services is needed.

4.6.1 Linux and Cryptography

Overview and User-API

The scatterlist cryptographic API in Linux has been introduced to support cryptographic functions used by IP Security (IPSEC). Later, all subsystems which need cryptographic support started using this API. Besides ciphering it also supports hash functions and compression.

On the user side, so called *transforms* are used which internally handle the algorithms. The actual cipher and mode of operation is encoded in a string which is parsed to decide which algorithm is used. Listing 16 shows an exemplary allocation of an AES transform with Cipher Block Chaining (CBC) mode of operation.

```
struct crypto_ablkcipher *tfm = crypto_alloc_ablkcipher("cbc(aes)", 0, 0);
```

Listing 16: Allocation of a transform for AES-CBC.

Instead of working with logical addresses, the API works with page vectors. In Linux, this is done with the *scatterlist* structure [Cor07]. Each *scatterlist* entry contains the memory page, offset and data length. The two lists (one for source and one for the destination buffer) are sent to the algorithm driver, so the implementation does not have to do the conversion between logical buffer and pages itself.

Algorithm API

On the other side of the cryptographic API, algorithms may register itself at run-time to provide different implementations for different ciphers. This work concentrates on asynchronous block cipher algorithms. However, other algorithms are implemented similar and a comprehensive overview is located at *include/linux/crypto.h* in the official Linux tree.

Asynchronous block ciphers in Linux are block cipher operations where the actual computation is not done in the caller's context as shown in Figure 4.7. When the implementation finishes the operation, a callback function is executed to notify the caller. These are properties which are very comfortable for hardware accelerated cipher modules because the caller is able to queue a number of cipher requests while the co-processor performs the actual operation.

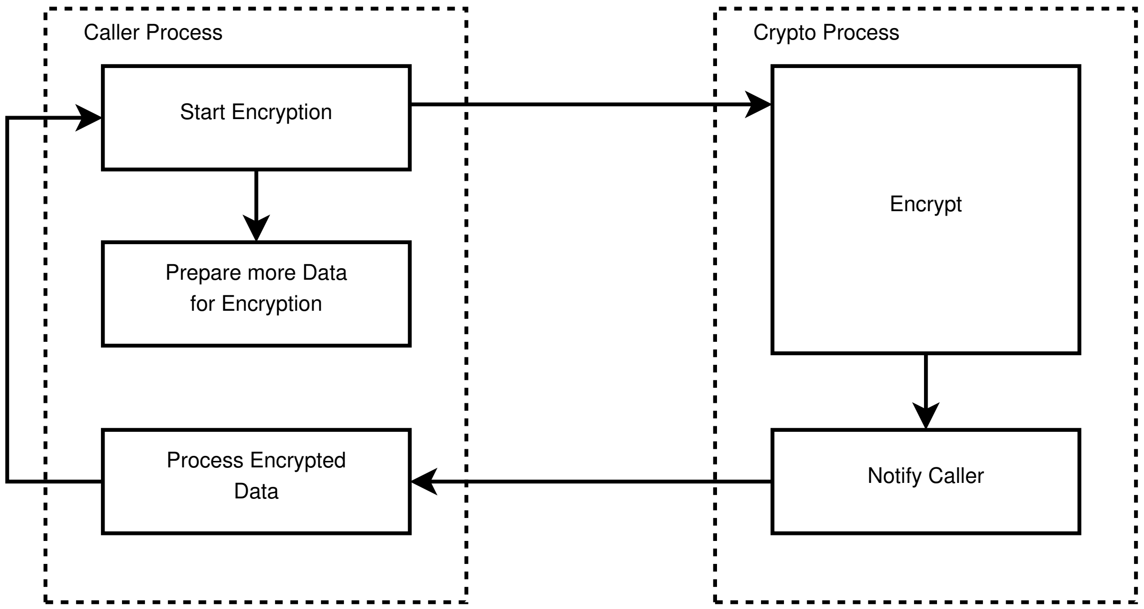

Figure 4.7: Asynchronous block cipher in Linux - exemplary usage: The calling process prepares the next buffer while the actual encryption is running on a different CPU or a dedicated hardware.

A new algorithm is registered with *crypto_register_alg(struct crypto_alg *)* where the *crypto_alg* structure holds all necessary information of the implementation as shown in Listing 17.

```c
struct crypto_alg {
        unsigned int cra_blocksize;
        unsigned int cra_alignmask;

        int cra_priority;

        char cra_name[CRYPTO_MAX_ALG_NAME];
        char cra_driver_name[CRYPTO_MAX_ALG_NAME];

        struct ablkcipher_alg {
                int (*setkey)(struct crypto_ablkcipher *tfm, const u8 *key,
                              unsigned int keylen);
                int (*encrypt)(struct ablkcipher_request *req);
                int (*decrypt)(struct ablkcipher_request *req);

                unsigned int min_keysize;
                unsigned int max_keysize;
                unsigned int ivsize;
        } cra_u;
```

Listing 17: Relevant parts of *crypto_alg* structure used to describe a cipher algorithm in Linux

- *cra_blocksize* describes the block size of the cipher
- *cra_alignmask* holds a mask for addresses used by this algorithm. According to the i.MX28 manual [Inc13], the DCP is performing better, if all addresses are aligned by 4 bytes. In this case, this mask is set to 3 so the cryptographic subsystem checks all addresses whether $address \& 0x03 = 0$ and uses bounce-buffers if the condition does not hold.
- *cra_priority* is used to prefer some implementations over others. If the kernel has soft and hardware AES, it might be useful to use the hardware version if not explicitly stated. A common value for hardware implementations is 300.
- The *cra_name* string describes the implemented function (like 'aes' or 'cbc(aes)') while *cra_driver_name* is used to store the driver's unique name.
- *cra_u* is actually an union for the different algorithm types (cipher, hash, and friends). In this case, only the asymmetric block cipher type is shown.
  - *setkey* points to the function used to set a key for an operation
  - *encrypt* and *decrypt* points to the corresponding functions of the implementation. The *ablkcipher_request* holds information like the source and destination *scatterlist* and data length.
  - The remaining parameters are used to set minimum and maximum key size, as well as the size of the initialization vector

Once the algorithm is registered, the subsystem provides it to the users. Especially in asynchronous block cipher mode, the algorithm driver has to ensure that no race conditions from multiple calls of the encryption or decryption functions can occur. Therefore, the subsystem provides a helper structure (*crypto_queue*) and associated functions to enable queuing.

4.6.2 Design of AES-Driver

Basic Approach

The basic approach to enable the DCP described in Section 3.3.5 has been implemented for Linux. At initialization time, the driver gathers all needed resources and registers the *crypto_alg*.

For actual usage, the following basic building blocks are implemented:

- *dcp_queue* queue
- *queue_task* tasklet [7]
- *done_task* tasklet
- *hw_pkg* ring buffer for hardware description packets

As shown in Figure 4.8, every encryption or decryption request is added to the queue. If no other request is processed, the *queue_task* tasklet is scheduled. At the time this function returns, the calling process is able continue with other work (or wait until the cipher operation finishes).

The *queue_task* de-queues the next element of *dcp_queue* and, if present, performs some initialization tasks like setting the mode of operation and the key before it calls the *dcp_op_proceed* function.

This function is in charge to fill the *hw_pkg* ring-buffer. Therefore, it calculates the length and addresses of the next data-buffer and writes them to the first free *hw_pkg*. The constraints for the buffer is taken from the reference manual [Inc13]: The length has to be a multiple of 16 (the block size) and it has to be physically continuous. The simplest way to achieve this is to use one hardware description package for each physical page. After the corresponding memory is mapped for DMA access, the packet is created and the DCP is notified.

After executing the operation, the DCP raises an interrupt. The Interrupt Request (IRQ) handler simply checks the error state of the DCP and schedules the *done_task* tasklet.

In *done_task*, the memory is un-mapped from the DMA subsystem and the *dcp_op_proceed* function is called if data is remaining. When no more data has to be processed, the tasklet performs some cleanup tasks, calls the callback method of the finished request and schedules *queue_task* in case of any pending requests.

Moreover, some error-recovery mechanisms are included. The *dcp_watchdog* timer aborts the operation, if the DCP takes more than $500ms$ to complete a packet(compared

[7] A *tasklet* is a structure holding a function which may be scheduled at a system-determined safe time [CRKH05]. The function is running in interrupt context, so it is not possible to call functions which may sleep (and therefore call the scheduler).

to about $7ms$ it takes the DCP to process one page). Every error is reported to the user as argument of the callback function.

Optimized Approach

As shown in Chapter 5, the performance of the driver depends heavily on the number of context switches (or scheduled tasklets). The performance has been significantly increased by the following two optimizations:

- The function of the *queue_task* tasklet is called directly the first time instead of scheduling the tasklet. Thus, the first function call is done in user-context.
- After finishing a packet, the *done_task* also calls the *queue_task* function directly.

4.6.3 Userspace access

Kernel Driver

In order to enable user-space access to the accelerated cryptography, the *cryptodev* [Var13] driver is used. This module provides a virtual file in */dev/crypto* to access the kernel's cryptography subsystem.

User-space

Similar to the TPM usage described in section 4.5.3 there exists an engine for openSSL to use the *cryptodev* module for ordinary AES ciphers in any application.

4.6.4 Disk Encryption

The device proxy described in Section 3.3.4 can be realized in Linux by using the *dm_crypt* module and *cryptsetup*. The first line in Listing 18 shows the steps used to setup the encryption on the */dev/mmcblkp2* partition with AES and 128 bit key size. The second line has to be executed to actually map the encrypted device. In this case the decrypted block device is mapped to */dev/mapper/myencpart*.

```
cryptsetup -y --cipher aes --key-size 128 luksFormat /dev/mmcblkp2
cryptsetup luksOpen /dev/mmcblkp2 myencpart
```

Listing 18: Encrypting a block device in Linux with *dm-crypt* and *cryptsetup*

In order to use the OTP key of the i.MX28, the *setkey* function of the DCP driver has been extended to use this key when an all-zero key is provided.

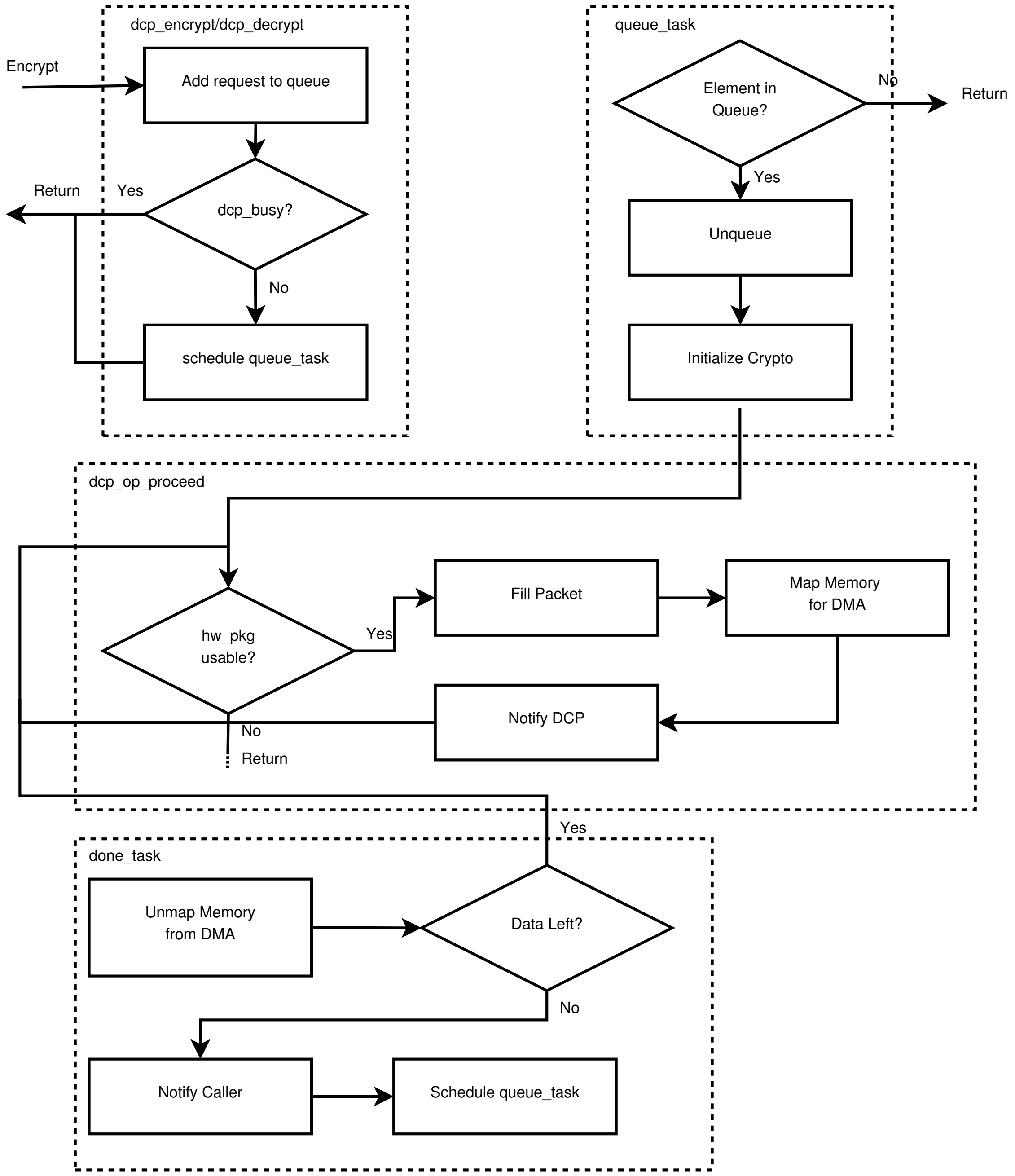

Figure 4.8: The basic building blocks of the implemented DCP driver for Linux.

Chapter 5

Results

This chapter covers the discussion of the impact of the implemented security enhancing features on the threats defined in Chapter 3. Additionally, an analysis of the impact on system performance in terms of boot and execution time is presented. Moreover, the hardware-based cryptography is compared to software-only approaches in user- and kernel-space.

5.1 Threat Mitigation

The threats defined in Section 3.2 are mitigated by the different security enhancements described in Chapter 3 and 4. Figure 5.1 and Table 5.1 show the different modules which were discussed before. Table 5.2 shows an overview of the used threat mitigation techniques followed by a more detailed discussion of every module. Mitigation strategies which are *italic* are not covered in this work and can be seen as assumptions and proposals.

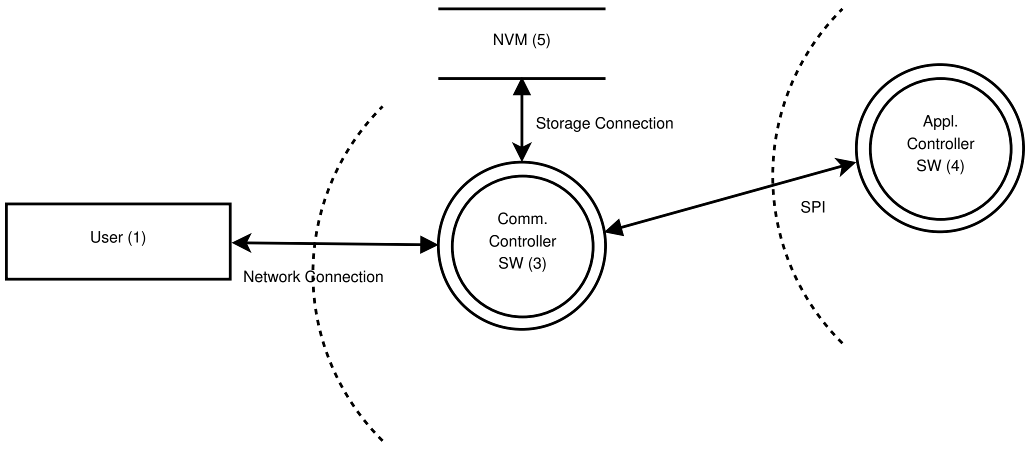

Figure 5.1: The data flow of the system with reduced complexity based on Assumption 2 and transparent client software.

Table 5.1: System modules used in threat model.

Module Type	Modules
External Identity	User (1)
Data Store	NVM (5)
Process	C-Software (3), A-Software(4)
Data Flow	Network (1-3), Storage Connection (3-5) SPI(3-4)

Table 5.2: Overview of the threat mitigation techniques per module.

	Threat	Mitigation Technique
User (1)	Identity Spoofing Credentials Disclosure on the Wire Replay Attacks Repudiation	*Password policy*, *mutual attestation*, DOR enc. *Network connection with TLS* *TLS* DOR encryption, *secure logging*
NVM (5)	Data Confidentiality Integrity of Executables	DOR encryption, *Linux access control* authenticated boot and IMA
Controller (3)	Spoof controller Exchange Executables Run arbitrary Code	authenticated boot, IMA, remote attestation authenticated boot, IMA, remote attestation IMA (partially)
NW (1-3)	Link Confidentiality Link Integrity	*TLS* *TLS*

5.1.1 User and Client PC

As mentioned before, the important problems regarding the user are identity spoofing and repudiation.

Identity Spoofing

Assuming a password-based authentication, there are basically two ways to spoof a user's identity:

Revealing User Secrets: The obvious way to spoof a user is to reveal it's secrets. Thus, a good password strength and update policy has to be chosen but this is not a part of this work. Moreover, the credentials have to be saved on the endpoint, the communication controller. Thus, this storage has to be secured against non-authorized access what is described in Section 5.1.2. To protect the data on the link between the user and the communication controller, the network connection has to be secured as explained in 5.1.4.

Tamper with the client software: The user communicates with the communication controller over an ordinary PC running a specific software. Compromising such a system enables an attacker the possibility to easily read user input or forge commands to the communication controller which are not actually placed by the real user. Section 3.3.3 described a method to attest the client computers state to the communication controller with the help of a TPM. This work does not cover an implementation of authenticated boot on an ordinary PC since the Static Root of Trust for Measurement (RTM) approach used on the communication controller might not be efficient as described in Section 2.2.1. However, after implementing a chain of trust on the client computer, the mutual attestation scheme can be implemented with little effort.

Repudiation

Besides identity spoofing, it is important to ensure non-repudiation of user actions. Therefore, log files have to be stored in a tamper resistant way. With DOR encryption, the files are secured while unmounted as described in Section 5.1.2. Detection of non-repudiation is not possible when a system is compromised, so this has to be done indirectly by logging unauthorized code execution as described in Section 5.1.3

5.1.2 Non-Volatile Memory

The NVM has to provide a secure partition which is not readable on other systems. The DOR encryption described in section 3.3.4 enforces this behavior.

Data Confidentiality

All data on this partition is encrypted with a per-system key which is only readable by the cryptographic co-processor of the SoC. Even if an attacker is able to read the entire memory of the system in software, he is not able to reveal the key. Thus, the NVM is bound to the SoC, so discarding of the memory card can be done without exhaustive recovery-resistant removal-technologies.

Data Integrity

Data integrity is ensured with authenticated boot technologies as described in Section 5.1.3.

5.1.3 Communication Controller

As mentioned before, the main attack vectors regarding the communication controller are spoofing a controller and running arbitrary code on an existing one. The former attack could reveal user secrets or forge information of the automation system while the latter case could additionally enable software modifications on an application controller.

Spoofing a Communication Controller

Assuming a non-compromised client computer, the communication controller has to attest it's state as described in 3.3.3. To successfully achieve this, the spoofing communication controller needs the following information:

1. All measurements of a clean communication controller, including the PCR values and the IMA measurement list.
2. A random value provided by the client software, the nonce.
3. An AIK which is known by the client.

While (1) and (2) are simple to reveal, the private part of the AIK does not leave the TPM of the real system and therefore it can be seen as non-readable. It is possible to forge the certificate-chain on the client computer to force the client to trust any key. However, in this case, it would not make sense to spoof a communication controller because it does not enable any benefits over access to the client computer.

Another possibility is to get access to an actual communication controller and generate a quote with the intention to replay it on the spoofed communication controller. This is countered with the 128 Bit nonce which is added to the measurement before singing.

Permanently change Executables on Communication Controller

According to Assumption 2 of Section 3.2.1 it is possible for an attacker to replace all system files if logical access to a communication controller is obtained. Another possibility to achieve this is to directly manipulate the NVM since the system binaries are not encrypted. The three types of software components are:

- Bootloader (U-Boot) and its Configuration
- Operation System (Linux)
- Userspace Applications

As described in Section 4.2 the signature of the bootloader and its configuration are checked by the HAB feature of the i.MX28. An altered bootloader would lead to a non-booting system, what is easily detectable.

The bootloader measures the operating system and extends the measurement to a PCR of the TPM. Whenever another entity starts a communication with the communication controller, it has to quote these values as described in the last subsection. An altered state will be detected and the communication refused.

Userspace applications are measured by the IMA, so a changed user-space binary or configuration file will be detected within the attestation process.

Code Execution at Runtime

An attacker may have the ability to run new code or code what is not intended to run on the communication controller. There are two components talking with the outer world: The sshd and the TCS. It may not be excluded that there will be any exploits for these parts at any time. Therefore, it has to be assumed that it is possible to send malicious messages to the system in order to execute unattended code.

Profound exploits whereat software is altered without starting a new process are not detectable by the system[8]. However, simpler attacks where an attacker is able to spawn a shell and other processes are reflected in the measurement list of the IMA and can be detected as described in the last subsection.

Impact on application controller

As mentioned before, it is not possible to completely prevent access to the communication controller since future exploits are unknown. However, it is possible to detect most malicious modifications on the communication controller. The application controller has to use the attestation scheme described in Section 3.3.3 to check the state of the communication controller. Since it can handle safety-critical operations autonomously, the communication to the communication controller may be safely interrupted while its state is not trustworthy. With this method it is not possible to send commands to the application controller with non-authenticated software running on the communication controller.

5.1.4 Network Connection

The connection between the communication controller and the client computer is done over the company network. Since it can not be assured that this connection is secure in terms of confidentiality and data integrity, some technical protections have to be implemented. These technologies are not covered by this work but an ordinary Transport Layer Security (TLS) connection between the endpoints enables all required features.

[8]There are mitigation strategies for this kind of attacks like Address Space Layout Randomization (ASLR) or Write XOR Execute (W$\oplus$E) but it cannot be assumed that these mitigations are able to completely prevent such attacks. A D-RTM based attestation mode where not only the binaries, but the RAM content is measured may detect such modifications but this is out of the scope of this work.

5.1.5 Limitations

This section concludes with the discussion of limitations of the security enhancing features described so far.

Physical Access

The most important limitation is the absence of protection against physical access: An adversary who is able to get physical access to the communication controller is able to perform the following tasks:

- Directly replace files on the NVM: This is the simplest way to alter the behavior of the device but detectable as described in Section 5.1.3.
- Tamper with the SPI bus: An attacker would be able to send malicious messages to the application controller.
- Tamper with the I2C connection between the SoC and the TPM: If an attacker is able to achieve this, the TPM can be considered useless as described in section 2.3. Moreover, any user would believe in the system's trustworthiness because the attestation features tell him so.

Thus, physical access has to be prevented for any adversary. However, hardware access to one communication controller does not compromise security of other systems because the keys used for encryption and signing are unique and their private parts are hardware protected by the TPM.

Run-Time Integrity Detection

Another limitation is the lack of detection of sophisticated exploits as described in Section 5.1.3. Whenever an attacker is able to execute arbitrary code on the communication controller without spawning a new process, this modification would remain undetectable. Such kind of attacks can be mitigated with technologies like ASLR or W$\oplus$E, and by reducing the complexity of the software interfaces. Whenever an attacker needs to spawn another process to perform an attack, it is detectable with the help of the IMA. Moreover, a reset of the platform restores the system to a trusted state.

5.2 Trusted Boot Performance

5.2.1 Overview

Trusted and/or secure boot add a significant overhead in terms of CPU-time. Both, the measurement itself (i.e., the hash function) and the extending to the TPM are relatively cost intensive. In order to picture this overhead, this section provides some measurements concerning the boot time differences of enabled and disabled integrity measurements. Table 5.3 shows the basic setup of the bootloader, kernel and hardware.

Table 5.3: Setup to measure authenticated boot performance impact.

Module	Version
System on Chip	Freescale i.MX287
Board	Freescale i.MX28evk Evaluation Board
Bootloader	U-Boot v2013.04 with TPM and bootstage support
Kernel	Linux v3.10-rc4 with DCP, Atmel TWI TPM and IMA

5.2.2 Bootloader

Measurement Setup

U-Boot provide a simple measurement interface called *bootstage* to incrementally take measurements of different actions. As shown in Figure 5.2 the measurement setup for U-Boot is defined as:

- Initialize the TPM after U-Boot has loaded.
- Load the kernel image from a server with TFTP.
- Set a *bootstage* mark and generate a hash of the kernel.
- Set another mark and extend the measured value to a PCR.
- Print the time measurements and hand over control to the kernel.

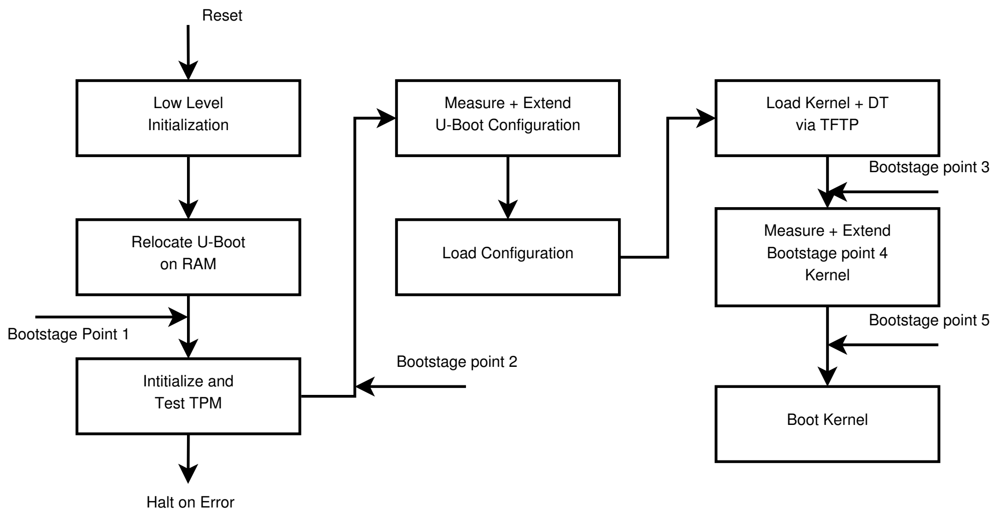

Figure 5.2: The bootloader is measured at 5 different points in order to measure the impact of authenticated boot on execution time.

This setup enables the measurement of U-Boot's performance while loading a kernel image with $3.19MB$ (uncompressed).

Table 5.4: Performance drawback in U-Boot when measuring a Kernel.

Action	Time
Load Kernel via TFTP	$4s$
Other Actions	$2s$
Boot Time Without TPM	$6s$
Measure Kernel	$222ms$
Extend to TPM	$700ms*$
Trusted Computing Overhead	$922ms*$

Measurement Results

Table 5.4 shows the performance impact of the authenticated boot process in U-Boot.

It has to be noted that the used I2C driver still had some bugs which causes the relatively long time used to extend the measurement to the PCR. However, assuming that the final version of the system does not load the kernel image via TFTP but boots it from memory (what reduces the boot time by 4 seconds), the impact on U-Boot boot time is about 45%.

5.2.3 Operating System

Measurement Setup

In order to measure the CPU-time used for integrity measurements, the IMA-module has been extended. As described in Section 4.4, the *process_measurement* function is the first function which is called from all measurement-hooks. Thus, it is the best candidate to measure the execution time.

The time is taken with the *local_clock* function what essentially calls the *read_sched_clock* function in the clocksource-driver of the i.MX28 platform which itself reads the *timrot* register 1 of the SoC. The complete boot time is measured with the help of a small kernel module which is late loaded after all other startup daemons just before the first shell is spawning. This is not a very accurate approach because it highly depends on scheduling slices but it provides a basic idea of the time consumed by the complete boot process.

The Dynamic Host Configuration Protocol (DHCP) client and other network functions are deactivated in order to minimize external dependencies. While the kernel is loaded via TFTP, the root filesystem is stored on a memory card.

Measurement Results

Table 5.5 shows the impact of the integrity measurements made by the IMA. Figure 5.3 shows the distribution of program sizes and the average time needed to measure in order to provide an idea of the amount of user-space applications. Since most of these files are relatively small ($< 500kB$), the time difference between measurement only and additional extending is very high (one extension takes about $20ms$).

Table 5.5: Performance drawback in Linux with activated IMA.

Action	Time
Boot Time Without IMA	$20s$
IMA without TPM enabled (measurement only)	$1695ms$
IMA with enabled TPM	$4748ms$

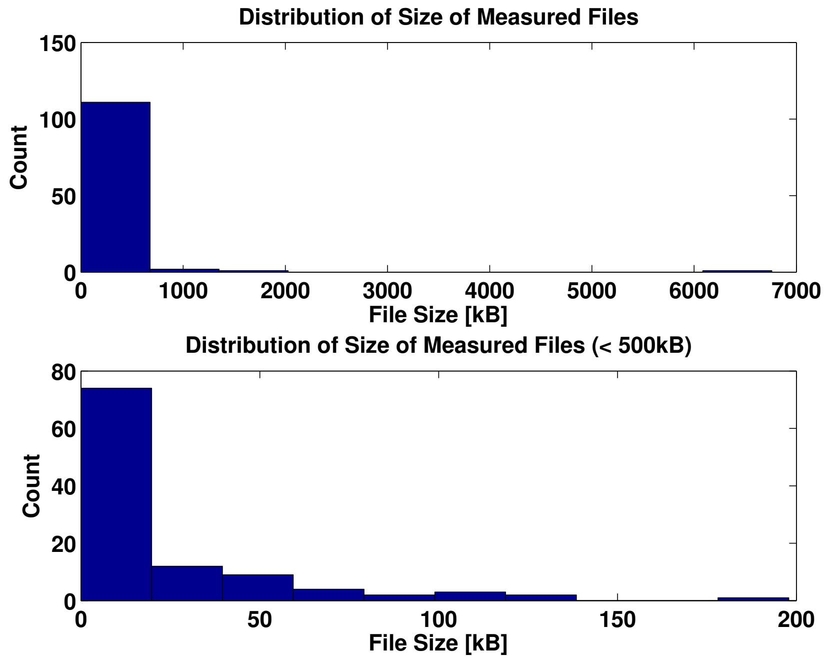

Figure 5.3: Distribution of sizes of measured files: The big time overhead used for extending the measured values to the TPM is explained by this distribution. Most of the 115 measured files are relatively small.

5.2.4 Conclusion

As seen in this section, the integrity measurements require a significant time in the boot process. However, after the system is started successfully, the IMA does not need to measure additional files, since all binaries needed for normal operation are loaded already.

5.3 Hardware AES Performance

The DCP driver's performance highly depends on implementation details and block sizes. This section describes different measurement setups used to analyze the behavior in kernel- and user-space. The differences in terms of performance for different blocks and use-cases are analyzed. The hardware and software setup is identical to Section 5.2.

5.3.1 Measurement Setup

Hardware Time

In order to face the time needed for configuration done by the CPU and the time needed for cryptography by the DCP, the driver is adopted as follows:

- Just before the DCP is notified in *dcp_proceed*, the current time is measured with *local_time* (see Section 5.2.3).
- When the last hardware interrupt raises the *done_task* tasklet, the time difference is measured and provided through a *sysfs* file.
- Moreover, an exported function to provide the measurements to other kernel modules is added.

In-Kernel Measurement

To measure the performance of the DCP as clean as possible in terms of minimizing computing overhead and context switches, the crypto-test module *tcrypt* has been adopted:

- The module is loaded with *insmod* and sets up the scatterlist and cipher transform for asynchronous encryption. The block size is set by a module parameter.
- In order to ensure that all needed code resides in RAM, the encryption function is run for 100 times for a small block (16 Bytes). After this run, the measurement value for hardware time of the DCP driver is set to 0.
- To prevent other processes to take CPU time, the scheduler is temporary disabled.
- The time is taken with *local_time* and one asynchronous request is set up.
- When the callback function is executed, the time difference is measured and the value is printed to the console

By running the encryption only once, it is possible to measure both, the complete CPU time and the time needed for the hardware encryption itself. This measurement is taken several times (100) to take a mean value to reduce fluctuations.

Userspace Measurement

In order to measure the performance in user-space, the *cryptodev* module is loaded and a slightly modified version of openSSL's *speed* tool is called as shown in Listing 19. The modifications of openSSL limit on block size alignment to make the results comparable to the in-kernel measurements.

```
openssl speed -evp aes-128-cbc
```

Listing 19: Measuring the encryption performance with openSSL

Data On Rest Encryption

The DOR encryption is set up for an ordinary partition on an memory card and on a RAM partition in order to clean the measurements form Input/Output (I/O) lags. The block size of the encryption is 512 bytes (defined by *dm-crypt*) and the measured time reflects a raw write form */dev/zero* to the mapped block device to mask the file system overhead.

5.3.2 In-Kernel Measurements

Figure 5.4 shows the performance of the DCP compared to software encryption for different block sizes. As seen in the chart, the performance heavily depends on the block size. Big blocks are encrypted more efficiently because the setup-time per block decreases. As shown in Figure 5.5, the CPU is idling about $30-40\%$ of the time. This can be seen as additional performance gain compared to software-only encryption as the CPU can be used for other computations.

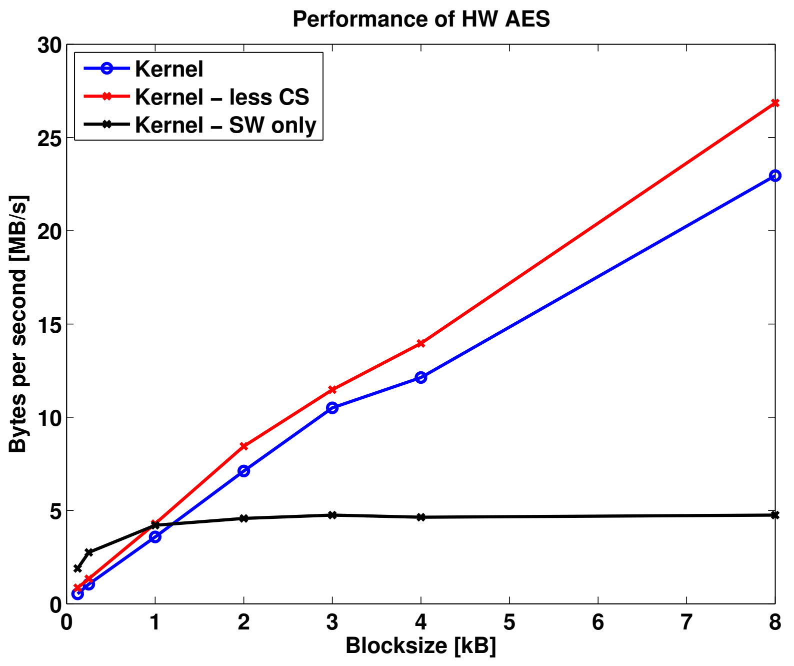

Figure 5.4: Performance of the hardware AES in kernel-space.

5.3.3 User-Space Measurements

Figure 5.6 faces the performance measured with openSSL's *speed* tool with software encryption. The user-space cryptography is generally slower than in-kernel encryption because of the context-switches to the kernel. Moreover, the graph flattens out much faster than the in-kernel measurement because the *cryptodev* module splits up the user-space memory to blocks of 1 page ($4kB$ on this platform) and calls the DCP driver for each of block separately.

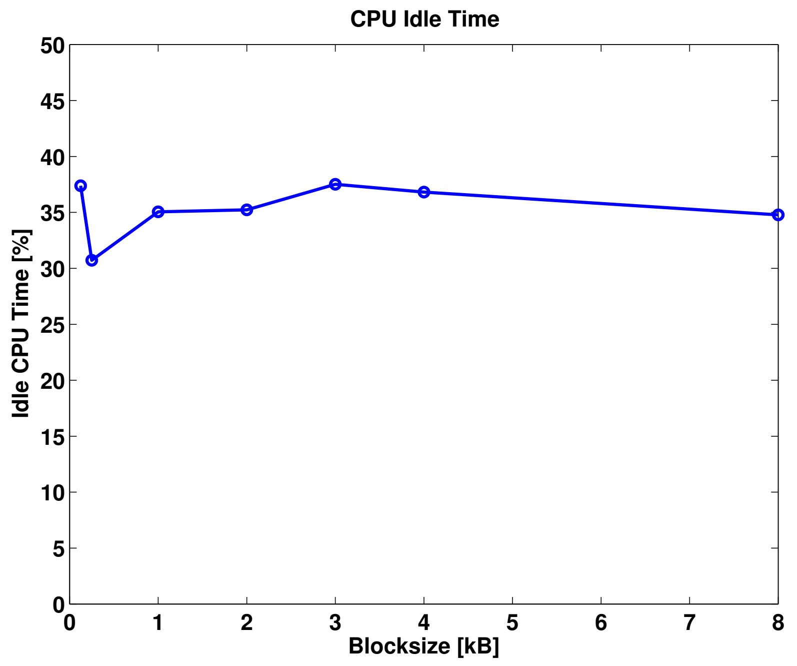

Figure 5.5: Idling time of the CPU while encrypting blocks in the co-processor.

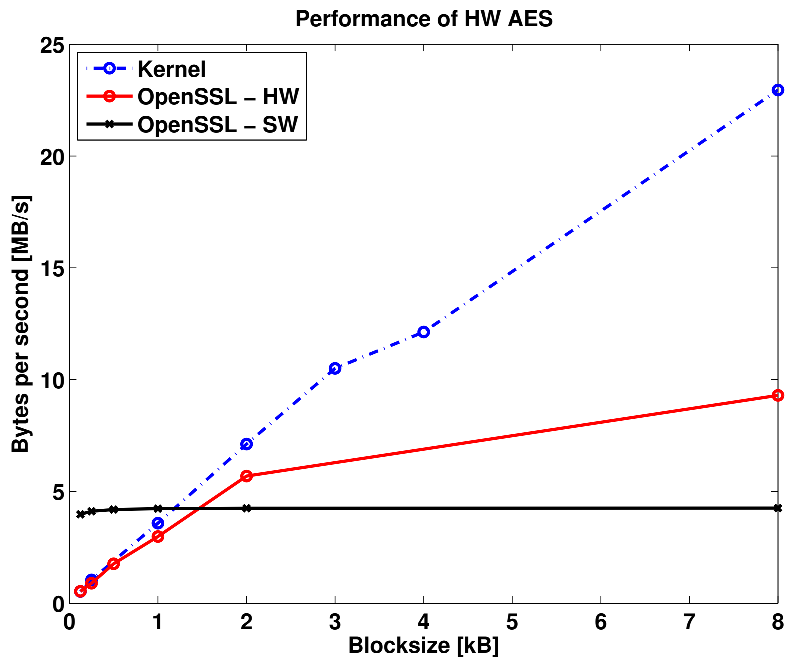

Figure 5.6: Performance of the hardware AES in user-space.

5.3.4 Data On Rest Encryption

Table 5.6 shows the results of the DOR performance measurement. The I/O performance does not seem to be a problem since the encryption is much slower. The hardware encryption is even less performing than the software-based approach. This gap results from the small block size ($512B$) used by *dm_crypt*. However, the time used for the hardware based approach has to be reduced by 33% to reflect the amount of time where the CPU is idle.

Table 5.6: Performance of DOR encryption.

Destination	No Encryption	Software	Hardware
Ramdisk	$18.8MB/s$	$2MB/s$	$1.8MB/s$
MMC	$7.4MB/s$	$2MB/s$	$1.8MB/s$

Chapter 6

Conclusion

Security should be an integral part of CPS as consequences are potentially enormous. Especially safety-critical applications like power plants need robust designs to prevent adversary from access to the system. Trusted computing technologies enable the possibility to ensure the integrity of a system. When a TPM is used, this can be achieved even if the system is compromised in software with little cost.

This work added these capabilities to an existing system used in hydro-electric power plants. Therefore, a security analysis with the help of the STRIDE process has been done and a subset of the possible threats has been mitigated.

A hardware TPM and an authenticated boot process has been implemented in order to prove the system's state to other entities what enables the possibility to isolate compromised subsystems.

Moreover, Data on Rest (DOR) encryption for the NVM has been implemented which is backed by the hardware accelerated AES module of the used SoC.

Additionally, the implemented features enabled some advantages with little effort:

- To simplify the use of the TPM, a wrapper library with an high-level API for the TrouSerS TSS has been implemented.
- The hardware-based RNG of the TPM is used to fill the entropy pool of the operating system.
- *OpenSSL* is able to use the TPM's RSA keys, what enables the possibility to securely store the private parts of the keys used for SSL connections on the TPM. This process is transparent to all applications using *OpenSSL* as cryptographic library.

The performance of the implemented features has been analyzed and the drawback in terms of boot time is significant. However, since the features does not consume time at run-time on a non-compromised system, this drawback is acceptable.

6.1 Future Work

With the conclusion of this work, some additional tasks remain:

Analyzing run-time behaviour: The security analysis has shown, that it might be possible to inject or execute arbitrary code on the communication controller without being noticed by the IMA. The IMA only recognizes opened files. Thus, an injection and execution of code without starting a new process is not covered in this work. Technologies like ASLR or W$\oplus$E might prevent these kind of threats, but further analysis has to be done.

Non-Repudiation: Since write access to log files is not covered by any of the introduced capabilities, non-repudiation is not ensured (except off-line tampering on the encrypted partition). Since a changed log file might mask malicious users, investigation of this problem should be done.

AES Performance: As seen in Chapter 4, there are some potentially performance gaps in the implementation of the hardware AES driver since it is oriented on similar drivers in the Linux tree and the organizational overhead is relatively high.

Acronyms

3DES Triple Data Encryption Standard. 21

AES Advanced Encryption Standard. 11, 32, 35, 44, 45, 66–68, 70

AIK Attestation Identity Key. 16, 18, 20, 39, 40, 75

API Application Programming Interface. 29, 57, 66, 67

ASLR Address Space Layout Randomization. 75, 77

BIOS Basic Input Output System. 15, 37

CBC Cipher Block Chaining. 66, 67

CLI Command Line Interface. 51

COT Chain of Trust. 14

CPS Cyber Physical System. 11, 82

CPU Central Processing Unit. 16, 21, 22, 24, 26, 28, 32, 35, 41–44, 67, 77–81

CRC Cyclic Redundancy Check. 52

CRTM Core Root of Trust for Measurement. 13, 15, 19, 20, 23, 25, 37, 52

D-RTM Dynamic Root of Trust for Measurement. 17, 24, 25, 28, 40, 75

DCP Data Co Processor. 45, 66, 68–70, 79–81

DES Data Encryption Standard. 21

DHCP Dynamic Host Configuration Protocol. 79

DMA Direct Memory Access. 42, 56, 57, 60, 69

DOR Data On Rest. 11, 41, 73, 74, 80, 81

DRM Digital Rights Management. 29, 33

E-TPM Extended TPM. 20–22, 37

EEPROM Electrically Erasable Programmable Read-Only Memory. 45

EK Endoresement Key. 15, 16, 18, 20

FOSS Free and Open Source Software. 31

FPGA Field Programmable Gate Array. 29

HAB High Assurance Boot. 37, 75

HMAC Hashed Message Authentication Code. 15

HRNG Hardware-based Random Number Generator. 64

I/O Input/Output. 80, 81

I2C Inter-IC. 16, 45, 53, 56–60, 76, 78

IMA Integrity Measurement Architecture. 23–25, 38, 40, 55, 60–64, 74–79

IOCTL Input/Output Control. 55

IPSEC IP Security. 66

IRQ Interrupt Request. 69

LEP Lying Endpoint Problem. 25

LOC Line of Code. 64, 65

LPC Low Pin Count. 16, 28, 29

MAC Mandatory Access Control. 60

MMU Memory Management Unit. 32, 42

NACMS Network Access Control Manager System. 25

NVM Non Volatatile Memory. 15–17, 19, 23, 34, 35, 41, 47, 50, 74–76

openSSL open Secure Socket Layer. 31, 64, 70, 80, 81

OTP One Time Programmable. 37, 70

PC Personal Computer. 15, 18, 28, 37, 38, 46–48, 66, 73, 74

PCR Platform Configuration Register. 16–19, 23–25, 27, 28, 35, 38, 40, 41, 55, 59–61, 63, 74, 75, 77, 78

PLC Programmable Logic Controller. 11, 45

PrivacyCA Privacy Certification Authority. 18, 20

Qt Qt - A framework for C++. 64, 65

RAM Random Access Memory. 41, 42, 75, 80

RNG Random Number Generator. 66

ROM Read Only Memory. 25, 37

RSA Rivest/Shamir/Adleman. 15, 65

RTM Root of Trust for Measurement. 15

RTM Static Root of Trust for Measurement. 74

RTR Root of Trust for Reporting. 15

RTS Root of Trust for Storage. 15

SDL Security Development Lifecycle. 46, 49, 77

SeLinux Security Enhanced Linux. 60

SHA-1 Secure Hash Algorithm. 15, 45

SMC Secure Monitor Call. 25

SMS4 Encryption Algorithm for Wireless Network. 21

SoC System on Chip. 25, 37, 41, 45, 74, 76, 79

SOF Start of Frame. 29

SPI Serial Periphial Interface. 32, 47, 48, 50, 73, 76

SRK Storage Root Key. 15–17, 66

sshd Secure Shell Daemon. 47, 75

SSL Secure Sockets Layer. 35

STRIDE Spoofing identity, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of Privilege. 12, 45, 49

TCG Trusted Computing Group. 13, 15, 16, 29, 31, 38

TCS Control Software. 29, 47, 64, 75

TDDL Trusted Device Driver Library. 29

TFTP Trivial File Transfer Protocol. 52, 77–79

TLS Transport Layer Security. 76

TNC Trusted Network Connection. 24, 25

TPM Trusted Platform Module. 11–20, 23–29, 31, 32, 34, 35, 37–41, 45, 46, 52, 54–56, 58–61, 64–66, 70, 74–77

TRNG True Random Generator. 15

TSP TSS Service Provider. 29, 64, 65

TSS Trusted Software Stack. 13, 21, 26, 29–32, 39, 64

TXT Trusted Execution Technology. 24

VM Virtual Machine. 27

W$\oplus$E Write XOR Execute. 75, 77

Bibliography

[AFS97] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 65–, Washington, DC, USA, 1997. IEEE Computer Society.

[AHYK11] L. H. Adnan, H. Hashim, Y.M. Yussoff, and M. U. Kamaluddin. Root of trust for trusted node based-on ARM11 platform. In *Communications (APCC), 2011 17th Asia-Pacific Conference on*, pages 812–815, 2011.

[CHL+09] Chris Codella, Arun Hampapur, Chung-Sheng Li, Dimitrios Pendarakis, and Josyula R. Rao. Continuous Assurance for Cyber Physical System Security [online]. 2009. visited: 08.09.2013. URL: `http://cimic.rutgers.edu/positionPapers/CPSSW09%20_IBM.pdf`.

[Cor05] Jonathan Corbet. The Integrity Measurement Architecture [online]. May 2005. visited: 08.09.2013. URL: `http://lwn.net/Articles/137306/`.

[Cor07] Jonathan Corbet. The chained scatterlist API [online]. October 2007. visited: 08.09.2013. URL: `http://lwn.net/Articles/256368/`.

[Cor11] Jonathan Corbet. Platform devices and device trees [online]. June 2011. visited: 08.09.2013. URL: `http://lwn.net/Articles/448502/`.

[Cor13] Jonathan Corbet. IBM Software TPM [online]. May 2013. visited: 08.09.2013. URL: `http://ibmswtpm.sourceforge.net/`.

[CPRS11] David Cooper, William Polk, Andrew Regenscheid, and Murugiah Souppaya. BIOS Protection Guidelines. Technical report, 2011.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[CYC+07] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A practical guide to trusted computing.* IBM Press, first edition, 2007.

[Den13a] Denx. Das U-Boot – the Universal Boot Loader [online]. May 2013. visited: 08.09.2013. URL: `http://www.denx.de/wiki/U-Boot`.

[Den13b] Denx. Das U-Boot - Supported Target Architectures [online]. May 2013. visited: 08.09.2013. URL: `http://www.denx.de/wiki/view/DULG/ELDKSupportedTargetArchitectures`.

[Dig13] Digia. Qt Project Site [online]. July 2013. visited: 08.09.2013. URL: `http://qt-project.org/`.

[FQmYF11] Wei Feng, Yu Qin, Ai min Yu, and Dengguo Feng. A DRTM-Based Method for Trusted Network Connection. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2011.

[GGKL89] A. Gasser, C. Goldstein, Kaufinan, and B. Lampson. The digital distribute system security architecture. In *In Proceedings of the National Computer Security Conference*, 1989.

[Gla13] Simon Glass. [U-Boot] [PATCH v3 0/12] Verified boot implementation based on FIT [online]. July 2013. visited: 08.09.2013. URL: `http://www.mail-archive.com/u-boot@lists.denx.de/msg115429.html`.

[HL09] M. Howard and S. Lipner. *Security Development Lifecycle*. Microsoft Press, 2009.

[IAI13] IAIK. jTSS Project Site [online]. July 2013. visited: 08.09.2013. URL: `http://trustedjava.sourceforge.net/`.

[IMA13] The Integrity Measurement Architecture - Documentation [online]. May 2013. visited: 08.09.2013. URL: `http://sourceforge.net/apps/mediawiki/linux-ima/index.php?title=Main_Page`.

[Inc13] Freescale Semiconductor Inc. *i.MX28 Applications Processor Reference Manual*, first edition, 2013.

[Kau07] Bernhard Kauer. OSLO: improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007. USENIX Association.

[Lan11] R. Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *Security Privacy, IEEE*, 2011.

[LZZ11] Jing Li, Huanguo Zhang, and Bo Zhao. Research of reliable trusted boot in embedded systems. In *Computer Science and Network Technology (ICCSNT)*, 2011.

[MWK$^+$13] K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson. Welcome to the Entropics: Boot-Time Entropy in Embedded Devices. In *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013.

[RNG13] Rngtools Project Site [online]. July 2013. visited: 08.09.2013. URL: `http://sourceforge.net/projects/gkernel/files/rng-tools/`.

[RNK$^+$11] Andreas Reiter, Georg Neubauer, Michael Kapfenberger, Johannes Winter, and Kurt Dietrich. Seamless Integration of Trusted Computing into Standard Cryptographic Frameworks. In *Trusted Systems*, volume 6802 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2011.

[Spa07] Evan R. Sparks. A Security Assessment of Trusted Platform Modules. Technical Report TR2007-597, Dartmouth College, Computer Science, Hanover, NH, June 2007. visited: 08.09.2013. URL: `http://www.cs.dartmouth.edu/reports/TR2007-597.ps.Z`.

[SS13] M Strasser and H. Stamer. Software-based TPM Emulator [online]. May 2013. visited: 08.09.2013. URL: `http://tpm-emulator.berlios.de/`.

[SZ10] Christian Stüble and Anoosheh Zaerin. uTSS – A Simplified Trusted Software Stack. In *Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*, pages 124–140. Springer Berlin Heidelberg, 2010.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[TCG05] TCG. TCG PC Client Specific TPM Interface Specification [online]. 2005. visited: 08.09.2013. URL: `http://www.trustedcomputinggroup.org/files/resource_files/87BCE22B-1D09-3519-ADEBA772FBF02CBD/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf`.

[TCG13] TCG. Trusted Computing Group [online]. May 2013. visited: 08.09.2013. URL: `https://www.trustedcomputinggroup.org/`.

[TRO13] Trousers Project Site [online]. July 2013. visited: 08.09.2013. URL: `http://trousers.sourceforge.net/`.

[Var13] Various. Cryptodev-linux - Project Page [online]. July 2013. visited: 08.09.2013. URL: `http://cryptodev-linux.org/`.

[WD12] Johannes Winter and Kurt Dietrich. A Hijacker's Guide to the LPC Bus. In *Public Key Infrastructures, Services and Applications*. Springer Berlin Heidelberg, 2012.

[Win08] Johannes Winter. Trusted computing building blocks for embedded Linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, New York, NY, USA, 2008. ACM.

[YDJ11] Hongfei Yin, Hongjun Dai, and Zhiping Jia. Verification-Based Multi-backup Firmware Architecture, an Assurance of Trusted Boot Process for the Embedded Systems. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011.

[YT95] Bennet Yee and J. D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *In Proceedings of The First USENIX Workshop on Electronic Commerce*, 1995.