Trusted Computing And Local Hardware Attacks

Master's Thesis

at

Graz University of Technology

submitted by

Johannes Winter

Institute for Applied Information Processing and Communication (IAIK),
Graz University of Technology
A-8010 Graz, Austria

22 May 2014

© Copyright 2014 by Johannes Winter

Advisor: Univ.-Prof. M.Sc. Ph.D. Roderick Paul Bloem

Trusted Computing und lokale Hardware-Attacken

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Johannes Winter

Institut für Angewandte Informationsverarbeitung und Kommunikation (IAIK),
Technische Universität Graz
A-8010 Graz

22. Mai 2014

© Copyright 2014, Johannes Winter

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Univ.-Prof. M.Sc. Ph.D. Roderick Paul Bloem

Abstract

Trusted Computing is one technical approach to solving the complex question, whether a computer platform, and the software currently running on this platform, is trustworthy, or untrustworthy. To address this situation, Trusted Computing defines mechanisms that allow local, and remote parties to attest the a platform. The technical realization of these attestation mechanisms relies on a combination of platform firmware, application software, and ultimately the Trusted Platform Module (TPM), which is a dedicated smart-card like hardware security element.

A common assumption Trusted Platforms states that attackers can only perform software attacks. Even simple hardware attacks are usually considered out of scope. One important problem with this assumption is that it only holds, if platforms owners, *and* all other parties (including attackers!) with physical access to the platforms in question play by the rules.

This yields to a rather paradox situation: On the one hand proponents of Trusted Computing, like the Trusted Computing Group (TCG), state that the technology is designed with software-only attackers in mind, and consider any form of hardware attacks to be out of scope. At the same time significant effort is put into making Trusted Platform Modules tamper resistant, or at least tamper evident. The result is trusted platforms that come with highly secure little hardware security modules, the Trusted Platform Modules, which are embedded into widely open "trusted" platforms.

In this master thesis we consider the gap between software-only attacks, and high-effort hardware attacks that directly target the TPM chip itself. Our focus is on simple, low-budget hardware attacks. We claim that current protection mechanisms of typical Trusted Computing enabled platforms are insufficient to defend even against these simple hardware attacks. To prove this claim we present several novel hardware attacks against Trusted Computing enabled platforms.

Kurzfassung

Trusted Computing ist ein technischer Ansatz zur Lösung der komplexen Frage, ob ein Computer und die darauf laufende Software vertrauenswürdig ist, oder nicht. Zur Beantwortung dieser Frage stellt Trusted Computing Mechanismen bereit, die es ermöglichen einen Computer entweder lokal, oder aus der Ferne, zu attestieren. Die technische Umsetzung dieses Attestierungsmechanismus basiert auf einer Kombination von Firmware, Software und dem Trusted Platform Modul (TPM).

Eine weitverbreitete Annahme zur Sicherheit von Trusted Computing Plattformen erlaubt es Angreifern ledligich Software-Attacken durchzuführen, während selbst einfachste Hardware-Attacken ausgeschlossen werden. Ein fundamentales Problem dieser Annahme ist, dass vorausgesetzt wird, dass sich alle Beteiligten, also auch die Angreifer, an die Regeln halten.

Daraus ergibt sich eine paradoxe Situation: Auf der einen Seite behaupten Trusted Computing Befürworter, wie die Trusted Computing Group (TCG), dass Hardware-Angriffe aus dem Bedrohungsmodell ausgeschlossen sind. Auf der anderen Seite werden Trusted Platform Module unter hohem Aufwand gegen Hardware-Angriffe und Manipulationsversuche abgesichert. Daraus resultieren Trusted Computing Plattformen, die sehr stark gesicherte kleine Hardware-Sicherheitsmodule, die Trusted Platform Module, einsezten um weit offene Plattformen abzusichern.

In dieser Diplomarbeit befassen wir uns mit der Lücke zwischen reinen Software-Angriffen und sehr aufwändigen Hardware-Angriffen, welche sich direkt gegen den TPM-Chip richten. Unser Augenmerk liegt auf einfache Hardware-Angriffe, die mit geringem Ressourcenaufwand durchführbar sind. Wir stellen die These auf, dass derzeitige Schutzmaßnahmen die auf typischen Trusted Computing-fähigen Plattformen implementiert sind nicht ausreichen, um sich selbst gegen einfachste Hardware-Angriffe zu schützen. Um unsere Behauptungen zu stützen, stellen wir in dieser Diplomarbeit eine Reihe sehr einfacher neuer Hardware-Angriffe gegen Trusted Computing Plattformen vor.

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.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I am indebted to my colleagues at the IAIK, in particular to Kurt Dietrich (who now works at NXP Seminconductors), Martin Pirker, Daniel Hein, and Ronald Tögl, who have provided invaluable help and feedback during the course of my work. I wish to specially thank my advisor, Roderick Bloem, for his patience and support during the long time required for the genesis of this master thesis.

My initial work on simple hardware attacks against trusted computing platforms was only possible due to a research grant (257433) from the European Commission for FP7 project SEPIA. Development of the "GUSTL" embedded trusted computing test platform, and the TPM v2.0 in-system emulator discussed at the end of this thesis were made possible by a research grant (317753) from the European Comission for FP7 project STANCE.

Last but not least, without the support and understanding of my family, in particular of my late father, this thesis would not have been possible.

Johannes Winter
Graz, Austria, April 2014

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [And12].

Earlier version of several chapters of this master thesis have been published previously in a journal paper [WD13] and a conference paper [WD12]. Material from these publications is reused in this thesis under the provisions granted the LNCS Copyright and the Elsevier Author Rights (see below). Chapters that reuse text from these publications are *clearly marked once* at the beginning of the chapter. Reused figures explicitly carry a reference to their original source of publication in their caption.

LNCS Copyright Notice

The LNCS copyright form[1] states:

> ... Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgment is given to the original source of publication. Author also retains the right to use, without having to pay a fee and without having to inform the publisher, parts of the Contribution (e.g. illustrations) for inclusion in future work, and to publish a substantially revised version (at least 30% new content) elsewhere, provided that the original Springer Contribution is properly cited ...

Elsevier Author Rights

The Elsevier Author Rights explicitly allow use in a thesis or dissertation. The journal publishing agreement for [WD13] explicitly states, that the authors retain rights for scholarly purposes, including *personal use*, without the need to obtain further permission. Personal use is defined by Elsevier[2] as:

> ... Use by an author in the author's classroom teaching (including distribution of copies, paper or electronic), distribution of copies to research colleagues for their personal use, use in a subsequent compilation of the author's works, inclusion in a thesis or dissertation, preparation of other derivative works such as extending the article to book-length form, or otherwise using or re-using portions or excerpts in other works (with full acknowledgment of the original publication of the article). ...

[1] `ftp://ftp.springer.de/pub/tex/latex/llncs/LNCS-Springer_Copyright_Form.pdf`

[2] `http://www.elsevier.com/journal-authors/policies/open-access-policies/article-posting-policy#published-journal-article`

Chapter 1

Introduction

"Who has begun has half done. Have the courage to be wise. Begin!"

[Horace, Epistles]

1.1 Overview

Desktop computers are complex systems comprising of a large number of interacting hardware, and software components. Users can customize their desktop computers to a large extent, for example by installing new software, changing the configuration of existing software, or installing new hardware components. For general purpose desktop computers there are hardly any restrictions on the extent of possible customizations: If the user does not like the operating system of his computer, he can simply replace it with an alternative operating system. Software settings, such as password policies, which may not match the preferences of the user, can typically be changed with small effort. Hardware components, like disk driver, network interface cards, or even the mainboards and processor can typically be changed. A direct consequence of this high degree of flexibility is that the number of possible system configurations, each comprising of a specific combination of hardware and software, is very large.

The situation is similar for many embedded computer platforms. Embedded system cover a broad range of diverse platforms. One end of the spectrum includes tiny microcontrollers used in washing machines, car engine control units, or similar devices, which typically only provide a very limited form of user-interaction, and limited possibilities for customization by normal users. The other end of the embedded systems' spectrum includes embedded x86-based computers that are used in industrial control systems, and multimedia oriented system-on-chip platforms that are used in set-top boxes, television sets, and mobile phones. This latter class of embedded system often provides rich user-interaction, and in particular in case of mobile phones, allows users to customize their system to a large degree.

Common to all types of platforms mentioned above is an increasing tendency to integrate remote control and management capabilities. In case of simpler systems, these capabilities may be as simple as a diagnostic port enabling read access to an on-device log buffer. In slightly more advanced systems these management capabilities can be used to modify device parameters, and to update the firmware of the entire device. Finally, complex embedded platforms, like mobile phones, can communicate with the outside world via the Internet.

Security increasingly becomes a major concern for all of these platforms. The overall complexity, and the large number of software, and hardware components, makes it extremely hard to maintain system security, over the entire lifetime of a platform. Even when all involved software and hardware components of the platform can be configured to operate securely, it is up to the user to do the required setup, and to ensure that components are kept up to date. In the worst case even a small error in either the hardware, the software, or the system configuration, can render an entire platform insecure.

Threats and Assets Typical assets that are found on user-centric devices like desktop computers, and mobile phones include authentication credentials, sensitive documents, privacy sensitive data, and cryptographic keys. In particular in context of embedded systems the firmware code, respectively the underlying intellectual properties, as well as the data generated by the system during operation, can be valuable assets.

Threats to these assets exist in manifold variety, and must generally be considered in context of the asset being protected, the owner of the asset, the user of the asset, and the adversary interested in the asset. Owner and user of an asset can be the same entity, for example in case of authentication credentials, where a user is interested in protecting the confidentiality of his or her password against hackers and malware. In other contexts, for example when viewing a digital rights management protected video broadcast, the situation can be more complicated: Here assets are the unencrypted video stream, and the corresponding content encryption keys. The owner of these assets is interested in protecting the confidentiality of these assets against unauthorized viewing, and copying. On the other hand the owner of the assets is also interested in ensuring availability to the users, given that the users stick to the policies set out by the owner. Apart from a content owner, and a user, there may be other stakeholders involved, such as the manufacturer of the computer hardware, or the vendor of the video playback software.

Trusted Computing as Solution? Failure of a computer system to appropriately defend against such threats may lead to breach of system security, and exposure of sensitive assets. In the worst case, a failure to properly defend against software threats can even lead to permanent physical damage.

Trusted Computing is one attempt to improve the situation. It aims to make guarantees about software load-time integrity, combined with a strong mechanisms to securely report the software configuration of a platform to local and to remote verifiers. To achieve this, Trusted Computing relies on a trustworthy log containing all security relevant software events that occurred since platform boot. Relevant events may include start of a program, or loading a configuration file. To protect the integrity of this log against any software attackers, a special smart-card like chip, the Trusted Platform Module (TPM) is used.

A standard assumption in common Trusted Computing settings is that attackers do not have unhindered access to the physical platform, or that they are incapable of modifying the platform hardware. Moreover, another common assumption in these settings is that the legitimate owner of a platform does not have any interest in attacking his or her own platform. These assumption on hardware attacks are summarized by a quote from David Grawrock:

"... What is the definition of a simple hardware attack? [...] Going to a local electronic store, purchasing twenty dollars worth of parts, putting the parts together and defeating the [...] protection is a simple hardware attack. ..." [Gra09, Ch. 10, p.132]

At the first glance, Grawrock's definition may seem overly pessimistic, and one may expect the actual protection to be much stronger. However, on second consideration, we can find valid valid arguments, such as previously known attacks discussed in Section 1.3, that let even this definition shine in a very optimistic light.

We claim that current general-purpose Trusted Computing platforms lack a *verifiable and tamper-proof* hardware binding between their Trusted Platform Module, and the remaining platform. Furthermore, we claim that the absence of this kind of secure binding opens the platform for man-in-the-middle attacks by local hardware attackers. To prove these claims, we develop variants of simple hardware attacks which are based around the man-in-the-middle idea. We take this as motivation to formulate three research questions, which we will try to answer in this master thesis:

1. Is it reasonable to assume that current trusted desktop computers can withstand simple hardware attacks according to the definition above?
2. Can we find attacks which are covered the definition above? What can be achieved by increasing the amount of resource available by raising to 15€ budget by one, or two orders of magnitude?
3. To whom would such attacks be a threat? Who, apart from curious master students, would be the attacker? What would be the assets?

Trusted Computing has received serious criticism right from its very beginning, for both, its technical realization, and its potential impact on users. Three of the main points of criticism brought up by security researchers were threats to privacy, threats to the freedom of users, and threats to the security of assets, software, and hardware. Within the scope of this master thesis, the former two types of threats will only receive a very brief treatment for sake of completeness. The main focus concentrates on threats to the security of software and hardware comprising a Trusted Computing enabled platform. The goal of this master thesis is to discuss technical aspects Trusted Computing, while trying to keep a neutral view on non-technical aspects.

Selecting a Suitable Attack The simple hardware attacks that are presented in this master thesis can be classified by their impact on platform security, and their implementation complexity. We implicitly assume that the platforms we are dealing with have their Trusted Platform Module soldered to the platform mainboard. Therefore, minimizing the hardware modifications required to perform an attack is always an implicit goal.

The existing classic TPM reset attack (cf. **??**) is the basis for our platform reset attack that is discussed later in Chapter 5. The classic TPM reset attack does not require any permanent changes to the platform, and allows attackers to exercise full control over large parts of the state measurements recorded in the Trusted Platform Module. It enables the attacker to construct arbitrary measurements chains from scratch, subject to restrictions imposed by the TPM and platform hardware.

Our platform reset attack from Chapter 5 uses a man-in-the-middle style approach to suppress certain bus signals. It requires a small, permanent modification to the trusted platform, and enables the attacker to boot the platform into a trusted platform state, freeze the measurements recorded inside the TPM, and then reboot the platform into an untrusted state, while the old measurements are retained.

The bus hijacking that we discuss in Chapter 6 uses a similar man-in-the-middle style approach to actively manipulate a single bus signal. The hijacking attack requires a small, permanent modification to the trusted platform. This attack allows the attacker to modify parts of the state measurements recorded inside the TPM, which can not be changed using the classic TPM reset attack. Our attack can be combined with the classic TPM reset attack, to create arbitrary measurement chain from scratch, given that the platform is compatible with the bus hijacking attack.

Use of an LPC bus emulator, as discussed in Chapter 7, enables much more powerful attacks. Instead of relying on small hardware modifications, which does *not* involve removal of the TPM from the platform, the attacker now can remove the TPM from the original trusted platform. In Chapter 8 we discuss how this capability can be used to construct arbitrary measurement chains from scratch. Moreover we show how a TPM, and thus the identity and state of its platform, can be physically moved from a running trusted platform to an untrusted LPC bus emulator.

Potential attackers can choose the most appropriate attack from several possibilities: If no permanent hardware modifications are desirable, the attacker may choose the classic TPM reset attack. When the classic TPM reset attack is not powerful enough, the attacker can combine the classic TPM reset attack with our frame hijacking attack. Alternatively the attacker can go for our platform reset attack, if only small modifications to the platform are desired, and if repeatedly booting the platform into a trusted state is easy.

If neither the frame hijacking attack, nor the platform reset attack are suitable, the attacker can always physically remove the TPM from the platform. This opens the possibility of using an LPC bus emulator to create arbitrary fake measurement chains from scratch. Alternatively the attacker can apply our physical TPM transfer attack to bring a platform into a trusted state, and then to perform arbitrary manipulations with the TPM measurements, without further involvement of the trusted platform.

1.2 Outline

The backbone of this master thesis is formed by two of our publications [WD13; WD12] on simple hardware attacks against Trusted Platforms that I authored as primary author. Several chapters of this master thesis are based in larger parts on the text found in these two publications.

The remainder of this master thesis is structured into two major parts. The first part, consisting of Chapters 1 to 4, establishes the background and the embedding of this work in the context of related work. The second part, consisting of Chapter 7 to 10, discusses the contributions of this master thesis. At the end of the second part we give a conclusion for this master thesis, point to (ongoing) further research, and try to answer the three research questions formulated earlier.

Chapter 1 briefly motivates the work in this master thesis and discusses the embedding of this work in the context of related work.

Chapter 2 discusses Trusted Computing, as understood by the Trusted Computing Group (TCG), and gives a brief overview of the primitives relevant to this work. The description of these Trusted Computing primitives is strongly driven by the attacks discussed in later sections, and omits details, such as key hierachies or migration, that are not directly relevant to this work.

Chapter 3 describes the *TPM Interface Standard* (TIS) that is used for communication between the Trusted Platform Module and its host platform. This chapter only discusses the interface protocol, without going into details of bus specific bindings.

Chapter 4 discusses the Low-Pin-Count (LPC) bus that is currently used on most x86-based platforms to connect the Trusted Platform Module to the platforms I/O hub (Southbridge), and thus to the main processor. A key part of this chapter is the bus specific binding between the TPM TIS protocol and the Low-Pin-Count bus.

Chapter 5 introduces a simple hardware attack with close relation to the well-known *TPM reset attack*. The existing TPM reset attack breaks the chain of trust by resetting the TPM without resetting the platform, while the attack discussed in Chapter 5 aims to reset the host platform without resetting the TPM. This apparently small difference has a number of significant implications in practice, which are discussed in detail in Chapter 5.

Chapter 6 presents a combined hardware and software attack that allows simulation of dynamic roots of trust, starting from an untrusted platform states. On the hardware side this attack exploits different length LPC bus transactions, combined with an FPGA-based *bus hijacking* device.

Chapter 7 presents an FPGA-based device that provides functionality to emulate the LPC bus host bridge found on typical x86 architectures.

Chapter 8 discusses simple hardware attacks, that extend beyond the reset attacks discussed earlier. The first part of this chapter investigate the suppression of framing signals on the LPC bus, to selective jam TPM communication. The second part of Chapter 8 discusses how the LPC emulator from Chapter 7 can be used to synthesize arbitrary measurement chains from scratch. The final part of Chapter 8 combines these results into a procedure for physically transferring a TPM between two platforms, without losing its current state.

Chapter 9 summarizes ongoing research, which was not yet concluded at the time of this writing, and gives an outlook to potential further research. The first part of Chapter 9 discusses work on the upcoming TPM 2.0 generation, and summarizes preliminary results. The second part of Chapter 9 summarizes preliminary results on trusted computing for embedded systems.

Finally, Chapter 10 concludes this master thesis.

1.3 Related work

Within this master thesis we restrict scope of our discussion to trusted platforms that build upon principles outlined by the Trusted Computing Group, and that use Trusted Platform Modules (TPMs) as their hardware anchor of trust. Alternative system-level approaches to building trusted platforms, such as ARM TrustZone [AF04], are out of scope for this master thesis.

Software Attacks Trusted Computing platforms are, in principle, susceptible to the same classes of software attacks as any standard computer platform without Trusted Computing. The architecture proposed by Trusted Computing Group does not explicitly require any special protection against buffer overflows, integer overflows, heap overflows, return-into-libX attacks, or return-oriented programming. We omit an explicit discussion of these standard attacks here, and concentrate on software attacks that specifically target Trusted Computing platforms in the remainder of this paragraph.

Several system-level software attacks against Trusted Platforms have been proposed in literature. In his paper on the OSLO boot-loader Kauer [Kau07] considers problems with malicious boot-loaders, BIOS firmware, and a software-only reset attack for version 1.1 TPMs from a particular vendor.

Kauer investigated a number several Trusted Computing enabled open-source bootloaders, including two variants of the well-known GRUB bootloader, and found discrepancies in the ways of how, and when parts of the boot-loaders, and the loaded kernel images were measured. One issue that appeared in some of the boot-loaders was to load the image twice, the first time for loading into system memory, and the second time for hashing into a PCR. According to Kauer's results [Kau07] this creates an exploitable time-of-check time-of-use (TOCTOU) vulnerability. To exploit the vulnerability, the attacker has to replace the kernel image on the boot medium between the two load operations, which is easily doable if the kernel is loaded via a network.

The BIOS firmware attack discussed by Kauer [Kau07] exploited a design weakness in the BIOS update mechanism of his target platform to flash a modified BIOS firmware image. This allowed Kauer to effectively disable the static root of trust for measurement in his BIOS, by patching the TPM transmit function in the BIOS firmware image, and flashing it to his test platform.

In addition to the hardware TPM reset attack, Kauer [Kau07] describes an implementation error of certain v1.1 TPM chips from a well-known vendor. By accident the vulnerable v1.1 TPM chips can be reset by malicious software. To trigger this reset it is sufficient to write a special value to a control register of the affected v1.1 TPM. The effects of this software attack on the TPM are similar to the hardware TPM reset attack discussed later. The notable difference to the hardware TPM reset attack is, that the vulnerable v1.1 TPMs can be reset by software with root access, without requiring the attacker to be physically present at the site where the victim platform is located.

In [DEG06] Duflot demonstrated a software attack, based on system management mode (SMM) BIOS vulnerabilities, against OpenBSD and NetBSD systems. Duflot's software attack took advantage of the graphic drivers of the vulnerable operating system to inject malicious code into a special region of system memory that holds the BIOS firmware code executing in system management mode. This system management code executes in parallel to the normal operating system, and is often responsible for low-level tasks, such as controlling the speed of the CPU's cooling fan. System management mode code runs with elevated privileges, and can control parts of the platform, where even a normal operating system kernel has no access to. Duflot's software attack does not directly try to attack the measurements held inside a TPM. Nevertheless, it is a good starting point for compromising roots of trust. Certain platforms implement parts of their BIOS flash write-protection unlock code in system management mode firmware, thus potentially enabling an attacker to reflash the BIOS, even when proper digital signature verification is implemented for BIOS update images.

A practical software attack against Intel Trusted Execution Technology was shown by Wojtczuk and Rutkowska in [WR09a]. Later, Wojtczuk et al. [WRT09] presented a second related software attack against Intel TXT, using a different attack path. Both attacks exploit implementation weaknesses in the Authenticated Code Modules used by Intel TXT to initialize its dynamic root of trust for measurement. These Authenticated Code Modules are digitally signed by Intel, to ensure the integrity of the dynamic root of trust on Intel TXT platforms. The software attacks by Wojtczuk and Rutkowska exploit weaknesses in these pieces of trusted code, and enable software attackers to create fake dynamic roots of trust. The team around Rutkowska and Wojtczuk reported a couple of other software attacks with implications on trusted platforms, including hypervisor subversion [RT08; RW08] attacks and system management mode BIOS exploits [WR09b].

Software attacks can only be used to attack a subset of the measurement values stored inside the platform configuration registers of current generation v1.2 TPMs. The reason for this limitation is linked to introduction of localities in the v1.2 TPM specification, and to locality restrictions introduced alongside. Localities will be discussed in greater detail later. For now it is sufficient to know that a certain locality is not directly accessible by any software on the platform. This special locality can only be used be trusted CPU microcode during startup of a dynamic root of trust

None of the software attacks discussed above are able to fully control *all* values that are to be measured into platform configuration registers of a v1.2 TPM. The methods discussed by Kauer [Kau07] can only be used to measure arbitrary data into platform configuration registers that are *not* subject to locality restrictions.

The attacks by Wojtczuk and Rutkowska [WR09a; WRT09] can control data measured into locality restricted platform configuration registers *except* for the measurement of the (vulnerable) Authenticated Code Module that is created by trusted CPU microcode.

In theory an adversary could attempt create malicious CPU microcode. Such an adversary would likely need access to unpublished internal details of the CPU, and to the cryptographic keys required for signing the crafted CPU microcode. At the time of this writing we are not aware of any published results on successfully creating such evil CPU microcode for state of the art x86 CPUs.

Simple Hardware Attacks Hardware attacks against Trusted Computing have been discussed publicly for an extended period of time. Probably the earliest publicly known active hardware attack against TCG-style trusted computing platforms is the *TPM Reset Attack*. This attack has been described independently by Kauer [Kau07] and by Sparks et al. [Spa+].

The principle behind the TPM reset attack is simple but powerful: Assuming that a Trusted Platform Module can be somehow tricked into performing a hardware reset *independently* of its host platform, it should be possible to reset the TPM and extend arbitrary fake measurement events into the TPM. The remaining platform would ideally remain unaffected.

Due to the implementation of the platform reset signal on typcial desktop computer, it is in principle possible to generate a fake reset condition by grounding the reset pin of the TPM. The video[1] by Sparks, and the related technical report [Spa07], clearly illustrates how easily the TPM reset attack can be mounted in practice.

One practical disadvantage of the original TPM reset attack is that other devices on the attacked bus, such as embedded controllers for power management and fan control, are affected by the fake reset as well. Depending on the exact design of the victim computer this can become an issue in practice. One problem we encountered while trying to reproduce the TPM reset attack in our own lab setup, was that PS/2 keyboard controllers did not always recover from the fake reset, leaving us with a system that was no longer accessible locally. A second problem we encountered was, that certain (newer) motherboards tended to immediately power off the entire platform on a fake reset, likely as a measure to prevent damage due to high current spikes caused by the short-circuit. The video by Sparks mentioned above shows similar problems, like stopping of the CPU fan, when the reset attack is performed.

The *locality* mechanism of v1.2 TPMs mentioned earlier provides a partial mitigation of the TPM reset attack, as it prevents fake measurements into platform configuration registers intended for use by dynamic roots of trust. Special boot-loaders, like the OSLO boot-loader presented by Kauer [Kau07] make use of special x86 processor extensions to set up a dynamic root of trust. As part of this setup process trusted CPU microcode produces an initial good measurement in a locality protected platform configuration register. Such measurements can not be reproduced with the TPM reset attack only.

Kursawe et al. [KSP05] pursued a different approach and passively listened to the communication between a (version 1.1) TPM and its host platform using a logic analyzer. One of their major observations was, that the protocols used by version 1.1 TPMs do not include sufficient cryptographic meausres, to protect the confidentiality of data exchanged over the bus.

In particular Kursawe et al. [KSP05] noted that, at least for version 1.1 TPMs, the *unsealing* command does not use any encryption, when transferring decrypted secret data from the TPM to the main processor. In response, the Trusted Computing Group added support for optionally encrypting the unsealed data returned by this command in version 1.2 of the TPM specification. To derive the required encryption keys, both the TPM and the software running on the main processor need a shared secret. Additionally *transport sessions* were introduced with in v1.2 TPMs, to enable establishment of trusted end-to-end channels between the TPM and the software running on the host processor. Transport sessions use a dedicated transport session command that accepts encrypted TPM commands as input, and produces encrypted TPM responses as output. To setup the session keys for a transport session, the TPM and the software on the host processor typically need a shared secret. We succeeded in reproducing the experiments discussed by Kursawe et al. [KSP05] in our own lab setup for the ETISS 2009 Educational Event[2].

In a chapter of his dissertation [Sch12] Schellekens discusses various hardware attacks against Trusted Platforms, and refers to our work on LPC bus manipulation [WD13; WD12], and our experimental results [Win09] with passive LPC bus monitoring. Similar variantes of the platform reset attack discussed

[1]`http://www.cs.dartmouth.edu/~pkilab/sparks/`

[2]Our results can be found at [Win09]

later in this master thesis were first discussed independently in our journal paper [WD13] and in the dissertation of Schellekens [Sch12].

Although the idea of attacking Trusted Platforms by attaching malicious devices, like micro-controllers, to an LPC bus has been around for several years[3] there exists relatively little published work on actual implementations of such devices targeting a TPM. Apart from the sources mentioned above, we are not aware of any practical implementations, apart from our own results, of malicious devices that specifically target the Trusted Platform Module on an LPC bus.

Passive LPC bus sniffing attacks, as discussed by Kursawe et al. [KSP05], are of limited use on v1.2 TPMs. The limitations are caused by improvements to the TPM unseal command, and the availability of transport sessions. Trusted software can, in principle, complicate the task of an attacker if at least one secure storage location is available for storing the required shared secrets. Arguably availability of a secure storage alleviates the need for a trusted platform module, while on the other hand absence of secure storage can lead to a chicken-and-egg situation, with respect to the shared key. Nevertheless, our own results [Win09] on passive LPC bus sniffing were crucial for implementing, and debugging, the bus modification attacks discussed in the remainder of this master thesis.

When applied against v1.2 TPMs, the TPM reset attack introduced by Kauer [Kau07] and by Sparks et al. [Spa+] suffers from similar limitations as the software attacks discussed earlier. Due to the restrictions on software access to certain TPM localities, an adversary can only control platform configuration registers that are used by the static root of trust. Platform configuration registers that are designated for use by a dynamic root of trust have locality constraints in place that thwart software attempts to measure fake data after a successful TPM reset attack.

Our work from [WD13; WD12] forms the backbone of this master thesis. We directly aim at the implementation of locality protection of v1.2 TPMs on the LPC bus. We build upon passive LPC bus sniffing techniques, and additionally allow active modification of a single bus signal. This enables us to circumvent the locality protection, and to allow *software on the victim machine* to simulate startup of am *arbitrary* dynamic root of trust. One primary design criteria for our simple bus modification attacks from [WD13; WD12] is to keep the hardware modifications to the victim platform as small as possible, and to ensure that the platform remains fully functional when no attack is in progress.

Lab Attacks At level of physical chip security Tarnovsky showed at the Black Hat 2012 conference [Tar10] that a TPM chip can be reverse engineered, and that its secrets can be extracted, by a knowledgeable adversary with sufficient resources.

Attacks similar to Tarnovsky's attack, which require access to highly specialized equipment like electron microscopes, are out of scope for this master thesis. Our focus is on low-budget and low-resource attacks that can by any motivated adversary with basic electronics knowledge.

[3]On of the earliest sources we could find, that formulates this idea is a blog posting in [Law07]

Chapter 2

Trusted Computing

"What trusted computing provides is a way to understand the current state of a platform, have some entity evaluate the state, and then make a decision whether the platform is appropriate for the current job."

[David Grawrock, The Intel Safer Computing Initiative]

This chapter discusses the principles of Trusted Computing, as proposed by the Trusted Computing Group (TCG). The informationsummarized in this chapter is based on the TCG's documents on architecture of Trusted Platforms[TCG07a], on the Trusted Platform Module[TCG07d; TCG07c; TCG07b] and on the books by Grawrock[Gra09; Gra06].

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to Communication Interfaces of the Trusted Platform Module". In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus". In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

2.1 History

Initially the Trusted Computing Group started as an industry consortium called *Trusted Computing Platform Alliance (TCPA)*, and later was reorganized into its current form as *Trusted Computing Group (TCG)* (cf. Berger [Ber05]). At the time of this writing, the homepage of the *TCG*[1] lists 12 active workgroups. Each of these workgroups is dedicated to a specific area related to the Trusted Computing Groups' view of what Trusted Computing should be. The topics include Trusted Platforms of varying type and size, infrastructure and architecture aspects, software stacks, Trusted Platform Modules, trusted storage and networking components, and virtualization. Given the scope and number of TCG workgroups, it becomes clear that Trusted Computing aims to be a security technology addressing platforms and their surrounding infrastructure as a whole.

2.2 Theory of Operation

The core idea of TCG-style Trusted Computing is to *measure* any events, like the start of a software components, which may affect the trustworthiness of the overall platform. Measurements in general include the type of event (e.g. start of a software component) being measured, combined with additional meta-data (e.g. cryptographic hash of the software component being started) describing the event.

This idea can be applied to produce a measurement log, which contains all events measured since the platform was booted, assuming that an initial *static root of trust for measurement* capable of performing an initial trustworthy measurement exists on the platform. The order of measurements in this log is important, as it reflects the temporal sequence of events. Any currently running *trusted* software component must ensure that it measures the next (potentially untrusted) component *before* executing it. This strategy implicitly creates a *chain-of-trust*, which starts at the *static root of trust* and extends over the measurements of trusted components in the measurement log. Measurement of the components does *not* automatically imply any kind of policy enforcement, such as rejecting software without a valid digital signature, or refusing to start malware.

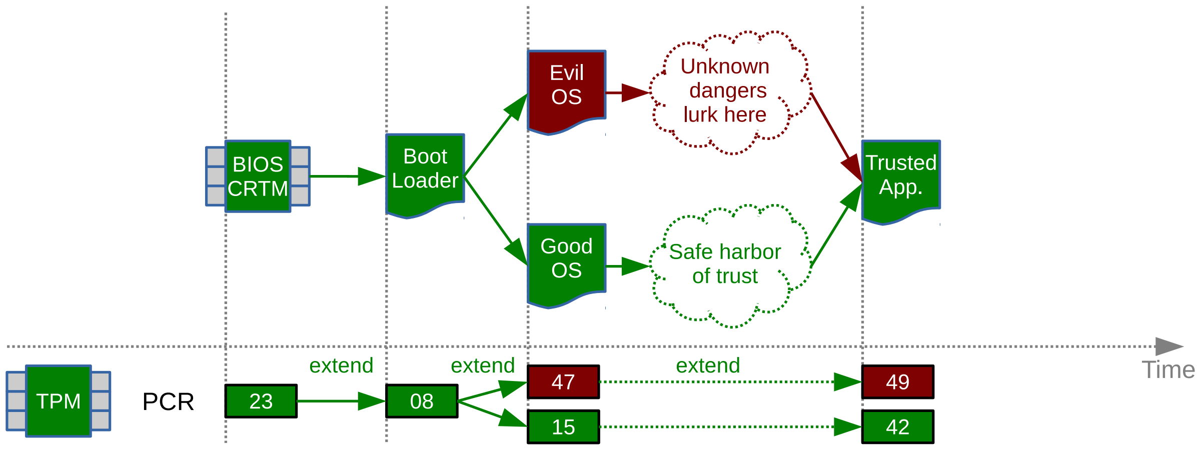

Figure 2.1: Constructing a Chain of Trust [Win11]

The invariant maintained in the chain of trust is that every component is measured by its parent, before execution. This way untrusted components can not lie about the fact that they have been executed, as there always will be a trusted measurement of the untrusted component. Figure 2.1 illustrates how a chain of trust can be constructed using this invariant. Both execution paths shown in the figure start with

[1] `https://www.trustedcomputinggroup.org/developers`; April 2014

the static root of trust for measurement, labeled in the figure as BIOS CRTM[2]. The the two execution paths shown in the figure start in the same trusted state, and end at the same trusted application. The upper path, however, includes an untrusted component. Measurements of individual components are sent to the Trusted Platform Module, and the measurement chains recorded by the TPM are shown near the bottom Figure 2.1. Before starting untrusted components — such as the “Evil OS” in Figure 2.1 — they are always measured by their trusted parent components — such as the trusted “Boot Loader” in Figure 2.1. This measurement is recorded by the Trusted Platform Module, and leads to a different measurement chain than in a trusted case.

Roots of Trust In order to realize the chain of trust it is necessary to have mechanisms to create, store, protect, and report the measurement log and the underlying measurements. The architecture proposed by the Trusted Computing Group [TCG07a; TCG07d] distinguishes between different roots of trust on a fine grained basis.

The following paragraphs give an overview of the roots of trust in a TCG-style Trusted Computing platform. More details on the operations and primitives provided by the Trusted Platform Module are discussed below in 2.6. Later in 3, and 4 we concentrate on the low-level software, and hardware interfaces of Trusted Platform Modules.

Roots of trust for measurement (RTMs) are responsible for establishing the initial good measurements at the start of a measurement chain. Typically a Trusted Platform has one *static root of trust for measurement (S-RTM)* that resides in the platform boot firmware or BIOS. This static root of trust for measurement should be immutable, and must not be modifiable by attackers, in order to ensure that an initial good measurement can be established at platform boot.

The length measurement chains produced by static roots of trust for measurement increases with software complexity. It is, by design, not possible to remove measurements from the measurement chain. Allowing such an “unmeasure” operation would enable malicious software to hide the fact that it had been started, which directly contradicts the basic idea of Trusted Computing. Long measurement chains are much harder to deal with by software that needs to attest the trustworthiness of a platform, based on measurement logs.

Newer platforms include a *dynamic root of trust for measurement (D-RTM)* that allow the running system to transition from an arbitrary software state into a well defined trusted state. The operations performed during start of the dynamic root of trust for measurement are similar to a special kind of software reset of the platform, combined with discarding parts of the measurements. Dynamic roots of trust require special support from the underlying platform, firmware and processor. Special boot-loaders, like the OSLO boot-loader by Kauer [Kau07], and operating system extensions, like the *Flicker* system by McCune et al. [McC+08] can take advantage of dynamic roots of trust.

Measurements produced by the root of trust for measurement must be protected against manipulation by malicious software. The challenge here is that measurement logs can, potentially, become infinitely large if the platform is never reset in case of a static root of trust, or the dynamic root of trust is never restarted. The TCG architecture addresses this problem by using chained cryptographic hashes as integrity protection for the entire measurement log. This way, it suffices to store integritycheck values of

[2]Core Root of Trust for Measurement

the measurement log inside special shielded locations, which form the *root of trust for storage (RTS)*, while the actual measurement log can be kept in normal (unprotected) memory. On current Trusted Platforms, the shielded locations forming the *root of trust for storage* are always the Platform Configuration Registers of the Trusted Platform Module.

The values protected by the *root of trust for storage* reflect the current software state of the Trusted Platform: The measurement log contains the measurements that were produced by trusted software, since the platform was rebooted, respectively since the dynamic root of trust for measurement was started. To protect the integrity of the measurement log, chained cryptographic hashes of the message log are stored in the Platform Configuration Register, safe from attacks by malicious software, inside the Trusted Platform Module.

The last root of trust considered here[3], the *root of trust for reporting (RTR)*, provides functionality to report the software state of the platform. On current Trusted Platforms, the shielded locations forming the root of trust for reporting is implemented inside the Trusted Platform Module, and can be accessed using special "seal" and "quote" commands, which are briefly discuss below in Section 2.6.7, and Section 2.6.5. The "seal command basically allows local users to cryptographically "seal" secrets to a given software state. The "quote" command enables the local platform to produce a digitally signed receipt vouching for the integrity of the measurement log that can be verified by remote users.

2.3 Criticisim

The "trusted platform" concept proposed by the TCG, and its predecessor the *TCPA*, was severely criticized as threat to the self-determination and freedom of platform users. Ross Anderson [And02] sketched a rather negative vision in context of Digital Rights Management (DRM) systems and of open-source software. The scenarios discussed in Anderson's paper depict a situation, where the choice of (trusted) hardware platform and installed software is imposed onto the users by large companies. The worst case fear in Andersons paper is, that any open-source software development could be killed by *TCPA*-style Trusted Platforms. Similar criticism and fears were shared by others. For example Stallman [Sta10] went to the extend to use the term *treacherous computing* instead of *trusted computing*, based on the reasoning that:

> "Treacherous computing" is a more appropriate name, because the plan is designed to make sure your computer will systematically disobey you. In fact, it is designed to stop your computer from functioning as a general-purpose computer. Every operation may require explicit permission. [Sta10, p.205]

Proponents of Trusted Computing point out that the use of TPM and Trusted Computing functionality is voluntary and under the strict control of the user[Gra09; Gra06; TCG07a]. Actually the trusted platform module provides a complex set of commands and states related to activation, enablement and assignment of ownership. From the point of view of the platform owner, these functions indeed allow a high degree of control of the Trusted Computing function which should be enabled on the platform.

[3]Additional roots of trusts that where proposed by the TCG for mobile systems (e.g. root of trust for verification, root of trust for enforcement) are out of scope here.

2.4 Platform Ownership

One of the main points of disagreement between proponents, and opponents of Trusted Computing is the question of who *owns* or *controls* a platform. For a desktop computer used at home, the expectation might reasonably be, that the owner of the platform, the owner of the TPM and the user of the platform are the same person. For desktop computer used in an office the situation can be significantly different: Here the owner of the platform might be the company, the manager of the platform might be an IT department, and the user of the platform might be an employee.

The TCG architecture considers these different ownership situations in [TCG07a, p.23]. Three possible deployment scenarios for Trusted Platforms are discussed in [TCG07a, p.23]:

Consumer Owned Platforms correspond to the home computing example discussed above. Platform and TPM management is under full control by the user (consumer).

IT-Owned and Managed Platforms correspond to the corporate computing example discussed above. The platform and TPM are managed by a central authority, for example an IT department. This scenario envisions periodical monitoring of the platform for compliance with IT policies.

Consumer Owned Platforms with Outsourced Management are a hybrid variant between user and third-party management of the platform. The scenario outlined in [TCG07a] assumes an external (outsourced) IT service provider, who has a service contract with the user. The IT service provider takes ownership of the TPM before the platform is shipped to the user. The user can decide if the TPM should be activated during normal platform operation. Interaction with the IT service provider requires the TPM to be (temporarily) activated.

This flexibility with respect to ownership is reflected in the TPM's command set (cf. [TCG07b]) by supporting separate authorization for the "owner" (`TPM_TakeOwnership` command) and the "operator" (`TPM_SetOperatorAuth`) of the TPM. Additionally the TPM provides a delegation mechanism that enables owners of TPM objects, like keys, to delegate certain rights, like using the key to sign data, to other users on a per-object basis.

From a Trusted Computing opponent's point of view the later two deployment scenarios are more problematic for the user or consumer, as they allow third-parties to enforce policies on the software installed on the user's platform. Such strong enforcement of software policies could prevent whistleblowers from leaking information as discussed in the examples of Anderson [And02] and Stallman [Sta10]. Similarly the consumer owned platform with outsourced management can be seen as possible instantiation of a Trusted Computing protected multimedia appliance with DRM capabilities.

2.5 Trusted Platform Modules

The *Trusted Platform Module (TPM)* is the primary hardware trust anchors of any TCG-style trusted computing platform. It implements the root of trust for reporting (RTR) and the root of trust for storage (RTS). Commonly TPMs for desktop usage are realized as dedicated hardware modules, either as standalone chips, or as IP building block integrated into larger platform elements like the Southbridge[4] of an x86 platform. Pure software realizations[5] of TPMs are in principle possible, as demonstrated by Strasser's software TPM emulator [SS08; SS04] and IBM's virtual TPM approach [Ber+06; Gol+10]. Other implementations of software TPMs for use on mobile and embedded platforms, have been proposed in literature e.g. by Ekberg and Bugiel [EB09], England and Tariq [ET09], Dietrich and Winter [DW10], Winter et al [Win+12], and Winter [Win08].

This section discusses the functionality of the TPM that is relevant to the attacks discussed later in this master thesis. Most of the attacks discussed later have in common, that they focus on the link between the root of trust for measurement, which resides *outside* the TPM, and the Trusted Platform Module of the target platform. The TPM functionality discussed here is limited to the commands, and primitives that are directly relevant for the attacks discussed in this master thesis. We deliberately omit large parts the TPM command set, including ownership management, key handling, delegation, and authorization as they are not directly related our attacks, and thus can be considered as being out of scope.

2.6 Platform Configuration Registers

Platform Configuration Registers (PCRs) are special registers used to store the current software state of the platform. PCRs are implemented in *shielded locations* and together form the *root of trust for reporting* (RTM) inside the TPM. These registers are volatile, and take a well-defined initial state whenever the platform — or more precisely the TPM — is reset. The content of all PCRs is lost whenever the TPM is reset or powered off. There exists no command to directly set arbitrary PCRs to arbitrary user-specified values. Software can only change the content of PCRs in two restricted ways: To record a measurement, software can *extend* a PCR, via the PCR extend command discussed below. Certain special PCRs, which are mainly intended for debug purposes, can be reset under well-defined conditions using a dedicated PCR reset command.

When a dynamic root of trust for measurement (D-RTM) is started a special sequence of hardware bus transactions is initiated by trusted CPU microcode, to generate an initial good measurement in a dedicated PCR. Normal software is, at least in theory, not able to generate the special bus cycles required to signal the start of the D-RTM. Below we discuss the effects of invoking a D-RTM on the PCR values held by the TPM. Next, in Chapter 3 we analyze the low-level interface used by software drivers, and by CPU microcode, to communicate with the TPM. As part of this analysis we discuss *localities*, which are a simple mechanism that allows the TPM to identify the origin (e.g. software, CPU microcode) of requests at hardware level.

2.6.1 Initial State

The initial content of a PCR at platform reset depends on its designated purpose. PCRs that are used by the static root of trust are reset to an all zeros initial value, while PCRs used by a dynamic roots of trust initially have all bits set to one. Let $PCR_{i,0}$ denote the value of the i-th PCR immediatly after platform

[4]For example on Intel ICH-10 chipsets [Int08, p.241]
[5]Which run as “processes” on the main platform CPU, instead of being firmware embedded into dedicated security chips

reset, and let l_h[6] be the fixed bitsize of a single PCR, then the initial PCR values of a typical version 1.2 TPM are given by:

$$\mathrm{PCR}_{i,0} := \begin{cases} \{0\}^{l_h} & \text{if } 0 \leq i \leq 16 \vee i = 23 \quad \text{(Static RTM)} \\ \{1\}^{l_h} & \text{if } 17 \leq i \leq 22 \quad \text{(Dynamic RTM)} \end{cases}$$

The design decision to initialize PCRs used by a dynamic root of trust to a well-defined non-zero value allows software to reliably detect if a dynamic root of trust was started since the last platform reboot, or not.

2.6.2 Extend Operation

Software uses the *extend* command to measure “interesting” platform events into the Platform Configuration Registers. It is up to the software performing the measurements to define what constitutes an “interesting” event. Relevant events can for example be the start of additional software components, loading of configuration data, security relevant policy decisions, or policy violations. At TPM command-level the *extend* command makes no assumptions about data being extended into a PCR. The only restriction is that the data being extended must be encoded as a bit-string of fixed length l_m. For version 1.2 TPMs the length l_m is fixed to 160 bits by the choice of SHA-1 as the TPM’s hash function. Trusted software stacks define their own mechanisms to map events to suitable bitstrings for the PCR extend primitive. The Trusted Software Stack (TSS) specified by the TCG [TCG06] defines an event data-structure which is mapped to the actual bit-string by application of a hash function.

Let $\mathrm{PCR}_{i,t}$ denote the value of the i-th PCR at time t, let H be the hash function[7] that is used by the TPM, and let l_m be a constant bitsize fixed[8] by the TPM specification. We assume for simplicity that all PCRs use a common global time t and that time increases whenever *any* PCR is extended. With $||$ denoting the concatenation of two bitstrings, the effects of a state update at time t caused by extending PCR_i with a bitstring $m \in \{0,1\}^{l_m}$ can then be described as:

$$\begin{aligned} \mathrm{PCR}_{i,t+1} &:= \begin{cases} \mathrm{H}\left(\mathrm{PCR}_{i,t} || m\right) & \text{if } \mathrm{canextend}_{j,t} \wedge i = j \quad \text{(Extend the target PCR with message } m\text{)} \\ \mathrm{PCR}_{i,t} & \text{if } \neg\, \mathrm{canextend}_{j,t} \vee i \neq j \quad \text{(Other PCRs are unaffected)} \end{cases} \\ \mathrm{canextend}_{i,t} &:= \begin{cases} 1 & \text{if PCR } i \text{ can be extended (from current locality)} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

The new value of the PCR after the extend operation is obtained by computing the hash value over the concatenation of the old PCR value with the data to be extended. The computation of the new PCR value does not involve any TPM internal secrets. To recompute the value of a PCR it is sufficient, to know its initial value, and the sequence of measurements that were extended to reach the the current value. This is exactly the property required to protected the integrity of measurement logs. Note that the $\mathrm{extendable}_{i,t}$ predicate captures the locality based PCR access restrictions discussed later n Section 3.2.

One important property of the PCR extend operation that it preserves the order of measurements in

[6] $l_h = 160$ for TPM version 1.2
[7] SHA-1 for TPM version 1.2
[8] $l_m = 160$ for TPM version 1.2

the final value. Extending two distinct measurements A and B yields a different result than extending B and A. This property is required to ensure that relative order of measurements, respectively the temporal order of events leading to this measurements, in a measurement log can be protected properly.

2.6.3 Software-controlled Reset

Certain special PCRs can be reset by trusted software with help of the `TPM_PCR_Reset` TPM command. Software reset of PCRs is useful for debugging trusted applications during development. Typical desktop TPMs provide one dedicated debug PCR (PCR_{16}) that can be reset by any normal application running on the platform, without requiring any special authorization. Applications must not be able use the PCR software reset command to reset any non-debug PCRs, as doing so would compromise the chain of trust. The reset behavior of non-debug PCRs is controlled by TPM internal data-structures, which are fixed at TPM manufacturing time. In principle they can vary between TPM vendors, and even different firmware versions of TPMs from the same vendor. TPMs for desktop use typically implement the reset behavior as discussed in [TCG05], which is suitable for desktop computer systems with static and dynamic RTMs.

Let $PCR_{i,t}$ denote the value of the i-th PCR at time t and let l_h be the bitsize of the PCR registers. Furthermore, let $\mathrm{resettable}_{j,t}$ denote if the j-th PCR can be reset at time t, and let $\mathrm{zeroreset}_{j,t}$ denote if the reset value of the j-th PCR at time t is all-zeros or all-ones. The effects of a `TPM_PCR_Reset` command on any PCR j at time t can be described as:

$$\mathrm{PCR}_{i,t+1} := \begin{cases} \{0\}^{l_h} & \text{if } i = j \wedge \mathrm{resettable}_{j,t} \wedge \mathrm{zeroreset}_{j,t} & \text{(Reset to } \texttt{0x0...}\text{)} \\ \{1\}^{l_h} & \text{if } i = j \wedge \mathrm{resettable}_{j,t} \wedge \neg\, \mathrm{zeroreset}_{j,t} & \text{(Reset to } \texttt{0xF...}\text{)} \\ \mathrm{PCR}_{i,t} & \text{if } i = j \wedge \neg\, \mathrm{resettable}_{j,t} & \text{(Preserve on error)} \\ \mathrm{PCR}_{i,t} & \text{if } i \neq j & \text{(Preserve others)} \end{cases}$$

$$\begin{aligned} \mathrm{resettable}_{j,t} &:= \begin{cases} 1 & \text{if } j = 16 \quad \text{(Debug PCR 16 can be reset)} \\ 0 & \text{if } j \neq 16 \quad \text{(Any other PCR can not be reset)} \end{cases} \\ \mathrm{zeroreset}_{i,t} &:= 1 \end{aligned}$$

The $\mathrm{resettable}_{j,t}$ and $\mathrm{zeroreset}_{j,t}$ predicates depend on TPM internal data-structures, and are fixed at manufacturing time. For typical version 1.2 TPMs intended for use on desktop computer, the only resettable PCR should be PCR_{16}. This leads to the definitions of the $\mathrm{resettable}$, and $\mathrm{zeroreset}$ predicates shown above.

2.6.4 Invocation of a D-RTM

Dynamic roots of trust for measurement (D-RTMs) try to overcome the problems associated with long measurement chains by bringing the platform into a well-defined, trusted state, and starting a fresh (dynamic) measurement chain. Trusted application like the OSLO boot-loader [Kau07] and the *Flicker* framework [McC+08] mentioned earlier use D-RTMs to establish fresh measurement chains after transitioning from an arbitrary platform start into a trusted platform state as part of the D-RTM invocation.

To practically realize D-RTMs special support from the TPM and the platform are needed. The TPM has to provide a special mechanism to atomically resets a certain group of PCRs, and to perform a single extend operation to measure the new "trusted" software state. Let $PCR_{i,t}$ denote the value of the i-th PCR at time t and let l_h be the bitsize of the PCR registers. Furthemore let and let l_m be a constant bitsize of PCR extend message and let $m \in \{0,1\}^{l_m}$ be the first measurement done during the dynamic RTM startup sequence. The effect of the dynamic RTM startup at time t can then be described as:

$$\mathrm{PCR}_{i,t+1} := \begin{cases} H\left(\{0\}^{l_h} || m\right) & \text{if } i = 17 & \text{(Reset PCR 17 and extend } m\text{)} \\ \{0\}^{l_h} & \text{if } 18 \leq i \leq 22 & \text{(Reset PCRs 18-22 to } \texttt{0x0000...}\text{)} \\ \mathrm{PCR}_{i,t} & \text{if } i \neq j & \text{(Other PCRs are unaffected)} \end{cases}$$

It is critical for the security of D-RTMs that the special PCR measurements during D-RTM invocation can not be simulated by normal application software (cf. [Gra09]). The TPM uses a hardware-assisted mechanism, called localities, to protect against malicious applications trying to simulate fake a D-RTM startup sequences, and the associated measurements. All special PCRs involved in the D-RTM invocation sequence can neither be extended, nor reset by software. On a typical PC platform PCR_{17} is designated to receive the initial measurement done by CPU microcode. It requires the current locality to be locality 4, which should, at least in theory, only be accessible from trusted CPU microcode.

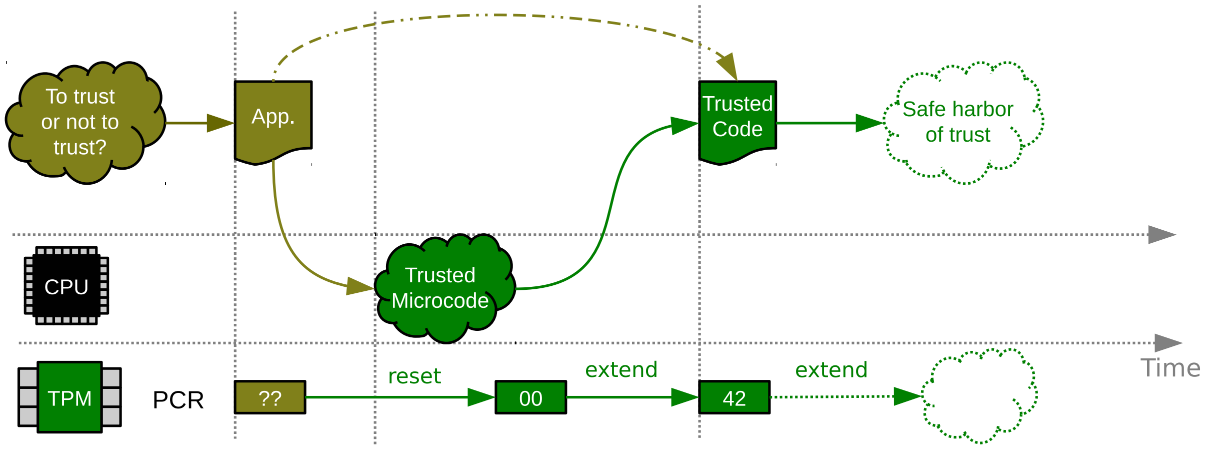

Figure 2.2: Start of a Dynamic Root of Trust for Measurement [Win11]

Figure 2.2 illustrates the startup process of a dynamic root of trust for measurement. To initiate the D-RTM invocation it is first necessary to load a special fragment of trusted application code into memory and to correctly setup the processor, and platform. These steps are performed while the system is still in a potentially untrusted, or unknown, system state. Once the preparation steps are complete, a special CPU instruction is used to trigger the actual start of the D-RTM startup sequence. In response trusted CPU microcode takes over control, produces the initial good measurement of the trusted application, and finally hands over the trusted application[9].

For example the OSLO boot-loader, which was proposed by Kauer [Kau07] as a possible mitigation against the TPM reset attack, uses this mechanism. This boot-loader initiates a D-RTM startup sequence to create a fresh measurement chain, which starts with a piece of trusted code from the boot-loader itself. The assumption is that software can not simulate the initial D-RTM measurement. This implies that a

[9]Depending on the CPU vendor there can be additional steps, which we omit here for brevity.

measurement of the trusted boot-loader in PCR_{17} indeed indicates that the trusted boot-loader has been started on the platform.

Our hardware attacks discussed later in Chapter 6, and Chapter 8 invalidate the assumption that *only* trusted CPU microcode can generate the low-level bus transaction needed to inform the TPM of D-RTM start.

2.6.5 Remote Attestation

Remote attestation allows a trusted platform to report its software configuration state to a remote party. During remote attestation, the TPM generates a digitally signed receipt of its current PCR values, using the *quote* command. This receipt is then used as evidence to "prove" that the platform was in a given state at the time the quote blob was created.

The Quote Operation The central TPM command required for remote attestation is the *quote* command. In principle the quote command generates a digital signature over a special message of fixed structure. The signed message contains a set of PCR indices, a cryptographic hash over the current values of the corresponding PCRs, and a cryptographic nonce.

Let $pcrs$ be a set of PCR indices of interest and let $PCR_{i\in pcrs,t}$ be the concatenation of the selected PCR values at time t. Furthermore let n be a user supplied bit-string of fixed length as defined by the TPM specification. Finally let H be a hash-function used by the TPM. The platform state $\mathrm{QuoteInfo}_t(n, \{pcrs\})$ at time t can now be encoded as[10]:

$$\mathrm{QuoteInfo}_t(n, pcrs) := (n||pcrs||H\left(\mathrm{PCR}_{i\in pcrs,t}\right))$$

To produce a trusted receipt — or *quote* — of the platform state at time t the TPM signs[11] the encoded platform state with the private key of a key pair K, to produce a signature σ. With $||$ denoting the concatenation of two bitstrings, construction of a *quote* blob by the TPM is given as:

$$\begin{aligned} m &:= \mathrm{QuoteInfo}_t\left(n, pcrs\right) \\ \sigma &:= \mathrm{Sign}_{K_{priv}}\left(m\right) \\ \mathrm{Quote}_t(K_{priv}, n, pcrs) &:= (m||\sigma) \end{aligned}$$

The *quote* blob is used to give proof of the current software state of the platform, as reflected in the Platform Configuration Registers of the TPM, to a remote verifier. Freshness of the quote blob can be ensured by incorporating a verifier provided nonce value into n when generating the quote. To check the validity of a quote blob the verifier extracts the QuoteInfo structure and checks the signature on that structure using the public key of key pair K. Furthermore the verifier has to check that the software configuration described by the QuoteInfo matches the requirements.

[10]The shown definition is simplified and omits several details of the real TPM structure for simplicity.

[11] to RSA in TPM version 1.2

2.6.6 Approaches to Remote Attestation

Attestation based on Known-good PCRs There are two major approaches for validating the software configuration encoded in the *quote* blob at the verifier side. The first, simple approach is to keep a database of known good PCR value sets representing trusted configuration and matching them against the *quote* blobs. This simple approach requires only little communication between the prover and the verifier.

Configuration Space Explosion The drawback of this simple approach is that a dedicated database of known good software configurations, and an approval mechanism to add new software configurations is needed. This likely will not be a problem if the number of software components, and good software configurations is small. With growing number of components, the number of configurations that need to be approved in the database grows quickly.

The problem here is caused in part by the granularity of measurements chains, and the significance of the order of measurements within a chain. To get a reasonably complete, and accurate view of the software running on a system, measurement chains need to be fine grained. This implies that virtually no events can be omitted, even if they have apparently no impact on overall system security.

To see the resulting problem with different loading orders, we consider a simple system consisting of only three components: an trusted operating system, a trusted device driver, and a trusted application program. For a first thought experiment we assume, that English, French, and German versions of all three components exist, and that each localized component results in a different binary[12]. Furthermore, we assume that the localized version of the three components can be mixed arbitrarily. It is for example possible, to run an English operating system, with a French version of the device driver, and a German version of the application. We assume that our trusted example system measures each component before execution, for example into PCR_{16}.

To realize the simple known-good PCRs approach for remote attestation, we need to build a database with possible values of PCR_{16}, which reflect a trusted system state resulting from starting the three components. In this first example it does not make sense allow configurations where the components are loaded in the wrong order, thus we fix one allowable load order with operating system first, driver next, and application last. We have three possible language version for the operating system, which can be combined with any of the three language versions of the device driver, which can be combined with any of the three language versions of the application. This already results in a total of nine trusted configurations that need to be kept in the database.

In a second thought experiment we now add a second trusted device driver, which exists in localized versions for English and German. With these two language variants of the same driver, the number of known-good configuration database entries increases by a factor of two to 18, given that the load order of the components is fixed, and that both device drivers must be loaded in a trusted configuration. Adding new components, again under the assumption of a statically fixed-load order, further increases the number of valid configurations.

Allowing different load orders complicates the situation further. In the worst case of k trusted com-

[12] Localization has been chosen to emphasize that *different* binaries can be *equally* trustworthy.

ponents, which can be loaded in an arbitrary order, we have to (additionally) consider all possible $k!$ permutations of the load orders. Unfortunately this explosion of the configuration space renders the simple known-good configuration database approach infeasible for many practical applications.

Attestation based on Measurement Logs To solve the problem of configuration space explosion, alternative attestation strategies based on transmission of the measurement logs to the verifier are typically used an practice. The idea here is to transfer and measurements log along with the quote blob that vouches for the log integrity. The PCR values in the quote blob protect the integrity of the entire log, and the digital signature on the quote blob ensures authenticity and integrity of the quote. To assert the trustworthiness of a given quote, the verifier now uses a database of known good software components in combination with the log entries. A configuration is rejected as untrustworthy if either the log entries can not be matched with the software database, or if the PCR values computed from the log do not match the values in the quote blob, or if the signature on the quote blob is invalid.

Nevertheless, the measurement log approach is still no silver bullet. Instead of having to store all good configurations in a database, the verifier now needs to assert the trustworthiness of an arbitrary configuration using a known good software database. In general it is insufficient to just check that each measurement in the log has a corresponding trusted entry in the software database, as the loading of executables can matter. In addition to measuring the binaries itself it will often be necessary to attest the trustworthiness of configuration files, complicating the process further.

2.6.7 Local Attestation (Sealing)

Local attestation assumes that the prover, and the verifier are both programs that execute on the *same* platform, with the *same* TPM, but may execute at different points in time. At a given time the prover decides that data should be protected, and only be released at a future time, when the platform has reached a specific software state that can be represented by particular set of future PCR values. The main primitive provided by the TPM to support local attestation is *sealing*. Sealing uses a TPM key K, to encrypt a small chunk of user-provided data, and to bind the encrypted data to a particular TPM, and a particular state. The binding to the TPM is achieved by requiring the key K to be a *non-migratable* key. Non-migratable keys K used for sealing can only be created inside the TPM, and their private parts can never leave the TPM in unencrypted form. It is *not* even possible to migrate sealing keys between two TPMs.

At a later point in time, when the verifier can ask the TPM to unseal the *sealed* blob, and to reveal the data stored inside. The TPM then compares its current PCR values to values expected in the *sealed* blob, and releases the data if and only if they match. Conceptually this comparison is similar to the known-good PCR database approach discussed earlier for remote attestation. One notable difference, which can be problematic in practical applications is, that sealing only supports specification of a single known good state[13]. When changing or updating software on the platform, this usually implies that any sealed blobs must be unsealed first, while the platform is still in the old state, and sealed again for the new platform state.

[13]For TPM v1.2

Chapter 3

The TPM Interface Standard (TIS)

“As soon as you trust yourself, you will know how to live.”

[Johann Wolfgang von Goethe, Faust]

In Chapter 2 we discussed principles of Trusted Computing and put a special emphasis on measurements and measurement logs. The communication path and the protocols used between the *root of trust for measurement* and the Trusted Platform Module is critical for transferring these measurements, and thus is essential for the security properties of the entire trusted platform. This chapter discusses the low-level details of the transport protocol used by hardware drivers to communicate with the TPM, based on Version 1.2 of the of the *TCG PC Client Specific Interface Specification (TIS)* as published in [TCG05].

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. “A Hijacker’s Guide to Communication Interfaces of the Trusted Platform Module”. In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018
- Johannes Winter and Kurt Dietrich. “A Hijacker’s Guide to the LPC Bus”. In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

3.1 The Initial Situation

Trusted PC Platforms are complex systems comprising of a large number of interacting hardware, and software components. Several layers of both hardware, and software, are involved whenever a normal

Linux application, like for example Trusted Computing enabled key store, communicates with the TPM.

Typically applications use a Trusted Software Stack to translate high level requests, such as “seal this data with that key”, into a sequence of byte blobs encoding the actual commands to be sent to the TPM. The actual TPM commands operate on a lower level of abstraction, and encode instructions like “load this key blob into the TPM”, “verify the authorization to use a key”, “seal a data blob with the loaded key”, or “unload this key blob from the TPM”. To send the byte blobs containing the encoded commands to the TPM, and to receive responses from the TPM, the Trusted Software Stack relies on the TPM driver of the operating system.

The TPM driver is, in general, the lowest-level software component that is involved in the process. Before version 1.2 of the TPM specification was published, there was no standard on how the TPM software drivers should communicate with the Trusted Platform Module. As a consequence, each TPM vendor had come up with its own proprietary interface between the software driver and the hardware TPM. Traces of this legacy interfaces can sill be found in the Linux kernel[1] sources. The `drivers/char/tpm` directory contains legacy drivers for communicating with pre-1.2 TPMs.

With release of the Version 1.2 TPM specification the Trusted Computing group defined a common low-level interface designed to allow a single TPM driver to be used with hardware TPMs from different manufacturers. The TIS specification describes a common, vendor-neutral interface for TPMs, and a mapping of this register interface to concrete hardware bus architectures, for example using memory mapped I/O. Instead of requiring one distinct TPM driver per vendor, the TIS specification strives to allow use of a single unified TPM TIS software driver for all TIS-compatible TPMs.

The TIS specification can be divided into a mostly hardware agnostic description of low-level TPM registers that are visible to software drivers, and hardware specific parts describing the binding to specific bus architectures. From the perspective of low-level software driver, a Trusted Platform Module looks like any other peripheral device with control and data registers. The driver communicates with the TPM by reading, and writing these low-level registers.

In the current chapter we briefly discuss the hardware agnostic details of the TIS register interface, and assume that some hardware mechanism to read, and write the low-level TPM registers just exists. In Chapter 4 we describe the Low-Pin-Count (LPC) bus, and analyze how this bus is used to implement the TIS standard.

We note that the term “register” as used in this section refers to the low-level interface used between the TPM and its driver software. These low-level registers should not be confused with higher level concepts, such as Platform Configuration Registers (PCRs) that were introduced earlier: PCRs in a TPM are accessed via dedicated TPM commands that are encoded as byte blobs. The low-level registers of the TIS interface provide the necessary transport functionality, to allow software drivers to send TPM command blobs to the TPM, and to receive response blobs from the TPM.

[1]`http://www.kernel.org/`

3.2 Localities

One of the major changes between earlier TPM interfaces and the v1.2 TIS interface is the introduction of *localities*. We already mentioned localities earlier, while discussing dynamic roots of trust. There we noted that localities prevent normal software from performing certain TPM operations, in particular from simulating the start of a dynamic root of trust.

Localities are indicators for the origin of read and write operations to the TPMs low-level register interface. This allows the TPM to detect if incoming register accesses come from trusted origins, such as trusted CPU microcode. Based on this origin information the TPM can decide if a register access should be processed, or dropped. At bus level localities are reported as part of the register address in a TPM register read, or write. Therefore, localities provide an access control mechanism to restrict access to certain TPM functionality, based on the logical, or physical origin of the platform component that accessing the TPM. Currently the TIS [TCG05] specification defines five different localities, plus one legacy locality intended for compatibility with older software. Locality 0 is the least privileged locality, and has no special access requirements. It can be used by static roots of trust for measurement, operating systems and applications.

Access to other localities is restricted by platform hardware, to ensure that normal software can not simulate a fake start of a dynamic root of trust for measurement (D-RTM). Chipsets that support D-RTMs implement hardware based filters, to restrict access to locality 4 to trusted platform hardware only. On x86 system with D-RTM support, the only part of the platform that can access locality 4 is trusted CPU microcode. Access to localities 1 to 3 can be unlocked during D-RTM startup, and can be used afterwards by trusted software. On x86 platforms that use an LPC bus, the five regular localities, and the legacy locallity, can be via special bus cycles that are considered in more detail in Chapter 4.

Certain operations performed by the TPM have locality restrictions in place, to prevent use by unauthorized parts of the platform. The valid target PCRs of the PCR reset command discussed in Section 2.6.3, and the PCR extend operation discusssed in Section 2.6.2, operation depend on the currently active locality. It is, for example, not possible to directly extend PCR 17, which is reserved for dynamic RTMs, using a standard PCR extend command issued from a locality other then locality 4. The only (intended) way to modify this particular PCR on a PC platform is to perform the special D-RTM invocation sequence outline in Section 2.6.4, which in turn can only be issued from locality 4.

Apart from being an access control mechanism, localities serve a second purpose: They coordinate access to the TPM by different, independent software components. To access the TPM, driver software needs to explicitly request access to the locality using the `ACCESS` register that is discussed below. This, in principle, allows independent software components on the platform, such as a trusted system management BIOS, and a trusted operating system, to synchronize their use of the TPM, without any extra software interfaces.

3.3 Register Interface

TPMs with a TIS-style interface expose a defined set of registers to the driver software on the host platform. Each of the five localities has its own view of the TPM register space. The registers of the TIS interface include device identification registers, interrupt status and control registers, data registers and per-locality control registers.

To successfully communicate with a TPM, driver software needs to work at least with the `ACCESS`, `STS`, and `DATA_FIFO` register discussed below.

3.3.1 Access register (`ACCESS`)

Software uses the access register to request access to a locality, and to detect when the access request has been granted. Each locality has its own separate copy of the access register. Parts of the value in the access register may be invalid, while the TPM is performing internal processing. A special validity bit in the access register indicates, if the remaining bits in the access register are currently valid. Drivers must use the validity bit, to determine if the value in the access registers is valid or not.

Acquiring and Releasing Localities In order to request access to the TPM, a driver first polls the access register, and waits until the validity bit indicates a valid register value. Next the driver tests if the read value indicates that the requested locality is already active., and if not issues a requests for the locality by writing a special value to the access register. Once polling of the access register value indicates that the requested locality is active, the driver can proceed to use the other TIS interface registers.

After completing its interaction with the TPM, the driver typically release the active locality by writing an appropriate value to the access register. This allows other (system) software on the platform to request access to the TPM.

As mentioned above, there is a separate instance of the access register for each locality. Together these per-locality interfaces form a simple arbiter that guards the other TIS registers against concurrency issues. Many of the TIS registers can only be granted to the currently active locality. Attempts to access these registers from any other, currently inactive, locality are silently ignored.

One example of concurrent usage of the TPM could be a trusted system management BIOS that utilizes parts of the TPM's non-volatile storage, or that measures certain platform events directly into the TPM's PCRs. This kind of trusted BIOS could work with arbitrary operating systems, given that it uses a different locality than the operating system. Measurements made by any of the involved component would be visible globally, as TPM resources such as PCRs are shared across all localities. Interference between the operating system, and the trusted BIOS, would not be possible, since the TPM's arbitration logic ensures that only one locality can be active at any given time.

Seizing During normal operation, the per-locality instances of the access register operate in fixed-priority arbitration order. When the current locality is released, the highest pending locality is granted access to the TPM. No attempt is made to ensure fairness to lower priority localities. The locality only changes after the currently active locality (if any) has been released by its owner.

This simple arbitration scheme would allow user of low-priority localities, such as locality 0, to block other localities from accessing the TPM, by simply not releasing the locality. To avoid this situation a preemption mechanism is included in the TIS specification: Software using higher priority localities can "seize" the TPM from lower priority localities, by writing to a special seize value to the access register.

Upon a successful completion of a seizing operation, the lower priority locality is forcefully released, and the higher priority locality that initiated the seizing request is granted access to the TPM. The access register of the lower priority locality is updated to indicate that a seizing operation took place, and that the lower priority locality no longer is active.

3.3.2 Status register (`STS`)

The status register enables TPM drivers to start, complete and abort TPM commands. Each locality has its own separate copy of the status register. Write attempts to the status register copy of an invalid locality are silently ignored, read attempts return an invalid value.

To send a command to the TPM, a driver first writes to the status register to indicate that a new command is being submitted, and then polls[2] the register until the read value indicates that the TPM is ready for command reception. Next the driver writes the TPM command blob byte-wise to the data FIFO register described below.

After submission of the last command byte the driver again writes a special "go" value to the status register, in order to start execution of the command. Finally the driver polls the `STS` register to detect that command execution has completed, and reads back the TPM response blob byte-wise from the data FIFO register.

3.3.3 Data FIFO register (`DATA_FIFO`)

Similar to the status register, copies of the data FIFO register exists for each locality, but can only be accessed by the currently active locality. Commands submitted to the TPM are writen byte-wise to this register. The status register indicates if the TPM expects more data to be written to this register. TPM responses blobs are read byte-wise from this register, after command execution has completed. Again the status register indicates if valid data is available in the data FIFO register, for reading.

[2]We ignore the status interrupts supported by TIS interface for simplicity.

Chapter 4

The Low-Pin-Count (LPC) bus

"I think complexity is mostly sort of crummy stuff that is there because it's too expensive to change the interface."

[Jaron Lanier]

In Chapter 3 we gave an overview of the TPM TIS interface, without going into hardware and bus specific details. This chapter considers hardware details of x86 based trusted platforms and describes the Low-Pin-Count (LPC) bus used to connect TPMs to such platforms. The preliminaries discussed in this chapter are used in the following chapters, to develop simple hardware attacks that directly aim at the LPC bus as weak hardware link in the chain of trust.

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to Communication Interfaces of the Trusted Platform Module". In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018
- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus". In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

4.1 Trusted PC Platforms

For the remainder of this chapter we assume that our Trusted Platform is a x86-based platform with a Version 1.2 Trusted Platform Module that is connected to the Low-Pin-Count (LPC) of the platform. The block diagram in Figure 4.1 gives a schematic overview of the relevant main components, and interfaces that can be found on this type of PC platform.

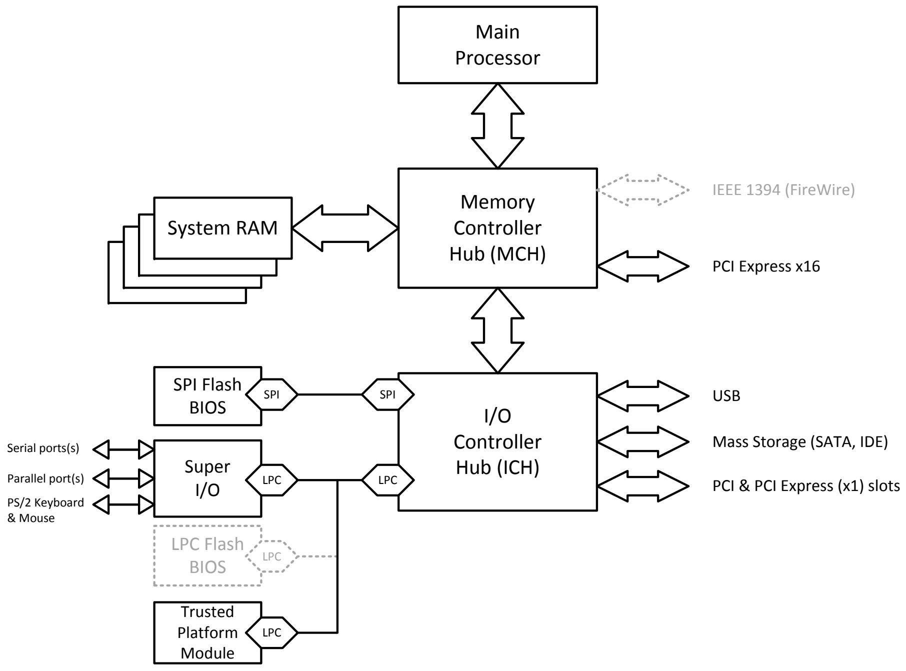

Figure 4.1: Components of a Trusted Platform [WD13]

The chipset of the shown example system is divided into three central components: The main processor, the memory controller hub, and the I/O controller hub.

BIOS firmware, operating system code, and application code execute on the main processor that is shown at the top of the block diagram. On desktop computers and servers the main processor generally is installed as replaceable module in a socket; notebooks and embedded x86 platforms may have there main processor permanently soldered soldered to the mainboard.

In contrast to many embedded system-on-chip platforms, x86 chipsets generally do not integrate memory controllers and peripheral I/O controllers in the main processor. Instead, x86 chipsets typically use memory controllers hubs, and I/O controllers hubs that are implemented in separate chips. Due to their connectivity and location relative to the main processor, these memory controller hubs and the I/O controller hubs are often referred to as “Northbridge”, and “Southbridge”.

The memory controller hub (MCH) is the only component with a direct interface to the x86 processor, and serves a threefold purpose: First, it connects the x86 processor, and the remaining chipset to the DRAM chips comprising the main system memory. Second, the MCH implements chipset devices with high memory band-with requirements, such as multi-lane PCI express hosts, integrated video controllers, or FireWire (IEEE 1394) host controllers. The third purpose of the MCH is, to provide an interface to the I/O controller hub (ICH), which in turn connects to most of the peripheral devices and externally accessible busses. The link between the MCH, and the ICH allows memory transactions in both directions. This enables the ICH to directly access main system memory.

Modern x86 chipsets, such as the Intel ICH10 chipset [Int08], use the LPC bus to connect the I/O controller hub (“Southbridge”) of the platform to peripheral devices on the mainboard. Typical devices found on an LPC bus include Super I/O controllers, and Trusted Platform Modules, as shown in Figure 4.1. Electrically the LPC bus is compatible with 3.3V PCI bus.

Depending on the chipset and flash memory type, the platform BIOS is either connected directly to the LPC bus, as indicated by the grey BIOS box in Figure 4.1, or to a dedicated serial peripheral interface (SPI) bus, as shown by the black BIOS box in Figure 4.1. LPC, and SPI flash memory chips have distinctpackages and chip markings making them easy to identify on the mainboard.

4.2 Properties of the LPC Bus

The Low Pin Count (LPC) bus is used on most current x86 platforms, as a replacement for older legacy bus systems, like the ISA bus. The details of the LPC bus and its interrupt signaling protocol are specified in two publicly available documents [Int02; C+95]. In this section we summarize the relevant parts of the main LPC bus specification [Int02], and briefly discusses the interrupt signaling protocol found in [C+95].

Three design properties of the LPC bus listed in [Int02] will be later of particular interest, when we start to consider simple hardware attacks against TPMs:

- *Systems with an LPC bus work without legacy bus systems. Due to the significantly lower number of bus signals, board layouts become simpler.*

  As already hinted at by its name, the LPC bus only requires a very small number of bus signals to work.

- *LPC bus host controllers and peripheral devices can be implemented as fully synchronous hardware designs. The bus does not use asynchronous signals (except system reset).*

- *Memory and firmware cycles on the LPC bus are able to address a full 32-bit address space. I/O cycles are able to address a 16-bit address space.*

Three major protocols are defined to access the LPC bus: The *target* protocol discussed in Section 4.2.2 is used for bus transfers initiated by masters, like the main processor that are *not* directly connected to the LPC bus. Communication between the main processor of a Trusted Platform, and the Trusted Platform Module is always done via special bus transaction adhering to the target protocol.

The two remaining protocols are used for direct memory access. We briefly briefly discuss them in Section 4.6, and Section 4.6.2 for completeness.

4.2.1 Bus Signals

The minimum configuration of the LPC bus requires seven bus signals, a connection to a 3.3V supply voltage, and common ground connection. In this minimum configuration, no support for interrupts, power management, and direct memory access is available. The basic signal required to implement a functional LPC bus are:

LRESET An active-low asynchronous reset signal, generated by the LPC bus host controller. The `LRESET` signal is the only asynchronous signal of the LPC bus. During platform reset, this signal is asserted (low level), to reset all peripherals on the bus to the initial state. Typically the LPC bus reset signal is directly connected to reset signal of the PCI bus on the platform.

LCLK This signal is the 33 MHz clock signal of LPC bus, and is generated by the LPC bus host controller. The LPC clock signal is normally derived from the PCI clock signal of the platform. On platforms with a 33 MHz PCI bus, the two clocks are usually connected directly. All other LPC bus signals, except for `LRESET` are synchronous to the clock signal, and change after the positive (rising) edge of the clock signal.

LFRAME Is the active-low framing signal of the LPC bus, and marks the start of bus transfer. The framing signal is generated by the LPC bus host controller. The `LFRAME` signal is normally asserted (low level) for one clock cycle, at the start of each new bus transfer. Peripherals on the bus detect the start of a new cycle, by watching for a low-to-high transition of the `LFRAME` signal in two consecutive clock periods. A second purpose of the framing signal is to abort pending bus transfers.

LAD0-3 Address and data is exchange over the four multiplexed `LAD` signals. These signals are bidirectional, and can be driven by the LPC bus host controller, and by peripherals, depending on the current phase of an ongoing bus transfer. The `LAD` signals are normally kept at a logic high level, when no bus transfer is active.

4.2.2 Target Protocol

The most common family of bus transfers seen on an LPC bus conforms to the *target protocol*. The target protocol family includes memory cycles, I/O cycles and firmware cycles, which we will discuss below. Target protocol cycles are always initiated by the LPC host controller in response to a read or write request directed to a peripheral device on the downstream LPC bus. The initiator of a target protocol cycle always is a master device, such as the main processor, which does *not* physically reside on the LPC bus.

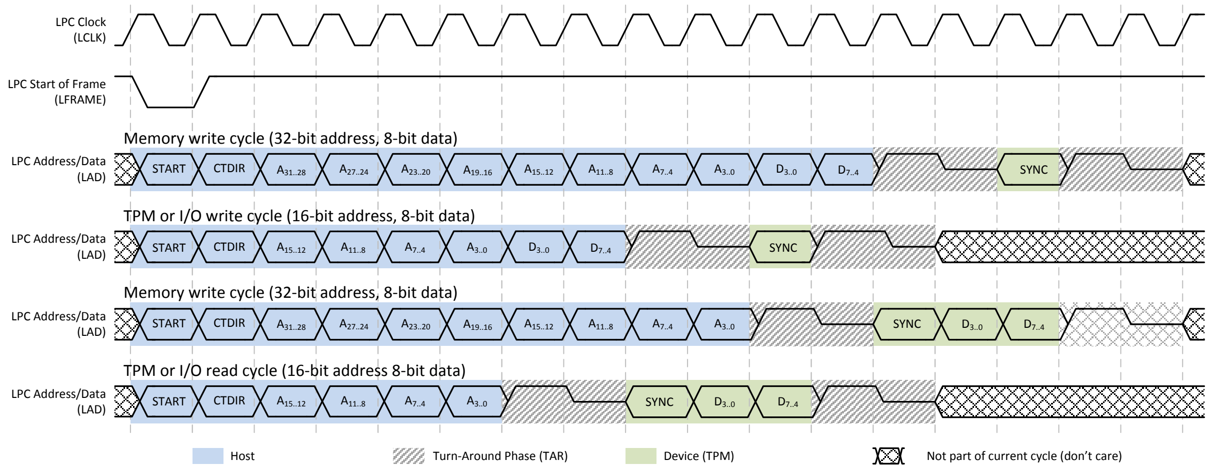
LPC Clock (LCLK)
LPC Start of Frame (LFRAME)
Memory write cycle (32-bit address, 8-bit data)
LPC Address/Data (LAD)
START CTDIR A31..28 A27..24 A23..20 A19..16 A15..12 A11..8 A7..4 A3..0 D3..0 D7..4 SYNC
TPM or I/O write cycle (16-bit address, 8-bit data)
LPC Address/Data (LAD)
START CTDIR A15..12 A11..8 A7..4 A3..0 D3..0 D7..4 SYNC
Memory write cycle (32-bit address, 8-bit data)
LPC Address/Data (LAD)
START CTDIR A31..28 A27..24 A23..20 A19..16 A15..12 A11..8 A7..4 A3..0 SYNC D3..0 D7..4
TPM or I/O read cycle (16-bit address 8-bit data)
LPC Address/Data (LAD)
START CTDIR A15..12 A11..8 A7..4 A3..0 SYNC D3..0 D7..4
Host
Turn-Around Phase (TAR)
Device (TPM)
Not part of current cycle (don't care)

Figure 4.2: LPC target Read and Write Cycle Waveforms

LAD[3:0]	Description
0000	Start of target cycle (see Section 4.2.2)
0001	Grant for bus master 0 (see Section 4.6)
0010	Grant for bus master 1 (see Section 4.6)
0101	TPM locality cycle (see Section 4.5)
1101	Firmware memory read cycle
1110	Firmware memory write cycle
1111	End of cycle (bus is idle or abort in progress)
(other values)	Reserved (ignored by devices)

Table 4.1: *START* values for LPC bus cycles defined in [Int02] and [TCG05]

Figure 4.2 show typical LPC bus target cycles. All bus activity on the LPC bus is synchronous to the 33 MHz LPC clock signal `LCLK`. Start and cancellation of LPC bus cycles is indicated via the active-low `LFRAME` framing signal. Address and data information for a specific bus cycle is exchanged over four bi-directional multiplexed address/data `LAD` signals.

A dedicated active-low `LRESET` reset signal, which is not shown in Figure 4.2, notifies all slave peripherals on the bus of a platform reset. These six signals (`LCLK`, `LFRAME`, `LRESET`, `LAD0`-`LAD3`) are sufficient to realize a minimal basic configuration of the LPC bus, without interrupts or DMA support.

Each bus cycle consists of several phases, or fields, during which either the LPC host controller or the peripheral device own the bus for a single clock cycle. During each phase 4 bits of information can be exchanged between host and device over the `LAD` signals.

4.3 Start Phase

To start a new bus cycle, the LPC host controller asserts the `LFRAME` signal for at least one clock cycle. The LPC bus specification in [Int02] explicitly allows assertion of the framing signal for than one clock cycle, and thus not define any upper limit on this duration.

Peripherals detect the transition of the `LFRAME` signal from low-to-high as *end* of the *START* phase, and thus as start of a new bus cycle. Since the LPC bus specification allows the *START* phase to be arbitrarily long, peripherals have to continuously monitor the bus for this transition. The last value seen on the bus, while the `LFRAME` was still low is the value of start field. Unknown or invalid start *START* values are silently ignored. Peripherals continue to monitor the bus for the start of the next valid cycle, if such an unknown encoding is found. Table 4.1 lists the valid *START* values defined in [Int02], and the TPM specific value added by the TPM TIS standard in [TCG05].

The *START* field of a bus cycle defines the protocol used to access the bus. Target, and TPM cycles use the LPC target protocol discussed in this section. Direct memory access cycles use a variant of the target protocol, as discussed below in Section 4.6. Bus master cycles use a separate bus mastering protocol, which is briefly explained in Section 4.6.

Cancellation of Bus Cycles The LPC host controller can abort pending bus cycles if it detects a locked up peripheral, or if no response is received at all. To abort a cycle, the host controller asserts the frame signal (low level) for several clock ticks. The bus specification at [Int02] mandates at least eight clock ticks, to allow all devices, and possibly bus bridges, to properly detect the abort sequence. During the last clock phase of the abort sequence the host controller drivers all `LAD` bus lines to high (logic 1) values, indicating the end of cycle pattern listed in Table 4.1.

For any devices which do not participate in the active LPC bus cycle the abort sequence looks as if the host controller was trying to start a new cycle with a reserved *START* value. The currently active (if any) device detects the abort sequence via the activity on the frame signal, and can take the required steps to handle the abort.

4.4 Target Cycles

Formally [Int02] specifies three sub-protocols of the target protocol that share the same `START` value. To distinguish between the sub-protocols, a `CTDIR` (cycle/type and direction) phase immediately follows the `START` phase, and indicates the sub-protocol and the transfer direction. Figure 4.2 shows examples of typical LPC target read and write cycles.

Target cycles for reading from or writing to the I/O address space use a 16-bit address, which is transferred with the most significant 4-bit nibble first, during four address phases. Similarly, target cycles intended to access the memory (mapped) address space use a 32-bit address which is transferred during eight address phases. The distinction between an I/O address space, and the memory address space relates to the dedicated “I/O port” instructions of x86 processors. Direct memory access target cycles use a different addressing scheme based on DMA channels, and contain an additional indication of the size of the DMA read or write transfer. Since DMA cycles are not relevant for communication with a TPM, we only briefly mention them in Section 4.6 for completeness.

In case of target write cycles, like the I/O and memory write cycles shown Figure 4.2, the data to be written to the device is sent by the host controller immediately after the last address phase. For read cycles, the device sends the data to be read of later after successful host-to-device synchronization.

Synchronization between Host and Device Bus ownership must be transferred once from the host to the device, and once from the device to the host for each target protocol cycle. The first bus turn-around happens after the host controller finished sending the address, and write data, to the device.

During the first clock cycle of the turn-around phase the host controller drives the `LAD` signals of the bus to logic high. In the second clock cycle of the turn-around phase the host controller electrically disconnects its `LAD` output drivers from the bus, by switching them to high-impedance state. The high level on the `LAD` lines is preserved during the second clock cycle of the turn-around phase by weak pull-up resistors on the `LAD` lines.

After the host-to-device turn-around has been completed, the host-controller waits for a synchronization response from the device. The device can insert wait-states into the transfer by sending appropriate *SYNC* values. Once the device has completed a write operation, or has data available for a read operation it responds with a special *ready* value in the *SYNC* field.

Alternatively a device may also respond with an *error* value in the *SYNC* field, to indicate that the read or write cycle failed at the device side. Reactions to a failed synchronization response may vary for

Host system address	TPM LPC address	TPM locality
`0xFED40...`	`0x0000–0x0FFF`	0 – Static RTM / Legacy
`0xFED41...`	`0x1000–0x1FFF`	1 – Trusted OS
`0xFED42...`	`0x2000–0x2FFF`	2 – Trusted OS Runtime Environment
`0xFED43...`	`0x3000–0x3FFF`	3 – Auxilary
`0xFED44...`	`0x4000–0x4FFF`	4 – Dynamic RTM / Trusted Hardware

Table 4.2: Locality Address Mapping according to [TCG05]

different host platforms and devices.

The bus specification in [Int02] even explicitly allows error responses to be silently ignored by the host platform. Typical desktop computer platforms follow this behavior: Failing write operations are usually ignored by the host controller or software; failing read operations are handled as if an all-ones value was read from the device.

After the device has send its final *SYNC* value a second turn-around phase transfers bus ownership from the device back to the host. This time the device drives the `LAD` signals of the bus to logic high during the first clock tick, and then disables its output driver after the second clock tick. Once the device-to-host turn-around phase has been completed, the bus is free for new transfers.

4.5 TPM-specific Extensions

Trusted Platform Modules use a variant of the LPC target protocol that is described in [TCG05]. The LPC bus cycles used by the TPM are almost identical to standard I/O target cycles, and only differ in the value sent in the *START* field.

Conceptually these TPM cycles add an additional 16-bit I/O address space that is exclusively reserved for the TPM. The TIS specification mandates that the chipset and its LPC host controller should translate host memory accesses in range `0xFED40000`-`0xFED44FFF` to TPM cycles on the LPC bus. The localities of version 1.2 TPMs are encoded into the host-side memory address according the mapping shown in the first column of Table 4.2.

On the LPC bus side, the top-most four bits of the 16-bit address specify the locality of the TPM access, as evident from the second column of Table 4.2. The lower 12-bits of address define the TPM register to be accessed, with byte-level granularity.

Register Access Example We assume, for example, that a TPM driver running on the main processor wants to read the TPM status register (`STS`) of locality 2. The TIS specification states that the locality 2 status register is at address `0x2018` from the TPM's point of view. Using the information from Table 4.2, we can determine that the driver has to read from physical memory address `0xFED42018` to generate the correct LPC bus cycle.

Figure 4.3 illustrates the components involved in our register access example, and the address translation performed. The TPM driver runs on the main processor an initiates a normal memory read from physical address `0xFED42018`. This read is decoded by the memory controller hub (Northbridge), and forwarded to the I/O controller hub (Southbridge). The I/O controller hub decodes the incoming memory

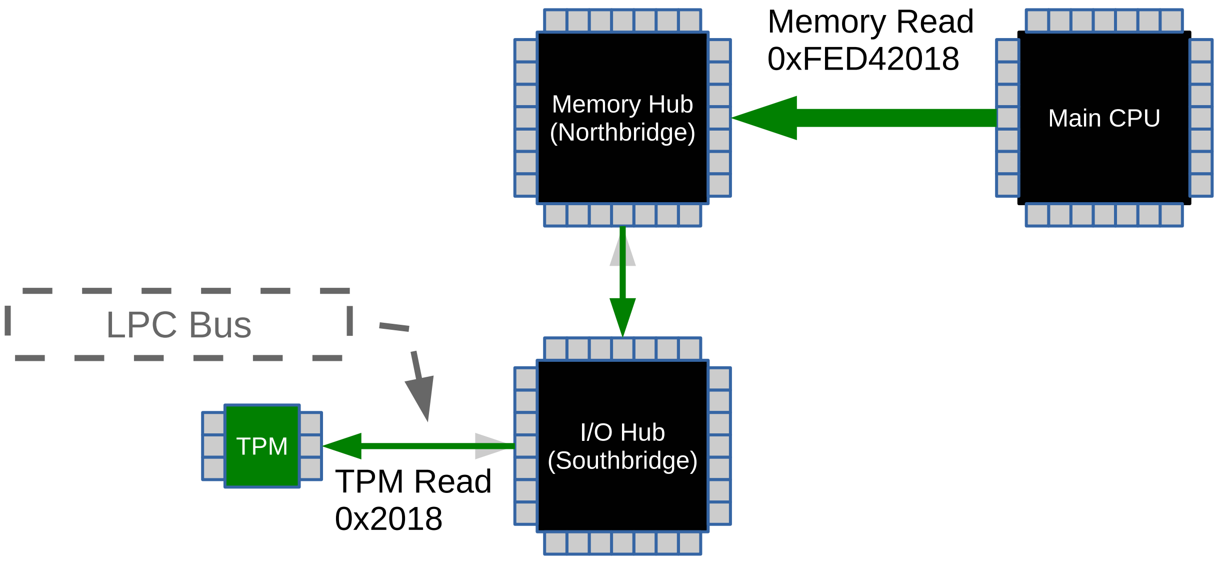

Figure 4.3: Reading the Locality 2 TPM Status Register

read and translates it into a corresponding TPM read cycle, with target address `0x2018`, which then is sent over the LPC bus to the TPM.

4.6 Direct Memory Access and Bus-mastering

Direct memory access (DMA) and bus-mastering requires that one additional `LDRQ#` bus signal is implemented, for each peripheral device which can initiate direct memory access as master. These `LDRQ#` signals are as uni-directional synchronous serial signals that transmit requests frames from the device to the host controller. When no requests are pending, all `LDRQ#` signals are at a logic high level. To request a transfer, a peripheral sends the desired DMA channel, or bus-master number over its private `LDRQ#` line to the LPC host controller.

4.6.1 DMA Transfers

To grant a DMA transfer, the LPC host controller issues special DMA cycles, and indicates the DMA channel number on the LPC bus. Depending on the direction of the DMA transfer the peripheral reads or writes the data. In case of DMA transfers, the peripheral has *no* direct control over the source or destination memory address. The transfer from and to memory is handled by the LPC host controller based on the values configured for the used DMA channel. The configuration of source and destination addresses is done by software.

4.6.2 Bus Mastering

The LPC bus supports a second form of direct memory access, which allows peripheral devices to act as bus masters on the LPC bus. To grant a bus master transfer the LPC host controller starts a special bus-master cycle. Immediately after the `START` phase, the LPC host controller relinquishes control of the bus, and hands over to peripheral, which requested the bus master transfer. The remaining bus transfer is driven by the master peripheral, and the slave peripheral.

To be compatible with the bus-master protocol, a slave peripheral must interpret the `START` field of the bus-master cycle, and the information sent be the master peripheral. Slave peripherals which do not support bus-master accesses simply ignore the bus-master LPC bus cycles.

The master peripheral has *full* control over the target I/O or memory address of the transfer. LPC host controllers can be implemented such that bus-mastering capable LPC peripherals are able to directly access main system memory.

4.6.3 Relations to the TPM

The Trusted Platform Module is designed to be a passive component (cf. [TCG07a]) that can not directly access system memory on its own. The decision to not allow TPMs to initiate direct memory access manifests in the absence of the `LDRQ#` signals in the recommended pinout for LPC TPMs found in [TCG05].

As major consequence of this design decision the TPM can *never* initiate a run-time integrity check of system memory, or directly modify any parts of system memory on its own. To implement this kind of functionality, the TPM *always* needs assistance from an external helper, such as BIOS firmware or (trusted) application software.

Chapter 5

Platform Reset Attack

"Don't let us make imaginary evils, when you know we have so many real ones to encounter."

[Oliver Goldsmith, The Good-Natured Man]

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to Communication Interfaces of the Trusted Platform Module". In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018
- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus". In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

5.1 Setting

One central assumption of Trusted Computing is that the measurements recorded inside a TPM reflect the actual chain of events that happened since the boot of its Trusted Platform. We recall, from Section 2.2 that Trusted Computing relies on *measuring* events, like the request to start a software component, *before taking any actions*, such as actually executing the program, in reaction to the events. The entire chain of trust relies on this principle, to guarantee that no untrusted action can be executed on the Trusted Platform *without being noticed*.

Remote attestation (cf. Section 2.6.5) enables remote verifiers to challenge the local Trusted Platform, which acts as prover, and to request evidence, by means of a digitally signed receipt of the PCR values that the platform is in a trusted state. Depending on outcome, the remote verifier may refuse to communicate with the local platform, or provide services requested by the local platform. Local attestation (cf. Section 2.6.7) allows the local platform itself, to seal secrets to a particular platform state. These secrets can only be revealed at a later time, on the same platform, with the same TPM, if the platform is in the correct state.

In both cases the chain of trust, at least in theory, ensures protection for assets by rejecting untrusted platform, and by refusing to release secrets when a platform is in an untrusted state. Arguably, an attacker restricted to software-only attacks, should not be able gain unauthorized access to assets protected by these Trusted Computing primitives, under the assumption that trusted software is implemented correctly. In practice the restriction to software-only attacks is often too strong, in particular in combination with remote attestation. We claim that attackers who desperately want to access a service, or extract a secret, will attack their own hardware, and the hardware of others, to gain access, given that the required effort is reasonable.

In this chapter we first consider the classic TPM reset attack of Kauer [Kau07], and Sparks et a. [Spa07], as a prime example of an extremely simple to mount hardware attack that breaks the relation between the actual platform state, and the measurements held by the TPM. Based on these considerations, we then investigate the effects of reversing the role of the TPM and its platform, which leads to our own *platform reset attack*.

The classic TPM reset attack does not require any permanent hardware modifications, and can be used to construct fake measurement chains from scratch, with the only limitation that PCRs designated for *dynamic roots of trust for measurement* can not be manipulated. Our platform reset attack requires minor hardware modification, and can be used to freeze the PCRs of the TPM, while rebooting the remaining platform into an untrusted state.

5.2 Classic TPM Reset Attack

The classic TPM reset attack was first described by Kauer [Kau07], and Sparks et al. [Spa07]. Figure 5.1 illustrates the control flow during a classic TPM reset attack. In this setting the platform under attack performs a normal trusted boot, and correctly constructs a chain of trust. At the time of attack, the software running on the platform remains largely unaffected, while the TPM is tricked to believe that a platform reset occurred. To simulate this situation, an attacker simply has to exploit the active-low nature of the LPC bus reset signal that we discussed in the previous section. The reset signal is active when it at a logic low (0V) level, which can easily be generated by grounding the TPM's reset pin using a piece of wire

The direct impact of this fake reset is that all PCRs are set to their initial state (cf. Section 2.6.1). Therefore, the adversary ends with a (still) running platform that has all PCRs in its initial state, and can proceed measuring the proper sequence of fake events into the PCRs.

Figure 5.1 shows the attack flow of the classic TPM reset attack, assuming that the attacker boots the system into a normal trusted state, performs the reset attack, and then proceeds with untrusted software.

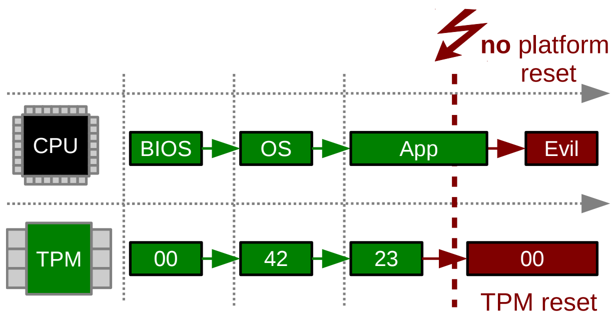

Figure 5.1: Classic TPM Reset Attack [WD13]

Alternatively the adversary can directly boot into an untrusted *evil* image, which eliminates any potential influence from the trusted OS and its user-space, and then perform the reset attack. To generate fake measurement values using the classic TPM reset attack usually the following steps are required:

1. Grab the measurement logs required to produce the “correct” platform state.
2. Reset the TPM by grounding the LPC bus reset signal (the remaining platform is unaffected).
3. Simulate the effects of BIOS startup sequence on the TPM. (Send a `TPM_Startup` command.)
4. Replay the measurement log captured earlier by issuing the proper PCR extend commands.

The classic TPM reset attack is well suited to simulate arbitrary fake measurements, which normally would be done by the *static root of trust for measurement*. It is, however, not possibly to create fake measurements in PCRs designated for a *dynamic root of trust for measurement (D-RTM)*, due to the restrictions (cf. Section 2.6.4) imposed by the hardware.

By using a D-RTM enabled boot-loader, such as the Kauer’s OSLO boot-loader [Kau07], the classic TPM reset attack can be mitigated, as the D-RTM measurement of the trusted boot-loader is easily distinguishable from the reset state (`0xFFFF...`, cf. Section 2.6.1) of the corresponding PCR.

5.3 Platform Reset Attack

A discussion with Dries Schellekens, one of the authors of [KSP05], after the first public presentation of our frame hijacking attack [WD12] at EuroPKI 2011 brought up an interesting variant of the reset attack, which reverses the roles of the TPM and the platform, leading to a “platform reset attack”. In this section, we discuss our results on the platform reset attack based on our earlier work done in [WD13]. We note that the outline of a very similar attack has been discussed by Schellekens in his dissertation [Sch12], while our journal paper was already under review[1]

From the point of view of an attacker, resetting the platform has the significant advantage that no prior knowledge about the exact sequence of measurements, and the measurement log is needed. Instead, it is sufficient to boot the platform into a trusted state, using the original *trusted* software image. Next, the attacker just has to trigger a hardware reset of the platform, for example by pressing the reset button,

[1]Thus creating a situation similar to the original TPM reset attack.

without resetting the TPM. The tricky part here is to find a reliable method, to isolate the platform reset from the TPM reset.

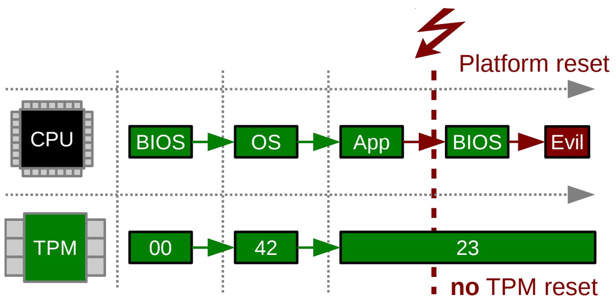

Figure 5.2: Platform Reset Attack [WD13]

Figure 5.2 illustrates the control flow of the platform reset attack. Initially we assume, that the adversary has found some way to prevent the TPM from receiving the platform reset signal. At the time of attack, the TPM keeps its PCR values, while the platform starts a normal boot sequence. This allows the adversary to boot his evil operating system image of choice, while the TPM still holds the PCR values from the previous *trusted* platform state.

Actually implementing the platform reset attack is slightly more complicated than just grounding a TPM pin with a piece of wire. The remaining sections of these chapter consider the practical issues that arise, and discuss the steps, and hardware modifications that are required to resolve the issues.

5.3.1 Triggering a Platform Reset

To mount a platform reset attack, the attack first has to find a reliable way for triggering a platform hardware reset, without turning off the power supply. At least the 3.3V power supply of the TPM must remain intact during the entire attack, to avoid losing the TPM state. Many modern desktop PCs do not have a dedicated hardware reset button. Some mainboards still provide a dedicated connector for an external reset button, which quickly solves that problem. Alternatively, an adversary can resort to tamper with the power-good signal of the power-supply, or try to identify the main reset signal manually on the mainboard.

In principle, the attacker can also try to force an immediate platform reboot in software. The disadvantage with this approach is, that a TPM-aware operating system, or BIOS may produce additional measurements during shutdown which interfere with the intents of the attacker.

5.3.2 Shielding the TPM from the Platform Reset

Once the issue of generating a suitable platform reset signal has been resolved, the adversary can proceed to shield the TPM reset from the platform reset, *and* the BIOS initialization sequence. As discussed earlier, the LPC bus uses an active-low reset signal, thus a 0V level on the reset line asserts the reset signal.

In case of the classic TPM reset attack this property was advantageous for the attacker, since it allowed assertion of a LPC bus reset by simply grounding the reset line. Unfortunately the very same property is a serious issue for the platform reset attack, depending on how the reset signal is generated on the target mainboard.

The LPC bus specification [Int02] does explicitly specify a particular way of implementing the reset signal at an electrical level. This, in principle, leaves room for both *pull-up* and *pull-down* realizations. In the *pull-up* case, the reset signal is connected to the positive supply voltage through a resistor and is activated by a driver transistor connected to ground. A *pull-down* realization works the opposite way, by placing the driver transistor between the reset line and positive supply voltage, while the resistor connects the reset line to ground potential.

On motherboards with a *pull-down* style reset it is possible to shield the LPC reset signal from the TPM, and all other LPC bus peripherals, by simply short-circuiting the reset signal to positive supply voltage. In this case the short-circuit does not cause any harm, as it corresponds to normal operating conditions when the reset is not asserted.

Pull-up style reset lines are more problematic, since the reset signal cannot be directly forced to positive supply voltage (inactive reset). Doing so would create a low-impedance path between positive supply voltage, and ground, whenever the platform tries to assert a reset signal. This in turn would cause excessive current flows during an LPC bus reset, and would likely destroy the output drivers of the platform's Southbridge after a short period of time.

5.3.3 Suppressing Resets

To reliably isolate the reset signal visible to the TPM, from the main platform reset signal, and attacker in general has to modify the platform hardware.

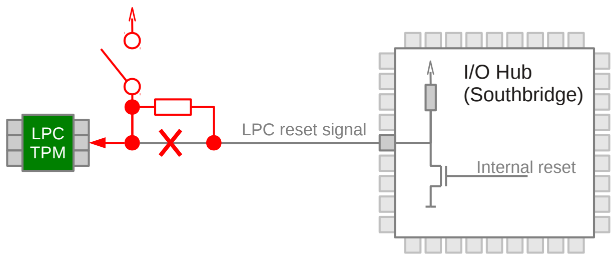

Figure 5.3: Simple Platform Reset "disable" Switch [WD13]

Figure 5.3 depicts a simple hardware modification for pull-up style reset lines that enables an attacker to properly decouple the LPC bus reset signal that is generated by the Southbridge, from the signal seen by the TPM. For this hardware modification, the attacker first cuts the LPC reset signal close to the the TPM, and inserts a large resistor. Next, the attacker installs a switch between the TPM-side reset signal, and the LPC bus supply voltage. This switch can later be used to selectively isolate the TPM side reset signal from the remaining LPC bus.

During normal operation, when the switch is open, no manipulations of the reset signal take place. In this mode the resistor has no noticeable effect on the LPC bus reset signal.

To start a platform reset attack, the adversary first closes the switch, and then triggers the platform reset. We assume that the platform power supply remains active during the reset.

Closing the switch connects the TPM-side of the LPC reset signal to supply voltage, pretending that the actuve-low reset signal is *inactive*. To assert an LPC bus reset, the Southbridge controller activates its internal driver transistor, to bring the Southbridge-side reset signal to a low (ground) level.

The voltage drop across the driver transistor, and thus the voltage level at the Southbridge-side reset output becomes almost zero. At the TPM-side, the reset signal is forced to high (supply) level by the attacker-installed switch. The external resistor added by the attacker ensures that the TPM-side reset is kept at high level, and that the current flow from supply voltage into the Southbridge-side reset output is limited. On the TPM-side we stills see the full supply voltage, indicating an *inactive* reset signal. The voltage drop across the attacker-installed resistor is close to full supply voltage, minus the negligible voltage drop across the output transistor in the Southbridge.

Using this simple hardware modification, an adversary can easily control the relation between the TPM reset signal, and the platform wide reset. Under normal operation conditions, when the *reset disable switch* is open, no changes of platform behavior are visible to software, and hardware.

5.3.4 Dealing with the BIOS

Triggering a platform reset, with the TPM reset disable switch activated, still causes software visible side-effects that can be observed by the platform BIOS. One of the first tasks trusted BIOS has to perform during boot, is to detected the presence of the TPM, and to submit a TPM startup command if a TPM is found.

Failure to detect a TPM normally does *not* cause the BIOS to abort the platform boot sequence. It should however be noted, that TPM detection and submission of the startup command are two different actions:

- To *detect* the presence of a TPM it is sufficient to check the identification registers, such as the vendor ID register, defined in the TIS [TCG05] specification.
- To actually carry out the TPM *startup sequence* several TPM commands, including `TPM_Startup` must be submitted using the TIS defined communication protocol. Each of the commands in the startup sequence can potentially fail, leaving different choices of error handling and recovery strategies up to BIOS vendors.

The error handling strategy of the BIOS directly affects the effort required by the attacker to carry out a platform reset attack. Simple BIOS implementations, we call them *attacker friendly*, silently ignore failures to initialize a TPM on platform boot. There are several possibly causes for TPM initialization to fail: First, there may be no TPM present on the platform at all. In this case any attempts to communicate with the (absent) TPM over the LPC bus will fail, and the BIOS will observe failed LPC bus reads and writes. Any attempts to read from the TPM's identification registers, will typically return an error value (cf. Section 4.4) in reaction to the failed bus transfers. Second, the hardware, or the TPM, may be defective. The effects observed by the BIOS in this second case can vary largely, and range from a non-responsive TPM, e.g. if a PCB trace is interrupted, to an error response from self-test commands.

The third likely reason for initialization errors is an ongoing platform reset attack that causes the platform to reboot independently of the TPM. In this case, the BIOS sees an error response when it tries to send the startup command to the TPM. From the point of view of the BIOS the startup command must be sent, since the platform is just being booted. From the perspective of the TPM, the startup command is unexpected, since no hardware reset of the TPM took place.

Most of the platforms that we investigated while developing the attacks described in this master thesis had *attacker friendly* BIOS implementations. The BIOS implementations encountered by us do not properly distinguish between the cases discussed above, and treat any failure to initalize the TPM, as if the TPM was absent, and continue to boot without a TPM. This behavior allows an attacker, to easily mount a platform reset attack, without having to pay special attention to the BIOS.

More elaborate BIOS firmware could, at least in theory, analyze the error code returned by a `TPM_Startup` command, and detect that something is wrong. We suggest, that there should be at least a check if the returned error result code indicates that TPM commands were submitted in the wrong sequence with respect to `TPM_Startup`. The TCG specifications define a special error code for this situation, and encountering this error code on a production platform is a sure sign of either a BIOS bug, or a tampering attempt, such as a platform reset attack.

5.3.5 Hiding the TPM

We just noted that the platform BIOS often *does not* interfere with the intents of the attacker during a platform reset attack. The situation can, however, be quite different for operating systems, like Linux, which come with their own TPM drivers, and TPM detection logic. For example, the standard Linux driver for TIS-compatible TPMs (`tpm_tis`) can be configured[2] to perform its own TPM detection, and to ignore the detection results of the BIOS. This override functionality of the driver is needed on some Trusted Platforms, where Linux does not recognize the presence of a TPM, based on the information (e.g. ACPI tables) that are provided by the BIOS.

When the platform under attack is reset, the TPM retains its current state. As discussed before, the BIOS sends a TPM startup command, receives an error response since the TPM was already started, and typically proceeds as if no TPM was present at all. The Linux kernel later loads its own TPM driver. We assume that the TPM driver is set to ignore the TPM detection results of the BIOS. At this point, the TPM is still in the same (fully operational state) as it was *before* the platform reset. In this case the TPM driver loads normally, and software can immediately start to use the TPM.

This can cause problems for the platform reset attack, if the attacker wants to reboot into an operating system image that uses the TPM, *and* that creates measurements *before* the attacker can interfere. The simplest solution here is to use an evil software image, which does not create unwanted measurements, instead of the original Trusted software image.

In the remainder of this section, we discuss an alternative solution that extends the hardware modification that we used earlier to isolate the TPM reset. This extended hardware modification will allow the attacker to hide the TPM from the BIOS *and* the operating system.

Extending the Reset Disable Switch We recall that the LPC bus framing signal can stay in a low level for an indefinite amount of time. An attacker can easily extend the reset disable switch discussed

[2]By loading the driver with the *force* option as: `modprobe tpm_tis force=1`

earlier, with a second switch, or a logic gate, to decouple the TPM-side of the LPC bus reset signal *and* the LPC framing signal from the main platform.

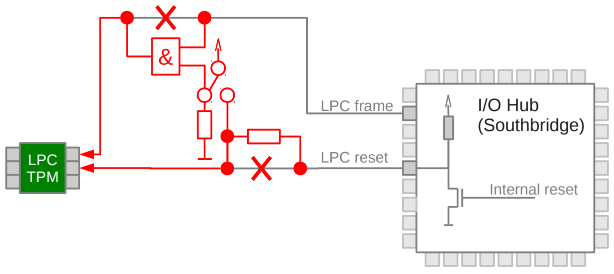

Figure 5.4: Extended Platform Reset "disable" Switch [WD13]

Figure 5.4 shows a possible extension of the simple *reset disable* logic discussed earlier into a *disable-and-hide* switch. Handling of the reset signal works exactly in the same way as discussed before. By decoupling the reset signal, as before, the TPM can be isolated from the platform reset. The hardware modification in Figure 5.4 additionally adds the logic, to decouple the LPC bus framing signal. Additional decoupling of the LPC bus framing signal prevents the TPM from detecting the start of any LPC bus transfers, and therefore effectively hides the TPM from the platform.

The switch position shown in Figure 5.4 indicates normal operation: One input of the logic AND gate is connected to a constant logic one value, while the second input is connected to the original LPC frame signal received from the Southbridge. In this mode the output of the AND gate simply follows the logic level present on the LPC frame signal input, and the TPM sees the original versions of the frame and the reset signals.

To prepare a platform reset attack, the attacker turns the switch into the other position. Now the reset signal seen by the TPM stays at its inactive level (logic one) regardless of the Southbrigde's reset output. The AND gate sees a constant logic zero at the input connected to the switch, causing it to generate a constant logic zero output regardless of the logic level of the LPC frame input.

The TPM now always sees an inactive reset signal, combined with a constant low level on the LPC framing signal. At the TPM side no bus transactions are received. At the platform side, no TPM appears to be present on the platform. Any communication attempts with the TPM end in timed-out LPC transactions.

Once the platform has booted the untrusted software image of the adversary, the switch can be turned back to normal operation. This allows the adversary to use the TPM again, which retained the state from *before* the platform reboot.

Chapter 6

Frame Hijacking Attack

"The only way to get rid of a temptation is to yield to it."

[Oscar Wilde, The Picture of Dorian Gray]

The reset attacks discussed in Chapter 5 are only one way to tamper with measurement chains of trusted platforms. Reset attacks are simple to mount, on require little to no permanent modifications to the platform under attack. From the poitn of view of an attacker the major drawback of reset attacks, is that they do not allow arbitrary control over the platform configuration registers used by the dynamic root of trust for measurement (D-RTM). This chapter discusses LPC *frame hijacking*, an active bus modification attack that we initially presented in [WD12], and later refined in [WD13]. Frame hijacking allows an attacker to exercise control over the D-RTM platform configuration register, while keeping the hardware modifications to the target platform comparable to the platform reset attack discussed in the previous chapter.

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to Communication Interfaces of the Trusted Platform Module". In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018

- Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus". In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

6.1 Setting

The setting for the frame hijacking attack discussed in this chapter is similar to the setting of the platform reset attack discussed in the previous chapter: We assume that the attacker has unrestricted physical access to the local platform, and is capable of doing simple hardware modifications. Furthermore, we assume that the platform under attack uses a dynamic root of trust for measurement (D-RTM). The target platform can be a trusted desktop computer with a boot-loader that starts a D-RTM (cf. Section 2.6.4), to mitigate the effects of a classic TPM reset attack. The goal of the attacker in this setting is to simulate the measurements that are generated during the start of the (trusted) D-RTM, without actually starting any trusted software.

We claim that this is possible on many platforms by using simple hardware modifications that are of similar complexity as the platform reset attack described in the previous. In the remainder of this chapter we describe the *frame hijacking* attack, which achieves this goal. The attack described in this chapter only requires permanent modification of the LPC bus framing signal, and passive probing of the LPC clock signal, and at least two LPC data signals.

6.2 D-RTM Startup Sequence

Before going into the details of the *frame hijacking* attack, we first take a closer look at the low-level bus transfers that are generated by trusted microcode, to establish the initial good measurement of a D-RTM. When a dynamic RTM is started. the platform hardware measures the initial trusted code being executed as part of this *late-launch* of the D-RTM. On current x86 systems these initial measurement is done by trusted microcode running on the main processor. To send the measurement to the TPM, the main processor uses three special TPM registers that are only accessible at the highest locality level (locality 4). Access to locality 4 is restricted to trusted CPU microcode. Software running on the platform can not access the D-RTM related TPM registers.

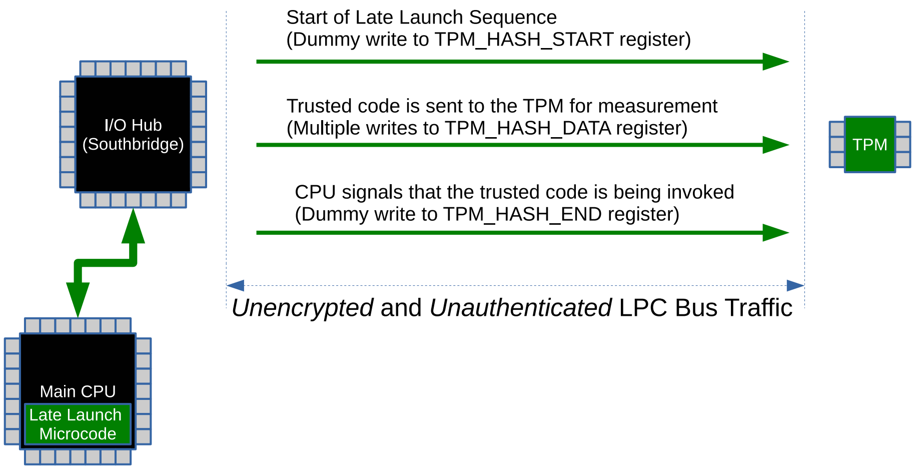
Start of Late Launch Sequence
(Dummy write to TPM_HASH_START register)
Trusted code is sent to the TPM for measurement
(Multiple writes to TPM_HASH_DATA register)
CPU signals that the trusted code is being invoked
(Dummy write to TPM_HASH_END register)
I/O Hub
(Southbridge)
TPM
Main CPU
Late Launch
Microcode
Unencrypted and *Unauthenticated* LPC Bus Traffic

Figure 6.1: TPM Interaction during D-RTM Startup [Win11]

Figure 6.1 illustrates the overall sequence of events, during the earliest stage of starting a D-RTM, when trusted microcode is measuring the first piece of trusted software to be started. The communication flow during this phase unidirectional, and data is only sent from the CPU to the TPM. Figure 6.2 lists the three TPM registers in the locality 4 address range, that are involved. The performed sequence of register writes consist of three phases:

1. First, the processor performs a dummy write to the `TPM_HASH_START` register. This register write signals the start of a D-RTM late-launch to the TPM.
2. Next, the processor sequentially writes the desired measurement value byte-wise to the `TPM_HASH_DATA` register. The value written by the processor typically is the hash of the initial trusted code to be executed by the D-RTM.
3. Finally, the processor signals the end of the D-RTM start sequence to the TPM by performing a dummy write to the `TPM_HASH_END` register.

Address	Register
`0x4020`	`TPM_HASH_END`
`0x4024`	`TPM_HASH_DATA`
`0x4028`	`TPM_HASH_START`

Figure 6.2: TPM 1.2 TIS Registers used during D-RTM Startup [WD12]

Upon completion of the last step, the TPM resets PCRs 17 to 22, and then extends the measurement data received via the `TPM_HASH_DATA` register into PCR 17, as we discussed earlier in Section 2.6.4. All of LPC bus writes to the TPM are sent in plain, without any kind of integrity protection. To simulate a D-RTM startup, it therefore suffices to simulate these simple register writes.

6.3 LPC Bus Memory and TPM Cycles

While discussing the basics of the LPC bus in Chapter 4 we already noted, that I/O bus cycles and TPM bus cycles only differ in the value transmitted during their start phases. Upon closer investigation of the bus cycles shown in Figure 6.3, a second similarity becomes obvious: I/O and TPM bus cycles only differ from memory cycle in the length of the address phase. The former two have a 16-bit address field, the latter one has 32-bit address field. Figure 6.3 compares an LPC memory write cycle with a TPM write cycle, to visualize this similarities.

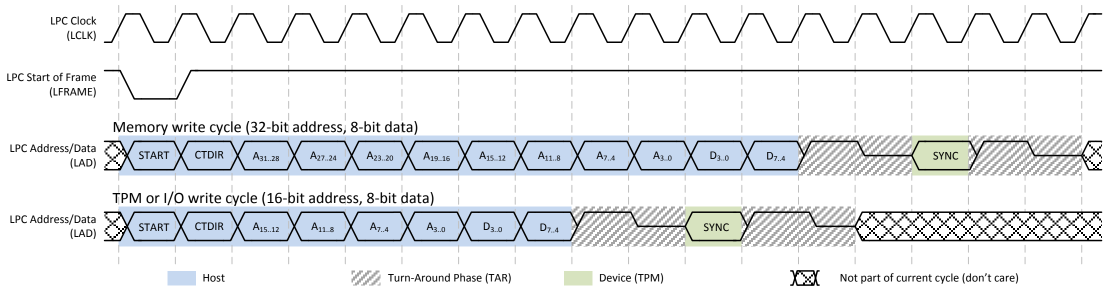

Figure 6.3: An LPC Memory Write Cycle compared to an LPC TPM Write Cycle

A memory cycle requires four additional clock periods, compared to a TPM cycle. The length difference is solely caused by the longer address phase of the memory cycle. In total the address and data phase of a TPM write cycle takes six clock periods, while the corresponding parts of a memory write cycle requires ten clock periods.

6.3.1 Hijacking LPC Memory Cycles

Apart from the length difference in the address phase the overall structure of TPM, and memory cycles shown in Figure 6.3 is identical. This can be used as basis for thought experiment: We can delay the start of the TPM cycle that is shown at the bottom of Figure 6.3 by four clock periods, while keeping the start of the memory cycle at its original position.

As a result, the trailing parts of the memory and the TPM cycles from Figure 6.3 would be in perfect alignment. In case of write cycles, the alignment starts at the data-phase. For read cycles, which we do not show here, the alignment starts at the host to peripheral turn-around phase.

We can use this observation as basis for a simple bus modification attack that places an attack device as man-in the middle on the bus. On the platform side, the attack device has to monitoring the LPC bus for specially crafted memory cycles on one side. On the TPM side, the attack device has to generate TPM cycles.

One conceptually simple, yet very powerful, approach to realizing the attacker device is to simple remove the TPM from the platform, and in its place install the attacker device. We will focus on this ideas later in Chapter 7, and Chapter 8. In the remainder of the current chapter, we consider a simple hardware attack that only requires manipulation of a single bus signal, without the need to remove the TPM from the platform.

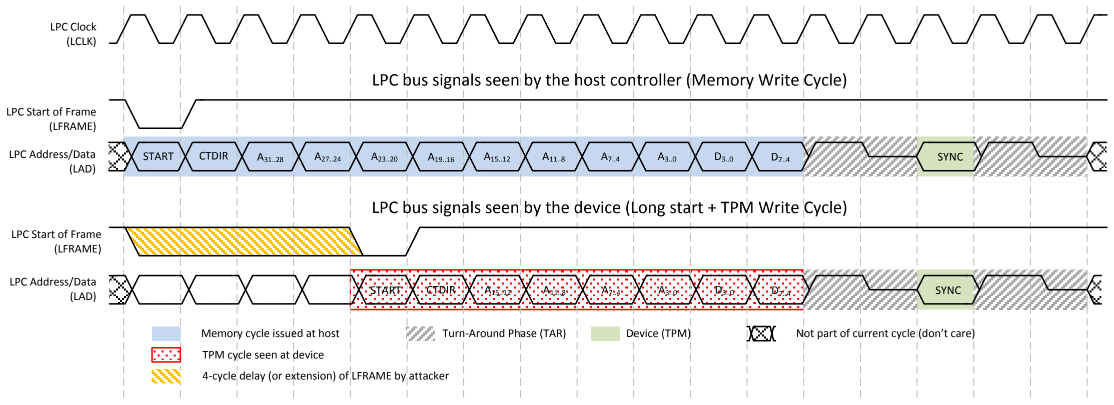
LPC Clock (LCLK)
LPC bus signals seen by the host controller (Memory Write Cycle)
LPC Start of Frame (LFRAME)
LPC Address/Data (LAD)
START
CTDIR
A31..28
A27..24
A23..20
A19..16
A15..12
A11..8
A7..4
A3..0
D3..0
D7..4
SYNC
LPC bus signals seen by the device (Long start + TPM Write Cycle)
LPC Start of Frame (LFRAME)
LPC Address/Data (LAD)
START
CTDIR
A15..12
A11..8
A7..4
A3..0
D3..0
D7..4
SYNC
Memory cycle issued at host
TPM cycle seen at device
4-cycle delay (or extension) of LFRAME by attacker
Turn-Around Phase (TAR)
Device (TPM)
Not part of current cycle (don't care)

Figure 6.4: Hijacking an LPC Memory Write Cycle [WD12]

Our aim here is to design a simple hardware device that is capable of hijacking an ordinary LPC memory write cycle, and promoting it to a TPM write cycles. To do so, our frame hijacker device has to monitor the LPC bus for the start of suitable memory cycles. Once a suitable memory cycle is detected, our device modifies the LPC frame signal seen by the the TPM. By correctly timing assertion of the TPM-side LPC frame signal, we can trick the TPM into detecting the start of the LPC bus cycle at the wrong time, and hijack attacker controlled parts of the memory cycle as TPM cycle. Figure 6.4 shows the results, when the frame signals is delayed by exactly four clock cycles. From the TPM's point of view, the LPC memory cycle shown in Figure 6.4 now looks like a normal TPM cycle. For the platform the entire bus transfer looks like a normal memory cycle.

Here we refer to the LPC bus specification [Int02] that explicitly allows the frame signal to be active for more than one consecutive clock period. The LPC bus specification mandates, that all devices on the bus *must* consider the last value observed on the `LAD` lines, while the frame signal was active, as `START` value. Coincidentally, this is just exactly behavior needed to trick the TPM: Our attack device can simply force the TPM-side frame signal to a low level whenever it detects the start of an LPC bus cycle on the platform side. The frame signal is released, as shown in Figure 6.4, after address bits 20 to 23 (`A5` nibble of the address) of the memory cycle have been transmitted. For the TPM the bus cycle starts now, and the attacker controlled value of the address bits is interpreted by the TPM as `START` field of the hijacked bus cycle.

Therefore, we now have a technique to hijack "harmless" LPC memory cycles, and to turn them into arbitrary TPM cycles. This technique allows the attacker to directly circumvent the locality filtering mechanism in the platform's Southbridge, that normally would prevent generation of arbitrary TPM cycles, with *any* locality. The remaining preliminaries for the attacker are to implement the described hijacking device, to carry out the required hardware modifications to the platform, and to find a way to create suitable memory cycles from software.

6.4 Practical Considerations

There are still some remaining practical considerations that need to be solved, before the the basic cycle hijacking technique outlined in Figure 6.4 can be applied:

1. To drive a logic low level on the `LFRAME` signal, which is an output of the Soutbridge, we must connect the signal to ground. To avoid creating short-circuits, we need to break up the frame signal between the Southbridge and the TPM. We have experimental evidence that a typical PC Southbridge withstands short-circuit condition for short periods of time, however, in the long term permanent damage to its I/O pads is likely.

2. While active, the hijacker device has to intercept all (candidate) memory cycles, in order to be able to cleanly produce the waveforms shown in Figure 6.4. Overriding the `LFRAME` signal for all memory cycles, on the entire bus, would effectively kills most non-TPM communication while the hijacker device is active.

3. Finally, bus-wide manipulation of a signal is easy to detect. It would suffice for a Southbridge, to compare the logic level expected at the output `LFRAME` pad, with the actual logic level observed at the pad.

All three issues can be resolved at once by permanently modifying the target platform: An attacker simply has to cut the LPC frame line along its path between the Southbridge and the TPM. As shown in Figure 6.5 the original LPC frame signal is rerouted through the hijacking device. With this hardware

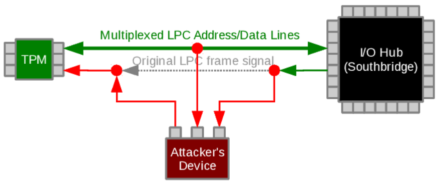

Figure 6.5: Hardware setup (principle) for the LPC frame hijacking attack [WD12]

setup, it becomes possible to *selectively* trigger the frame hijacking device on memory cycles targeted to special trigger memory addresses, without interfering with any other devices on the LPC bus. To detect those special trigger addresses, we monitor the `START`, `CTDIR`, `A7`[1], and optionally the `A6`[2] phases, visible in Figure 6.4.

6.5 Experimental Lab Setup

To validate our frame hijacking attack in hardware, we developed the experimental lab setup shown in Figure 6.6. Our setup consists of two standard FPGA boards, and a TPM daughterboard mounted on a breadboard socket adapter. One of the FPGA boards, and the TPM daughterboard model the platform under attack from Figure 6.4. The second FPGA board implements the cycle hijacking device.

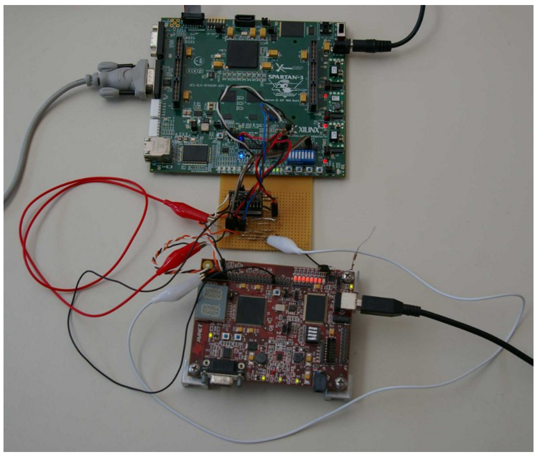

Figure 6.6: Experimental Setup for the LPC Frame Hijacker [WD12]

PC Southbridge Emulator The upper (green) FPGA board in Figure 6.6 is a Xilinx Spartan-3A DSP 1800 Starter Kit that simulates the I/O controller hub (Southbridge) of a typical PC platform. Inside the Spartan-3A FPGA, we synthesized a simple microprocessor system designed around a Xilinx MicroBlaze soft-core processor, with 16K of on-chip RAM. The processor is clocked at 66 MHz, and interfaces to custom LPC bus master core clocked at 33 MHz. The LPC bus master core, used in this experiment supports I/O, memory, and TPM cycles, as well as single-byte firmware reads and writes. It is an earlier version of the core discussed later in Chapter 7. Our LPC bus master consists of in approximately 490 lines of VHDL code. The core is encapsulated in a mostly auto-generated bus wrap-

[1]Address bits 31 to 28
[2]Address bits 27 to 24

per[3] to connect to the the MicroBlaze processor used inside our FPGA design. This wrapper code adds approximately 850 lines of of VHDL source code.

TPM Socket Adapter The center of Figure 6.6 shows the breadboard socket adapter that is required to connect our setup to the actual TPM daughterboard. This adapter board contains a 2x10 pin header to host the TPM daughterboard, a few smaller pin headers to connect to the FPGA boards, and some pull-up resistors. For the experiment we installed a TPM version 1.2 daughterboard from a major vendor on the socket adapter. TPM modules like the one used by us are available as add-on modules for PC motherboards[4] that optionally support a TPM. We note, that our attack works with *any* version 1.2 TPMs implementing an LPC bus interface conforming to the TPM Interface Standard (TIS). We do not rely on any vendor-specific properties of the TPM.

Cycle Hijacker Device The actual LPC cycle hijacker device is implemented inside an FPGA on the smaller (red) board shown at the bottom of Figure 6.6. This board is an older evaluation kit for Xilinx Spartan-3E100 FPGAs[5]. It contains a Xilinx Spartan-3E FPGA, and a Cypress-FX2 microcontroller with USB 2.0 interface. For the setup discussed in this chapter we do *not* use the Cypress-FX2 microcontroller. The USB connector of the board is only used to provide the power supply for the hijacker device.

The inbound LPC clock, frame and data signals are routed from the breadboard adapter to the hijacker device in two groups of smaller wires. The large red wire visible at the left of Figure 6.6 carries the hijacked TPM frame signal to the TPM. The white wire visible in Figure 6.6 connects the signal ground of the hijacker device to the signal ground of the Southbridge simulator..

Hardware Implementation The VHDL source code implementing the LPC frame hijacker device core can be found in Appendix A of this master thesis. By default, our implementation monitors all `LAD` lines of the LPC bus. Our hijacker device can be configured, to work with a reduced number of LPC bus lines. This can be an advantage, if a PC motherboard with a soldered on TPM is being attacked instead of our experimental setup. Reducing the number of probed bus lines adds some additional constraints that are discussed in Appendix A. The minimum configuration of the frame hijacker discussed here can work reliably with only two probed `LAD` lines, in addition to the modified `LFRAME` line.

6.5.1 Software Considerations

To simulate a D-RTM late-launch using the LPC hijacker device the attacker has to generate LPC bus memory write cycles on the platform. This can be achieved by running specially crafted software, which tries to write to memory mapped I/O locations. The physical memory addresses used by these writes can be chosen such that the platform Southbridge translates them to memory cycles on the LPC bus.

We assume that the attacker is able to reconfigure the Southbridge to generate LPC memory cycles, whenever the main processor performs read or writes to the `0xA0000000-0xA0FFFFFF` physical address range. The justification for this, seemingly random, choice of addresses will become clearer

[3]The MicroBlaze processor uses a PLB bus to communicate with its peripherals.
[4]For example for certain ASUS, or GIGABYTE brands
[5]AVnet Spartan 3ES100 Evaluation Kit

in the next paragraphs. The assumption that the attacker may be able to reprogram the Southbridge is justified by the experimental results discussed below.

The LPC cycle hijacker device can transform an attacker controlled part of a specially crafted LPC memory write cycle into an arbitrary TPM write cycle. To find the required physical memory addresses, we reconsider the overlap between memory write cycles and TPM write cycles shown earlier in Figure 6.4. To find the correct physical memory addresses the following four steps are necessary:

1. Our choice of the `0xA0000000–0xA0FFFFFF` physical address range fixes the two highest order address nibbles `A7` to `0xA` and `A6` to `0x0`. This seemingly arbitrary choice is justified by the experimental results in following section.

2. The `A5` address nibble of the hijacked memory cycle corresponds to the LPC `START` field of the piggy-backed bus cycle. The `START` field of TPM cycles must be `0x5`, thus we can fix the `A5` address nibble to this value.

3. The `A4` address nibble corresponds to the LPC `CTDIR` field of the piggy-backed bus cycle. For TPM write cycles this value must by `0x2`, for read cycles it must `0x0`. To simulate the write operations during D-RTM startup we fix `A4` to `0x2`.

4. Finally, the lowest four address nibbles `A3`, `A2`, `A1`, and `A0` of the original memory cycle directly map to the 16-bit address of the piggy-backed TPM cycle.

Following these four steps the `0xA0520000-0xA052FFFF` physical address range is for memory write cycles, and thus for the piggy-backed TPM write cycles. The read address range for memory cycles, and their piggy-backed TPM read cycles, can be found in the `0xA0500000-0xA050FFFF` physical address range using the same procedure. To obtain the final memory address for accessing TPM register it is sufficient to add `0xA0520000` for writing, or `0xA0500000` for reading, to the 16-bit TPM register address found in the TPM TIS specification.

The crucial point here is that an attacker can freely choose all 16 bits of the TPM registers address, which consists of 4 bits indicating the locality, and 12 bits indicating the actual register offset.

Validation using the Southbridge Simulator To validate the functionality of the frame hijacker device, and to verify that the physical addresses found above are correct, we implemented a simple D-RTM startup simulator program on our Southbridge emulator device.

The small program in Listing 6.1 simulates the TPM register writes during early stages of a D-RTM startup that we discussed above in Section 6.2. We note that our program runs on the main processor of the platform, and uses the LPC frame hijacker, to generate bus cycles that normally can only be generated by trusted CPU microcode.

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define LOC4_WR_BASE 0xA0524000
static char pcr17_data[] = { ... };

int main(int argc, char **argv) {
   int fd = open("/dev/mem", O_RDWR);
```

```c
  char *tpm = mmap(0, 4096, PROT_READ|PROT_WRITE,
       MAP_SHARED, fd, LOC4_WR_BASE);
  char *src = pcr17_data;
  unsigned size = sizeof(pcr17_data);

  tpm[0x28] = 0x00; // TPM_HASH_START
  while (size--) {
    tpm[0x24] = *src++;  // TPM_HASH_DATA
  }
  tpm[0x20] = 0x00; // TPM_HASH_END
  return 0;
}
```

Listing 6.1: Proof-of-concept D-RTM Startup Simulator Code [WD12]

The `LOC4_WR_BASE` constant in the code snippet was computed by adding the base address (`0x4000`) of the locality 4 TPM register range to memory mapped I/O ”write” address (`0xA052000`) that we found earlier. The offsets to access the `TPM_HASH_START`, `TPM_HASH_DATA`, and `TPM_HASH_END` TPM registers via the memory mapped `tpm` base pointer were obtained from the values given in Figure 6.2. The sequence of register writes generated by our program corresponds to the sequence illustrated earlier in Figure 6.1.

6.5.2 Validation on a Desktop PC

The initial validation of the LPC frame hijacker attack was done with the experimental setup discussed ealier, using the Southbridge emulator. In order to validate our attack on a real PC platform, we attached the LPC bus hijacker device to an Intel TXT enabled HP Compaq DC7900 desktop PC. The mainboard of this computer is designed around an Intel ICH10 Southbridge. To avoid permanent modifications to this computer, which otherwise was used as normal office workstation, we connected the hijacker device in a passive “sniff-only” setup, without cutting any circuit board traces.

To observe the inputs and outputs of the LPC hijacker device we integrated a logic analyzer core[6] into the FPGA containing the hijacker device. Using this integrated logic analyzer core we were able to compare the waveforms observed during tests with the Southbridge emulator to the waveforms generated on the TXT capable PC platform.

Reconfiguring the ICH10 Southbridge When deriving the physical addresses for use be the hijacker device earlier we selected the `0xA0000000-0xA0FFFFFF` physical address range without further justification. The choice of this particular address range depends on the chipset of the desktop PC used to validate our setup.

We note that the ICH10 Southbridge found in our target PC contains a programmable decode area for LPC memory cycles, and that this area is *not* locked down by the BIOS. The decode address area for LPC memory cycles can be configured via the `LGMR` register of the ICH10 Southbridge (cf. [Int08, Ch. 13.1.27]). This register is accessible via PCI device index 31, function 0 at offsets `0x98-0x9B`, and can be configured via the Linux `setpci` program from the `pciutils` package.

To reprogram the `LGMR` register with the address range, that we obtained above for TPM write cycles it is sufficient to run the `setpci` command as root user:

[6]Xilinx ChipScope Integrated Logic Analyzer (ILA)

```
# setpci -s 00:1f.0 0x98.L=0xA0520001
```

Once the register has been configured, the program from Listing 6.1 can be used to generate specially crafted memory cycles on the LPC bus. Our LPC hijacker device can then translated these memory cycles, into arbitrary TPM write cycles. This in turn allows us to simulate the start of a dynamic root of trust for measurement, without ever being in a trusted software state.

To generate TPM read cycles, we have to program a slightly different value range into the `LGMR` register:

```
# setpci -s 00:1f.0 0x98.L=0xA0500001
```

6.6 Using LPC Firmware Cycles Instead of Memory Cycles

The success of the frame hijacking attack crucially depends on an adversary's ability to generate long LPC bus cycles which can be used to piggyback the faked TPM cycles. On the Intel ICH10 based mainboard , it is particularly easy to generate suitable LPC memory cycles due to the freely accessible, programmable memory decode window of the LPC host controller. On other motherboards, or chip-sets we have to resort to alternative means of generating the required cycles.

One alternative solution to hijacking LPC memory cycles is to use LPC firmware cycles. Firmware cycles are intended to access LPC flash memory chips, which containing the BIOS or option ROMs, during platform boot. This type of bus cycles slightly differs from memory cycles in the used addressing scheme, and the support for different data transfer sizes.

From the point of an attacker, firmware cycles are usable, but are often an inferior to LPC memory cycles. The combination of data transfer sizes supported by the platform, and the memory mapped I/O ranges that are mapped to firmware cycles often imposes severe restrictions on the TPM register reachable via frame hijacking. The address phase of a typical LPC firmware cycle is followed by an extra field indicating the size of the transfer. For one byte firmware cycles the size field is fixed to the value `0x0`. Combined with restriction that can present on the address fields these limitations can lead to situations, where only TPM registers at offset evenly divisible by 16 are accessible via LPC frame hijacking. Another severe difficulty with firmware cycles is that Southbridges usually block firmware write cycles, in order to prevent unauthorized re-flashing of the platform BIOS. To generate usable firmware write cycles, an adversary first has to circumvent to BIOS re-flash protection mechanism.

The LPC bus specification defines firmware read and write cycles with different sizes ranging from single byte access up to 128-byte blocks. Apart from mandatory support for single byte cycles all other sizes are optional. We want to point out that firmware write cycles with data sizes larger than one byte can be hijacked as TPM cycles without suffering the modulo 16 TPM address constraint discussed above. We have successfully verified the use of hijacked LPC firmware cycles instead of memory cycles on our Southbridge simulator. Best results, in terms of reachable TPM TIS registers were obtained with 2-byte, and 16-byte firmware write cycles. Worst results were obtained with 1-byte firmware write cycles. Actually the case of a hijacked 16-byte firmware write cycle was our initial idea and implementation of the frame hijacking technique, before we even considered the LPC memory cycle based method outlined in this chapter.

Chapter 7

LPC Bus Emulation

"All things truly wicked start from an innocence."

[Ernest Hemingway, A Moveable Feast]

Experiments with Trusted Platform Modules are often complicated by security properties of the Trusted Platforms. In particular TPM localites (cf. Section 3.2), and the lack of a possibility to reset most of the PCRs (cf. Section 2.6.3) complicate things in practice. From a security perspective these two features are essential for Trusted Platforms, as they are both related to the integrity of the chain of trust.

In the previous chapter we introduced an active bus modification attack that manipulates bus signals of a platform under attack, to work around limitations related to the TPM locality mechanism. To validate our attack, we developed an FPGA-based model of the Trusted Platform that includes a simple LPC bus host controller. In this chapter we present a *standalone* LPC Bus Emulator that reuses parts of the simple LPC host controller from our FPGA-based platform model from Section 6.5. Our LPC Bus Emulator was originally developed by me as part of joint work with Pirker et la. [PWT12] to connect standard TPMs with LPC interface to embedded platforms that do not provide an LPC bus natively.

7.1 Emulating an LPC Bus Host Controller

When we started our joint work with Pirker et al. [PWT12] on Android-based Trusted Platforms, we quickly encountered two major problems: First, our target platform, a development board for ARM-based Freescale i.MX51 processors, did not natively provide the LPC bus interface required to connect a standard LPC-based TPM. Second, no embedded v1.2 TPMs that satisfied our requirements were available at the time we started to work on [PWT12].

To resolve these problem, we decided to built upon our previous results on LPC frame hijacking attacks, and to refactor the Southbridge simulator discussed in Section 6.5 into a general purpose LPC bus emulator.

Figure 7.1 gives an overview of the system architecture that we implemented in Pirker et al. [PWT12]. Due to its modular design, no major changes were needed to extract the LPC bus master core from the

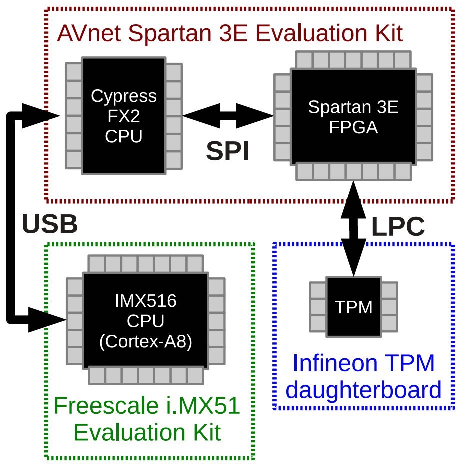
AVnet Spartan 3E Evaluation Kit
Cypress FX2 CPU
Spartan 3E FPGA
SPI
USB
LPC
IMX516 CPU (Cortex-A8)
TPM
Infineon TPM daughterboard
Freescale i.MX51 Evaluation Kit

Figure 7.1: Embedded Platform Setup used in [PWT12]

Southbridge simulator that we discussed in the previous chapter, in Section 6.5. The resulting implementation of the corresponding VHDL entity can be found in the appendix of this master thesis in Listing B.1, and is closely based on the earlier version that we used in Section 6.5.

To keep the overall LPC bus emulator design simple, we replaced the soft-core processor used to control bus activity in Section 6.5 by a small finite state machine that accepts control commands via a serial peripheral bus interface. The complete implementation of this serial peripheral interface (SPI), including the finite state machine and control logic for the LPC bus controller can be found in Listing B.2 in the appendix.

This decision allowed us to keep the FPGA implementation small, and to fit the entire bus emulator on the same type of Spartan-3E FPGA evaluation kit that we already used in Section 6.5. The resulting FPGA design, and one of our major contributions to Figure 7.2, is the standalone LPC bus emulator shown in Figure 7.2.

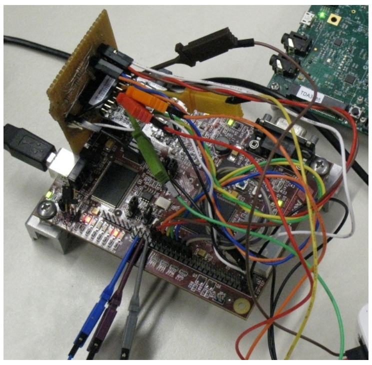

Figure 7.2: LPC Bus Emulator Hardware Setup [PWT12]

For our experiments in Pirker et al. [PWT12] we ended up with the system architecture shown in Figure 7.1. In this architecture the embedded ARM board uses a standard USB interface to communicate with a Cypress FX2 microcontroller on the used FPGA evaluation board. Additionally, the USB port provides the power supply for the interface logic and the TPM. The Cypress FX2 microcontroller communicates via an SPI bus with the FPGA that contains the actual LPC bus interface logic. The Trusted Platform Module connects directly to the LPC bus provided by the FPGA.

In principle, we can simplify the architecture from Figure 7.1, by directly connecting one of the SPI controllers of our embedded ARM board to the FPGA. This would avoid the intermediate hop over the USB bus, and the Cypress FX2 controller of the FPGA evaluation board. However, using the USB interface to connect to the LPC bus emulator, and the TPM attached to it, has the practical advantage that the LPC bus emulator can be used with desktop computers, which commonly do not have externally accessible SPI busses, too.

7.2 Implementation Details

The SPI to LPC bridge, which forms the core of our LPC bus emulator, is designed as synchronous digital logic that is clocked by the 33 MHz LPC bus clock. The implementation that we used in Pirker et al. [PWT12] uses a 48 MHz signal provided generated by the microcontroller on the FPGA evaluation board, and a digital clock management block of the FPGA to synthesize The SPI to LPC bridge includes serial peripheral slave interface (SPI) for command and data input from an external microcontroller, as well as an LPC bus master interface for communication with LPC bus devices such as the TPM.

The LPC bus master interface of our bridge only implements the bare minimum LPC bus signals (`LRESET`, `LFRAME`, `LCLK`, `LAD0-3`) that are required to operate an LPC bus. This minimum set of signals is sufficient to communicate with a TPM. Interrupts, power management, and clock management facilities are currently not implemented are currently not implemented.

The complete VHDL implementation of the control interface, including the instantiation of the LPC bus master core, can be found in Appendix B (see Listing B.2).

Handling of LPC Aborts Our bus bridge automatically detects if no LPC slave is responding to a transaction and ensures proper termination of LPC bus cycle. Long LPC synchronization cycles are not automatically terminated to simplify debugging of hardware issues, such as stuck LPC slaves, during experiments. The SPI control interface provides a commands to manually abort a stuck transfer on the LPC side. This design decision significantly increases the robustness of the LPC bus bridge against “strange” activity on the LPC bus, such as sudden appearance and removal of a TPM.

SPI Hardware Interface In its current version the bridge can be attached to any microcontroller with an serial peripheral interface (SPI) compatible bus interface. The clock polarity and phase of the SPI slave are configured at synthesis time using the VHDL generics, the default configuration is SPI Mode 0 (CPOL=0, CPHA=0).

At the moment the core can be configured for SPI Mode 0 (CPOL=0, CPHA=0) or SPI Mode 1 (CPOL=1, CPHA=0). The other two SPI modes (with CPHA=1) are not supported.

The SPI MISO (master-in slave-out) port can be configured as three-state port in the FPGA top-level design, to allow use of multiple SPI slaves on the same bus.

Command Interface

The command interface of the SPI to LPC bus bridge is based on simple packets that are sent by an external microcontroller via the SPI slave interface to the bridge. The first byte of a packet always indicates the command type. All following bytes are command specific data.

Our bridge requires the external microcontroller, to assert the SPI slave select line before sending commands, and to release the SPI slave select line between any two consecutive commands send over the control interface. This requirement ensures that the bridge can recover from errors, when receiving unimplemented, or corrupted commands.

The following paragraphs discuss all commands that we implemented for the version of the SPI to LPC bus bridge found in Listing B.2 the appendix.

0xC0 - Read Device ID This command checks connectivity and presence of the bridge device on the SPI bus. After receiving the command byte the core starts to repeatedly send the magic value `0x2A` to the SPI master, until the SPI transfer is terminated by releasing the slave select line.

0xC1 - Read Control/Status Byte Using this command, the SPI master can read the current status of the LPC bus master core. The status response generated by this command indicates if the LPC bus is currently busy, or if a new transaction can be started safely. Figure 7.3 shows the format of the control/status byte.

7..4	3	2	1	0
SYNC	0	ABT	0	BUSY

```
SYNC: The raw LPC "SYNC" value of the last bus transfer
ABT:  LPC core is in abort mode (aborting a transfer)
BUSY: LPC core is busy (doing a transfer)
```

Figure 7.3: Format of LPC Bus Emulator Control/Status Byte

After receiving the command byte, the core repeatedly samples the internal status and sends status bytes until the SPI transfer is terminated by releasing the slave select line.

0xC2 - Write Control/Status Byte This command modifies the control/status register of the core. The value to be written immediately follows the command byte in the SPI transfer.

The layout of the core's control/status byte is shown in Figure 7.3. Writes to the `SYNC` and `BUSY` status fields are silently ignored.

Currently the only modifiable bit of the control register is the abort flag (`ABT`). This flags can be set by the SPI master to terminate any pending LPC transactions. It must be cleared explicitly before the core accepts any new requests to start LPC transfers.

0xC3 - Read LPC Data Byte (Lazy Read) The lazy read command allows the SPI master to lazily read the data byte that has been returned by the most recent LPC read transaction. See the LPC read commands below for more details on its uses.

0xD1, 0xD2, 0xD4, 0xD8 - LPC IO/Memory/TPM/Firmware Read Transaction The LPC read transaction commands initiate read cycles on the LPC bus. The type of LPC cycle is indicated by the lower 4 bits of the command byte, the bit assignment is given in Table 7.1

Code	Cycle Type
`0x1`	I/O Read
`0x2`	Memory Read
`0x4`	TPM Read
`0x8`	Firmware Read (1-byte)

Table 7.1: Cycle Types supported by the LPC Bus Emulator

The command byte is immediately followed by the 16- or 32-bit target address in big-endian byte order. I/O (0xD1) and TPM (0xD4) read commands expect a 16-bit address, while memory (0xD2), and firmware (0xD8) read commands require a 32-bit address.

An LPC transaction is started after the last address byte has been successfully transmitted over the SPI control interface. The SPI master can release the slave select line and use the *Read Control/Status Byte* (0xC1), and *Read LPC Data Byte* (0xC3) commands to fetch the results later.

Alternatively the SPI master may continue the SPI transfer, to poll for completion of the LPC cycle. The bus bridge starts to repeatedly send the status byte (at least once) until the LPC cycle has completed. When the LPC cycle is done, the core repeatedly sends the LPC data register content. The following code snippet in Listing 7.1 shows an excerpt from our test firmware, making use of this feature.

```c
...
// I/O read from address 0x1234
spi_select_slave();
spi_send_byte(0xD1); // Assume I/O read
spi_send_byte(0x12); // Upper address nibble
spi_send_byte(0x23); // Lower address nibble

unsigned char status;
do {
  // Read status until the core is idle
  status = spi_recv_byte();
} while (status & 0x1);

// Read the data byte
unsigned char data = spi_recv_byte();

// Deselect the slave
spi_deselect_slave();
...
```

Listing 7.1: Using to LPC Bus Emulator to Read Data

0xE1, 0xE2, 0xE4, 0xE8 - LPC IO/Memory/TPM/Firmware Write Transaction The LPC write transaction commands start write cycles on the LPC bus. Similar to the read commands, the type of LPC cycle is indicated by the lower 4 bits of the command byte. The available types are shown in Table 7.1.

The command byte is immediately followed by the 16- or 32-bit target address in big-endian byte order. I/O (0xE1) and TPM (0xE4) write commands expect a 16-bit address, while memory (0xD2), and firmware (0xD8) write commands require a 32-bit address. The data byte to be written immediately follows the last address byte.

The LPC bus cycle is started after the data byte has been successfully transmitted. Similar to the read command, the SPI master can release the slave select line and use the *Read Control/Status Byte* (0xC1) command to detect completion of the LPC bus transaction.

```c
...
// I/O write value 0xCA to address 0x1234
spi_select_slave();
spi_send_byte(0xE1); // Assume I/O write
spi_send_byte(0x12); // Upper address nibble
spi_send_byte(0x23); // Lower address nibble
spi_send_byte(0xCA); // SPI data byte

unsigned char status;
do {
  // Read status until the core is idle
  status = spi_recv_byte();
} while (status & 0x1);

// Deselect the slave
spi_deselect_slave();
...
```

Listing 7.2: Using to LPC Bus Emulator to Write Data

Alternatively the SPI master can continue the SPI transfer, to poll for completion. Listing 7.2 shows corresponding example code from our test firmware.

Chapter 8

Beyond the Platform Reset Attack

"The more corrupt the state, the more numerous the laws."

[Tacitus, The Annals of Imperial Rome]

In Chapter 5, and Chapter 6 we discussed simple hardware attacks against trusted platforms that required only small modifications to the target hardware. We, at most, had to break up two PCB traces on the motherboard, to gain access to the LPC bus frame and reset signals. The previous chapter introduced an LPC bus emulator that we successfully used in [PWT12], to provide TPM connectivity for embedded systems. In this chapter we first extend our results from Chapter 5, and Chapter 6, on manipulation of the LPC bus frame signal. Next we discuss, how the LPC bus emulator from Chapter 7 can be used to construct arbitrary measurement chains without involving a trusted platform at all. Finally we relax our assumption on the attacker, and allow physical removal removal of the TPM from the motherboard, and manipulation of all LPC bus signals.

Based on these preliminaries we show at the end of this chapter, how a TPM can be physically moved from a specially modified running trusted platform, to an untrusted platform under control of the attacker. The state of the TPM will be preserved during this physical transfer.

8.1 Setting

The setting for the attacks in this chapter is an extension of the setting of the attacks in Chapter 5, and Chapter 6. We assume that the attacker has unrestricted physical access to the platform, and is capable of doing simple hardware modifications. These simple hardware modification now include physical removal of the TPM, and *arbitrary* manipulation of *all* LPC bus signals between the TPM, and the platform. The attacker in this chapter can act as a man-in-the-middle between the TPM, and the platform.

The ultimate goal of the attacker in this chapter is to fool any users of the TPM, to believe that the platform is in an *arbitrary* trusted state, whereas really the attacker has full control over the *platformn and the state* that is reported by the TPM.

Thus security is breached, because the attacker will be able to decouple the actual platform state from the state recorded by the TPM at any time, and moreover will be able to construct arbitrary fake measurements chains using the TPM of the platform under attack.

8.2 LPC Frame Suppression

Our variant of the platform reset attack from Chapter 5 manipulates the LPC bus framing signal, to simulate a long LPC start frame at the TPM-side, and this way prevents the platform BIOS from detecting the TPM after reboot. The technique discussed in Section 5.3.5 relies on an attacker controlled switch to force a logic low level of the LPC framing signal.

An alternative approach is to force the LPC framing signal seen at the TPM-side, to a constant logic high level. Similarly to the technique in Section 5.3.5 communication attempts between the TPM and the remaining platform are disrupted by this manipulation. The difference here is that the TPM now sees a long bus idle phase, instead of a long start phase, on the LPC bus.

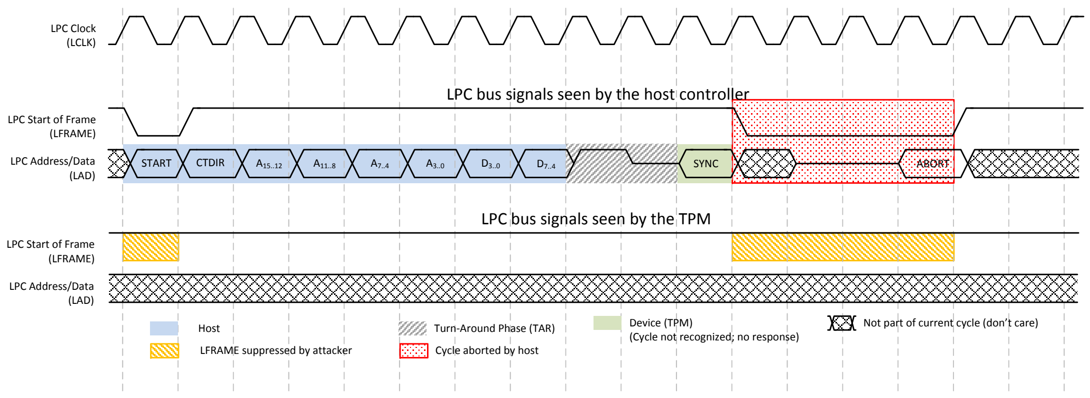
LPC Clock (LCLK)
LPC bus signals seen by the host controller
LPC Start of Frame (LFRAME)
LPC Address/Data (LAD)
START
CTDIR
$A_{15..12}$
$A_{11..8}$
$A_{7..4}$
$A_{3..0}$
$D_{3..0}$
$D_{7..4}$
SYNC
ABORT
LPC bus signals seen by the TPM
LPC Start of Frame (LFRAME)
LPC Address/Data (LAD)
Host
LFRAME suppressed by attacker
Turn-Around Phase (TAR)
Cycle aborted by host
Device (TPM) (Cycle not recognized; no response)
Not part of current cycle (don't care)

Figure 8.1: LPC Frame Suppression

For the host controller (Southbridge), no TPM appears to be present on the bus. As discussed in Section 4.2.2 host to device synchronization fails, and in response the LPC host controller aborts the bus cycle. Software, and drivers usually see resulting failed read cycles as all-one reads, while failing write cycles are typically ignored. In principle, the platform chipset could generate interrupts, to notify driver software of bus error. In practice we found that current Linux kernels do not use such fault reporting mechanisms, and, in this respect, behave attacker-friendly.

Use of a bus idle phase instead of a long start phase to isolate the TPM can be advantageous for the attacker, since the risk of bus data being misinterpreted by the TPM as start of new cycles is minimized. To cleanly isolate the TPM and suppress further communication it thus suffices to first wait until the bus enters an idle state, and then to force the TPM-side LPC bus framing signal to a constant high level. Figure 8.1 shows the resulting waveforms that are generated by the LPC host controller (top), and that are seen by the TPM (bottom).

8.2.1 Conditional Cycle Dropping

Simulated bus idle phases can be use similar to what we did in Section 5.3.4, for isolating the TPM from the platform. An advantage of the idle phase method discussed here over method given in Section 5.3.5

is that it never generates spurious cycle aborts on the TPM-side, as it never forces the LPC frame signal to logic low level.

We can easily modify the frame hijacking technique from Chapter 6 technique, to selectively drop individual read or write operations of TPM registers. This in turn can disturb the normal flow of TPM communication via the TPM TIS protocol.

One possible attack strategy is to selectively block execution of TPM commands, by dropping writes to the `STS` register (see Section 3.3.2) register, after the last command byte has been received. To implement this attack strategy it suffices to monitor reads from the `STS` register, and to isolate the TPM once the last command byte has been received.

Another attack strategy is to selectively prevent software from reading reading TPM responses after command execution has completed. To implement this strategy the attacker can monitor reads from the `STS` register, and isolate the TPM once the `STS` register indicates that command execution has completed.

Both of these attack strategies can be exploited to attack the error handling paths in trusted software, by selectively disrupting TPM command execution at critical points. Security is breached, if trusted software can be tricked into ceasing to function normally, because of an assumed TPM hardware failure.

8.3 Synthesis of Arbitrary Measurement Chains

One simple, yet effective, method to attack trusted platforms is to synthesize is to synthesize arbitrary measurement chains. The hardware attacks discussed in earlier chapters try to achieve this, by targeting a TPM that is installed within a trusted platform: Reset attacks, as discussed in Chapter 5, do so by manipulating the LPC bus reset signal, and in some cases the LPC bus framing signal. The LPC bus cycle hijacking attack, discussed in Chapter 6, exploits a combination of hardware and software manipulation to piggy-back fake TPM bus cycles onto harmless memory bus cycles. Common to these attacks is, that they circumvent platform protection features, in order to privileged access to the TPM, thus reaching their ultimate goal manipulating an existing measurement chain.

An LPC bus emulator, as the one described in Chapter 7, allows attackers to address the problem from the opposite direction: Instead of manipulating an existing measurement chain, an attacker can construct a completely new measurement chain from scratch. This can even be done with the TPM of the trusted platform, if the attacker removes the TPM.

The underlying major problem is that no *verifiable and tamper-proof binding* exists between *all* TPMs and their host platforms. Without a *verifiable binding*, an attacker can easily obtain an arbitrary, genuine TPM, and use an LPC bus emulator to synthesize the arbitrary measurements chain from scratch. Without a *tamper-proof* binding, the attacker can try to remove the TPM from the platform, and install it in the LPC bus emulator.

8.4 Physical TPM Transfer

The platform reset attack from Chapter 5 depends on the attacker's ability to isolate the TPM. To succeed, it is crucial for to shield the TPM from both, the platform reset signal (cf. Section 5.3.2), and the LPC framing signal (cf. Section 5.3.5). To achieve this isolation, the attacker typically cuts the LPC bus `LFRAME`, and `LRESET` signals close to the TPM, and then install *ad-hoc* attack circuitry, to act as man-in-the-middle.

The man-in-the middle idea can be extended to all LPC bus signals. This raises the question, if it is possible to isolate the entire TPM, maybe even the power supply, without losing the current state of the TPM.

Assuming that this physical *hot* removal of the TPM from the platform is possible, the next question is if the TPM can be successfully attached to a platform of the attacker's choice, such as the LPC bus emulator from Chapter 7. Ideally, the state of the TPM will be preserved during this process.

Assuming that it is feasible, this process constitutes a very powerful attack. The attacker needs physical access to the victim platform, and the knowledge required to boot the platform into a desired trusted state. Unhindered physical access allows the attacker to perform modifications to the victim platform. With necessary knowledge about the trusted state the attacker then can boot the physically modified platform with unmodified, thus trusted, software.

After the platform has booted into the desired trusted state, the attacker physically removes the TPM from the victim platform. At this point, software defense mechanisms on the victim platform that try to record the break-in attempt in the TPM are rendered useless. In the next step, the attacker simply has to connect the TPM to a rouge platform, or LPC bus emulator, of his choice. The net effect of this physical TPM transfer is that the TPM-based identity, and TPM-visible state of the trusted victim platform is transferred to rouge attacker-controlled platform.

The attack idea outlined above assumes, that it actually *is* possible to perform the TPM transfer at runtime. Results on the other attacks, which we discussed earlier, indicate that this *may* be possible.

8.4.1 Experimental Results

To validate that our attack idea is feasible in practice, we devised a series of experiments on an older HP Compaq nx6325 notebook. This notebook is powered by an AMD Turion X64 X2 that supports AMD's D-RTM implementation, and contains a v1.2 TPM.

After obtaining the platform, our first step was to locate the TPM on the mainboard. TPM chips, which follow the recommended packaging and pinout given in [TCG05], are relatively easy to find on printed circuit boards. Their chips packages have a distinct rectangular shape, with 28 fine-pitched pins in two columns along the longer sides of the package.

The TPM of our test notebook is hidden below the integrated memory card reader of the notebook. Figure 8.2 shows the TPM, after the memory card reader has been desoldered with a hot-air gun. We assume that the TPM was placed below the memory card reader to simplify board routing, and to reduce manufacturing costs. We note that it was *not* glued to the mainboard, thus making removal relatively easy.

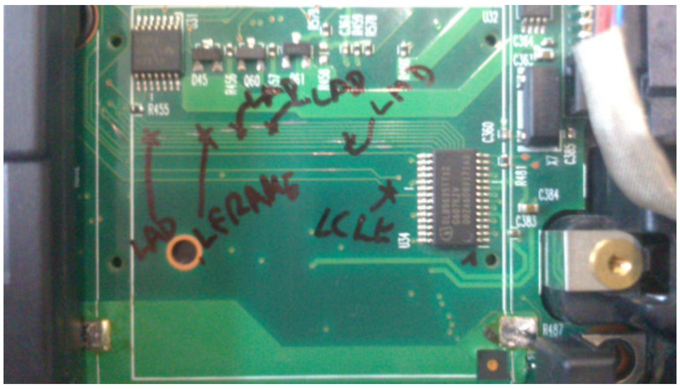

Figure 8.2: TPM of an HP Compaq nx6325 Notebook

Passive Bus Probing During the next experiment, we tried to passively monitor LPC bus activity, and TPM communication on the notebook platform. For passive bus monitoring we used a digital oscilloscope[1], to help identifying the bus signals, and later a low-cost USB logic analyzer[2], to capture bus traffic.

In Figure 8.2 we already identified and labeled the corresponding board traces. To be able to solder-on probe wires, we first tried to scratch off the insulation of the relevant copper traces on the board. This simple technique turned out to work quite well, while only requiring moderate soldering skills.

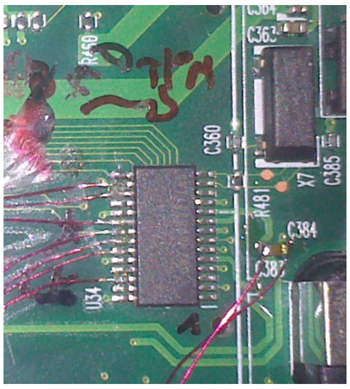

Figure 8.3: TPM with Probe Wires

Figure 8.3 shows an alternative method to probe LPC bus signals, by directly soldering the probe wires to the TPM pins. The main advantage of this second method is that the pin assignment of the TPM is well known, and that no guess-work is necessary to identify the relevant board traces. Moreover, the probe wires can be easily removed after the experiment, respectively the attack, to conceal visible traces.

Attack Procedure After completing the bus probing experiment discussed above, we removed the original TPM of the notebook and routed the relevant LPC bus signals to the outside of the notebook case. Combined with the breadboard socket adapter from Chapter 6 this allows us to connect a standard off-the-shelf TPM daughterboard to our test platform. Figure 8.4 shows the resulting final hardware setup.

[1]Tektronix TDS 2104
[2]ZeroPlus LAP-C (16032)

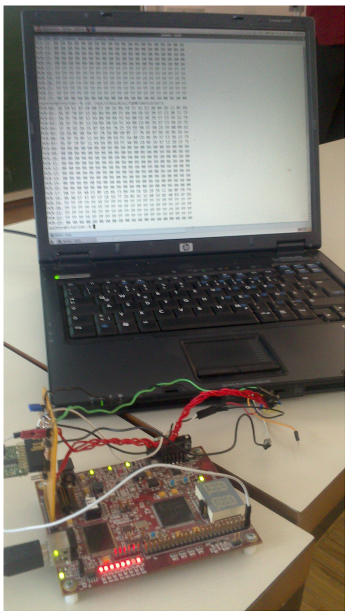

Figure 8.4: Physical TPM Transfer Hardware Setup

In front of Figure 8.4 the LPC bus emulator from Chapter 7 is already setup to validate the physical TPM transfer attack. We initially expected complications with the LPC bus clock signal during the transfer, and planned on using a clock multiplexer cell[3] to switch between the notebook LPC clock, and the LPC bus emulator clock. To our surprise, we found switching clocks by manually disconnecting the clock wire from the notebook side, and reconnecting it the the LPC bus emulator to work very well.

Thus, we ended with the following simple 9 step procedure for physical TPM transfer, between a prepared “trusted” platform, and our “untrusted” LPC bus emulator:

1. Prepare the victim platform, remove its original TPM, and install LPC bus probe wires.

2. Connect an adapter board, which holds the TPM, to the probe wires. The adapter should provide a pull-up resistors for the `LFRAME` and `LRESET` signals. For correct TPM operation we additionally need the `LCLK`, and `LAD0` to `LAD3` signals. All unneeded signals should be set the appropriate logic levels, via pull-up and pull-down resistors on the adapter board.

3. Setup the LPC bus emulator, and connect its ground signal to the adapter board.

4. Power-on the victim platform, and boot it into the desired trusted state. The platform will not notice any difference to a normal boot.

5. Disconnect the `LRESET` signal from the victim platform, and attach it to the LPC bus emulator. The pull-up resistor on the adapter board ensures that the reset signalis kept in its inactive (high) level during the transfer.

6. Disconnect the `LFRAME` signal from the victim platform, and attach it to the LPC bus emulator. Similar to `LRESET` the pull-up resistor on the adapter board maintains the signal in its inactive

[3]e.g. Xilinx `BUFGMUX`

(high) level during transfer. After this step, the TPM is isolated from the victim platform, and no further communication can take place.

7. Disconnect the `LAD0` to `LAD3` signals of the TPM from the victim platform, and attach them to the LPC bus emulator. After this step all control and data signals have been transferred from the host to the emulator. The only remaining signal is the clock signal, that will be addressed next.

8. Disconnect the `LCLK` signal from the victim platform, and attach it to the LPC bus emulator. In principle the LPC bus specification allows platforms to stop their clocks without notifying devices. This is exactly what happens during the transfer, and is tolerated by the TPM modules we tested.

9. Use the LPC bus emulator to communicate with the transferred TPM, which still holds the *trusted* state of the victim platform.

For the outlined attack procedure we assumed that the victim platform remains powered on, even after the TPM transfer, and only connected the ground signal in step 3. It would also be possible, to power-off the victim platform after the transfer, if the power-supply for the TPM is provided by the LPC bus emulator in step 3.

The order of transferring `LRESET` and `LFRAME` is arbitrary. We chose to transfer `LRESET` first, in case that the victim platform resets later during the procedure, either due to an accident[4], or to potential future countermeasures.

Impact Successfully mounting a TPM transfer attack allows the attacker to take a snapshot of trusted victim platform, consisting of TPM-based identity and state, and to prove to verifiers, that the trusted platform's state is his own state.

This attack effectively makes the TPM-based identity of the attacker, and his platform, indistinguishable from the trusted victim platform. Conceptually this attack can be seen as TPM variant of identity theft. For remote attestation verifiers have to trust in the physical integrity of all prover platforms. Thus security is breached for remote attestation, because we created a situation where this trust assumptions is no longer not justified.

The impact of this attack on local attestation (sealing) strongly depends on the use case. For scenarios where local users with physical access to the platform may have an incentive to attack their own platforms, for example in case of DRM applications, the impact is similar to the impact on remote attestation. When local users have no incentive to attack their own platforms this attack may appear to not be that much of an issue. However, this is only true, if device theft by an attacker can be ruled out. Otherwise, when attackers may steal the platforms of legitimate users, the situation becomes similar to DRM, with the user seeking protection of her assets, and the attacker trying to break that protection.

[4]e.g. Attacker accidentally causes a short circuit ...

Chapter 9

Outlook

"The past gives you an identity and the future holds the promise of salvation, of fulfillment in whatever form. Both are illusions."

[Eckhart Tolle, The Power of Now]

9.1 Next Generation TPMs

Since we started our work on simple TPM hardware attacks, a new major revision of the TPM specification (TPM v2.0) has been published. In a joint paper with Pirker [PW13] we discussed how a working TPM 2.0 emulator can be extracted from the source code fragments found in the TCG TPM 2.0 specifications.

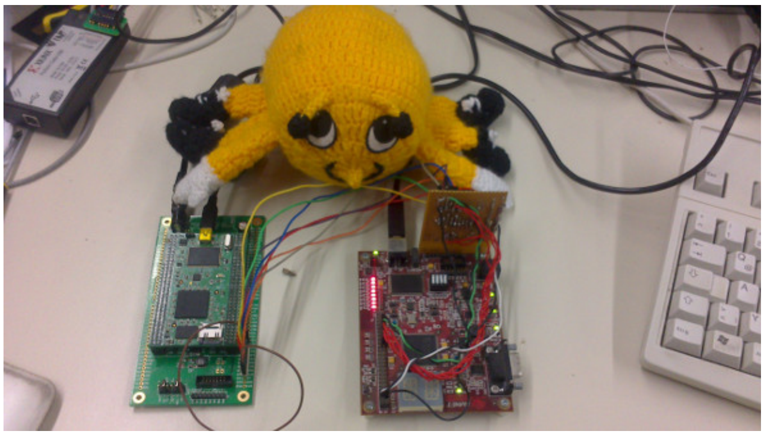

Figure 9.1: Prototype TPM v2.0 In-System Emulation

As part of our joint work we developed a prototype emulator for TPM v2.0 that we presented in [PW13]. Our prototype emulator is intended to work with desktop, and embedded platforms. Therefore, it provides a variety of external interfaces, including a TIS 1.3 compatible LPC bus interface, an I2C bus interface, and an Ethernet interface for communication between the host platform and the emulated TPM.

9.1.1 Preliminary Results

Due to the significant changes in design principles, and support commands, between TPM v1.2 and TPM v2.0 it is not immediately clear if all of the attacks discussed earlier are applicable to TPM 2.0. We intuitively expect some of these attacks to become less relevant, due to changes in the semantics of TPM v2.0 primitives, while other new types of attacks will appear.

As a basis for future work in these areas, we started to verify bus manipulation methods introduced in this master thesis, against our TPM v2.0 emulator. Figure 9.1 shows a simplified version of the emulator prototype from [PW13], which only includes an LPC bus interface, used during our initial experiments.

The test setup in Figure 9.1 consists of the our simplified TPM v2.0 emulator, implemented in the left (green) FPGA board, and the LPC bus emulator discussed in Chapter 7, implemented on the right (red) FPGA board.

First results obtained with the shown test setup suggest[1] that the bus modification attacks discussed in this master thesis are applicable to TPM v2.0 based platforms, which use an LPC bus.

9.1.2 TIS 1.3 SPI Protocol

As part of TPM v2.0 specification, the Trusted Computing Group updated the TPM TIS interface specification. The updated TPM TIS 1.3 specification adds a standardized SPI bus binding, in addition to the existing LPC bus binding.

As basis for future work, we started to analyze the differences between the SPI bus protocol used by our LPC bus emulator from Chapter 7, and the TPM TIS 1.3 SPI protocol. Based on preliminary results from this analysis, we expect that modification of our LPC bus emulator to the TPM TIS 1.3 SPI protocol should be possible without major issues.

The outcome of such a modification would be an adapter that enables host platforms with a TPM TIS 1.3 SPI interface to communicate with LPC bus based TPMs. Implementing these modifications is part of possible future work. A next follow-up step would be to research how our LPC bus modification attacks can be adapted for SPI-based TPMs.

9.1.3 Relay Attacks

Our TPM 2.0 in-system emulator can be combined with the LPC bus emulator from Chapter 7 to relay TPM communication between a modified local system, and an attacker-controlled remote system. This setup can be used in to trick an unsuspecting verifier, who can either be a local user, or a locally running program, to communicate with an attacker-controlled remote TPM. The result would be a *mafia fraud* (cf. Desmedt et al [DGB88]) attack, where the local verifier corresponds to the *customer*, the modified local platform corresponds to the *mafia-owned restaurant*, the LPC bus emulator corresponds to the *mafia gang member*, and the remote TPM corresponds to the *jeweller*.

A similar attack, the "Cuckoo Attack" has been discussed by Parno in [Par08]. The class of attacks described by the Cuckoo Attack considers adversaries who try to trick local users into communicating

[1]We could not (yet) verify our findings against commercial TPM v2.0 chips, due to a lack of suitable test chip samples at the time of this writing.

with attacker-controlled TPMs, instead of the genuine TPMs, on the local trusted platform. Parno tried to formalize his assumption on the local trusted platform, and the attacker platform, requiring physical security for the local platform only.

Preliminary Results For a first preliminary experiment with hardware TPM relay attacks, we modified the firmware of our TPM in-system emulator shown on the left side of Figure 9.1 to directly forward TPM command and response blobs, from the target platform to an attached attacker platform. We succeeded in getting the resulting man-in-the-middle setup to work with the Linux TPM TIS driver running on the target platform. On the attacker platform we experimented with a software TPM simulator, and with the LPC bus emulator introduced earlier in this master thesis.

9.2 TPMs and Embedded Platforms

In recent joint work with Weiser at al. [WTW14] we designed, and implemented, a tiny trusted embedded platform called “GUSTL”. The tiny platform is built around an 8-bit Atmel AVR microcontroller, and contains an embedded TPM with I2C interface. Figure 9.2 shows one of our “GUSTL” circuit boards next to a LPC TPM daughterboard for size comparison.

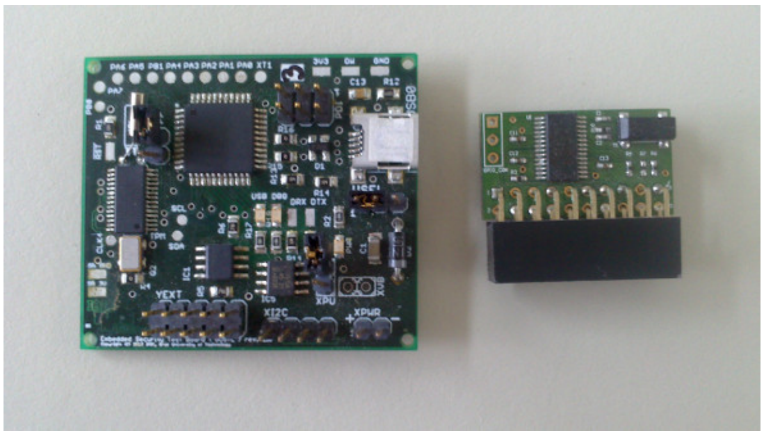

Figure 9.2: The “GUSTL” Embedded Platform

The code size the main firmware of the “GUSTL” platform is limited to 32 kilobytes, by the flash size of the used microcontroller. Additional 4 kilobytes of code memory can be used for an optional on-chip bootloader. The total amount of RAM available on a “GUSTL” is 4 kilobytes. Given these constraints Weiser at al [WTW14] successfully implemented a proof of concept remote firmware update mechanism that uses the on-board embedded TPM to protect secret key material for firmware encryption.

Preliminary Results In our journal paper [WD13] we sketched several attacks on embedded platforms with I2C based TPMs. These attacks covered manipulations of the I2C bus clock and data signals, as well as man-in-the-middle style attacks. The “GUSTL” platform introduced in Weiser at al [WTW14] provides one key element for further work on refining the I2C bus attacks sketched in [WD13], and for investigating potential countermeasures for embedded platforms.

One idea for countermeasures against simple hardware attacks, which do not involve modification of victim the platform, would be to exploit the fact that microcontroller general-purpose I/O pins typically continue to function as inputs, allowing the CPU to read the logic level at the pad, even when they are configured as outputs. In first preliminary experiments on the “GUSTL” platform, we successfully applied this idea to detect unexpected glitches on the TPM reset line.

Chapter 10

Concluding Remarks

"Experience is the name everyone gives to their mistakes."

[Oscar Wilde, Lady Windermere's Fan]

In this thesis we analyzed simple hardware attacks against trusted desktop platform, assuming low to moderate resource effort on the attacker side.

After discussing the motivation for this work, and establishing the context of related work in Chapter 1, we covered the background on trusted computing in Chapter 2, on the protocols for communicating with a TPM in Chapter 3, and on the low-level hardware aspects in Chapter 4.

Platform Reset In Chapter 5 we developed a new variant of the previously known TPM reset attack. Our variant reverses the role of the TPM and the platform. We first discussed the basic principles of the *platform reset attack*, and then proceeded to realize it in a step by step manner. At the end of this process, we found a reliable technique to isolate a TPM from its host platform. Our technique enabled us to isolate the reset signal, and to jam any further communication, at will. The techniques from Chapter 5 allowed us to freeze the current state of a trusted platform across a reboot, assuming that the power supply of the TPM was not lost in the process.

Frame Hijacking Next, in Chapter 6 we considered dynamic roots of trust (D-RTMs), and the protection mechanism that prevents normal platform software from tampering with D-RTM measurements. To bypass these mechanisms, we developed a bus modification technique to *hijack* harmless bus cycles, and to convert them to arbitrary TPM bus cycles. The *frame hijacking* technique discussed in Chapter 6 was based on a combined hardware and software attack, and allowed tampering with D-RTM measurements on the target platform. As part of experimental validation of the frame hijacking attack, we developed a simple FPGA-based simulator of an PC Southbridge. This simulator formed the basis for the LPC bus emulator in Chapter 7.

Physical TPM Transfer The results from Chapter 5, Chapter 6 and Chapter 7 are combined in Chapter 8. Chapter 8 starts with reconsidering details of the steps taken earlier in Chapter 5 to isolate the TPM from the platform. Upon reconsideration, we note that the same technique can be used to selectively drop individual read or write operations to the TPM, thus selectively jamming its communication flow with the host platform. Next, we consider how arbitrary measurement chains can be synthesized from scratch, by utilizing the LPC bus emulator from Chapter 7, instead of the target platform. Synthesis of arbitrary measurement chains breaches security by lying about the state of a platform, given that no strong (cryptographic) binding between particular platforms, and particular TPMs exists. At the end of Chapter 8 we develop the probably most powerful attack in this master thesis that is *physical TPM transfer* between platforms. In the preparation phase of the TPM transfer, we remove the TPM of the target platform, and replace it by an adapter board. The adapter board is then populated with a replacement TPM[1], and allows us got easily access all bus signals for man-in-the-middle attacks. Finally we discuss a nine-step procedure to physically transfer the prepared TPM from the *running* victim platform, to our LPC bus emulator, *without* losing its state.

Using *physical TPM transfer* attackers have the capabilities to freeze an arbitrary good state of the victim platform, physically move the TPM to an attacker controlled platform, and continue to work with the TPM's view of this good state in an attacker-controlled environment.

Outlook to TPM 2.0 A brief outlook on ongoing and possible further work is found in Chapter 9. In the first half of Chapter 9, we discuss preliminary results on our attempts to emulate next generation version 2.0 TPMs on an FPGA platform. We note that TPM 2.0 *is* susceptible to the bus modification attacks discussed in this master thesis, at least on LPC bus and TIS protocol level. We mention preliminary results on an attack that relays TPM communication between modified trusted platform, and an attacker controlled TPM.

Outlook to Embedded Trusted Platforms With the recent availability of TPMs with I2C and SPI interfaces, a second promising direction for future work is embedded systems. In the second half of Chapter 9 we briefly mention the "GUSTL" embedded trusted computing test platform that models a small embedded system with an I2C TPM. As part of future work, we plan to investigate both, attacks, and mitigation methods for this small platform. In [WD13] we already outlined, how our bus modification attacks could be mapped to embedded trusted platforms like the "GUSTL" board. One possible idea on mitigating such attacks is given at the and of chapter Chapter 9. We expect mitigation techniques for simple hardware attacks against embedded systems that are similar to "GUSTL" to become an important part of possible future work.

Conclusions At the beginning of this master thesis, we asked three questions related to simple hardware attacks that can be answered now:

Our first question was, whether current trusted desktop computers can withstand simple hardware attacks. Even when we follow the definition of simple hardware attacks given in the introduction to the letter, our answer here is clearly *no*. The justification for this answer should be obvious from the multitude of very simple attacks discussed in earlier chapters. We point out that the *physical TPM*

[1]If preservation of identity is needed, the original TPM can be solder to that board.

transfer attack can be implemented with a few wires, resistors, and a piece of prototyping board, with total costs being significantly below ten Euros.

Our second question was, whether we can find simple hardware attacks that satisfy this condition, and what we can achieve by increasing the amount of resources available. To answer this question we refer to the LPC bus emulator from Chapter 7, and the discussion on synthesizing arbitrary measurement chains in Chapter 8: By increasing the amount of resources by a factor of ten, thus allowing a budget of approximately one-hundred Euros instead of ten Euros, an attacker can easily buy the FPGA board and equipment required to build an LPC bus emulator. Using the LPC bus emulator, an attacker gains the capability to fake arbitrary measurement chains, given that the values to be measured are known.

Our third and last question was, to whom simple hardware attacks may be a threat, and who, apart from curious master students, could be an attacker. The answer to this question heavily depends on the actual application of the TPM. Based on Chapter 8, we conclude that both, applications using remote attestation, and applications using local attestation can be potential targets.

Both types of attestation have been proposed in literature as building blocks for Digital Rights Management systems using TPMs, and Trusted Computing. In this scenario, the attacker often is the platform owner, or a user with unrestricted physical platform access, who wants to extract media encryption keys from the platform, or who wants to authenticate a rouge platform with a content provider. Based on the results presented earlier, we strongly recommend to carefully consider physical platform security, in particular tamper resistance, when designing such systems. Our conclusion is that DRM designs that use trusted computing, without considering physical platform security at all, are very likely to fall short for simple hardware attacks in the foreseeable future.

Trusted computing provides rich, and powerful, set of primitives that can be used to make systems more secure and trustworthy. The effort to attack an individual quality TPM chip, with the goal to extract secrets from the chip, is extremely high. Software attacks, and simple hardware attacks are usually much easier for an attacker, and usually provide satisfying results. Therefore, we finally conclude that our work emphasizes on the importance of correctly understanding threats and assets, in order to build secure systems.

Appendix A

VHDL Sources of the LPC Bus Frame Hijacker

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- Johannes Winter and Kurt Dietrich. “A Hijacker’s Guide to the LPC Bus”. In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12

References to these source are not always made explicit.

This appendix contains the complete VHDL implementation of the LPC frame hijacker device discussed in Chapter 6. The four most-significant bits of the memory cycle trigger address are configured via the `C_A7_ADDR` parameter. The default value shown in Listing A.1 corresponds to the the read and write trigger addresses discussed in Chapter 6.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sp3e_tpm_hijacker is
  generic (C_A7_ADDR : std_logic_vector(3 downto 0) := x"A");
  port (lclk       : in  std_logic;
        lreset     : in  std_logic;
        lframe_in  : in  std_logic;
        lframe_out : out std_logic;
        lad        : in  std_logic_vector(3 downto 0);
        enable     : in  std_logic);
end entity sp3e_tpm_hijacker;

architecture rtl of sp3e_tpm_hijacker is
  constant START_TGT : std_logic_vector(3 downto 0) := "0000";
  constant CT_MEM_RD : std_logic_vector(3 downto 0) := "0100";
  constant CT_MEM_WR : std_logic_vector(3 downto 0) := "0110";
  type lad_array_t is array(2 downto 0) of std_logic_vector(3 downto 0);
  signal q_lad           : lad_array_t;
  signal q_lframe       : std_logic_vector(2 downto 0);
  signal force_frame    : std_logic;
  signal is_tgt_cycle   : std_logic;
  signal is_mem_cycle   : std_logic;
  signal is_valid_frame : std_logic;
  signal is_valid_a7    : std_logic;
  signal is_triggered   : std_logic;
begin  -- architecture rtl

  -- LAD and LFRAME input shift registers
  lad_shift_reg : process (lclk, lreset) is
  begin
    if lreset = '0' then
      q_lframe <= (others => '1');
      q_lad    <= (others => (others => '-'));
    elsif lclk'event and lclk = '1' then
      q_lframe <= (2 => lframe_in, 1 => q_lframe(2), 0 => q_lframe(1));
      q_lad    <= (2 => lad, 1 => q_lad(2), 0 => q_lad(1));
    end if;
  end process lad_shift_reg;

  -- Trigger logic
  is_tgt_cycle   <= '1' when q_lad(0) = START_TGT else '0';
  is_mem_cycle   <= '1' when (q_lad(1) = CT_MEM_RD) or
                  (q_lad(1) = CT_MEM_WR) else '0';
  is_valid_a7    <= '1' when q_lad(2) = C_A7_ADDR else '0';
  is_valid_frame <= q_lframe(2) and q_lframe(1)
                  and not q_lframe(0);
  is_triggered   <= enable and is_valid_frame and is_tgt_cycle
                  and is_mem_cycle and is_valid_a7;

  -- Force FRAME output register
  delayed_frame_gen : process (lclk, lreset) is
  begin
    if lreset = '0' then
      force_frame <= '1';
    elsif lclk'event and lclk = '1' then
      force_frame <= not is_triggered;
    end if;
  end process delayed_frame_gen;

  -- Output frame multiplexer
  lframe_out <= force_frame when enable = '1' else lframe_in;
end architecture rtl;
```

Listing A.1: VHDL Source Code of the LPC Bus Frame Hijacker [WD12]

Our prototype implementation includes an additional `enable` input to control the hijacker core. When the disabled, the hijacker core does not interfere with the LPC frame signal and directly forwards `lframe_in` to `lframe_out`, thus allowing normal operation of the bus.

A.0.1 Reducing the number of LPC address/data lines to probe

By default the cycle hijacker device evaluates all LPC address/data lines (`lad` input signal) to distinguish LPC memory cycles from other types of LPC bus cycles. A closer look at the LPC bus start fields defined in [Int02, Ch. 4.2.1.1, p.15] and the type/direction values for LPC target cycles[Int02, Ch. 4.2.1.2, p.15] reveals that probing the `lad[3]` and `lad[2]` lines is sufficient to reliable detect LPC memory cycle, under certain preconditions:

Currently the only defined LPC `START` values with `lad[3:2] = 2b00` are used for bus master grant cycles and target cycles. For target cycles the actual cycle type, and direction is indicated by the LPC `CTDIR` field. Coincidentally LPC memory read and write cycles can be distinguished from all other LPC target cycles by the `lad[3:2] = 2b01` value in their `CTDIR` field.

By using these properties we can tie the `lad[1]`, and `lad[0]` inputs to a constant logic zero value. Thus, the LPC hijacker device only needs to probe the `lad[3]`, and `lad[2]`) inputs on the actual bus.

For the attacker the main advantage of the reduced setup is, that the reduced setup requires only half of the `LAD` lines to be probe. On the downside, the two least-significant bits of the trigger address specified by `C_A7_ADDR` are lost. Thus the reduced setup can only be used, if no bus master grant cycles occur while the hijacker device is active.

Appendix B

VHDL Sources of the LPC Bus Emulator

This appendix contains the core implementation of the LPC bus emulator discussed in Chapter 7. The VHDL code shown in Listing B.1 has its origins in the Southbridge emulator (see Section 6.5) that we develop for testing the LPC bus frame hijacking attack.

```vhdl
--------------------------------------------------------------------------------
-- Simple LPC bus master code
--
-- Copyright (C) 2009-2014, IAIK, Graz University of Technology.
-- Author: Johannes Winter <johannes.winter@iaik.tugraz.at>
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.lpc.all;

entity lpc_bus_master is
  generic (
    C_SYS_CLK_POLARITY      : std_logic                := '1';
    C_SYS_RST_POLARITY      : std_logic                := '0';
    C_MAX_SHORT_SYNC_CYCLES : integer range 0 to 65535 := 16;
    C_MAX_LONG_SYNC_CYCLES  : integer range 0 to 65535 := 0;
    C_ABORT_CYCLES          : integer range 1 to 15    := 8);

  port (
    -- System interface signals
    sys_clk : in std_logic;
    sys_rst : in std_logic;

    cyc_adr   : in  std_logic_vector(31 downto 0);
    cyc_wdat  : in  std_logic_vector(7 downto 0);
    cyc_rdat  : out std_logic_vector(7 downto 0);
    cyc_type  : in  lpc_cycle_type_t;
    cyc_write : in  std_logic;
    cyc_start : in  std_logic;
    cyc_busy  : out std_logic;
    cyc_done  : out std_logic;
    cyc_abort : in  std_logic;
    cyc_sync  : out std_logic_vector(3 downto 0);

    -- LPC interface signals
    lad_o  : out std_logic_vector(3 downto 0);
    lad_i  : in  std_logic_vector(3 downto 0);
    lad_oe : out std_logic;
    lclk   : out std_logic;
    lframe : out std_logic;
    lreset : out std_logic);
end entity lpc_bus_master;

architecture rtl of lpc_bus_master is
  -- LPC bus-master core FSM states
  type lpc_master_fsm_state_t is (H_IDLE, H_START, H_CTDIR,
                                  H_ADR7, H_ADR6, H_ADR5, H_ADR4,
                                  H_ADR3, H_ADR2, H_ADR1, H_ADR0,
                                  H_MSIZE, H_DAT0, H_DAT1,
                                  H_TAR1, H_TAR2,
                                  P_SYNC, P_WAIT,
                                  P_DAT1, P_DAT0, P_TAR1, P_TAR2,
                                  H_ABORT);

  -- LPC target cycle types (as seen by the master core)
  type    lpc_ctdir_t is (TARGET_IO, TARGET_MEM, FIRMWARE);
  subtype sync_counter_t is unsigned(15 downto 0);
  subtype abort_counter_t is unsigned(4 downto 0);

  -- LPC bus-master core state
  type lpc_master_core_state_t is record
    fsm_state      : lpc_master_fsm_state_t;
    adr            : std_logic_vector(31 downto 0);
    busy           : std_logic;
    done           : std_logic;
    is_write       : std_logic;
    ctype          : lpc_ctdir_t;
    sync           : std_logic_vector(3 downto 0);
    data           : std_logic_vector(7 downto 0);
    lad            : std_logic_vector(3 downto 0);
    lad_oe         : std_logic;
    ld_data        : std_logic;
    lreset         : std_logic;
    lframe         : std_logic;
    sync_cnt       : sync_counter_t;
    sync_abt_short : std_logic;
    sync_abt_long  : std_logic;
    abort_cnt      : abort_counter_t;
  end record lpc_master_core_state_t;

  -- Reset state of the LPC core
  constant INIT_STATE : lpc_master_core_state_t := (
    fsm_state      => H_IDLE,
    adr            => (others => '0'),
    busy           => '0',
    done           => '0',
    is_write       => '-',
    ctype          => TARGET_IO,
    sync           => (others => '-'),
    data           => (others => '-'),
    lad            => "1111",
    ld_data        => '0',
    lad_oe         => '0',
    lframe         => '1',
    lreset         => '0',
    sync_cnt       => (others => '-'),
    sync_abt_short => '-',
    sync_abt_long  => '-',
    abort_cnt      => (others => '-'));

  -- LPC master core LAD input shift register.
  type lad_array_t is array(0 to 1) of std_logic_vector(3 downto 0);

  -- LPC slave core input registers
  type lpc_master_core_inputs_t is record
    lad : lad_array_t;
  end record;

  -- Initial input values (on reset)
  constant INIT_INPUTS : lpc_master_core_inputs_t := (
    lad => (others => (others => '1')));

```

```
                                          -- Sampled pad inputs
  signal inputs : lpc_master_core_inputs_t;

  -- Current and next state
  signal state      : lpc_master_core_state_t;
  signal next_state : lpc_master_core_state_t;

begin  -- architecture rtl

  ---------------------------------------------------------------------------
  -- PAD input registers
  --
  -- These registers are responsible for sampling the LAD/LFRAME PAD inputs
  -- as close to the PADs as possible.
  ---------------------------------------------------------------------------
  q_pad_regs : process (sys_clk, sys_rst)
  begin  -- process q_lad_reg
    if sys_rst = C_SYS_RST_POLARITY then
      inputs <= INIT_INPUTS;
    elsif sys_clk'event and sys_clk = C_SYS_CLK_POLARITY then
      inputs.lad(1 to inputs.lad'high) <= inputs.lad(0 to inputs.lad'high - 1);
      inputs.lad(0)                    <= lad_i;
    end if;
  end process q_pad_regs;

  -- Bus master core FSM (combinational logic)
  fsm0_comb : process (state, cyc_adr, cyc_wdat, cyc_type, cyc_write,
                       cyc_start, cyc_abort, inputs, lad_i) is
    variable v : lpc_master_core_state_t;
  begin  -- process fsm0_comb
    -- Initialisation
    v                 := state;
    v.done            := '0';
    v.lad             := "1111";
    v.lad_oe          := '1';
    v.lframe          := '1';
    v.lreset          := '1';
    v.abort_cnt       := to_unsigned(C_ABORT_CYCLES, v.abort_cnt'length);
    v.sync_cnt        := (others => '-');
    v.sync_abt_short  := '-';
    v.sync_abt_long   := '-';
    v.ld_data         := '0';

    -- Main state machine
    case state.fsm_state is
      -----------------------------------------------------------------------
      -- H_IDLE: Bus master is ready to start new cycles.
      -----------------------------------------------------------------------
      when H_IDLE =>
        if cyc_start = '1' then
          -- Setup registers
          v.adr      := cyc_adr;
          v.data     := cyc_wdat;
          v.is_write := cyc_write;
          v.ctype    := TARGET_IO;


          case cyc_type is
            when LPC_MST_CYC_IO =>      -- I/O target cycle
              v.lad       := LPC_START_TARGET;
              v.lframe    := '0';
              v.busy      := '1';
              v.ctype     := TARGET_IO;
              v.fsm_state := H_START;

            when LPC_MST_CYC_MEM =>     -- Memory target cycle
              v.lad       := LPC_START_TARGET;
              v.lframe    := '0';
              v.busy      := '1';
              v.ctype     := TARGET_MEM;
              v.fsm_state := H_START;

            when LPC_MST_CYC_TPM =>     -- TPM target cycle
              v.lad       := LPC_START_TPM;
              v.lframe    := '0';
              v.busy      := '1';
              v.ctype     := TARGET_IO;
              v.fsm_state := H_START;

            when LPC_MST_CYC_FIRMWARE =>  -- Firmware cycle
              if cyc_write = '1' then
```

```
                v.lad := LPC_START_FW_WRITE;
              else
                v.lad := LPC_START_FW_READ;
              end if;

              v.lframe    := '0';
              v.busy      := '1';
              v.ctype     := FIRMWARE;
              v.fsm_state := H_CTDIR;

            when others =>               -- Unsupported cycle type
              v.sync      := LPC_SYNC_NONE;
              v.done      := '1';
              v.busy      := '0';
              v.fsm_state := H_IDLE;
          end case;
        else
          v.fsm_state := H_IDLE;
          v.busy      := '0';
        end if;

        -------------------------------------------------------------------------
        -- H_START: Bus master has started a new LPC cycle.
        -------------------------------------------------------------------------
      when H_START =>
        v.fsm_state := H_CTDIR;

        case state.ctype is
          when TARGET_IO =>
            v.lad := LPC_CTYPE_IO & state.is_write & "0";

          when TARGET_MEM =>
            v.lad := LPC_CTYPE_MEM & state.is_write & "0";

          when others =>
            v.lad := (others => '-');
        end case;

        -----------------------------------------------------------------------
        -- H_CTDIR/H_IDSEL: Bus master is about to indicate cycle type and
        --                  direction (or IDSEL if firmware cycle)
        -----------------------------------------------------------------------
      when H_CTDIR =>
        case state.ctype is
          when TARGET_IO =>
            v.fsm_state := H_ADR3;
            v.lad       := state.adr(15 downto 12);

          when TARGET_MEM | FIRMWARE =>
            v.lad       := state.adr(31 downto 28);
            v.fsm_state := H_ADR7;

          when others =>                 -- Error recovery
            v.fsm_state := H_IDLE;
        end case;

        -----------------------------------------------------------------------
        -- H_ADRx: Bus master is about to assert peripheral address
        -----------------------------------------------------------------------
      when H_ADR7 =>
        v.lad       := state.adr(27 downto 24);
        v.fsm_state := H_ADR6;

      when H_ADR6 =>
        v.lad       := state.adr(23 downto 20);
        v.fsm_state := H_ADR5;

      when H_ADR5 =>
        v.lad       := state.adr(19 downto 16);
        v.fsm_state := H_ADR4;

      when H_ADR4 =>
        v.lad       := state.adr(15 downto 12);
        v.fsm_state := H_ADR3;

      when H_ADR3 =>
        v.lad       := state.adr(11 downto 8);
        v.fsm_state := H_ADR2;

      when H_ADR2 =>
        v.lad       := state.adr(7 downto 4);
```

```
          v.fsm_state := H_ADR1;

        when H_ADR1 =>
          v.lad       := state.adr(3 downto 0);
          v.fsm_state := H_ADR0;

        when H_ADR0 =>
          case state.ctype is
            when FIRMWARE =>
              v.lad       := LPC_MSIZE_8_BIT;
              v.fsm_state := H_MSIZE;

            when others =>
              if state.is_write = '1' then
                v.lad       := state.data(3 downto 0);
                v.fsm_state := H_DAT0;
              else
                v.lad       := "1111";
                v.fsm_state := H_TAR1;
              end if;
          end case;

          -----------------------------------------------------------------------
          -- H_MSIZE: Firmware cycle memory size (currently fixed to 8-bit)
          -----------------------------------------------------------------------
        when H_MSIZE =>
          if state.is_write = '1' then
            v.lad       := state.data(3 downto 0);
            v.fsm_state := H_DAT0;
          else
            v.lad       := "1111";
            v.fsm_state := H_TAR1;
          end if;

          -----------------------------------------------------------------------
          -- H_DATx: Host->Peripheral data transfer
          -----------------------------------------------------------------------
        when H_DAT0 =>
          v.lad       := state.data(7 downto 4);
          v.fsm_state := H_DAT1;

        when H_DAT1 =>
          v.lad       := "1111";
          v.fsm_state := H_TAR1;

          -----------------------------------------------------------------------
          -- H_TARx: Host->Peripheral Turn-Around
          -----------------------------------------------------------------------
        when H_TAR1 =>
          v.lad_oe    := '0';
          v.fsm_state := H_TAR2;

        when H_TAR2 =>
          v.lad_oe          := '0';
          v.fsm_state       := P_SYNC;
          v.sync_cnt        := to_unsigned(0, v.sync_cnt'length);
          v.sync_abt_short  := '0';
          v.sync_abt_long   := '0';

          -----------------------------------------------------------------------
          -- P_SYNC, P_WAIT: Peripheral->Host Synchronization
          --
          -----------------------------------------------------------------------
        when P_SYNC =>
          -- Latch the sync word
          v.lad_oe    := '0';
          v.fsm_state := P_WAIT;

        when P_WAIT =>
          v.lad_oe := '0';
          v.sync   := inputs.lad(0);

          -- Update the sync counter
          if C_MAX_SHORT_SYNC_CYCLES > 0 or C_MAX_LONG_SYNC_CYCLES > 0 then
            v.sync_cnt := state.sync_cnt + 1;

            if C_MAX_SHORT_SYNC_CYCLES > 0
              and state.sync_cnt = to_unsigned(C_MAX_SHORT_SYNC_CYCLES, state.sync_cnt'length) then
              v.sync_abt_short := '1';
            end if;

            if C_MAX_LONG_SYNC_CYCLES > 0
              and state.sync_cnt = to_unsigned(C_MAX_LONG_SYNC_CYCLES, state.sync_cnt'length) then
              v.sync_abt_long := '1';
            end if;
          end if;

          -- Now decode the active sync word
          case inputs.lad(0) is
            when LPC_SYNC_ERROR | LPC_SYNC_READY =>  -- READY OR ERROR
              if state.is_write = '1' then
                v.fsm_state := P_TAR1;
              else
                v.fsm_state := P_DAT1;
              end if;

            when LPC_SYNC_SHORT =>         -- Short sync
              if state.sync_abt_short = '1' then
                -- Abort the transaction
                v.fsm_state := H_ABORT;
              end if;

            when LPC_SYNC_LONG =>          -- Long sync
              if state.sync_abt_long = '1' then
                -- Abort the transaction
                v.fsm_state := H_ABORT;
              end if;

              -- Failure
            when others =>                 -- READY or ERROR
              -- Abort the transaction
              v.lframe    := '0';
              v.fsm_state := H_ABORT;
          end case;

          ----------------------------------------------------------------------
          -- P_DATx: Peripheral->Host data transfer
          ----------------------------------------------------------------------

          -- P_DAT0 is merged into P_WAIT
          -- when P_DAT0 =>
          -- v.lad_oe    := '0';
          --  v.fsm_state := P_DAT1;

        when P_DAT1 =>
          v.lad_oe    := '0';
          v.fsm_state := P_TAR1;
          v.ld_data   := '1';

          ----------------------------------------------------------------------
          -- P_TARx: Peripheral->Host Turn-Around
          --
          -- Note: P_TAR2 is merged with the bus-master idle state.
          ----------------------------------------------------------------------
        when P_TAR1 =>
          v.done      := '0';
          v.busy      := '1';
          v.fsm_state := P_TAR2;

        when P_TAR2 =>
          v.done      := '1';
          v.busy      := '0';
          v.fsm_state := H_IDLE;

          ----------------------------------------------------------------------
          -- Host side abort
          ----------------------------------------------------------------------
        when H_ABORT =>
          -- Signal the pending abort
          v.lad_oe := '0';
          v.lad    := LPC_START_ABORT;
          v.lframe := '0';
          v.sync   := LPC_SYNC_NONE;

          -- Update the abort counter
          v.abort_cnt := state.abort_cnt - 1;

          if state.abort_cnt = to_unsigned(0, state.abort_cnt'length) then
            -- Abort count down finished
            v.lad_oe    := '1';
            v.lad       := LPC_START_ABORT;
            v.done      := '1';
```

```vhdl
          v.busy      := '0';
          v.fsm_state := H_IDLE;

        else
          -- Still counting down
          v.fsm_state := H_ABORT;
        end if;

      when others =>                    -- FSM error, recover to idle state
        v.fsm_state := H_IDLE;
    end case;

    -- Input data handling
    if state.ld_data = '1' then
      v.data := inputs.lad(0) & inputs.lad(1);
    end if;

    -- Uniform handling of user abort
    if state.fsm_state /= H_IDLE and cyc_abort = '1' then
      v.fsm_state := H_ABORT;
    end if;

    -- Next state output
    next_state <= v;
  end process fsm0_comb;

  -- Bus master core FSM (sequential logic)
  fsm0_seq : process (sys_clk, sys_rst) is
  begin
    if sys_rst = C_SYS_RST_POLARITY then
      state <= INIT_STATE;
    elsif sys_clk'event and sys_clk = C_SYS_CLK_POLARITY then
      state <= next_state;
    end if;
  end process fsm0_seq;

  -- Outputs
  lad_o  <= state.lad;
  lad_oe <= state.lad_oe;
  lreset <= state.lreset;
  lframe <= state.lframe;

  cyc_rdat <= state.data;
  cyc_done <= state.done;
  cyc_sync <= state.sync;
  cyc_busy <= state.busy;

  -- Pass-throught the LPC clock signal
  lclk <= sys_clk;
end architecture rtl;
```

Listing B.1: LPC Bus Master Core

For our experiments with embedded trusted computing discussed in [PWT12], we added an SPI control interface. This additional interface turns the LPC bus master core shown above into a fully fledged SPI-to-LPC bus bridge, that supports the software interface that has been discussed in Section 7.2. The VHDL code of the final bus bridge is shown below in Listing B.2.

```vhdl
-------------------------------------------------------------------------------
-- Simple SPI slave for the TPM interface
--
-- Copyright (C) 2011-2014 IAIK, Graz University of Technology.
-- Author: Johannes Winter <johannes.winter@iaik.tugraz.at>
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.lpc.all;

entity tif_spi_slave is
  generic (
    C_RST_POLARITY   : std_logic := '0';   -- System reset polarity
    C_CLK_POLARITY   : std_logic := '1';   -- System clock polarity
    C_SPI_BIG_ENDIAN : boolean   := true;  -- Bit endianess
    C_SPI_CPHA       : std_logic := '0';   -- SCK clock phase
    C_SPI_CPOL       : std_logic := '0');  -- SCK clock polarity
  port (
    -- System interface
    sys_rst_i : std_logic;
    sys_clk_i : std_logic;

    -- SPI bus interface (slave)
    spi_sck_i  : in  std_logic;          -- SPI clock
    spi_ss_i   : in  std_logic;          -- SPI slave select
    spi_miso_o : out std_logic;          -- SPI slave data output
    spi_mosi_i : in  std_logic;          -- SPI master data output

    -- LPC bus interface (master)
    lpc_lad_o     : out std_logic_vector(3 downto 0);
    lpc_lad_i     : in  std_logic_vector(3 downto 0);
    lpc_lad_oe    : out std_logic;
    lpc_lclk_o    : out std_logic;
    lpc_lframe_o  : out std_logic;
    lpc_lreset_o  : out std_logic;

    -- Status interface (debugging)
    dbg_lpc_busy_o : out std_logic;      -- LPC bus is busy
    dbg_spi_busy_o : out std_logic);     -- SPI state machine is non-idle
end tif_spi_slave;

architecture rtl of tif_spi_slave is
  -- Fixed parameters for the LPC bus master
  --
  -- We do NOT enforce any built-in limit for short or long bus
  -- cycles. (The user can request an abort explicitly over the
  -- SPI interface.)
  constant C_MAX_SHORT_SYNC_CYCLES : integer range 0 to 65535 := 0;
  constant C_MAX_LONG_SYNC_CYCLES  : integer range 0 to 65535 := 0;
  constant C_ABORT_CYCLES          : integer range 1 to 15    := 8;
  -----------------------------------------------------------------------------
  -- SPI commands
  --


  --
  -- Read Device ID
  --
  constant SPI_CMD_READ_ID : std_logic_vector(7 downto 0) := x"C0";

  constant SPI_DEVICE_ID : std_logic_vector(7 downto 0) := x"2A";

  --
  -- Read the control/status register
  --
  constant SPI_CMD_READ_STATUS : std_logic_vector(7 downto 0) := x"C1";

  --
  -- Write the control/status register
  --
  constant SPI_CMD_WRITE_CTRL : std_logic_vector(7 downto 0) := x"C2";

  --
  -- Lazily read the result of the last LPC bus transaction.
  --
  constant SPI_CMD_READ_LAZY : std_logic_vector(7 downto 0) := x"C3";

  --
  -- Start LPC read or write transaction
  --
  -- The 5th bit of the command encodes the direction (0=read, 1=write)
  -- The lower 4 bits of the command indicate the LPC_MST_CYC_xxx cycle
  -- type as defined in lpc_pkg.vhd.
  --
  --
  constant SPI_CMD_LPC_READ_IO   : std_logic_vector(7 downto 0)  := x"D1";
  constant SPI_CMD_LPC_READ_MEM  : std_logic_vector(7 downto 0)  := x"D2";
  constant SPI_CMD_LPC_READ_TPM  : std_logic_vector(7 downto 0)  := x"D4";
  constant SPI_CMD_LPC_READ_FW   : std_logic_vector(7 downto 0)  := x"D8";
  constant SPI_CMD_LPC_WRITE_IO  : std_logic_vector(7 downto 0)  := x"E1";
  constant SPI_CMD_LPC_WRITE_MEM : std_logic_vector(7 downto 0)  := x"E2";
  constant SPI_CMD_LPC_WRITE_TPM : std_logic_vector(7 downto 0)  := x"E4";
  constant SPI_CMD_LPC_WRITE_FW  : std_logic_vector(7 downto 0)  := x"E8";

  -----------------------------------------------------------------------------
  -- Discrete FSM state values.
  type tif_fsm_state_t is (IDLE, ADR3, ADR2, ADR1, ADR0, WDAT, SYNC, RDAT, RID, CTRL, BADCMD);
  type tis_spi_data_src_t is (INVALID, STATUS, LPCDATA, DEVICEID);

  -- FSM state and registers
  type tif_state_t is record
    fsm_state : tif_fsm_state_t;                 -- FSM state
    lpc_write : std_logic;                       -- LPC cycle direction
    lpc_type  : lpc_cycle_type_t;                -- LPC cycle type
    lpc_adr   : std_logic_vector(31 downto 0);   -- LPC target address
    lpc_start : std_logic;                       -- LPC start flag
    lpc_abort : std_logic;                       -- LPC abort flag
    lpc_wdat  : std_logic_vector(7 downto 0);    -- LPC write data
  end record;

  -- Initial state value
  constant INITIAL_STATE : tif_state_t := (
    fsm_state => IDLE,
    lpc_write => '-',
    lpc_adr   => (others => '-'),
    lpc_type  => (others => '-'),
    lpc_start => '0',
    lpc_abort => '0',
    lpc_wdat  => (others => '-'));

  -- Synchronizer FFs for SPI control signals
  signal ss_sync   : std_logic_vector(1 downto 0);
  signal sck_sync  : std_logic_vector(1 downto 0);
  signal mosi_sync : std_logic_vector(1 downto 0);

  -- Internal versions of SPI control signals
  signal ss_int   : std_logic;
  signal sck_int  : std_logic;
  signal mosi_int : std_logic;

  -- Internal signals
  signal sck_dly           : std_logic;  -- Delayed version of sck_int
  signal sck_leading_edge  : std_logic;  -- Leading SCK edge indicator
  signal sck_trailing_edge : std_logic;  -- Trailing SCK edge indicator

  signal bit_cnt  : std_logic_vector(2 downto 0);   -- RX/TX bit counter
  signal rx_valid : std_logic;                      -- RX byte valid
  signal rx_byte  : std_logic_vector(7 downto 0);   -- RX byte data

  signal tx_byte      : std_logic_vector(7 downto 0);   -- TX byte data
  signal tx_load      : std_logic;                      -- TX byte load
  signal tx_byte_next : std_logic_vector(7 downto 0) := x"CA";

  signal state        : tif_state_t;          -- Current interface FSM state
  signal next_state   : tif_state_t;          -- Next interface FSM state
  signal spi_data_sel : tis_spi_data_src_t;   -- Data select

  -- LPC control and status signals
  signal cyc_adr   : std_logic_vector(31 downto 0);
  signal cyc_wdat  : std_logic_vector(7 downto 0);
  signal cyc_rdat  : std_logic_vector(7 downto 0);
  signal cyc_type  : lpc_cycle_type_t;
  signal cyc_write : std_logic;
  signal cyc_start : std_logic;
  signal cyc_busy  : std_logic;
  signal cyc_done  : std_logic;
  signal cyc_abort : std_logic;
  signal cyc_sync  : std_logic_vector(3 downto 0);

begin  -- rtl

  -----------------------------------------------------------------------------
  -- SPI input synchronization stage
  --
```

```vhdl
  -- The SPI clock domain is asynchronous to the internal system
  -- clock domain of this core. We two synchronization flip-flops
  -- to avoid meta-stability issues on the SPI input signals.
  --
  -- SPI timing must be chosen such that the data intput is stable
  -- (and synced to the system domain) before the clock signal is
  -- pulsed.
  --
  sync_ffs : process (sys_clk_i, sys_rst_i)
  begin  -- process sync_ffs
    if sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      ss_sync   <= spi_ss_i & ss_sync(ss_sync'high downto 1);
      sck_sync  <= spi_sck_i & sck_sync(sck_sync'high downto 1);
      mosi_sync <= spi_mosi_i & mosi_sync(mosi_sync'high downto 1);
    end if;
  end process sync_ffs;

  ss_int   <= ss_sync(0);
  sck_int  <= sck_sync(0);
  mosi_int <= mosi_sync(0);

  -----------------------------------------------------------------------------
  -- SPI clock edge detection
  --
  sck_dly_ff : process (sys_clk_i, sys_rst_i)
  begin  -- process sck_dly_ff
    if sys_rst_i = C_RST_POLARITY then
      sck_dly <= not C_CLK_POLARITY;
    elsif sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      sck_dly <= sck_int;
    end if;
  end process sck_dly_ff;

  sck_leading_edge  <= (sck_int xor sck_dly) and not (sck_int xor C_SPI_CPOL);
  sck_trailing_edge <= (sck_int xor sck_dly) and (sck_int xor C_SPI_CPOL);

  -----------------------------------------------------------------------------
  -- SPI bit counter
  --
  bit_counter : process (sys_clk_i, sys_rst_i)
  begin  -- process bit_counter
    if sys_rst_i = C_RST_POLARITY then
      bit_cnt  <= (others => '0');
      rx_valid <= '0';
      tx_load  <= '0';
    elsif sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      if sck_leading_edge = '1' and bit_cnt = "111" then
        -- RX valid/TX load detection
        rx_valid <= '1';
        tx_load  <= '1';
      else
        -- Inside a bit
        rx_valid <= '0';
      end if;

      if sck_leading_edge = '1' then
        -- Normal bit counter update
        bit_cnt <= std_logic_vector(unsigned(bit_cnt) + to_unsigned(1, bit_cnt'length));
      end if;

      if sck_trailing_edge = '1' then
        -- Clear TX load on trailing edge
        tx_load <= '0';
      end if;
    end if;
  end process bit_counter;

  -----------------------------------------------------------------------------
  -- SPI RX/TX shift registers
  --
  rx_shift_reg : process (sys_clk_i, sys_rst_i)
  begin  -- process rx_shift_reg
    if sys_rst_i = C_RST_POLARITY then
      rx_byte <= (others => '0');
    elsif sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      if sck_leading_edge = '1' then
        if C_SPI_BIG_ENDIAN then
          rx_byte <= rx_byte(6 downto 0) & mosi_int;
        else
          rx_byte <= mosi_int & rx_byte(7 downto 1);
        end if;
      end if;
    end if;
  end process rx_shift_reg;

  --
  -- Reloading of the shift register content happens one clock
  -- cycle after the SCK edge for the final beithas been detected.
  --
  -- This simplifies the control state machine below by allowing
  -- the states to directly specify their intended selector value.
  --
  --
  tx_shift_reg : process (sys_clk_i, sys_rst_i)
  begin  -- process tx_shift_reg
    if sys_rst_i = C_RST_POLARITY then
      tx_byte <= (others => '0');
    elsif sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      if sck_trailing_edge = '1' and tx_load = '1' then
        tx_byte <= tx_byte_next;
      elsif sck_trailing_edge = '1' then
        -- Cycle the current byte
        if C_SPI_BIG_ENDIAN then
          tx_byte <= tx_byte(6 downto 0) & tx_byte(7);
        else
          tx_byte <= tx_byte(0) & tx_byte(7 downto 1);
        end if;
      end if;
    end if;
  end process tx_shift_reg;

-- Transmit data source multiplexer
  tx_mux : process (spi_data_sel, cyc_sync, cyc_abort, cyc_done, cyc_rdat, cyc_busy, state)
  begin  -- process tx_mux
    case (spi_data_sel) is
      when STATUS =>
        -- LPC bridge status
        tx_byte_next(7 downto 4) <= cyc_sync;
        tx_byte_next(3)          <= '0';         -- Reserved
        tx_byte_next(2)          <= cyc_abort;   -- Abort pending
        tx_byte_next(1)          <= '0';         -- Reserved
        tx_byte_next(0)          <= cyc_busy or state.lpc_start;

      when LPCDATA =>
        -- LPC data read register
        tx_byte_next <= cyc_rdat;

      when DEVICEID =>
        -- SPI device ID register
        tx_byte_next <= SPI_DEVICE_ID;

      when others =>
        -- Invalid register access
        tx_byte_next <= (others => '1');
    end case;
  end process tx_mux;

-- Output data handling
  spi_miso_o <= tx_byte(7) when C_SPI_BIG_ENDIAN else tx_byte(0);

-----------------------------------------------------------------------------
-- LPC bus master
--
  lpc_bus_master_1 : lpc_bus_master
    generic map (
      C_SYS_CLK_POLARITY      => C_CLK_POLARITY,
      C_SYS_RST_POLARITY      => C_RST_POLARITY,
      C_MAX_SHORT_SYNC_CYCLES => C_MAX_SHORT_SYNC_CYCLES,
      C_MAX_LONG_SYNC_CYCLES  => C_MAX_LONG_SYNC_CYCLES,
      C_ABORT_CYCLES          => C_ABORT_CYCLES)
    port map (
      sys_clk => sys_clk_i,
      sys_rst => sys_rst_i,

      cyc_adr   => cyc_adr,
      cyc_wdat  => cyc_wdat,
      cyc_rdat  => cyc_rdat,
      cyc_type  => cyc_type,
      cyc_write => cyc_write,
      cyc_start => cyc_start,
      cyc_busy  => cyc_busy,
      cyc_done  => cyc_done,
```

```vhdl
      cyc_abort => cyc_abort,
      cyc_sync  => cyc_sync,

      lad_o  => lpc_lad_o,
      lad_i  => lpc_lad_i,
      lad_oe => lpc_lad_oe,
      lclk   => lpc_lclk_o,
      lframe => lpc_lframe_o,
      lreset => lpc_lreset_o);

-- Control signals
  cyc_start <= state.lpc_start and not state.lpc_abort;
  cyc_abort <= state.lpc_abort;
  cyc_write <= state.lpc_write;
  cyc_adr   <= state.lpc_adr;
  cyc_type  <= state.lpc_type;
  cyc_wdat  <= state.lpc_wdat;

-----------------------------------------------------------------------------
--
-- Bridge interface state machine
--

  fsm_comb : process (state, ss_int, rx_valid, rx_byte, cyc_done)
    variable v              : tif_state_t;
    variable v_spi_data_sel : tis_spi_data_src_t;
  begin  -- process fsm_comb
    -- Assume no state change
    v := state;

    v_spi_data_sel := INVALID;
    v.lpc_start    := '0';

    if ss_int = '1' then
      -- We have been deselected
      v.fsm_state := IDLE;

    else
      -- Main state machine
      case state.fsm_state is
        when IDLE =>
          if ss_int = '0' and rx_valid = '1' then
            case rx_byte is
              when SPI_CMD_READ_ID =>
                v.lpc_write := '-';
                v.lpc_type  := (others => '-');
                v.fsm_state := RID;

              when SPI_CMD_READ_LAZY =>
                -- Lazy read operation - directly go to RDAT state
                v.lpc_write := '-';
                v.lpc_type  := (others => '-');
                v.fsm_state := RDAT;

              when SPI_CMD_WRITE_CTRL =>
                -- Control register write
                v.lpc_write := '-';
                v.lpc_type  := (others => '-');
                v.fsm_state := CTRL;

              when SPI_CMD_READ_STATUS =>
                -- Directly go to sync status (as if a write were done)
                v.lpc_write := '1';
                v.lpc_type  := (others => '-');
                v.fsm_state := SYNC;

              when SPI_CMD_LPC_READ_IO | SPI_CMD_LPC_READ_TPM =>
                -- Read/write with 16-bit
                v.lpc_write := '0';
                v.lpc_type  := rx_byte(3 downto 0);
                v.fsm_state := ADR1;

              when SPI_CMD_LPC_WRITE_IO | SPI_CMD_LPC_WRITE_TPM =>
                -- Read/write with 16-bit address
                v.lpc_write := '1';
                v.lpc_type  := rx_byte(3 downto 0);
                v.fsm_state := ADR1;

              when SPI_CMD_LPC_READ_MEM | SPI_CMD_LPC_READ_FW =>
                -- Read/write with 32-bit address
                v.lpc_write := '0';
                v.lpc_type  := rx_byte(3 downto 0);
                v.fsm_state := ADR3;

              when SPI_CMD_LPC_WRITE_MEM | SPI_CMD_LPC_WRITE_FW =>
                -- Read/write with 32-bit address
                v.lpc_write := '1';
                v.lpc_type  := rx_byte(3 downto 0);
                v.fsm_state := ADR3;

              when others =>
                -- Invalid/byte command
                v.lpc_write := '-';
                v.lpc_type  := (others => '-');
                v.fsm_state := BADCMD;
            end case;
          end if;

        when ADR3 =>
          if rx_valid = '1' then
            v.lpc_adr(31 downto 24) := rx_byte;
            v.fsm_state             := ADR2;
          end if;

        when ADR2 =>
          if rx_valid = '1' then
            v.lpc_adr(23 downto 16) := rx_byte;
            v.fsm_state             := ADR1;
          end if;

        when ADR1 =>
          if rx_valid = '1' then
            v.lpc_adr(15 downto 8) := rx_byte;
            v.fsm_state            := ADR0;
          end if;

        when ADR0 =>
          if rx_valid = '1' then
            v.lpc_adr(7 downto 0) := rx_byte;

            if state.lpc_write = '1' then
              -- Get the write data
              v.fsm_state := WDAT;

            else
              -- Start a read transaction
              v.fsm_state := SYNC;
              v.lpc_start := '1';
            end if;
          end if;

        when WDAT =>
          if rx_valid = '1' then
            v.lpc_wdat  := rx_byte;
            v.lpc_start := '1';
            v.fsm_state := SYNC;
          end if;

        when SYNC =>
          -- Select the status output
          v_spi_data_sel := STATUS;

          -- TODO: Check rx_valid test (likely buggy)
          if rx_valid = '1' and cyc_done = '1' and state.lpc_write = '0' then
            -- Read completed go to read data state
            v.fsm_state := RDAT;
          end if;

        when RDAT =>
          -- Select the data output
          v_spi_data_sel := LPCDATA;

        when CTRL =>
          -- Write the control register
          if rx_valid = '1' then
            v.lpc_abort := rx_byte(2);  -- Set/clear the abort flag
          end if;

        when RID =>
          -- Read the device ID register
          v_spi_data_sel := DEVICEID;

```

```vhdl
        when BADCMD =>
          -- Bad command, wait for deselect
          v_spi_data_sel := INVALID;

        when others =>
          -- Recover to sane state
          v := INITIAL_STATE;
      end case;
    end if;

    -- Assign the ouputs
    next_state   <= v;
    spi_data_sel <= v_spi_data_sel;
  end process fsm_comb;

  fsm_seq : process (sys_clk_i, sys_rst_i)
  begin  -- process fsm_seq
    if sys_rst_i = C_RST_POLARITY then
      -- Check proper core configuration
      assert C_SPI_CPHA = '0' report "SPI␣CPHA=1␣modes␣are␣not␣(yet)␣supported␣by␣this␣core" severity failure;

      state <= INITIAL_STATE;
    elsif sys_clk_i'event and sys_clk_i = C_CLK_POLARITY then
      state <= next_state;
    end if;
  end process fsm_seq;

  -- Debug outputs
  dbg_lpc_busy_o <= cyc_busy;
  dbg_spi_busy_o <= '1' when (state.fsm_state /= IDLE) else '0';
end rtl;
```

Listing B.2: SPI Control Interface

Bibliography

[AF04] Tiago Alves and Don Felton. *TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems*. July 2004. `http://www.arm.com/pdfs/TZ_Whitepaper.pdf` (cited on page 6).

[And02] Ross J. Anderson. *Security in open versus closed systems — the dance of Boltzmann, Coase and Moore*. Technical Report. England: Cambridge University, 2002. `http://www.cl.cam.ac.uk/~rja14/Papers/toulouse.pdf` (cited on pages 14, 15).

[And12] Keith Andrews. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. Oct. 22, 2012. `http://ftp.iicm.edu/pub/keith/thesis/` (cited on page xi).

[Ber05] Brian Berger. "Trusted computing group history". In: *Information Security Technical Report* 10.2 (2005), pages 59–62. ISSN 1363-4127. doi:http://dx.doi.org/10.1016/j.istr.2005.05.007. `http://www.sciencedirect.com/science/article/pii/S1363412705000233` (cited on page 12).

[Ber+06] Stefan Berger et al. "vTPM: Virtualizing the Trusted Platform Module". In: *Proceedings of the 15th USENIX Security Symposium*. USENIX. Aug. 2006, pages 305–320. `https://www.usenix.org/legacy/event/sec06/tech/full_papers/berger/berger.pdf` (cited on page 16).

[C+95] Compaq Computer Corporation, Cirrus Logic Incorporated, et al. Revision 6.0. Sept. 1995. `http://www.smsc.com/Downloads/SMSC/Downloads_Archive/papers/serirq60.doc` (cited on page 31).

[DGB88] Yvo Desmedt, Claude Goutier, and Samy Bengio. "Special uses and abuses of the Fiat-Shamir passport protocol". In: *Advances in Cryptology – CRYPTO'87*. Springer. 1988, pages 21–39 (cited on page 72).

[DW10] Kurt Dietrich and Johannes Winter. "Towards Customizable, Application Specific Mobile Trusted Modules". In: *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*. STC '10. Chicago, Illinois, USA: ACM, 2010, pages 31–40. ISBN 978-1-4503-0095-7. doi:10.1145/1867635.1867642. `http://doi.acm.org/10.1145/1867635.1867642` (cited on page 16).

[DEG06] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. “Using CPU system management mode to circumvent operating system security functions”. In: *CanSecWest/core06* (2006) (cited on page 6).

[EB09] Jan-Erik Ekberg and Sven Bugiel. “Trust in a small package: minimized MRTM software implementation for mobile secure environments”. In: *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM. 2009, pages 9–18 (cited on page 16).

[ET09] Paul England and Talha Tariq. “Towards a programmable TPM”. In: *Trusted Computing*. Springer, 2009, pages 1–13 (cited on page 16).

[Gol+10] Kenneth A. Goldman et al. *IBM’s Software Trusted Platform Module*. SorceForge open-source project. 2010. `http://ibmswtpm.sourceforge.net/` (cited on page 16).

[Gra06] David Grawrock. *The Intel Safer Computing Initiative*. Intel Press, 2006. ISBN 0-9764832-6-2 (cited on pages 11, 14).

[Gra09] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009. ISBN 978-1934053171 (cited on pages 3, 11, 14, 19).

[TCG05] Trusted Computing Group. *TCG PC Client Specific TPM Interface Specification (TIS)*. Version 1.2 FINAL. For TPM Family 1.2; Level 2. Nov. 2005. `http://www.trustedcomputinggroup.org/files/resource_files/87BCE22B-1D09-3519-ADEBA772FBF02CBD/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf` (cited on pages 18, 23, 25, 33, 35, 37, 44, 66).

[TCG07a] Trusted Computing Group. *TCG Specification Architecture Overview*. Revision 1.4. Feb. 2007. `http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf` (cited on pages 11, 13–15, 37).

[TCG07d] Trusted Computing Group – TPM Working Group. *TPM Main Part 1 Design Principles*. Specification version 1.2 Level 2 Revision 103. Sept. 2007. `http://www.trustedcomputinggroup.org/files/resource_files/ACD19914-1D09-3519-ADA64741A1A15795/mainP1DPrev103.zip` (cited on pages 11, 13).

[TCG07c] Trusted Computing Group – TPM Working Group. *TPM Main Part 2 Structures*. Specification version 1.2 Level 2 Revision 103. Sept. 2007. `http://www.trustedcomputinggroup.org/files/resource_files/8D3D6571-1D09-3519-AD22EA2911D4E9D0/mainP2Structrev103.pdf` (cited on page 11).

[TCG07b] Trusted Computing Group – TPM Working Group. *TPM Main Part 3 Commands*. Specification version 1.2 Level 2 Revision 103. Sept. 2007. `http://www.trustedcomputinggroup.org/files/static_page_files/ACD28F6C-1D09-3519-AD210DC2597F1E4C/mainP3Commandsrev103.pdf` (cited on pages 11, 15).

[TCG06] Trusted Computing Group - TSS Working Group. *TCG Software Stack (TSS) Specification Version 1.2 Level 1*. Part1: Commands and Structures. June 2006. `https://www.`

trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf (cited on page 17).

[Int02] Intel. *Intel Low Pin Count (LPC) Interface Specification*. Revision 1.1. Aug. 2002. http://www.intel.com/design/chipsets/industry/25128901.pdf (cited on pages 31, 33–35, 43, 51, 80).

[Int08] Intel. *Intel I/O Controller Hub 10 (ICH10) Family Datasheet*. Oct. 2008. http://www.intel.com/content/dam/doc/datasheet/io-controller-hub-10-family-datasheet.pdf (cited on pages 16, 30, 55).

[Kau07] Bernahrd Kauer. "OSLO: improving the security of trusted computing". In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Boston, MA: USENIX Association, 2007, 16:1–16:9. ISBN 111-333-5555-77-9. http://portal.acm.org/citation.cfm?id=1362903.1362919 (cited on pages 6–9, 13, 18, 19, 40, 41).

[KSP05] Klaus Kursawe, Dries Schellekens, and Bart Preneel. "Analyzing trusted platform communication". In: *In: ECRYPT Workshop, CRASH – CRyptographic Advances in Secure Hardware*. 2005, page 8. https://www.cosic.esat.kuleuven.be/publications/article-591.pdf (cited on pages 8, 9, 41).

[Law07] Nate Lawson. *TPM hardware attacks (part 2)*. http://rdist.root.org/2007/07/17/tpm-hardware-attacks-part-2/. Blog disucssion. 2007 (cited on page 9).

[McC+08] Jonathan M. McCune et al. "Flicker: An Execution Infrastructure for Tcb Minimization". In: *SIGOPS Oper. Syst. Rev.* 42.4 (Apr. 2008), pages 315–328. ISSN 0163-5980. doi:10.1145/1357010.1352625. http://doi.acm.org/10.1145/1357010.1352625 (cited on pages 13, 18).

[Par08] Bryan Parno. "Bootstrapping Trust in a" Trusted" Platform." In: *HotSec*. 2008 (cited on page 72).

[Pea02] Siani Pearson. *Trusted Computing Platforms, the Next Security Solution*. Technical report. HP Laboratories Bristol HPL-2002-221: Trusted E-Services Laboratory, May 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-221.pdf.

[PW13] Martin Pirker and Johannes Winter. "Semi-automated Prototyping of a TPM v2 Software and Hardware Simulation Platform". In: *Trust and Trustworthy Computing*. Springer, 2013, pages 106–114 (cited on pages 71, 72).

[PWT12] Martin Pirker, Johannes Winter, and Ronald Toegl. "Lightweight Distributed Heterogeneous Attested Android Clouds". In: *TRUST*. 2012, pages 122–141 (cited on pages 57–59, 63, 89).

[RT08] Joanna Rutkowska and Alexander Tereshkin. "Bluepilling the xen hypervisor". In: *Black Hat USA* (2008) (cited on page 7).

[RW08] Joanna Rutkowska and Rafał Wojtczuk. "Preventing and detecting Xen hypervisor subversions". In: *Blackhat Briefings USA* (2008) (cited on page 7).

[SVW04] Reiner Sailer, Leendert Van Doorn, and James P Ward. *The role of TPM in enterprise security*. Technical report. Technical Report RC23363 (W0410-029), IBM Research, 2004.

[Sch12] Dries Schellekens. “Design and Analysis of Trusted Computing Platforms”. Ph. D. thesis, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium. PhD thesis. 2012 (cited on pages 8, 9, 41).

[Spa+] Evan R. Sparks et al. *TPM Reset Attack*. `http://www.cs.dartmouth.edu/~pkilab/sparks/` (cited on pages 7, 9).

[Spa07] Evan R. Sparks. *A Security Assessment of Trusted Platform Modules*. Technical report TR2007-597. Hanover, NH: Dartmouth College, Computer Science, 2007. `http://www.cs.dartmouth.edu/reports/TR2007-597.ps.Z` (cited on pages 8, 40).

[Sta10] Richard M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Second Edition. Free Software Foundation, 2010. ISBN 978-0-9831592-0-9. `http://www.gnu.org/doc/fsfs-ii-2.pdf` (cited on pages 14, 15).

[SS04] Mario Strasser and Heiko Stamer. *Software-based TPM Emulator*. berlios.de open-source project. 2004. `http://http://tpm-emulator.berlios.de/` (cited on page 16).

[SS08] Mario Strasser and Heiko Stamer. “A Software-Based Trusted Platform Module Emulator”. In: *Trusted Computing - Challenges and Applications*. Edited by Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch. Volume 4968. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pages 33–47. ISBN 978-3-540-68978-2. doi:10.1007/978-3-540-68979-9_3 (cited on page 16).

[Tar10] Christopher Tarnovsky. *Hacking the Smartcard Chip*. Presentation at Black Hat DC 2010 conference. 2010. `http://www.blackhat.com/html/bh-dc-10/bh-dc-10-archives.html` (cited on page 9).

[WTW14] Samuel Weiser, Ronald Tögl, and Johannes Winter. “Measured Firmware Deployment for Embedded Microcontroller Platforms”. In: *MeSeCCS Proceedings*. in press. SCITEPRESS, 2014 (cited on page 73).

[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*. ACM. 2008, pages 21–30 (cited on page 16).

[Win09] Johannes Winter. *Eavesdropping Trusted Platform Module Communication*. Presented at 4th European Trusted Infrastructure Summerschool (ETISS) 2009. July 2009. `http://embedded.iaik.tugraz.at/` (cited on pages 8, 9).

[Win11] Johannes Winter. *A Hijacker’s Guide to the LPC Bus (presentation slides)*. Slides presented at EuroPKI 2011 workshop. Available online at: `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=203846`. 2011 (cited on pages 12, 19, 48).

[WD12] Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to the LPC Bus". In: *Public Key Infrastructures, Services and Applications*. Edited by Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul. Volume 7163. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pages 176–193. ISBN 978-3-642-29803-5. doi:10.1007/978-3-642-29804-2_12 (cited on pages vii, xi, 4, 8, 9, 11, 23, 29, 39, 41, 47, 49, 50, 52, 55, 79, 80).

[WD13] Johannes Winter and Kurt Dietrich. "A Hijacker's Guide to Communication Interfaces of the Trusted Platform Module". In: *Computers & Mathematics with Applications* 65.5 (2013), pages 748–761. ISSN 0898-1221. doi:10.1016/j.camwa.2012.06.018 (cited on pages xi, 4, 8, 9, 11, 23, 29, 30, 39, 41–43, 46, 47, 73, 76).

[Win+12] Johannes Winter et al. "A flexible software development and emulation framework for ARM TrustZone". In: *Trusted Systems*. Springer, 2012, pages 1–15 (cited on page 16).

[WR09a] Rafal Wojtczuk and Joanna Rutkowska. "Attacking intel trusted execution technology". In: *Black Hat DC* (2009) (cited on page 7).

[WR09b] Rafal Wojtczuk and Joanna Rutkowska. "Attacking SMM memory via Intel CPU cache poisoning". In: *Invisible Things Lab* (2009) (cited on page 7).

[WRT09] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. "Another way to circumvent Intel trusted execution technology". In: *Invisible Things Lab* (2009) (cited on page 7).