Authenticated booting for L4

Bernhard Kauer

kauer@os.inf.tu-dresden.de

November 16, 2004

Contents

1 Introduction **3**

2 Fundamentals **4**
2.1 Basics 4
2.1.1 Authenticated vs. Secure Booting 4
2.1.2 Sealed Memory 4
2.1.3 Environment 4
2.2 Trusted Computing Group: TCG 5
2.2.1 Trusted Platform Module Main Specification 5
2.2.2 TPM - General Layout 5
2.2.3 PCR 5
2.2.4 Keys 6
2.2.5 Attestation 6
2.2.6 Others 6
2.3 Related Work 7
2.3.1 TrustedGRUB 7
2.3.2 TCPA Device Driver for Linux 7
2.3.3 Enforcer 7
2.3.4 Authenticated booting with Linux 7
2.3.5 AEGIS 7

3 Design **9**
3.1 Authenticated Booting 9
3.2 The traditional approach 9
3.2.1 Secure booting - the idea 9
3.2.2 Adopting to Authenticated Booting 9
3.2.3 Multiple layers 10
3.2.4 Sealed Memory 10
3.2.5 Summary 11
3.3 The TCG approach 11
3.3.1 Authenticated Booting 11
3.3.2 Secure Booting 12
3.3.3 Sealed Memory 12
3.3.4 Problems 12
3.4 The hybrid approach 12
3.4.1 The idea 12
3.4.2 Linking the two 13
3.4.3 Sealed Memory 13
3.4.4 Summary 13
3.5 Other questions 13
3.5.1 Refreshing keys 14
3.5.2 BIOS Extension ROM 14

4 Implementation **15**
4.1 System Components . . . 15
4.1.1 Basic System . . . 15
4.2 Implementation steps . . . 16
4.3 GRUB - the bootloader . . . 16
4.3.1 stage1 . . . 16
4.3.2 start of stage2 . . . 17
4.3.3 stage2 . . . 17
4.3.4 Race Condition . . . 17
4.4 InfTPM - a device driver . . . 17
4.4.1 polling and waiting . . . 17
4.4.2 abort . . . 18
4.5 STPM . . . 18
4.5.1 Interface . . . 18
4.5.2 Porting device drivers to L4 . . . 18
4.5.3 libtcg - the user mode library . . . 18
4.6 VTPM . . . 18
4.6.1 Interface . . . 18
4.6.2 Internals . . . 19
4.7 Implementation Status . . . 19

5 Evaluation **20**
5.1 Trusted Computing Base: TCB . . . 20
5.1.1 Components of the TCB . . . 20
5.1.2 Reasons . . . 20

6 Summary **22**
6.1 Future work . . . 22

A Appendix **23**
A.1 Glossary . . . 23

Chapter 1

Introduction

With the increasing integration of computers into worldwide networks, the question of trustworthiness of computing devices and the included software is refreshed. Trusted computing stands for technologies that want to build computer architectures, in which trust could be established.

Authenticated Booting is such a technology. It logs, which software is started on a computer and allows to verify that a particular program is securely running.

Current available designs for authenticated booting systems either require new hardware or lack the support of starting untrusted applications.

This work has the goal to implement authenticated booting for microkernel based OS's and to extend the trust provided by authenticated booting to userlevel servers and applications. The design is implemented on a L4 microkernel. This give the opportunity to build small secure components, which run in parallel to big untrusted components like a complete Linux system [17].

Outline

The next chapter is about fundamentals needed for this work and related work already done in the field of authenticated booting. Chapter 3 introduces two different approaches of authenticated booting and gives a solution to combining them. Chapter 4 illustrates implementation issues. Evaluation of security aspects is presented in chapter 5. At the end a summary including future work areas is given.

Chapter 2

Fundamentals

2.1 Basics

2.1.1 Authenticated vs. Secure Booting

There exist two types of booting a system in a trusted manner: authenticated and secure booting.

Secure booting compares the hashes of loaded modules with a list to decide whether they are authorized to run them or not. If the hash is not found on the list, the boot process is stopped. This means only particular operating systems can be started at this platform.

Authenticated booting systems log only the hashes of the files, which are loaded. Remote entities decide whether the system is trustworthy or not. This allows to boot every operating system on the platform.

Secure booting could be used to build secure closed platforms, where the software running on the platforms is limited.

Authenticated booting enables the developing of secure open platforms[1] and give the choice to the end user whether to trust a system or not.

2.1.2 Sealed Memory

Sealed memory is based on the idea that programs could read and write private memory. No other program could access or tamper it. Only if for example the right operation system is loaded, the data is readable.

Although sealed memory could be build in hardware[2] a software solution is often preferred. Software based sealed memory decrypts data with an asymmetric key. The data is only revealed to the program if the hash of the program is equal to the hash of the program running while encrypting the data.

2.1.3 Environment

L4 [19] is a microkernel interface initially developed by Jochen Liedtke [18]. There exist different implementations like Fiasco [10] and L4Ka::Pistachio [16]. This work is independent from the implementation but we use Fiasco in a DROPS environment.

The DROPS (Dresden Realtime OPerating System) [8] project wants to build a system were real-time and secure applications run in parallel to non real-time and to insecure applications without influencing real-time and security properties of each others.

The microkernel approach results in a client-server architecture. Small servers with a clean interface provide the functionality, which is normally embedded into a monolithic kernel. The verification of security properties of the microkernel and the small servers is easier as in big

[1] open platforms are modifiable and the source can be disclosed
[2] using a key into an associative memory

monolithic systems because of a smaller code-base and cleaner interfaces. This advantage makes microkernel-based systems suitable for security-critical applications.

2.2 Trusted Computing Group: TCG

The Trusted Computing Group (TCG) [26] was founded in 2003 as a group of computer manufacturers with the mission to define specifications how trusted platforms should be built. It is the successor organization of the Trusted Computing Platform Alliance (TCPA), which was founded in 1999 with the same goals. All major vendors of computer hard- and software are members [25] of the TCG.

2.2.1 Trusted Platform Module Main Specification

The TPM main specification version 1.2 was published in November 2003 and consists of 4 parts. They define design, data structures and commands of a Trusted Platform Module. However the fourth part of it (Compliance) is not yet published.

The available TPM chips support only the predecessor version 1.1 of this specification. Because the version 1.2 is mainly an extension to the old one, in the following the new specification is used. For changes between them see [27].

2.2.2 TPM - General Layout

The Trusted Platform Module is a hardware chip [29] that consists of:

- Execution Engine
- Cryptographic Coprocessor
- Random Number Generator
- Non-volatile Memory
- Volatile Memory

The execution engine is the core of the TPM. It executes code — the TPM commands.

The cryptographic coprocessor provides cryptographic routines to the execution engine like RSA [15] key generation, RSA de- and encryption and calculation of SHA1 [6] hashes.

The Random Number Generator RNG is internally used to create nonce values[3] or while creating RSA keys. It can also be used outside the TPM to get random data or to create symmetric keys.

Non-volatile memory is used to store keys and internal state. A small part of it is available from the outside.

Volatile memory hold state of the TPM that must not persist after a platform reset. This state contains for example the Platform Control Registers (*PCR*s), session data and flags. Most of the memory is shielded, which means that it cannot be read or written from the outside.

2.2.3 PCR

The Platform Control Registers *PCR*s are 160 Bit wide registers in the TPM that hold a SHA1 hash. They could be used to implement authenticated booting, see section 3.3. At least 16 *PCR*s are available in a TPM.

These registers cannot be written. Instead they can only be modified using the *extend* operation. The $extend(x)$ operation calculates the new value of a PCR as a SHA1 hash $H()$ of the concatenation of the old value and x:

[3]random data in messages to prevent replay attacks

$$PCR(t+1) = H(PCR(t)||x).$$

This operation is not associative because $H(A||H(B||C))$ is in general[4] not equal to $H(H(A||B)||C)$. Thus a given hash value represents the sequence of *extend*() operations used to create it.

Prior *extend*()s could not be revoked by later ones. This means the prefix of such a extend chain could not be changed afterwards.

The extend operation allows storing of a hash value of a chain of hashes into a single register. This could be used for authenticated booting to store a hash of the chain of loaded programs.

2.2.4 Keys

The TPM uses several asymmetric keys.

The Endorsement Key (EK) is a 2048 Bit RSA key stored in the TPM. It identifies the TPM in a unique way. A certificate of the manufacturer ensures that the key is a key of a valid TPM. The private part of the EK is used only to decrypt data send to the TPM. This happens for example when creating new identities.

The EK never leaves the TPM, but since specification 1.2 it is possible to revoke the EK and create a new one. To use the new EK an expensive manufacturer certificate is needed. This makes sure that revoking the EK will be rarely used.

The Storage Root Key (SRK) is used as the root of the key hierarchy. It never leaves the TPM, but a new one could be created by the owner of the TPM. If a new one is created all derived keys and the entire sealed memory content is lost.

An identity or Attestation Identity Key (AIK) aliases the EK and is a child[5] of the SRK. It is used for attestation purposes. Multiple AIKs could be created. Different identities could be used for different remote entities to be anonymous between them.

2.2.5 Attestation

Attestation is normally done by signing data, including *PCR* values, with an asymmetric key that could be verified at the other side. The *PCR* values are used to identify the software stack running on the computer. To prove that the signing key is trustworthy (at least that it belongs to a TPM) a certificate is checked. The EK and the manufacturer certificate could not be used here, because they provide no anonymity which is often needed.

Certificates could be generated in 2 ways. The old way[6] is based on a trusted third party (TTP). The TTP get a public identity key, the manufacturer certificate and the public endorsement key. It checks the public EK with the manufacturer certificate and issues a certificate that proves that the public identity key belongs to a TPM. The TTP hides the relation between the EK and the identity, making the identity anonymous.

The new way[7] called Direct Anonymous Attestation (DAA) is based on a zero-knowledge group-signature scheme [5]. The main idea is that the certificate is not send to a verifier. Instead it could be proved that such a certificate exists, without revealing it [3, 4]. With this a certificate could be made anonymous in the group of certificates issued by the same certification issuer. The issuer could be the same as the one who verifies that such a certificate exists. Even publishing the certificates is useful to check that the group is large enough.

2.2.6 Others

There are several things in the TPM main specification, which are not used for authenticated booting, like Ownership, Delegation, Monotonic Counters or Maintenance. They are described in detail in the TPM specification [29].

[4] the hash is collision resistant

[5] a child key is encrypted with the parent key when it is outside the TPM

[6] available since version 1.0

[7] new in version 1.2

2.3 Related Work

2.3.1 TrustedGRUB

Developed at the Ruhr-University Bochum as a bootloader for PERSEUS [23] TrustedGRUB [28] extends GRUB (GRand Unified Bootloader) [12] with secure booting. It stores hashes of the commands given to GRUB and all files loaded by GRUB (kernel and modules) into the TPM.

It could check hashes of files on the hard disk against a given list. The checking of files at boot time is a feature that overloads a bootloader because of the following point. If a secure system is loaded, which could protect the integrity of files, it is possible to check them after booting for example in the init phase. If the secure system could not protect files against bad program's it is useless to check them at boot time because they could be changed later.

TrustedGRUB version 0.7.1 contains the same race condition like the GRUB extension implemented in this work (see section 4.3.4), which makes it possible to load other data than hashed.

2.3.2 TCPA Device Driver for Linux

A open source device driver for Linux was published [13] by IBM in August 2003. It is a small driver that defines a clean user mode interface via a device named /dev/tpm. It works with the Atmel TPM AT97SC3201.

Because of a different interface of the Infineon TPM, this driver cannot be used with it.

2.3.3 Enforcer

Enforcer [9] is a Linux security module initially developed at the Dartmouth College [20, 21]. It monitors files in a Linux system and checks whether a file has changed by comparing the SHA1 hash of it with a database of file hashes build by the system administrator.

Enforcer uses the TPM to seal a secret key of an encrypted loopback filesystem. It use a modified LILO as TPM-enabled bootloader that fits on a floppy but not in the MBR of a hard disk[8].

2.3.4 Authenticated booting with Linux

In [24] an implementation of authenticated booting in a Linux system is presented solely based on a TPM. All started programs are hashed into a *PCR* for the first time they are used or if they are changed on disk.

It suffers from all problems related to the exclusive usage of a TPM (see section 3.3.4). For example with this design it is not possible to load untrusted applications or self compiled ones without losing the ability for a successful authentication.

Every change of the memory of a running process from another is forbidden in this design. But some ways of changing the memory like */proc/PID/mem* or the *ptrace*-systemcall used by debuggers are not mentioned there.

There is also a race condition detecting writes to a file. If a file is changed a flag is set to indicate that is has to be checked at next execution. But if a running process accesses a page which needs to be reloaded from disk[9] it could get a modified version.

2.3.5 AEGIS

In AEGIS [2] secure booting is implemented with a changed BIOS. Starting from the basic layer, BIOS section1, which is in a ROM and assumed to be trusted, all other layers are hashed and verified with a stored signature. The trust in this system depends on the integrity of the first section of the BIOS and a trusted PROM where signatures are stored.

[8] A floppy MBR does not contain a partition table. So there are 64 Bytes more available for code.

[9] because it is not in the page cache

AEGIS is on of the first systems implementing a recovery mode. It seems to be useless to recover BIOS extension ROM data from extension cards or recovering the MBR, because if the verification of this code failed, stored code from the trusted PROM is used. It is simpler to use always code from trusted memory.

sAEGIS [14] is an extension to AEGIS that implements secure booting where the needed trust in the system administrator is reduced. This reduce of needed trust is achieved by using a smart card to store certificates and hashes of trusted systems. This gives the choice to the user, whether a system component is trusted or not.

Both projects do not support authenticated booting and are based on a modified BIOS, which prevent a wider usage of them.

Chapter 3

Design

3.1 Authenticated Booting

There are two different solutions to implement authenticated or secure booting. The traditional one, based on asymmetric cryptography, and the solution from the TCG, which is based on *PCR*s. After a short introduction into both I discuss a hybrid solution based on both ideas.

3.2 The traditional approach

In March 1991 Michael Gross presented a solution [11] for secure booting based on asymmetric cryptography. In the following I resume the main idea and present small enhancements.

3.2.1 Secure booting - the idea

The CPU contains an asymmetric key pair (K_{CPU},PK_{CPU}) that is only available during the boot process. After that it must be disabled until the next reboot.

After loading, but before giving control to it, a certificate of the OS is checked with the public key PK_{OS} of the manufacturer. If it is incorrect, the boot process is stopped. Otherwise a "loaded-OS" asymmetric key pair (K_{LOS},PK_{LOS}) is created and a boot certificate (ID_{OS},PK_{LOS},PK_{OS}) is signed with K_{CPU}.

The "loaded-OS" key, the boot certificate and a certificate of the CPU is given to the operating system. With this the OS could attest to remote entities that it was booted in the right way by checking the boot certificate with the public CPU key and the CPU key with the certificate of the CPU.

The root of trust in this system is based on the assumption that the CPU key is never revealed or that it could be guessed.

3.2.2 Adopting to Authenticated Booting

We could adopt this idea to authenticated booting in the following way:

- do not check the certificate of the loaded OS
- use the hash H_{los} of the loaded OS instead of the id in the boot certificate

This means only (H_{los},PK_{los}) remain as the boot certificate. The hash could be used to find an OS manufacturer certificate stored in a public database.

Neither signing of every bootable OS nor the checking against a master certificate in the CPU is needed anymore. This assures that every operating system could be loaded.

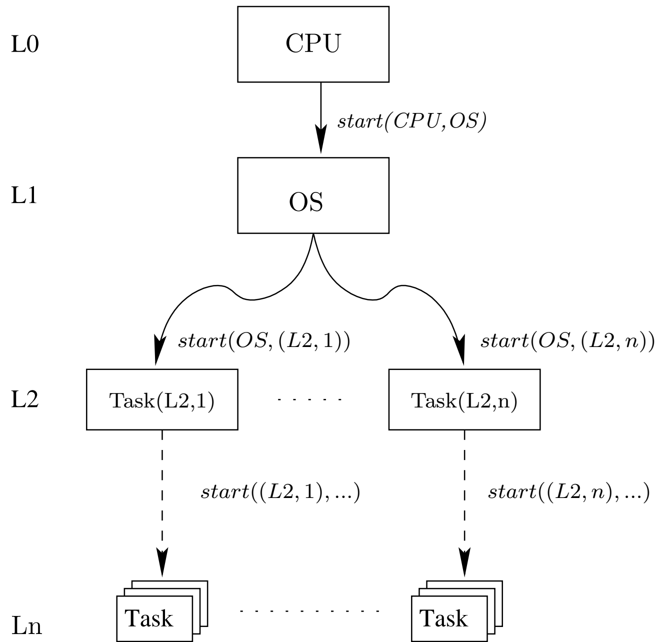

Figure 3.1: Booting Multiple Layers

3.2.3 Multiple layers

This general idea could be chained and applied to a layered model.

To boot a program in the layer L_{n+1}, a Hash H(L_{n+1}) is calculated by the layer L_n, which identifies the program in a unique way. An asymmetric key pair (K(L_{n+1}),PK(L_{n+1})) is created and the boot certificate for this program H(L_{n+1}),PK(L_{n+1})) is signed with K(L_n). The key and all boot certificates of parent layers are given to the program.

Figure 3.1 shows Multiple Layers using a start(Parent-Layer,Child-Layer) function for the above calculations. To authenticate a single program the authentication of every parent layer has to be checked. The trust in a program is independent from the trust in sibling programs in this tree of authenticated booted programs, provided that the parent implements proper isolation of the sibling programs.

Every layer must protect the storage of their own key against programs running in higher layers. If a child layer replaces a parent one, for instance an OS replaces the bootloader, it must be ensured that the key of the lower layer is not available anymore.

The root of this chain of trust, the layer 0, could be the CPU, which is certified by the manufacturer of the CPU.

A similar idea named *chaining layered integrity checks* was discussed in [1].

3.2.4 Sealed Memory

How could sealed memory be implemented with this approach?

To seal data for a program in layer L_{n+1} the hash of the program is added in front and the whole block is encrypted with the public key PK(L_n) of the parent layer. The data could be decrypted at startup and given to the program if the beginning of the block matches the hash H(L_{n+1}) of the loaded layer. To make sure the sealed memory was created by L_n the encrypted memory could be signed with K(L_n).

The initial sealed memory could be small because at most a symmetric key should be stored there. This key could be used to encrypt a larger amount of memory for example other keys or arbitrary data. By using sealed memory in this way, every data or hard disk could easily be sealed.

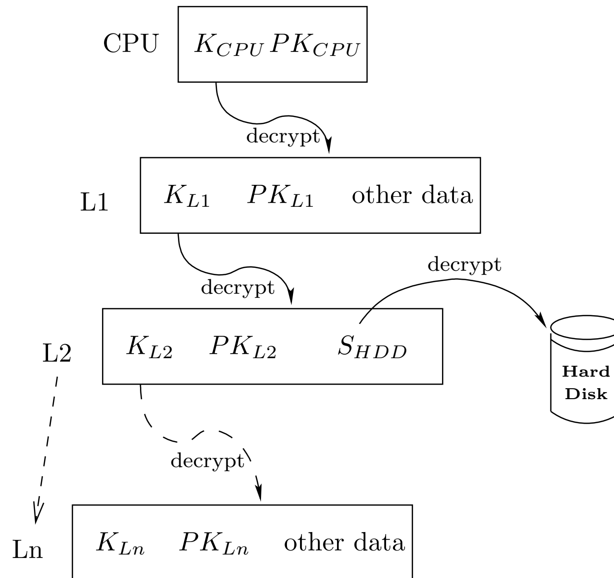

Figure 3.2: sealed memory in the traditional approach

The sealed memory could be stored everywhere for example on a flash chip or on a hard disk.

3.2.5 Summary

The main problem of this approach is the needed CPU extensions that include key storage and functions to sign, encrypt and decrypt. It cannot be solved without new hardware.

Another problem with this approach is the time consuming creation of asymmetric key pairs every time the system boots, which significantly increase the boot time. To circumvent this, the "loaded-OS" key is created once and could be stored encrypted and signed in sealed memory, only readable with the CPU key.

The advantage of this approach is the ability to split a system in compartments that could be independently authenticated.

3.3 The TCG approach

The TCG (see section 2.2) defines a way to implement authenticated booting based on *PCR*s in a TPM. In this section I describe their approach shortly.

3.3.1 Authenticated Booting

Authenticated Booting works in a simple way. A *PCR* (see section 2.2.3) is extended by a hash of every loaded program before it is executed. So the chain of all loaded programs could be reconstructed with the values in the *PCR*s.

Because a *PCR* could be extended(see section 2.2.3) an unlimited number of times only one *PCR* is normally needed. But the TCG defines an assignment that split the input to 8 registers. For example *PCR* 0 is extended with the hash of the BIOS and *PCR* 4 is extend with the hash of the MBR.

For remote attestation the *PCR* values are signed with a key and sent together with certificates that allow to trust this key. The remote entity checks the signature, reconstructs the chain of extend operations. After that it tries to calculate the same *PCR* value and decides whether it

trust all programs in this chain. If nothing fails, the remote attestation was successful and the computer is trustworthy.

To simplify the reconstruction of the extend chain, a log could be written into insecure memory that contains the hash of the measured program and other information like the name or version of it. This log is used as a hint for reconstructing the chain during a remote attestation.

3.3.2 Secure Booting

Secure booting is here nearly the same as authenticated booting. In addition on every boot step the hashes of the modules or the values of the *PCR*s could be compared with securely stored[1] values. If they mismatch the boot process is stopped.

3.3.3 Sealed Memory

To seal data a set of *PCR*s is added in front before encrypting the data with a key. After decrypting, the data is only released outside the TPM if the *PCR*s are the same as the stored ones. This allows to bind the access to private data to a specific platform configuration.

3.3.4 Problems

To attest a system where every program execution is measured into a *PCR* the whole log of program hashes have to be traversed and checked against a database to verify that the *PCR* value is correct. A single unknown program, for instance a self compiled one, could easily make the whole chain of hashes not verifiable.

With this approach, the trust in a program of a multi user system depend also on the programs executed by other users.

Using *PCR*s to store hashes could not protect the privacy of a user. Why should a remote entity know all software that is and that was running on a trusted client? A remote entity should only get minimal information. It should only get to know the programs that had control over the secure program, with which they are communicating.

The TCG solution for authenticated booting, providing *PCR*s, is suitable for the booting of the first parts of a system. At system booting time programs execute in a strict sequential order. But after that many programs run in parallel and thousands of processes could be started in the uptime of a system. Hashing all of them into the TPM *PCR*s is not acceptable.

Instead compartments of trusted programs are needed, which the trust in them could be reported independently. But this cannot achieved with the TCG approach.

3.4 The hybrid approach

The hardware problem of the traditional approach and the inflexibility of the TCG approach lead to the idea to combine them into a hybrid system.

3.4.1 The idea

Until the basic system has started, which is booted in a strict sequential order by the bootloader, the system is booted with the TCG approach[2]. After that the system switches to chaining the traditional approach, which lets us boot a tree of programs without interfering the attestation data of different paths in this tree.

Now there are two subsystems for authenticated booting: the first, based on a TPM and the second based on a “loaded-OS” key. To attest that an application is running securely at the system, the attestation of both is needed. The first subsystem makes clear that a trusted OS is loaded and the second what application chain is running on top of it.

[1] in non-volatile memory of the TPM

[2] The traditional one does not work without hardware support and the TCG one is sufficient for this.

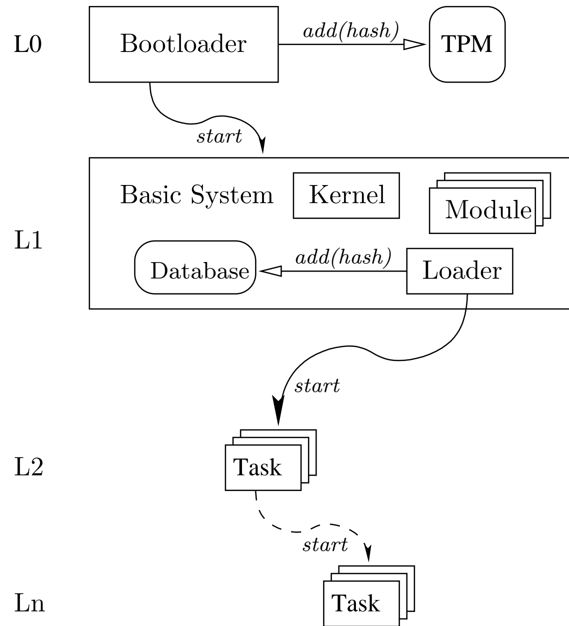

Figure 3.3: The hybrid approach

3.4.2 Linking the two

How to assure that the second subsystem runs on top of the loaded OS and is not independent from it? A cryptographic link between the two different subsystems is needed.

This could be done for example by issuing a certificate for the public OS key with a key belonging to the TPM. But it is simpler to extend a *PCR* with a hash of the public OS key. This makes the check, whether the TPM and the public key are accessible on the same machine easier. Only a comparison of the hash of the public OS key with the value in the *PCR* is needed.

3.4.3 Sealed Memory

In the hybrid approach every layer provides sealed memory. Either by using the TPM in the first subsystem or by using encryption with an asymmetric key in the second subsystem.

If the asymmetric key of the layer is constant between different boot cycles it could be used to encrypt the data. Otherwise a second but constant key is needed in every layer.

3.4.4 Summary

Using the TPM for authenticated booting of the first basic part of the system, resolves the hardware problem of the traditional approach. By using asymmetric cryptography for the tasks started on top of the basic system, the hashes could be independently reported and booting of untrusted and self compiled applications is possible.

In summary the hybrid approach takes the best from both ideas and makes authenticated booting usable with current hardware up to the application layers. Thus this approach was chosen for further work.

3.5 Other questions

In the following I discuss the questions whether keys should expire and the open problem of BIOS Extension ROMs.

3.5.1 Refreshing keys

Is it necessary to give the asymmetric keys an expiration date and to refresh them after some time?

An expiration date does not work with the root keys (Endorsement, CPU or vendor key), because refreshing them could be expensive. The keys created for higher layers that depend on the root keys could have an expiration date. Creating new keys is possible if the access to the parent keys is permitted.

The weakest point in increasing security by expiration of keys is that the roots of a key hierarchy could not be protected in this way. Strengthening of child keys does not help if the root key is attackable.

3.5.2 BIOS Extension ROM

BIOS Extension ROMs come from flash memory or ROM on PCI or ISA cards, like graphic cards or SCSI controllers, which extend the BIOS for example by installing new BIOS interrupt handlers. They are security critical because they gain control of the hardware from the BIOS before the MBR is loaded.

What to do with these extensions?

By changing the memory of a card, any code could be inserted into the boot sequence. Therefore the TCG defines that all extension code must be hashed into *PCR* 2 before it is used. This solution is secure, if in addition the extensions hash all memory they load.

The main disadvantage is that if a card is changed, the sealed memory and all keys are lost, unless a card with the same ROM is used. This applies also to inserting new cards or to updates of card memory.

Another solution is presented in [2]. Only the extensions of which the hashes are known to the BIOS, could be used there. This list should not be static, because new cards and updates to old cards are available after a short time. But how to update this list of known hashes? If the prerequisites for an update are to low, like proving only physical presence, simple hardware attacks could happen. If they are to high, like sending the whole computer to the manufacturer, an update could be expensive.

In summary all currently available solutions are dissatisfying, so further research is needed.

Chapter 4

Implementation

After an overview of the system components I discuss implementation issues in detail.

4.1 System Components

What are the needed components until an L4 system could be authenticated booted in the hybrid approach?

A modified boot loader is needed to hash a basic system into the TPM. This makes the first part of the hybrid approach. The second part could be implemented in an L4 server called VTPM (Virtual TPM) from now.

4.1.1 Basic System

A (minimal) basic L4 system consists of the following parts:

- Fiasco - the l4 microkernel
- Sigma0 - the initial pager
- RMGR - the resource manager
- Loader - load new applications
- STPM - access to the TPM
- VTPM - the second subsystem

The first three are used to provide the basic functions of an L4 system.

The Loader loads new applications. The Loader of DROPS needs a file provider[1] and a dataspace manager[2] to load an application. Both should be part of the basic system, too.

STPM is used to access the TPM. VTPM is used to implement the traditional approach, which means it stores the hashes of loaded programs and outputs signed hashes and certificates.

The whole basic system could be loaded and hashed by the bootloader. STPM and VTPM had to be written from scratch and the Loader had to be modified to call VTPM before starting the loaded application. All other components could be used unchanged.

[1] The file provider supplies the file to load.
[2] The dataspace manager is used to get the memory for the application to load.

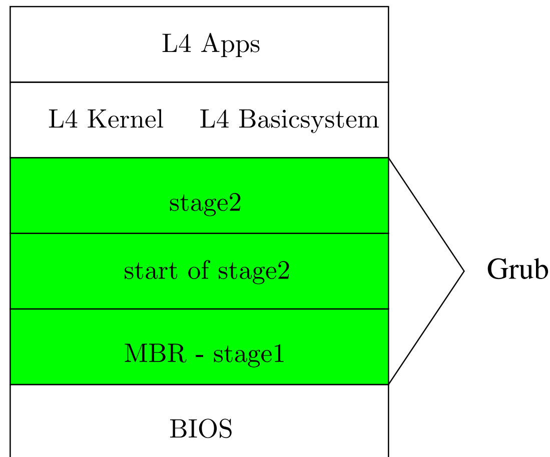

Figure 4.1: Boot sequence with GRUB

4.2 Implementation steps

The steps in implementation were:

- enhancing the bootloader to call the TPM
- implementing a driver for a TPM and porting it to L4
- extending a TPM library
- implementing the VTPM server

In the following issues and details of the implementation are mentioned.

4.3 GRUB - the bootloader

A modified GRUB [12] is the bootloader used to load the basic servers of DROPS. It is extended in this work to support authenticated booting.

The boot sequence with grub is shown in Figure 4.1. The BIOS loads the MBR from the boot device, which is the stage1 of GRUB. Stage1 loads the first sector of stage2 called start. Start than loads the remainder of stage2 from a list of sectors. Stage2 provides a menu interface to select the kernel or the multiboot [22] modules to load. It loads them from a disk or from the network and gives control to the kernel.

To support authenticated booting in GRUB all loaded code must be hashed before it is executed.

4.3.1 stage1

The main problem in stage1 is the size limitation. Only 20 bytes are available in stage1 to hash the start of stage2 and extend a *PCR*. This could be enough but the TCG defines an impractically interface so that at least 80 bytes are needed.

Instead of using a simple BIOS interrupt with parameter transfer in registers, the parameters are transfered in a big memory block. Independent from this the function *TCG_HashLogExtendEvent()* does not always work, so two calls to the BIOS are needed. First the data is hashed with *TCG_HashAll()* and after that *TCG_PassTroughToTPM()* is called to extend a *PCR*.

Stage1 could boot in two modes: in CHS mode, which today is only needed for booting floppies and in LBA mode, which is needed to boot from a hard disk. To circumvent the size problem these two modes are separated into two different versions of stage1. This could only be a workaround until a better BIOS interface is available, because it has the disadvantage of increasing the maintenance time of this code.

4.3.2 start of stage2

The implementation of hashing stage2 in the start sector of it is straightforward and similar to stage1. So the same code, with a little preface to adopt to the changes in stage2, is used.

4.3.3 stage2

To support TCG commands a function *TCG_call()* was written in assembler, which calls the BIOS in real mode. Because GRUB uses linear memory[3] in protected mode, memory copy is not needed. So this function could be used as a gate to all other TCG BIOS functions like *TCG_HashAll()* or *TCG_PassTroughToTPM()*. After that the implementation of hashing the kernel, initrd and modules was easy.

No command line of grub is hashed in stage2. This omission is justifiable because most of them are not security critical to the system. Only the setting of the next module load address, the *"modaddr"*[4] command, seemed to be a problem. Perhaps this could be disabled in the future or a *PCR* should be extended.

The hashing of boot parameters to modules or kernel was not implemented. The main reason was that a trusted operating system should be trusted independently from its parameters. If it is possible to break into a system or bypass security checks through boot parameters, the kernel should hash the parameters itself before using them.

4.3.4 Race Condition

Due to the bad modularization of GRUB there is a race condition in the code. Every kernel and module is hashed and loaded. This job is done by reading the complete file twice, first for hashing and than for the real loading. This makes it possible to change the file between the two reads. For example a bad TFTP server, which is used for network booting, could easily deliver two different files for the two reads. To avoid this, rewriting of large parts of GRUB code is required.

4.4 InfTPM - a device driver

The mainboard of the test system used contains an Infineon TPM. At the time of this project only a windows driver existed. Therefore I write InfTPM, a device driver for the Infineon TPM for Linux 2.6.

Different layers (transport and vendor layer) and many registers on different levels (pci, dev, io) makes the driver more complex than the IBM driver (see section 2.3.2).

It is written with the same user interface as the IBM device driver, so porting of it to L4 was easy.

4.4.1 polling and waiting

The driver does not yet use interrupts. Instead a status register in the chip must be polled to decide when data could be send or received. Because polling consumes to much CPU time, it waits a longer time after every 10 polls. This results in an overhead of 25% according to plain polling but with much less CPU consumption.

[3] virtual and physical addresses are the same

[4] an extension for DROPS

4.4.2 abort

The asynchronous abort of a command sent to the TPM is not implemented. The strategy to get this to work is a global variable in the driver that could be checked in the loops. The state of the low level communication with the TPM, which is completely synchronous, should be considered before implementing this command.

4.5 STPM

4.5.1 Interface

STPM is an L4 server used to access the TPM and encapsulate the drivers for it. At least 2 commands are needed from the TPM:

- *TPM_Extend()* - to extend a *PCR* to link the subsystems (see section 3.4.2)
- *TPM_Quote()* - to get the signed *PCR*s

For key handling (create, load, evict) or sealed memory (seal, unseal) much more commands are used from the TPM. So a lower level interface with read/write semantic will be implemented. The higher-level code could be written as a library.

The interface of STPM is quite simple. Only 3 functions are needed. The *stpm_read()* and the *stpm_write()* provide the read/write calls of the Linux device. The *stpm_abort()* is mapped to an ioctl of the drivers.

4.5.2 Porting device drivers to L4

Porting of the Linux device drivers for Atmel and Infineon TPM's was, thanks to the L4 Device Driver Environment DDE [7], extremely fast. Less than 200 lines of glue code were needed to integrate them. Nearly the same number of lines were needed to build an L4 server named STPM and a user library named libstpm that communicates with it. So less than 400 lines of code were needed to make a Linux device, provided by two drivers, available under L4.

4.5.3 libtcg - the user mode library

There are many TPM commands defined. To use them in normal applications a library named libtcg was written by myself. It is based on libtcpa from IBM, but main parts are rewritten or extended. By introducing a set of macros the code duplication and the code size could be significantly reduced.

Due to a bug in the authentication of commands, which works with Atmel but not with the Infineon TPM, all commands that require a key, like *TPM_CreateWrapKey()* or *TPM_Quote()*, fail with Infineon chips, until now.

4.6 VTPM

VTPM is an L4 server that implements the second subsystem. In the following the interface and some internals of it are presented in detail.

4.6.1 Interface

The second subsystem should be implemented in a single L4 server. Therefore a interface is needed that could be used by the loader to store the hashes of loaded applications or by applications to ask for remote authentication.

The interface contain at least two functions: *add()* to add hashes and *quote()* to get authentication data. A method to establish a cryptographic link between another subsystem or another

layer with the VTPM is useful to implement multiple layers: *link()*. Because applications are not running endless a function to delete a hash *del()* is required, too. To implement sealed memory a function *seal()* to encrypt data and a function *unseal()* to decrypt data is needed.

This interface of 6 functions contains everything needed to implement authenticated booting and sealed memory with different layers. It could be used with a TPM, with the exception that *del()* is not possible with it[5].

4.6.2 Internals

The implemented VTPM uses RSA to sign the authenticated data. The hashes are stored in a single-linked list because the search of hashes is not time critical. To every hash a name and a parent of the application is stored. The path traversal in the tree of hashes is not done in *quote()*, so every client has to implement this functionality themself.

To use the VTPM in a productive system DROPS need a simple way to store persistent data. With it TPM key handling, taking of ownership and sealed memory could be implemented.

4.7 Implementation Status

By the time this document was written, basic parts are completed. The system boots with the modified GRUB. The STPM server could be used to access Atmel and Infineon TPM's. The VTPM could be used to use authenticated booting of applications. A prototype of a Loader, using the libtcg to extend a *PCR*, and a python program on a PDA, sniffing at the serial line for *TPM_Quote()* output, work successfully with an IBM Thinkpad T30.

The open implementation issues are the mentioned bug in the authentication of commands with the Infineon TPM, a missing random number source for nonces and the key handling in DROPS.

[5] *PCR*s are not resetable.

Chapter 5

Evaluation

The focus of evaluation is on the security aspects of this work. Execution-time aspects are nearly negligible because booting a system or starting applications takes a long time so that hashing of the code introduces only a small overhead [2].

5.1 Trusted Computing Base: TCB

To enforce a security policy a set of hardware and software, called the Trusted Computing Base (TCB), is needed.

5.1.1 Components of the TCB

If a remote entity want to trusts a quote for an authenticated booted application, it has to trust the following components:

- TPM and the TPM keys
- bootloader
- microkernel
- memory managers
- loader
- STPM
- VTPM.

These components are the parts of the TCB required for reporting authenticated booting in our L4-based system.

5.1.2 Reasons

Normally a component will be trusted to work in a proper way. What are the work the previous mentioned components have to do correctly?

The TPM must provide extending *PCR*s and provide a quote output of them. The keys of the TPM must be kept private. The bootloader must hash the right values into the TPM. The microkernel must provide distinct address spaces for applications. And it must provide secure communication where no third program could listen or change the message. The memory managers[1] must make sure that memory is kept private to an application and that it is not changed

[1] e.g., sigma0, rmgr and DMphys

by other applications. The loader must hash the right memory and report the right values to the VTPM. The STPM must extend a *PCR* on request of the VTPM. The VTPM must initialize correctly and it must report the right hashes.

In summary every program that could access the memory of the TCB components have to be trusted. But shared memory regions between two programs could be allowed if the code and the private data of the trusted component is protected.

So utility applications like a 'file provider', which loads and gives a file to the loader or a decoder for a binary format, which only extracts the file, are not part of the TCB. Also a network stack or a hard disk driver is not part of the TCB, because modifications of the data are detected in higher layers.

Chapter 6

Summary

In this work I present a hybrid approach for authenticated booting (see section 3.4). It avoids the problems of the TCG (see section 3.3) and the traditional approach (see section 3.2) and allows a system architecture where authenticated booted applications attested independently from each other.

I implemented (see section 4) the basic parts of it in an L4 environment and worked out a common interface (see section 4.6.1) for multi-layered authenticated booting and sealed memory.

6.1 Future work

A bridge between a real TPM and the VTPM interface should be implemented to provide a consistent interface for applications.

To make the check easier, whether to trust an authenticated booted system or not, a smartcard or small USB stick should be built, that compares the stored hashes of trusted applications with the ones given in the attestation.

Research is needed, to answer the open questions of hardware extension cards mentioned in Section 3.5.2. Another field of research is: Instead of using a full TPM for a trusted platform, what are the minimal hardware extensions needed for authenticated booting and sealed memory?

Appendix A

Appendix

A.1 Glossary

AIK Attestation Identity Key; an anonymous alias of the EK

DROPS Dresden Realtime OPerating System [8]

EK Endorsement Key; a 2048-bit RSA key that identifies the TPM; used on identity creation

L4 a second generation microkernel interface [18, 19]

multiboot a standard to load boot modules and give parameters to them [22]

MBR Master Boot Record; the first 512 Bytes of a drive

Nonce random data in messages to prevent replay attacks

PCR Platform Control Register; 160 Bit register in the TPM

RNG Random Number Generator; based in a TPM on the noise of a diode as physical random source

RSA an asymmetric encryption system [15]

SHA1 a 160 Bit hash function [6]

SRK Storage Root Key; the parent key of all created keys in a TPM; could be deleted by the owner

TPM Trusted Platform Module; a small hardware chip

TCG Trusted Computing Group [26]

TCPA Trusted Computing Platform Alliance; the predecessor of the TCG

References

[1] W. Arbaugh. *Chaining Layered Integrity Checks*. PhD thesis, University of Pennsylvania, Philadelphia, 1999.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, pages 65–71, Oakland, CA, May 1997.

[3] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. Technical Report HPL-2004-93, HP Laboratories, Bristol, UK, June 2004. URL: `http://www.hpl.hp.com/techreports/2004/HPL-2004-93.pdf`.

[4] Jan Camenisch. Direct Anonymous Attestation: Achieving Privacy in Remote Authentication, June 2004. URL: `http://www.zisc.ethz.ch/events/ISC2004Slides/folien-jan-camenisch.pdf`.

[5] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In S. Cimato, C. Galdi, and G. Persiano, editors, *Security in Communication Networks, Third International Conference, SCN 2002*, volume 2576 of *LNCS*, pages 268–289. Springer Verlag, 2003.

[6] 3rd D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, September 2001.

[7] DDE - L4 Device Driver Environment. URL: `http://os.inf.tu-dresden.de/L4/`.

[8] Dresden Realtime OPeration System. URL: `URL:http://os.inf.tu-dresden.de/drops/`.

[9] Enforcer website. URL: `http://enforcer.sourceforge.net/`.

[10] The Fiasco microkernel. URL: `http://os.inf.tu-dresden.de/fiasco/`.

[11] Michael Gross. Vertrauenswürdiges Booten als Grundlage authentischer Basissysteme. In *Verläšliche Informationssysteme, Tagungsband*, Informatikfachberichte 271 , Bremen, 1991. Springer Verlag.

[12] GNU GRUB (GRand Unified Bootloader). URL: `www.gnu.org/software/grub/`.

[13] IBM TCPA Resources. URL: `http://www.research.ibm.com/gsal/tcpa/`.

[14] Naomaru Itoi, William A. Arbaugh, Samuela J. Pollack, and Daniel M. Reeves. Personal secure booting. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 130–144. Springer-Verlag, 2001.

[15] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313, March 1998.

[16] The L4Ka::Pistachio microkernel. URL: `http://www.l4ka.org/projects/pistachio/`.

[17] L4 Linux. URL: `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

[18] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[19] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[20] Rich MacDonald, Sean W. Smith, John Marchesini, and Omen Wild. Bear: An open-source virtual secure coprocessor based on tcpa. Technical Report TR2003-471, Dartmouth College, Hanover, NH, August 2003.

[21] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with tcpa/tcg hardware, or: How i learned to stop worrying and love the bear. Technical Report TR2003-476, Dartmouth College, Hanover, NH, December 2003.

[22] Multiboot Specification manual. URL: `www.gnu.org/software/grub/manual/multiboot/`.

[23] Perseus. URL: `http://www.perseus-os.org`.

[24] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*, pages 223–238, IBM T. J. Watson Research Center, August 2004.

[25] Members of the TCG. URL: `https://www.trustedcomputinggroup.org/about/members/`.

[26] Trusted Computing Group TCG. URL: `https://www.trustedcomputinggroup.org`.

[27] TCG. TPM v1.2 Specification Changes. URL: `https://www.trustedcomputinggroup.org/downloads/TPM_1_2_Changes_final.pdf`, November 2003.

[28] TrustedGRUB. URL: `http://www.prosec.rub.de/trusted_grub.html`.

[29] TCG TPM Specification Version 1.2 Design Principles. URL: `https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part1_Design_Principles.pdf`.