Semester Thesis

A Software-based TPM Emulator for Linux

Mario Strasser

Department of Computer Science
Swiss Federal Institute of Technology Zurich

Summer Semester 2004

Supervisors:
Paul E. Sevinç
Prof. Dr. David Basin

Abstract

The *Trusted Computing Group* (TCG) has produced several specifications for trusted computing such as for a security chip, called *Trusted Platform Module* (TPM), and for related software interfaces (*TCG Software Stack Specification* (TSS)).

Although a TPM is probably going to be part of more and more state-of-the-art personal computers, there are and will always be situations where a TPM is unavailable or unaccessible. Furthermore, the TCG and the TPM in particular are controversial and disputed matters. Therefore, the goal of this semester thesis was not to show that TPMs are either good or bad, but to implement a software-based TPM emulator for Linux to give people the means to easily explore TPMs for educational and experimental purposes.

In the course of this semester thesis, about 50 out of 120 ($\sim$42%) TPM commands have been implemented and for the remaining, appropriate function-stubs have been provided. Additionally, a complete TCG Device Driver Library (TDDL) to access the TPM has been created to provide future applications with a suitable and standardized device interface. Despite the limited functionality, almost all available TPM applications work well with the current emulator implementation as the most important and frequent commands are already supported. Currently, the TPM emulator has been successfully tested by the developers as well as by two other people and on three different Linux platforms.

Contents

1 Introduction **1**

1.1 Motivation and Goals 1

1.2 Tasks 1

1.3 Typographic Conventions 2

1.4 Outline 2

1.5 Acknowledgments 2

2 Trusted Computing and Trusted Platform Module Basics **3**

2.1 Trusted Computing 3

2.2 Trusted Platform Module 4

3 Installation and Usage **11**

3.1 TPM Emulator 11

3.2 TCG Device Driver Library 13

4 Implementation of the TPM Emulator **19**

4.1 Concept 19

4.2 Structure 19

4.3 Naming and Coding Conventions 20

4.4 License and Copyright 20

4.5 Kernel Module/Interface 21

4.6 TPM Data Structures 21

4.7 Internal Data 23

4.8 Marshaling and Unmarshaling 25

4.9 Authorization 27

4.10 Cryptographic Functions 28

4.11 Initialization, Self-Test, and Shutdown 29

4.12 Command Execution 30

4.13 Command Summary 31

4.14 TCG Device Driver Library 38

5 Conclusion **39**

5.1 Summary 39

5.2 Outlook 39

A FIPS Tests **41**

A.1 Statistical Random Number Generator Tests [FIPS140] 41

A.2 SHA-1 Test Vectors [FIPS180] 42

A.3 HMAC Test Vectors [RFC2202] 42

B Source and Documentation Files **45**

B.1 CD-ROM Content 45

B.2 TPM Emulator Package 45

B.3 TDDL Package 46

Chapter 1

Introduction

What I cannot create I do not understand.
– R. Feynman

1.1 Motivation and Goals

The *Trusted Computing Group* (TCG) [TCGBG] – successor of the *Trusted Computing Platform Alliance* (TCPA) – has produced several specifications for trusted computing such as for a security chip, called *Trusted Platform Module* (TPM) or *Fritz-Chip*[1], and for related software interfaces (*TCG Software Stack Specification* (TSS)) [TPMPart1, TPMArch, TSS11].

Although a TPM is probably going to be part of more and more state-of-the-art personal computers, there are and will always be situations where a TPM is unavailable or unaccessible. Furthermore, the TCG and the TPM in particular are controversial and disputed matters. Therefore, the goal of this semester thesis was not to show that TPMs are either good or bad (whatever this means), but to implement a software-based TPM emulator for Linux to give people the means to easily explore TPMs for educational and experimental purposes.

Linux was chosen as the target platform for two main reasons: First, almost all currently available TPM-based applications and projects (e.g., IBM's TPM utilities [IBMSW], tcgLinux [IBMTL] or the enforcer project [DCEF]) run under Linux. Second, we are more familiar with device-driver development under Linux than under any other operating system.

1.2 Tasks

- Implementation of a software-based TPM emulator for Linux by means of a Linux kernel module.
- Implementation of an appropriate TCG Device Driver Library (TDDL) to access the emulator.
- Making the device interface of the emulator compatible to IBM's device driver [IBMSW].
- At least supporting Linux kernel release 2.4.

[1] Named after the famous US-senator Fritz Hollings who supports the ideas of the TCG very much.

1.3 Typographic Conventions

- Functions, variables, and constants are set in a mono-spaced typewriter font: `function()`, `data`, `CONSTANT`.
- Shell commands are marked with a leading #:

```
# ls *.c
```

- Code snippets and listings contain colored and highlighted keywords and are printed in a smaller font:

```
/* listing example */
for (int i = 0; i < 10; i++) ...
```

1.4 Outline

This report is structured as follows: Chapter two gives a brief introduction into Trusted Computing and highlights the capabilities of the Trusted Platform Module (readers already familiar with these topics might skip this chapter). In chapter three, the installation and usage of the TPM emulator as well as of the dedicated device driver library are explained. Chapter four, describes the implementation of the TPM emulator. Note that this chapter only points out the most important implementation and design issues and decisions. For a more detailed description we refer to the documented source code and [TPMPart1, TPMPart2, TPMPart3]. In chapter five, we conclude by giving a summary and a short outlook of future work.

1.5 Acknowledgments

I would like to extend my gratitude to all people who made this semester thesis possible. First of all I would like to thank my supervisors Prof. Dr. David Basin and Paul E. Sevinç.

A special thank you goes to Achim D. Brucker and Michael Näf for supplying me with the necessary hardware and software.

Furthermore, I would like to thank Jeff Kravitz and David Safford from the IBM Watson Research Center for sharing information about their TPM libraries and examples with me, as well as Omen Wild from Dartmouth College and Jesus Molina from the University of Maryland for testing the emulator.

Chapter 2

Trusted Computing and Trusted Platform Module Basics

A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.
– D. Adams

Walking on water and developing software from a specification are easy if both are frozen.
– E. Berard

2.1 Trusted Computing

The following chapter gives a brief overview of trusted computing and TPMs. The interested reader is referred to [TPMPart1, TPMPart3, TPMArch, TPMNew, TCPHP] for a complete and comprehensive description.

2.1.1 Trusted Platform

A *Trusted (Computing) Platform* (TP) is a platform that is trusted by local users and remote entities including users, software and web sites. To enable a user to trust such a platform, a relationship of trust must be established between the user and the computing platform so that the user believes that an expected boot process, a selected operating system, and a set of selected security functions in the computing platform have been properly installed and operate correctly. The user makes his or her own judgment on whether or not he or she trusts the relationship.

2.1.2 Roots of Trust

The so called *Roots of Trust* (RT) are components within a TP that must be trusted because misbehavior might not be detected. They provide at least functionality for measurement, storing, and reporting of characteristics that affect the trustworthiness of the platform. Commonly there is one root of trust for each capability: a *Root of Trust for Measurement* (RTM), a *Root of Trust for Storage* (RTS), and a *Root of Trust for Reporting* (RTR).

In the trusted platform specified by the *Trusted Computing Group* (TCG), the *Trusted Platform Module* (TPM) acts as the RTS as well as the RTR whereas the *Core Root of Trust for Measurement*

(CRTM) is most often part of the BIOS.

2.2 Trusted Platform Module

The Trusted Platform Module is a hardware component that provides four major classes of functions:

1. Cryptographic functions: (P)RNG, SHA-1, HMAC, RSA.
2. Secure storage and reporting of hash values representing a specific platform configuration.
3. Protected key and data storage.
4. Initialization and management functions.

In addition, the following auxiliary functions exist:

- Monotonic counters and timing-ticks
- Non-volatile storage
- Auditing
- Delegation

2.2.1 Cryptographic Functions

It should be mentioned that the TPM is not a cryptographic accelerator. There are no specified minimum throughput requirements for any of the cryptographic functions.

Random Number Generator

The *Random Number Generator* (RNG) is the source of randomness in the TPM. It is used for the generation of nonces, keys and the randomness in signatures. The TPM specification allows for both true hardware-based and for algorithmic pseudo random-number generators.

SHA-1 Engine

A SHA-1 [FIPS180] message digest engine is primarily used for computing message or data signatures and for creating key blobs. The hash interfaces are also exposed outside the TPM to support measurement during the boot phases.

HMAC Engine

The HMAC [RFC2104] calculation provides two pieces of information to the TPM: proof of knowledge of the authorization data (shared secret key K) and integrity of the message $\mathcal{M}$. The used algorithm implementation uses SHA-1 as the hash function and a padding *ipad* (*opad*) consisting of 64 repetitions of byte `0x36` (`0x5C`). In the following formula $\otimes$ denotes the bitwise xor-operation and $||$ concatenation:

$$\text{HMAC}(K, \mathcal{M}) = \text{SHA-1}(K \ \otimes \ \textit{opad} \ \| \ \text{SHA-1}(K \ \otimes \ \textit{ipad} \ \| \ \mathcal{M}))$$

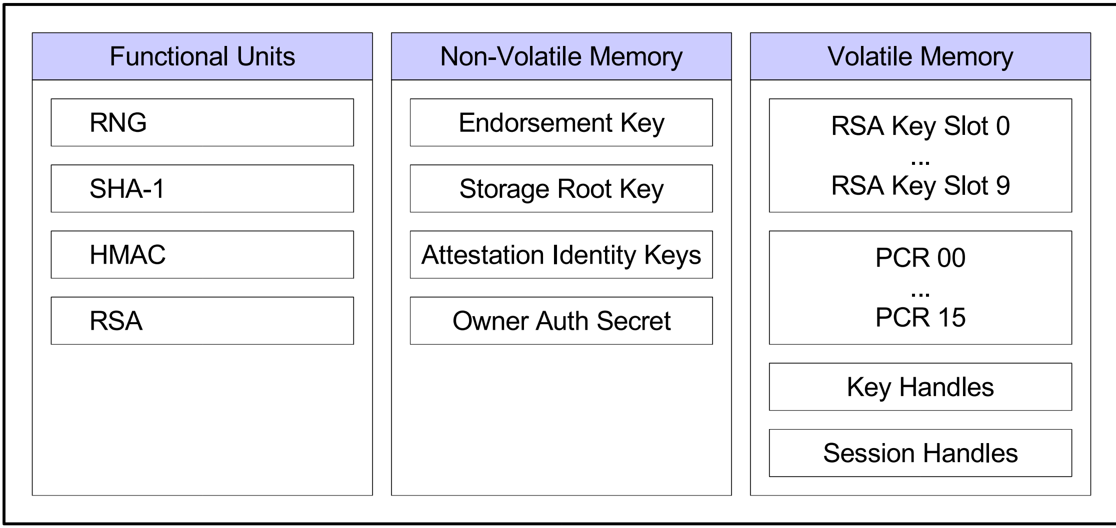

Figure 2.1: Schematic representation of a Trusted Platform Module

RSA Engine

The RSA asymmetric algorithm is used for digital signatures and for encryption. The PKCS #1 standard [PKCS1] provides the implementation details for digital signature, encryption, and data formats. The RSA key generation engine is used to create signing keys and storage keys. A TPM must support up to 2048-bit RSA keys, and certain keys must have at least a 2048-bit modulus. There is no requirement concerning how the RSA algorithm is to be implemented. TPM manufacturers may use *Chinese Remainder Theorem* (CRT) implementations or any other method.

2.2.2 Platform Integrity

Platform Configuration Register (PCR)

A *Platform Configuration Register* (PCR) is a 160-bit storage location for discrete integrity measurements in form of SHA-1 digests. There are a minimum of 16 PCR registers which are all inside the shielded location of the TPM. The correlation of PCR registers to specific measurements is specified in [TPMPC] (see table 2.1 on the following page).

To allow the storage of an unlimited number of measurements in a specific PCR register, its new value is dependent on both its old value and the new value to be added:

$$\text{PCR}[i] \leftarrow \text{SHA-1}(i \parallel \text{PCR}[i] \parallel \textit{new value to add})$$

As a result, updates to PCRs are not commutative. For example, measuring first A then B is not the same as measuring first B then A. Furthermore, subsequent updates to a PCR cannot be determined without knowledge of the previous PCR values or all previous input messages provided to a PCR register since the last reset.

Integrity Measurement

Integrity measurement is the process of obtaining metrics that reflect the integrity of a platform, storing them, and putting digests of those metrics in the PCRs. Examples for such metrics are the

PCR Index	PCR Usage
0	CRTM, BIOS, and Platform Extensions
1	Platform Configuration
2	Option ROM Code
3	Option ROM Configuration and Data
4	IPL Code (usually the MBR)
5	IPL Code Configuration and Data
6	State Transition and Wake Events
7	Reserved for future usage

Table 2.1: PCR usage on PC platforms.

opcode of the operating system or the BIOS configuration settings. The philosophy of integrity measurement, storage, and reporting is that a platform may be permitted to enter any state, including undesirable or insecure states, but that it cannot lie about states that it was or was not in.

Starting with the CRTM, there is a bootstrapping process by which a series of trusted subsystem components measure the next component in the chain and record the value in a PCR register (e.g., CRTM → BIOS → MBR → OS → Application). By these means, software is measured and recorded before it is executed. The recordings cannot be undone until the platform is rebooted.

Integrity Reporting

Integrity reporting is used to determine a platform's current configuration state. The reports are digitally signed, using therefore created *Attestation Identity Keys* (AIK), to authenticate the PCR values as created by a trusted TPM. To ensure anonymity, different AIKs should be used with different parties. Attestation that a specific AIK really belongs to a trusted platform without disclosure of the actual TPM identity can either be done by using a trusted third party (privacy CA) or by means of Direct Anonymous Attestation (DAA) [HPDAA]. The latter has the advantage that it avoids a possible linkage of the several AIK by the privacy CA.

2.2.3 Protected Key and Data Storage

Key Storage

The *Root of Trust for Storage* (RTS) protects keys and data entrusted to the TPM. Due to size limitations, only a small amount of volatile memory, where keys are held while performing signing and decryption operations, is provided. Inactive keys are encrypted using an adequate storage key (SK), often referred to as the keys parent key (see figure 2.2 on the next page). The so called *Storage Root Key* (SRK) acts as the root of this key hierarchy and cannot be removed from the TPM. However, a new SRK may be created as part of the take-ownership process. This has the side-effect that data objects controlled by the previous SRK are no longer accessible.

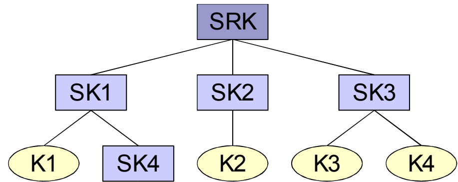

Figure 2.2: Key storage hierarchy.

Further, the TPM can also be used to create new signing or storage keys that can either be bound to it or marked as migratable. In addition to the specific key parameters (key length, usage, authorization data, etc.) the keys parent key has also to be specified. A new created key is not automatically loaded into the TPM but encrypted using the given parent key and returned to the user. Hence, it has to be explicitly loaded before usage.

Data Protection

The TPM specification defines four classes of data protection: Binding, Sealing (Sealed-Binding), Signing, and Sealed-Signing. Due to the limited data size that can be directly protected (∼210 bytes with a 2048-bit RSA key) not the confidential data itself but a symmetric key which is used to (de-)encrypt the data is typically protected.

Binding Binding is the operation of encrypting data using a public key. The data is only recoverable by decrypting it with the corresponding private key. If the private key is managed by the TPM as a non-migratable key, only the TPM that created the key may use it. Hence, the data might be seen as *bound* to a particular TPM. However, as it is possible to create migratable private keys that are transferable between multiple TPM devices, binding has no special significance beyond encryption.

Sealing Sealing takes binding one step further inasmuch as the data are not only encrypted but also bound to a specific platform configuration. Sealing associates the encrypted data with a set of PCR-register values and a non-migratable asymmetric key. The TPM only decrypts the data when the platform configuration matches the specified PCR-register values. Sealing is a powerful feature of the TPM as it provides assurance that the protected data is only recoverable when the platform is in a specific configuration.

Signing Signing of a data digest is only possible with signing only keys. These keys cannot be used for encryption and therefore avoid that a malicious user can encrypt an accordingly prepared data object and misuse it as a valid signature.

Sealed-Signing Signing operations can also be linked to PCR registers to ensure that the platform that signed the data meets a specific configuration. When a verifier demands that a signature must include a particular set of PCR registers, the specified registers are added to the data and included into the computation of the signature.

2.2.4 Initialization and management functions

Opt-In

TPM modules are shipped in the state the customer desires. This ranges from disabled and deactivated to fully enabled; ready for an owner to take possession.

TPM Ownership

Taking ownership of a TPM is the process of inserting a shared secret into a TPM's shielded-location. Proving knowledge of this authentication value proves that the calling entity is the TPM owner. The owner of the TPM has ultimate administrative control, in particular it can enable or disable the TPM, create AIKs and set policies for the TPM. There is no mechanism to recover a lost TPM owner authentication value. Recovery from a lost or forgotten authentication value involves removing the old value and installing a new one, thereby invalidating all information associated with the previous value.

The semantics of platform ownership are tied to the root of trust for storage (RTS). During the process of taking ownership, a new Storage Root Key (SRK) and a new (unique) TPM proof-value are created. It follows that objects owned by a previous owner will not be inherited by the new owner and must (if still needed) be explicitly transferred.

Authorization

The purpose of the authorization protocols and mechanisms is to prove to the TPM that the initiator of a command has permission to execute it and to access the involved data objects. For instance, they are used to establish platform ownership, restrict key usage, and to apply access control to (opaque) objects. The proof comes from the knowledge of a shared secret, the so called authorization data. The TPM treats knowledge of the authorization data as complete proof, no other checks are necessary.

There are three protocols to proof the knowledge of a specific piece of authorization data. The *Object-Independent Authorization Protocol* (OIAP) supports multiple authorization sessions for arbitrary entities whereby each entity has to be authorized individually. The *Object-Specific Authorization Protocol* (OSAP) supports an authorization session for a single entity only, but has the advantage that the authorization data has to be used only once during the whole session. The *Delegate-Specific Authorization Protocol* (DSAP) supports the delegation of owner or entity authorization.

New authorization information is inserted by the *Authorization Data Insertion Protocol* (ADIP) during the creation of an entity. The *Authorization Data Change Protocol* (ADCP) and the *Asymmetric Authorization Change Protocol* (AACP) allow the changing of the authorization data for an entity.

The protocols use a rolling nonce paradigm that is, they require that a nonce from one side is in use only for one message and its reply. For instance, a TPM's nonce which is returned as part of its command reply is part of the next command request. This mechanism is used to prevent reply as well as man-in-the-middle attacks.

2.2.5 Auxiliary Functions

Monotonic Counters

Monotonic counters provide an ever-increasing incremental value which is designed to not wear out in the first 7 years of operation (with an increment once every 5 seconds). A TPM must support a minimum of at least 4 concurrent counters.

Non Volatile Storage

The TPM contains a protected and shielded non-volatile storage area. Its main purpose is to provide the manufacturers and owners with a area for storing protected information. The TPM owner, when defining the area to use, will set the access and use policy for the area. He can set authorization values, delegations, PCR values and other controls (e.g., whether and how often data might be written to this area). The TPM does not control, edit, validate or manipulate in any manner the information in the non-volatile store. It is merely a storage device that enforces the specified access rules.

Auditing

To give the TPM owner the ability to determine that certain operations on the TPM have been executed, auditing of commands is possible. The audit value is a digest held internally to the TPM and externally as a log of all audited commands. The internal digest is only used to detect manipulations of the external log and does not contain any actual log information. It is in the responsibility of the TPM owner to specify which commands generate an audit event and to change the selection at any time.

Delegation Model

The TPM owner is an entity with a single super user privilege to control the TPM. Thus, if a TPM requires some kind of management, the TPM owner must perform that task himself or reveal his privilege information to another entity. This other entity thereby obtains the privilege to operate all TPM controls, not just those intended by the owner. Therefore, the owner often must have greater trust in the other entity than is strictly necessary to perform an arbitrary task. The delegation model addresses this issue by allowing delegation of individual TPM owner privileges (the right to use individual owner authorized TPM commands) to individual entities.

2.2.6 Command Structure

TPM commands (responses) are byte arrays which consist of three main parts: a request (response) header, some command-specific input (output) parameters, and an optional authorization trailer. Presence or absence of the authorization trailer depends on the used command (response) type, defined by the tag field. If it is set to `TPM_TAG_RQU_COMMAND`, no authorization data is available, if it is set to `TPM_TAG_RQU_AUTH1_COMMAND`, only auth1 is present, and if it is set to `TPM_TAG_RQU_AUTH2_COMMAND`, both auth1 and auth2 are given.

Integer values which are larger than one byte are encoded in network byte-order and might have to be converted into the local byte-order of the host platform. Since version 1.2 of the TPM

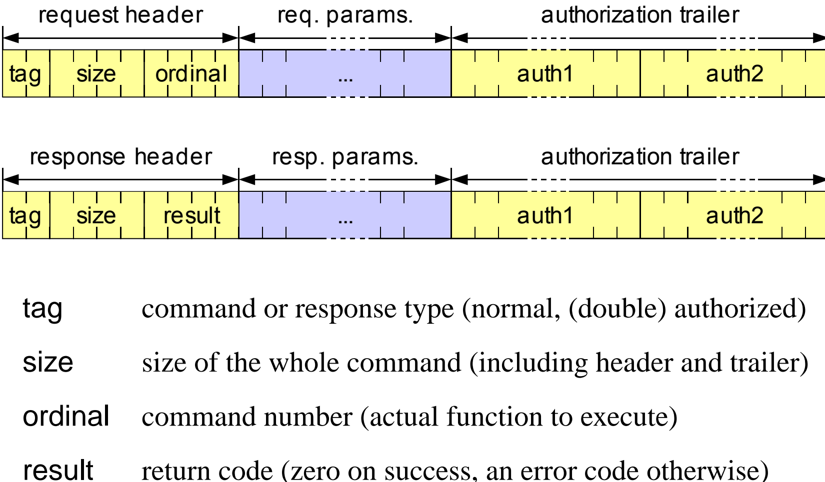

Figure 2.3: TPM command sructure

specification commands can also be sent from and to the TPM in an encrypted form using the transport encryption command.

Chapter 3

Installation and Usage

Captain Kirk: "You ought to sell an instruction and maintenance manual with this thing."
Cyrano Jones: "If I did, what would happen to man's search for knowledge?"
– Star Trek, The Trouble with Tribbles (1967)

3.1 TPM Emulator

3.1.1 Configuration

Kernel Release

Per default, the module is compiled and installed for the currently running kernel release, as indicated by 'uname -r'. To compile and install the module for an alternative (installed) kernel release, the variable RELEASE in the main make file (Makefile) has to be adapted accordingly.

GNU MP Support

By defining the variable USE_GMP in the main make file to point to the static GNU MP library (libgmp.a), some of its highly optimized core functions (mpz_*, mpn_*) are linked against the module. Otherwise, if USE_GMP is not defined, the generic GNU MP sources shipped with the emulator are used. Although linking the module against the GMP library results in a larger module, it is recommended because of the significant performance gain.
Example:

```
USE_GMP := /usr/lib/libgmp.a
```

3.1.2 Installation

First unpack the archive, then compile and install it:

```
# tar xvzf tpm_emulator-X.Y.tar.gz
# cd tpm_emulator-X.Y
# make
# make install
```

The automatic installation procedure (`make install`) consists of the following steps: First, a TPM device with major number 10 and minor number 224 as well as the directory to store the persistent TPM data (`/var/tpm/`) are created. Afterwards, the TPM kernel module is copied into the `misc` directory of the dedicated kernel release and its dependencies are updated.

3.1.3 Loading the Module

The startup mode of the TPM emulator (see [TPMPart1, section 7.3]) can be defined with a module parameter called `startup` that can either be set to `clear`, `save` (default) or `deactivated`. If the emulator is started in mode `save` and fails to load a previously stored TPM state it will go into a fail-stop mode and has to be reloaded to be usable again. Therefore, `startup` should be set to `clear` the first time the emulator module is loaded.

```
# modprobe tpm_emulator startup="clear"
```

3.1.4 Example Programs from IBM

IBM provides a TPM device driver for their ThinkPads as well as some example applications [IBMSW]. Unfortunately, the current release contains some minor byte-order conversion bugs and thus a patch has to be applied before the examples can be used with the emulator.

```
# tar xvzf tpm-1.1b.tar.gz
# cd TPM
# patch -p1 < tpm-1.1b.patch
# cd libtcpa/
# make
# cd ../examples/
# make
```

In the following, it is shown how these tools can be used to test the TPM emulator's capabilities. For a more detailed description and explanation of the library and tools we refer to the package documentation as well as to an article in the Linux Journal [IBMLJ] written by the authors of the software.

For a first verification of whether the TPM emulator has been loaded correctly or not, the `tcpa_demo` application can be used.

```
# ./tcpa_demo
```

In case of errors or unexpected behavior, the extensive logging output of the module might be a good starting point to localize the problem source.

```
# tail -f /var/log/messages | grep tpm_emulator
```

Taking Ownership

In order to use the TPM emulator one has to perform the take-ownership command when the module is loaded for the first time. The process includes the insertion of owner authorization data and the creation of the storage root key (SRK).

```
# ./takeown <owner password> <SRK password>
```

Key Generation and Key Loading

The programs `createkey` and `loadkey` allow the generation and loading of a RSA signing key, respectively. In the following example the SRK – whose key handle is always `0x40000000` – is used as the parent key. The specified key name is only used by the programs as a file name to store the returned key-blob and its dedicated public key; it has no TPM specific relevance.

```
# ./createkey 40000000 <key name> <key password>
# ./loadkey 40000000 <SRK password> <key name>
loaded key <key name>, returned handle <key handle>
```

File Signing and Verification

Once a signing key has been installed, it can be used to verify the integrity of files by signing their content digest.

```
# ./signfile <key handle> <key password> <data file> <sig file>
# ./verifyfile <sig file> <data file> <key name>.pem
```

Data Sealing

In addition to signing, two more programs for sealing and unsealing files are provided. It should be noted that signing keys cannot be used for encryption but only for signing. In particular, if the previously generated signing key were used for sealing, this would cause an invalid key usage exception. In addition to the key password, an additional data password has to be defined. Unsealing is only possible if both the key password and the data password are known.

```
# ./sealfile <key handle> <key password> <data password> \
<input file> <seal file>
# ./unsealfile <key handle> <key password> <data password> \
<seal file> <output file>
```

Key Eviction

If the loaded keys are no longer needed, they can be evicted using the `./evictkey` tool.

```
# ./evictkey <key handle>
```

3.2 TCG Device Driver Library

This section gives a short overview of the *TCG Device Driver Library* (TDDL) for the TPM emulator. For a detailed specification of the mentioned functions, constants, and error codes we refer to [TSS11].

3.2.1 Configuration

Per default, the TDDL library is installed into `/usr/local/lib` and the TDDL header file into `/usr/local/include`, respectively. To use different installation locations (e.g., `/usr/lib`

and /usr/include) the variables LIBDIR and INCDIR in the main configuration file (Makefile) have to be set accordingly. For example:

```
LIBDIR := /usr/lib/
INCDIR := /usr/include/
```

In addition, if not already done so, the path defined by LIBDIR has to be added to the dynamic linking configuration file /etc/ld.so.conf.

3.2.2 Installation

First unpack the archive, then compile and install it. Finally, update the dynamic linking cache.

```
# tar xvzf tddl-X.Y.tar.gz
# cd tddl-X.Y
# make
# make install
# ldconfig
```

3.2.3 Testing

To test the installation of the device driver library a small testing application is provided. One should note that the TPM emulator module must be loaded before the test can be executed.

```
# make test
# ./test_tddl
```

3.2.4 Function Interface

Tddli_Open

This function establishes a connection with the TPM device driver. The application utilizing the TPM device driver library is guaranteed to have exclusive access to the TPM device. This function must be called before any call to Tddli_GetStatus(), Tddli_GetCapability(), Tddli_SetCapability(), or Tddli_TransmitData(). On success zero is returned, otherwise a TDDL error code.

```
TSS_RESULT Tddli_Open()
```

Tddli_Close

This function closes a connection with the TPM device driver. Following a successful response to this function, the TPM device driver can clean up any resources used to maintain a connection with the TDDL. On success zero is returned, otherwise a TDDL error code.

```
TSS_RESULT Tddli_Close()
```

Tddli_Cancel

This function cancels an outstanding TPM command. An application can call this function (in a separate context) to interrupt a TPM command that has not completed. The TPM returns `TDDL_COMMAND_ABORTED` for the call in process. The function returns zero on success, a TDDL error code otherwise.

```
TSS_RESULT Tddli_Cancel()
```

Tddli_GetCapability

This function queries the TPM hardware, firmware, and device-driver attributes such as firmware version, driver version, etc. On success zero is returned, otherwise a TDDL error code.

```
TSS_RESULT Tddli_GetCapability(
  UINT32 CapArea, UINT32 SubCap,
  BYTE* pCapBuf, UINT32* puntCapBufLen)
```

`CapArea`	[in] Partition of capabilities to be interrogated.
`SubCap`	[in] Subcode of the requested capabilities.
`pCapBuf`	[out] Pointer to a buffer containing the received attribute data.
`puntCapBufLen`	[in] Size of the receive buffer in bytes.
	[out] Number of written bytes.

Tddli_SetCapability

This function sets parameters in the TPM hardware, firmware and device driver attributes. An application can set TPM device driver and operating parameters that may be defined by the TPM vendor. For now, the parameter definitions are vendor-defined. On success zero is returned, otherwise a TDDL error code.

```
TSS_RESULT Tddli_SetCapability(
  UINT32 CapArea, UINT32 SubCap,
  BYTE* pCapBuf, UINT32* puntCapBufLen)
```

`CapArea`	[in] Partition of capabilities to be set.
`SubCap`	[in] Subcode of the capabilities to be set.
`pCapBuf`	[in] Pointer to a buffer containing the capability data to set.
`puntCapBufLen`	[in] Size of the request buffer in bytes.

Tddli_GetStatus

This function queries the status of the TPM driver and device. On success zero is returned, a TDDL error code otherwise.

```
TSS_RESULT Tddli_GetStatus(
  UINT32 ReqStatusType,
  UINT32* puntStatus)
```

`ReqStatusType`	[in] Requested type of status information.
`puntStatus`	[out] Requested status.

Tddli_TransmitData

The function sends a TPM command directly to a TPM device driver, causing the TPM to perform the corresponding operation. On success zero is returned, otherwise a TDDL error code.

```
TSS_RESULT Tddli_TransmitData(
  BYTE* pTransmitBuf, UINT32 TransmitBufLen,
  BYTE* pReceiveBuf, UINT32* puntReceiveBufLen)
```

`pTransmitBuf`	[in] Pointer to a buffer containing TPM transmit data.
`TransmitBufLen`	[in] Size of TPM transmit data in bytes.
`pReceiveBuf`	[out] Pointer to a buffer containing TPM receive data.
`puntReceiveBufLen`	[in] Size of TPM receive buffer in bytes.
	[out] Number of written bytes.

3.2.5 Example Application

tddl_example.c

```
/*
 * TPM Device Driver Library Example
 * Copyright (C) 2004 Mario Strasser <mast@gmx.net>,
 *                      Swiss Federal Institute of Technology (ETH) Zurich
 */

#include <stdio.h>
#include <tddl.h>

int main()
{
  TSS_RESULT res;
  BYTE buf[256];
  UINT32 buf_size;
  UINT32 tpm_res;
  /* TPM_Reset() */
  BYTE reset[] = {0, 193, 0, 0, 0, 10, 0, 0, 0, 90};

  res = Tddli_Open();
  if (res != TDDL_SUCCESS) {
    printf("Error: Tddli_Open() failed: %04x\n", res);
    return -1;
  }
```

```
  buf_size = sizeof(buf);
  res = Tddli_TransmitData(reset, sizeof(reset), buf, &buf_size);
  if (res != TDDL_SUCCESS) {
    printf("Error: Tddli_TransmitData() failed: %04x\n", res);
    Tddli_Close();
    return -1;
  }

  tpm_res = ntohl(*(UINT32*)&buf[6]);
  if (tpm_res != 0) {
    printf("Error: TPM_Reset() failed: %04x\n", tpm_res);
    Tddli_Close();
    return -1;
  }

  res = Tddli_Close();
  if (res != TDDL_SUCCESS) {
    printf("Error: Tddli_Close() failed: %04x\n", res);
    return -1;
  }

  return 0;
}
```

Make File

```
#
# TPM Device Driver Library Example
# Copyright (C) 2004 Mario Strasser <mast@gmx.net>,
#                     Swiss Federal Institute of Technology (ETH) Zurich
#

all:
	gcc -o tddl_example -ltddl tddl_example.c

clean:
	rm -f tddl_example

PHONY:	all clean
```

Chapter 4

Implementation of the TPM Emulator

Programs must be written for people to read, and only incidentally for machines to execute.
– H. Abelson and G. Sussman

Real programmers don't comment their code.
If it was hard to write, it should be hard to understand.
– Anonymous

4.1 Concept

Our concern was less with building a high-performance emulator than it was with making the emulator as easy extensible and portable as possible. Hence, it was important to have a meaningful modularity and well-defined and well-documented interfaces.

When to have emulating a TPM, in principle there are two possibilities for attaching the software emulation layer. First, at the level of the device driver library and second, in the device driver itself. The former is easier to implement as it is completely located in user-space whereas the latter runs in kernel-space and thus cannot make (or only very limited) use of the standard and system libraries. However, as the device-driver library specification has only been released a short time ago, all current TPM applications do not use it but access the TPM by means of the hardware device (`/dev/tpm`) and hence could not be used with a library-based emulator. Such an implementation would then be of very limited use. Therefore, and to be compatible with the driver from IBM on the lowest possible level, this emulator uses the latter approach.

4.2 Structure

Basically, the TPM emulator consists of two main parts: the kernel interface and the emulator engine (see figure 4.1 on the following page). The latter can be further divided into the parameter (un)marshaling and (de)coding entity, the command execution engine, the cryptographic engine and the key, data, and state storage entity. Each part interacts with the others over a small set of well-defined functions.

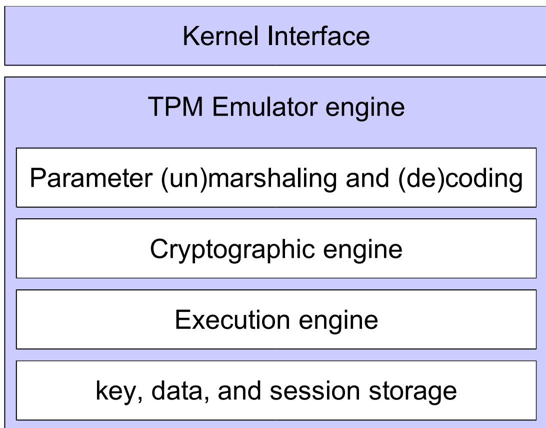

Figure 4.1: Structure of the TPM emulator

4.3 Naming and Coding Conventions

To avoid ambiguities and to keep the sources more readable, the following naming- and coding-conventions are used throughout the whole project:

- In general, code is formatted in the *Kernighan and Ritchie* style [KRCS].
- Variables, types, and functions of the kernel-module interface (`module.c`) are named according to the Linux Kernel Coding Style [KernelCS].
- Types and functions defined by the TPM specification start with `TPM_`.
- Globally visible types and functions start with `tpm_`.
- SHA-1-, HMAC-, and RSA-related types and functions start with `sha1_`, `hmac_`, and `rsa_`, respectively.
- Local utility functions and types do not start with one of the above prefixes.

4.4 License and Copyright

The TPM emulator was always intended to be an open-source project. In the decision about which of the many open-source licenses [OSI] to apply, the following criteria were considered:

1. responsibility and liability of the institute
2. the frequency of use of the license
3. the degree of compatibility of the available components with the chosen license
4. the protection granted by the license against proprietary uses of the software produced

Finally, the decision was made in favor of the GNU General Public License [GPL]; mainly because it has the greatest degree of compatibility with respect to the usage of other open-source products and due to its frequent usage:

This TPM emulator was written by Mario Strasser and is Copyright © 2004 by the Swiss Federal Institute of Technology (ETH) Zurich.

This TPM emulator is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This TPM emulator is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

4.5 Kernel Module/Interface

File: `module.c`

4.5.1 Device Driver

When the kernel module is loaded, it first registers itself as the device driver for the TPM device, using the minor-number 224 reserved for this purpose. It provides device functions for opening (`tpm_open()`), closing (`tpm_close()`), reading (`tpm_read()`), writing (`tpm_write()`), and controlling (`tpm_ioctl()`). Afterwards, the emulator is set up by calling `tpm_emulator_init()` as well as by initializing some variables, flags, and semaphores. In case of an error, the driver is unregistered and the module-initialization aborts with a dedicated error code.

4.5.2 Startup Parameter

The startup mode of the TPM emulator (see [TPMPart1, section 7.3]) can be defined with a module parameter called `startup`, which can either be set to `clear`, `save` (default), or `deactivated`.

4.5.3 Synchronization

To simplify matters and to simulate the behavior of a real hardware-based TPM, only one TPM command can be executed at once. This is achieved by means of a binary semaphore which is decremented (incremented) on entering (leaving) each of the device functions. Further, only the response to the last executed command is stored. That is, if one executes two commands without reading the response of the first one in between, it will be lost.

4.6 TPM Data Structures

File: `tpm_structures.h`

All public TPM data structures (i.e., those who are used for communication or external storage), are defined in [TPMPart2]. For simplicity and to avoid errors, all necessary types and definitions were extracted out of the original specification in PDF with some awk, sed, and Perl scripts. After rearranging the structures so that each type/structure is defined before its first usage, two

important utility macros have been added for most types. The first is called `sizeof_<type>()` and provides – following the well known C-function `sizeof()` which returns the *memory* usage of a type – the size of the *marshaled* type. The second function (`free_<type>`) releases all allocated memory for a type including all embedded sub-structures. A concrete example is shown in figure 4.3 on page 27.

There exist only five basic types: a single byte (`BYTE`), some unsigned integers (`UINT16`, `UINT32`, and `UINT64`) as well as a not further specified array of bytes (`BYTE*`), sometimes also referred to as BLOB. Any other (more complex) types are just compositions of these types, either in form of structures or of arrays. The most important ones are the authentication values `TPM_SECRET` and `TPM_AUTHDATA`, the nonce type `TPM_NONCE`, and the SHA-1 digest type `TPM_DIGEST`. For a more detailed and complete description of all TPM types we refer to [TPMPart2].

4.6.1 Backward Compatibility

Some of the standardized TPM structures have been changed in the new specification 1.2. While unmarshaling the parameters, several commands therefore have to detect whether a new or an old type is used and behave accordingly. Deciding which kind of parameter type is present can be done by examining the structure tag which is heading all new 1.2 structures. More precisely, it is first verified whether the first two bytes of the encoded data correspond to the expected 1.2 structure type. If so, all data is marshaled as a new structure; if not, the data is assumed to be an old structure.

Because most modifications are just additions of new parameters, or enlargement of existing ones, the emulator internally uses only one single structure (i.e., the greatest common element set) to store both variants, and uses the above-mentioned type tag to decide whether a specific structure element is valid or not. Having only one data type to deal with makes life much easier, in particular when it is used as an input or output argument.

Not all changed types but only the ones which are used at least once have been *merged* so far. Hence, when implementing new capabilities one has to make sure that all necessary data types can be used as described above in order to guarantee consistency.

4.6.2 Optional Structure Elements

There are elements in the defined TPM structures which are either optional or whose content depends on some other elements. For example, the structure to describe the parameters of a key (`TPM_KEY_PARMS`) contains the element `parms` (see [TPMPart2, section 10.1]) whose content is dependent on the defined key algorithm (`algorithmid`). However, to keep things simple, the structure internally used by the emulator contains references to all possible types whose validity is determined by the other elements.

4.6.3 Internal TPM Types

For storing emulator-internal administration data, some additional TPM types have been defined (e.g., for storing keys, sessions, handles, etc.) which are also based on the above-mentioned base-types. For a better understanding, the description of these types is postponed to sections where their usage is discussed.

4.7 Internal Data

Files: `tpm_data.[ch]`

All internal data of the TPM emulator are held in a global variable called `tpmData` of type `TPM_DATA`. It consists of three main parts appropriate for the three possible types of data a TPM can hold. Data and flags in the first part are reset in any startup mode of the TPM (`tpmData.stany`), elements in the second part are only reset if the TPM starts in the *clear* mode (`tpmData.stclear`), data in the third part is non-volatile and thus not reset on startup (`tpmData.permanent`).

In the following sections only the TPM emulator-specific structures are discussed. For a detailed description of the other data elements we refer to [TPMPart2, section 7].

4.7.1 Handles

Files: `tpm_handles.[ch]`

So far, four different types of handles have been implemented: key handles, authentication handles, transport handles, and counter handles. In this implementation, handles are just unsigned 32-bit integers which are constructed as follows: The most significant byte (bit 24-31) contains the resource type of the handle [TPMPart2, section 7.1], the remaining three bytes are used as an (array) index for the dedicated resources. An exception are the predefined key handles [TPMPart2, section 4.4.1] which do not follow the rules just described but are merely numbers. An invalid handle is indicated by setting all bits to one.

To simplify the application of the several handle types, functions for converting a specific resource index into a corresponding resource handle (e.g., `INDEX_TO_KEY_HANDLE()`) as well as for getting the resource for a given handle (e.g., `tpm_get_key()`) are provided.

4.7.2 Keys

Files: `tpm_structures.[ch], tpm_storage.c`

One of the main tasks of a TPM is the secure storage of keys. In addition to the actual key, several attributes have to be stored (e.g., authorization information, key usage, encryption and signing schemes, status of the PCR registers, flags, etc.). To meet these requirements, a new structure (`TPM_KEY_DATA`) was defined. For the storing, an array of `TPM_MAX_KEYS` keys (`tpmData.permanent.data.keys`) is provided. To identify whether a specific array element is already in use, the key structures contain an additional valid-flag which is set to false if the element is free and to true otherwise.

4.7.3 Sessions

Files: `tpm_structures.[ch], tpm_authorization.c`

Up to now, the TPM emulator supports object-independent and object-specific authentication sessions as well as transport sessions. For simplicity only one single session type is used to internally

represent all these sessions (TPM_SESSION_DATA). A detailed specification of which kind of information has to be stored for which session type is given in [TPMPart2, section 11]. The sessions are stored in an array (tpmData.stany.data.sessions) of size TPM_MAX_SESSIONS. Whether a specific array element is free or not is indicated by the type field of the session structure; if its value is TPM_ST_INVALID the element is unused, otherwise it already contains a valid session.

4.7.4 Persistent Storage

Difficulties and Challenges

Several parameters of a TPM (e.g., the endorsement key) are supposed to be permanently stored, that is, should persist restarts of the TPM or of the whole computer system. At first sight this behavior seems easy to emulate; just store the parameters on the hard disk. A closer look at the problem, however, shows that because we emulate the TPM using a kernel module, things are slightly more complicated. In general there are three possibilities for kernel modules to store persistent data, which all have their advantages and disadvantages:

Module parameters: Current kernel releases and module utilities allow the persistent storage of kernel module parameters by specifying a special argument when (un)loading the module. However, module parameters are just alphanumeric strings and meant for storing a few small arguments such as port numbers or device names. Hence, this approach is not adequate for storing the whole content of TPM which is in the order of some kilo-bytes, depending on the number of RSA keys to store.

Direct file access: By means of the kernel VFS-API, modules have full access to the local file systems an can therefore use a file for storing all persistent data. This approach has the big advantage that the storage process is completely under control of the kernel module which knows best when (re)storing data is advisable. However, having the kernel directly access files is a very controversial issue and one of the don'ts in several kernel programming guides [KernelHT]. Then, by design, Unix kernels should not access the hard disk directly especially not without a specific user context.

Although technically this works perfectly fine, one of the major problems apart of the aesthetic issue is that each process has its own name space (and hence effectively its own root directory). Auto-loading the driver will result in the module using init's name space, while manually loading it or within a chroot environment might result in the module using a different root directory.

User utility: Before a module is unloaded, a user utility reads the internal module data with some special ioctl commands and writes it into a file. After the module is reloaded, the utility can restore the internal data by first reading it from the file and then providing it to the module by using still another ioctl command. The advantage of this approach is that (re)storing the TPM data is completely done in userspace and is therefore most flexible. For example, it is not only possible to store the data into a file but to further manipulate and save it in any possible way.

However, having the storage of the internal data under complete control of the user is also the biggest disadvantage of this approach since usually it is the TPM who knows when (re)storing data is most advisable and not the user.

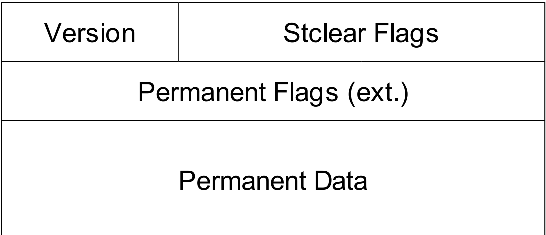

Figure 4.2: Schematic format of the persistent data file.

Current Approach

Files: `tpm_data.[ch], tpm_startup.c`

For simplicity and because it seems important to us that (re)storing the persistent data should be under control of the TPM, the second approach (direct file access) has been implemented. We are aware of the limitations and problems of this controversial solution and the mechanism is therefore very likely to change in the future. Nevertheless, at the moment, to us this seems to be the most appropriate solution. Especially, as all other solutions do also suffer under the changing root directory problem (using the utility in different contexts, for example in a chroot environment, would have almost the same effect). Those who do not want the emulator to persistently store its data in any form can disable the mechanism by undefining `TPM_STORE_TO_FILE` in the file `tpm_emulator.h`.

In the current approach, the persistent data of the emulator is stored on shutdown and restored on initialization with the functions `tpm_store_permanent_data()` and `tpm_restore_-permanent_data`, respectively. If stronger persistence is desired (i.e., that the data is stored after each successful command), it can be enabled by defining `TPM_STRONG_PERSISTENCE` in the file `tpm_emulator.h`. For the actual storage, the file `TPM_STORAGE_FILE` (`/var/-tpm/tpm_emulator-1.2.0.1`) is used; its schematic format is shown in figure 4.2. To make the file interoperable, all data is marshaled before it is written and unmarshaled after it is read, using the common (un)marshaling functions.

4.8 Marshaling and Unmarshaling

Files: `tpm_marshaling.[ch]`

Unmarshaling and marshaling of the TPM requests and responses, respectively, is realized by utility functions for each TPM type. To avoid errors and for efficiency reasons, only the (un)marshaling functions for the fundamental base types have been implemented manually. All other functions have been auto-generated out of the TPM header files using a Perl script created for this purpose. Because all non-basic TPM types are just compositions of either basic or other non-basic TPM types (see section 4.6 on page 21), (un)marshaling is straightforward by calling the dedicated (un)marshaling function for each element recursively.

In table 4.1 on the following page a short description of all base-type functions is given. The difference between `(un)marshal_BLOB()` and `(un)marshal_BYTE_ARRAY()` is that in the former only a *shallow* copy is made whereas in the latter the data is duplicated. An example of a more complex TPM structure and the resulting, generated code is shown in figure 4.3 on page 27.

`(un)marshal_BYTE()`	(un)marshals a single byte
`(un)marshal_UINT16()`	(un)marshals an unsigned 16-bit integer
`(un)marshal_UINT32()`	(un)marshals an unsigned 32-bit integer
`(un)marshal_BLOB()`	(un)marshals a blob of bytes
`(un)marshal_BYTE_ARRAY()`	(un)marshals an array of bytes
`(un)marshal_UINT16_ARRAY()`	(un)marshals an array of 16-bit integers
`(un)marshal_UINT32_ARRAY()`	(un)marshals an array of 32-bit integers

Table 4.1: (Un-)marshaling functions for the TPM base-types

4.8.1 Marshaling

In general, a function which performs the data marshaling for a value `v` of type `T` looks as follows:

```
int tpm_marshal_T(BYTE **ptr, UINT32 *length, T *v)
```

The first variable `ptr` points to the target buffer of size `length` in which the marshaled value should be stored. On success, 0 is returned and the values of `ptr` as well as `length` are updated (i.e., the size of the marshaled value is added to `ptr` and subtracted from `length`). In case of an error, -1 is returned and the values of `ptr` and `length` are undefined.

4.8.2 Unmarshaling

In analogy to the marshaling function, a function which performs data unmarshaling for a value `v` of type `T` is of the following form:

```
int tpm_unmarshal_T(BYTE **ptr, UINT32 *length, T *v)
```

The first variable `ptr` points to the source buffer which contains the marshaled value. On success, 0 is returned and the values of `ptr` as well as `length` are updated (i.e., the size of the marshaled value is added to `ptr` and subtracted from `length`). In case of an error, -1 is returned and the values of `ptr` and `length` are undefined.

4.8.3 Marshaling RSA Keys

Files: `tpm_marshalling.[ch], crypto/rsa.[ch]`

As mentioned before, (un)marshaling complex types is a straightforward process as they are just compositions of other basic or non-basic types. However, there is no rule without exception. The internally used RSA keys of the emulator are based on GNU MP integers [GNUMP] and therefore need special treatment. To unambiguously define a RSA private key, its modulus (m), public exponent (e) as well as one of its modulus primes (p) have to be known. All other parameters such as the private exponent (d), the other modulus prime (q) and all CRT parameters can be computed. Thus, only the former three parameters have to be (un)marshaled. This is done as follows: First, the lengths of m, e and p are marshaled, each as type `UINT16`. Afterwards, their

binary representations with the most significant bit first are stored; first m, then e, and finally p. When a key has to be unmarshaled, it is first determined whether the sizes for m, e, and p are appropriate (i.e., greater than zero and that p is half as long as m). Converting the GNU MP integers from and into their binary representation is done by means of the `rsa_import_*()` and `rsa_export_*()` functions, respectively.

```
typedef struct tdTPM_SYMMETRIC_KEY {
  TPM_ALGORITHM_ID algId;
  TPM_ENC_SCHEME encScheme;
  UINT16 size;
  BYTE* data;
} TPM_SYMMETRIC_KEY;

#define sizeof_TPM_SYMMETRIC_KEY(s) (4 + 2 + 2 + s.size)
#define free_TPM_SYMMETRIC_KEY(s) { tpm_free(s.data); }

...

int tpm_marshal_TPM_SYMMETRIC_KEY(BYTE **ptr, UINT32 *length,
                                  TPM_SYMMETRIC_KEY *v)
{
  if (tpm_marshal_TPM_ALGORITHM_ID(ptr, length, v->algId)
      || tpm_marshal_TPM_ENC_SCHEME(ptr, length, v->encScheme)
      || tpm_marshal_UINT16(ptr, length, v->size)
      || tpm_marshal_BLOB(ptr, length, v->data, v->size)) return -1;
  return 0;
}

int tpm_unmarshal_TPM_SYMMETRIC_KEY(BYTE **ptr, UINT32 *length,
                                    TPM_SYMMETRIC_KEY *v)
{
  if (tpm_unmarshal_TPM_ALGORITHM_ID(ptr, length, &v->algId)
      || tpm_unmarshal_TPM_ENC_SCHEME(ptr, length, &v->encScheme)
      || tpm_unmarshal_UINT16(ptr, length, &v->size)
      || tpm_unmarshal_BLOB(ptr, length, &v->data, v->size)) return -1;
  return 0;
}
```

Figure 4.3: Definition of the `TPM_SYMMETRIC_KEY` type and its generated (un)marshaling functions.

4.9 Authorization

Files: `tpm_authorization.c, tpm_cmd_handler.c`

The purpose of the authorization protocols and mechanisms [TPMPart1, section 11] is to prove to the TPM that the requester has the permission to perform a function and use some object. The proof comes from the knowledge of a 160-bit large shared secret (`TPM_SECRET`) which is most often derived from a user password.

Initially, our intention was to verify the command authorization right after unmarshaling the command parameters, so that whenever a command was executed, authorization and parameter integrity would have been guaranteed. However, for some commands, part of the needed authorization information is encrypted or is not available before executing most of the actual command.

Hence, to be consistent with all functions, the authorization mechanism of the emulator has been split up in three parts (see also figure 4.5 on page 31):

1. The parameter digest is computed right before the parameters are unmarshaled (`tpm_compute_in_param_digest()`) and stored as part of the authorization structure.
2. After all needed key handles and secrets are available the actual verification of the authorization value is performed. The therefore dedicated function `tpm_verify_auth()` takes the authorization handle, the shared secret of the object, and the handle of the object as an argument.
3. The setup of the response authorization (`tpm_setup_rsp_auth()`) is done as the last step before the response is marshaled and returned to the caller.

It should be mentioned that the TPM specification [TPMPart1, TPMPart3] is inconsistent regarding of what should be part of the output parameter digest. The current implementation was designed to compatible with currently available hardware TPMs.

4.10 Cryptographic Functions

In order to be functional and to meet the TCG specifications, a TPM must support a minimum set of cryptographic algorithms and operations: the SHA-1 hash function, a SHA-1-based message authentication code (HMAC), and RSA signing and key-generation. Additional algorithms and protocols may be available (e.g., AES or a stronger hash function such as SHA-256).

Since release 2.6 of the Linux kernel, SHA-1 and HMAC are directly supported through the so-called kernel Crypto-API. RSA and other asymmetric ciphers are not (and most probably will never be) supported. Most Linux systems, however, are still based on the 2.4 kernel series which support the Crypto-API only as an optional extension that, moreover, is hardly ever included. In order not to unnecessarily restrict the usability of the TPM emulator, we decided not to rely on the Crypto-API but to also include implementations for SHA-1 and HMAC in addition to the RSA engine.

4.10.1 SHA-1

Files: `crypto/sha1.[ch]`

The implementation of the SHA-1 hash algorithm is based on Steve Reid's public domain implementation and was tested according to [FIPS180].

4.10.2 HMAC-SHA-1

Files: `crypto/hmac.[ch]`

The SHA-1-based hashed message authentication code (HMAC) was implemented according to [RFC2104] and tested with the test vectors given in [RFC2202].

4.10.3 RSA

Multiple Precision Integer Support

Files: crypto/gmp_kernel_wrapper.c, crypto/gmp/*

Due to the already mentioned lack of multiple precision integer (MPI) support in the kernel, all necessary MPI functions are supplied by the emulator. Standard MPI libraries such as GNU MP, however, cannot easily be linked to kernel modules as they are not designed to run in kernel mode (e.g., there is no realloc() or stdin in the kernel).

The TPM emulator provides two possibilities for adding MPI support which can be selected at compile time (see section 3.1.1 on page 11). Both solution are based on the well known *GNU Multiple Precision Arithmetic Library* (GMP) [GNUMP] and depend on several function wrappers and auxiliary functions, in particular for memory and error management. In the first solution the generic (i.e., not processor specific) GMP source code of all needed arithmetic functions has been adapted and added to the emulator; whereas in the second solution the mandatory functions are (statically) linked against the static GNU MP library. Linking the module against the GMP library results in a larger module and requires the availability of the library but has the advantage that highly optimized and hence very high-performance functions are used.

Encryption/Decryption and Signing/Verification

Files: crypto/rsa.[ch]

Due to a lack of freely available RSA implementations that support all required encryption and signing schemes (with the eception of maybe the OpenSSL package which, however, is almost unadaptable due to its size and complexity) a new RSA engine according to [PKCS1] has been implemented. The engine uses CRT computation for better performance and supports the following encryption and signing schemes: RSAES-OAEP [PKCS1, section 7.1] using SHA-1 as the hash function, RSAES-PKCS1-v1_5 [PKCS1, section 7.2] and RSASSA-PKCS1-v1_5 [PKCS1, section 8.2] using SHA-1 as the hash function.

4.11 Initialization, Self-Test, and Shutdown

Files: tpm_startup.c, tpm_testing.c, tpm_data.c

4.11.1 Initialization

The initialization of the TPM emulator proceeds in three steps:

1. tpm_init_data() is executed and thereby all persistent data elements are set to their default values. The only exceptions are the disable and deactivated flags which are set to false although their default value is specified as true. This is done because loading the module can be seen as the users indication to use (i.e., enable and activate) the TPM emulator.

2. On executing TPM_Init() an internal self-test (TPM_SelfTestFull()) is first preformed comprising the following tests:

(a) Statistical random number generator tests [FIPS140].

(b) Verification of the SHA-1 algorithm using the test vectors given in [FIPS180].

(c) Verification of the HMAC algorithm using the test vectors given in [RFC2202].

(d) Test of the RSA engine and the endorsement key by generating a new key pair as well as by performing several en- and decryptions.

A more detailed description of the above-mentioned test cases is given in appendix A on page 41. If one of the tests fails, the emulator goes into a fail-stop mode and returns a dedicated error message on each command call. The only exception is the `TPM_GetTestResult()` function which can be used to get the exact cause of fault.

3. If all tests succeed, the emulator is started in the specified startup mode and the remaining internal data is initialized as described in section 4.7 on page 23. In the particular case of startup mode `save` this includes the restoration of all persistent data by means of `tpm_restore_permanent_data()`.

4.11.2 Shutdown

On shutdown, first `TPM_SaveState()` is called to store all persistent data. Next, `tpm_release_data()` is executed which releases all allocated resources such as memory or handles.

4.12 Command Execution

The execution of a TPM command is initiated by writing a marshaled TPM command (i.e., a byte array) to the TPM device (see figure 4.4). By doing so, the registered device-function (`tpm_write()`) is called which in turn forwards the command to the emulator by calling `tpm_handle_command()`.

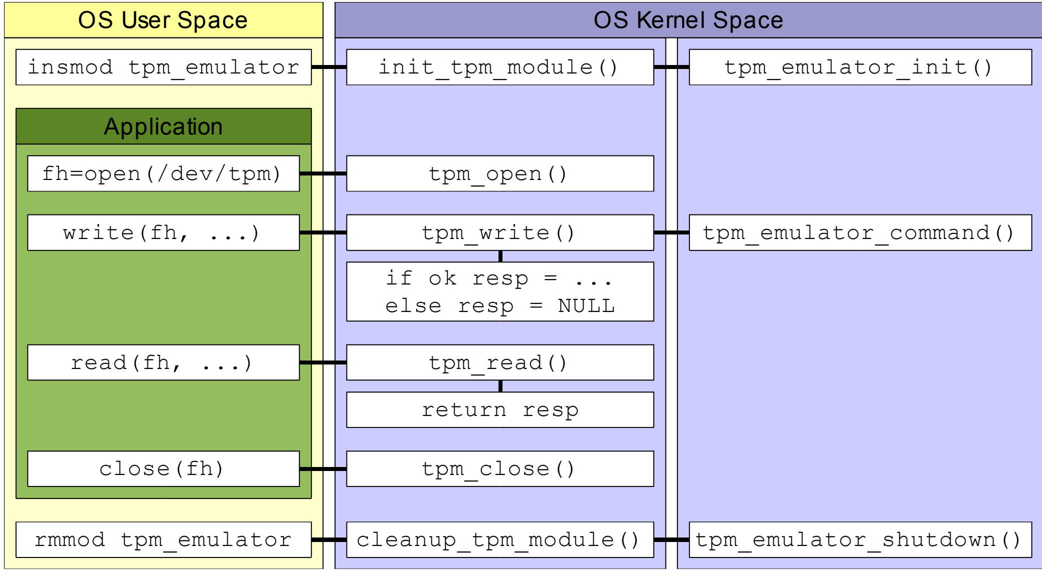

Figure 4.4: TPM command execution (a)

In the emulator, the command is processed in three steps (see figure 4.5):

In the first step, the command is first decoded into its three main components: the request header, the command parameters, and the (optional) authorization trailer. After verifying whether the command is allowed to be executed in the current state of the TPM, the input parameter digest is computed and the parameters are further unmarshaled according to the specific TPM command.

The second step performs the actual execution of the command and sets the command response parameters up. Before the execution (if mandatory) the authorization trailer is verified to guarantee command authorization and integrity of the input parameters.

In the last step, the response authorization is computed and composed together with the response parameters and the response header. Finally, the response is marshaled into its byte representation and returned to the caller.

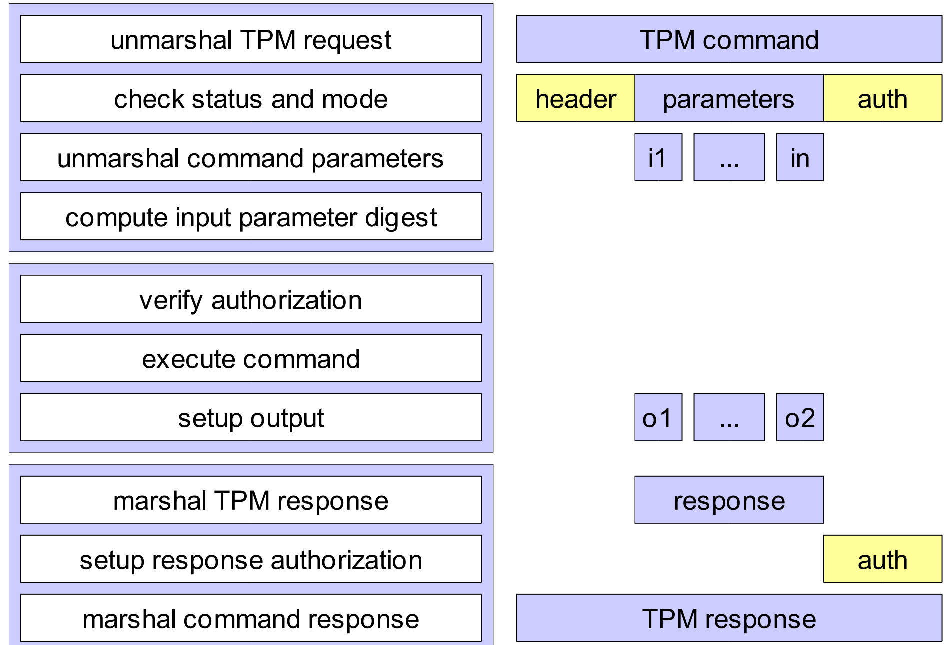

Figure 4.5: TPM command execution (b)

The received command response is then stored in the module until it is replaced by the next TPM command. A user can receive the command response by performing read operations on the TPM device file. The thereby implicitly called `tpm_read()` command copies the received response from kernel into user memory before it is returned.

4.13 Command Summary

fully implemented: +	partially implemented: ±	not implemented yet: –

4.13.1 Startup and State

File: `tpm_startup.c`

TPM_Init	+
TPM_Startup	+
TPM_SaveState	+

4.13.2 Testing

File: `tpm_testing.c`

TPM_SelfTestFull	+
TPM_ContinueSelfTest	+
TPM_GetTestResult	+

4.13.3 Opt-In

File: `tpm_testing.c`

TPM_SetOwnerInstall	+
TPM_OwnerSetDisable	+
TPM_PhysicalEnable	+
TPM_PhysicalDisable	+
TPM_PhysicalSetDeactivated	+
TPM_SetTempDeactivated	+
TPM_SetOperatorAuth	+

4.13.4 Ownership

File: `tpm_owner.c`

TPM_TakeOwnership	+
TPM_OwnerClear	+
TPM_ForceClear	+
TPM_DisableOwnerClear	+
TPM_DisableForceClear	+
TSC_PhysicalPresence	+
TSC_ResetEstablishmentBit	+

4.13.5 Capabilities

File: `tpm_capability.c`

TPM_GetCapability	±

4.13.6 Auditing

File: `tpm_audit.c`

TPM_GetAuditDigest	−
TPM_GetAuditDigestSigned	−
TPM_SetOrdinalAuditStatus	−

4.13.7 Administrative Functions

File: `tpm_management.c`

TPM_FieldUpgrade	−
TPM_SetRedirection	−

4.13.8 Storage Functions

File: `tpm_storage.c`

TPM_Seal	+
TPM_Unseal	+
TPM_UnBind	−
TPM_CreateWrapKey	+
TPM_LoadKey	+
TPM_GetPubKey	+

4.13.9 Migration

File: `tpm_migration.c`

TPM_CreateMigrationBlob	–
TPM_ConvertMigrationBlob	–
TPM_AuthorizeMigrationKey	–
TPM_CMK_CreateKey	–
TPM_CMK_CreateTicket	–
TPM_CMK_CreateBlob	–
TPM_CMK_SetRestrictions	–

4.13.10 Maintenance Functions

File: `tpm_maintenance.c`

TPM_CreateMaintenanceArchive	–
TPM_LoadMaintenanceArchive	–
TPM_KillMaintenanceFeature	–
TPM_LoadManuMaintPub	–
TPM_ReadManuMaintPub	–

4.13.11 Cryptographic Functions

File: `tpm_crypto.c`

TPM_SHA1Start	+
TPM_SHA1Update	+
TPM_SHA1Complete	+
TPM_SHA1CompleteExtend	+
TPM_Sign	+
TPM_GetRandom	+
TPM_StirRandom	+
TPM_CertifyKey	–
TPM_CertifyKey2	–

4.13.12 Credential Handling

File: `tpm_credentials.c`

TPM_CreateEndorsementKeyPair	+
TPM_CreateRevocableEK	+
TPM_RevokeTrust	+
TPM_ReadPubek	+
TPM_DisablePubekRead	+
TPM_OwnerReadInternalPub	+

4.13.13 Identity Creation and Activation

File: `tpm_credentials.c`

TPM_MakeIdentity	–
TPM_ActivateIdentity	–

4.13.14 Integrity Collection and Reporting

File: `tpm_integrity.c`

TPM_Extend	+
TPM_PCRRead	+
TPM_Quote	+
TPM_PCR_Reset	+

4.13.15 Authorization Changing

File: `tpm_authorization.c`

TPM_ChangeAuth	–
TPM_ChangeAuthOwner	–
TPM_OIAP	+
TPM_OSAP	+
TPM_DSAP	–
TPM_SetOwnerPointer	–

4.13.16 Delegation Commands

File: `tpm_delegation.c`

TPM_Delegate_Manage	–
TPM_Delegate_CreateKeyDelegation	–
TPM_Delegate_CreateOwnerDelegation	–
TPM_Delegate_LoadOwnerDelegation	–
TPM_Delegate_ReadTable	–
TPM_Delegate_UpdateVerification	–
TPM_Delegate_VerifyDelegation	–

4.13.17 Non-volatile Storage

File: `tpm_nv_storage.c`

TPM_NV_DefineSpace	–
TPM_NV_WriteValue	–
TPM_NV_WriteValueAuth	–
TPM_NV_ReadValue	–
TPM_NV_ReadValueAuth	–

4.13.18 Session Management

File: `tpm_context.c`

TPM_KeyControlOwner	–
TPM_SaveContext	–
TPM_LoadContext	–

4.13.19 Eviction

File: `tpm_eviction.c`

TPM_FlushSpecific	+

4.13.20 Timing Ticks

File: `tpm_ticks.c`

TPM_SetTickType	–
TPM_GetTicks	–
TPM_TickStampBlob	–

4.13.21 Transport Sessions

File: `tpm_transport.c`

TPM_EstablishTransport	−
TPM_ExecuteTransport	−
TPM_ReleaseTransportSigned	−

4.13.22 Monotonic Counter

File: `tpm_counter.c`

TPM_CreateCounter	+
TPM_IncrementCounter	+
TPM_ReadCounter	+
TPM_ReleaseCounter	+
TPM_ReleaseCounterOwner	+

4.13.23 DAA commands

File: `tpm_daa.c`

TPM_DAA_Join	−
TPM_DAA_Sign	−

4.13.24 GPIO

File: `tpm_gpio.c`

TPM_GPIO_AuthChannel	−
TPM_GPIO_ReadWrite	−

4.13.25 Deprecated commands

File: `tpm_deprecated.c`

TPM_EvictKey	+
TPM_Terminate_Handle	+
TPM_SaveKeyContext	−
TPM_LoadKeyContext	−
TPM_SaveAuthContext	−
TPM_LoadAuthContext	−
TPM_DirWriteAuth	−
TPM_DirRead	−
TPM_ChangeAuthAsymStart	−
TPM_ChangeAuthAsymFinish	−
TPM_Reset	+
TPM_CertifySelfTest	−
TPM_OwnerReadPubek	−

4.14 TCG Device Driver Library

Although the TPM device driver interface, as described in the previous sections, is provided by both, the TPM emulator and IBM's device driver and is used by all current applications, it is not part of the TSS specification and thus not standardized. To provide future applications with a more suitable and standardized device interface, a *TCG Device Driver Library* (TDDL) according to [TSS11] has been implemented and is shipped together with the emulator.

To guarantee the mandatory TPM command serialization, the library uses a POSIX-thread mutex. Furthermore, no vendor-specific capabilities have been defined yet, but might be in the future to allow the manipulation of some TPM emulator-specific settings.

Chapter 5

Conclusion

The best way to predict the future is to invent it.
– A. Kay

5.1 Summary

Summarized we can state that the main goal of this thesis, namely the development of a *functional* software-based Trusted Platform Module emulator for Linux, has been reached.

More precisely, about 50 out of 120 ($\sim$42%) TPM commands have been implemented and for the remaining, appropriate function-stubs have been added. These stubs already contain the needed parameter unmarshaling and marshaling pre- and postambles and also do some error handling, allowing a minimum-effort completion of the corresponding TPM commands. Additionally, a complete *TCG Device Driver Library* (TDDL) to access the TPM has been developed in order to provide future applications with a suitable and standardized device interface.

Almost all available TPM applications work perfectly well with the current emulator implementation as most of the important and frequent commands are already supported. Some of the applications which do not yet work will most probably never do, as the required functionality, e.g., measurements of the BIOS or operating system on system boot, cannot be provided by means of a kernel module (the module is not loaded until the operating system has already started).

It might be worth mentioning that by now, the TPM emulator has been successfully tested by the developers as well as by two other people and on three different Linux platforms.

5.2 Outlook

As the emulator is still a work in progress, there remains some future work to be done; either in form of completion or enhancements. The most beneficial tasks are listed below:

- *Delegation* and *non-volatile storage* functionality.
- Step-wise completion of the `TPM_GetCapabiliy` command along with the addition of commands and other capabilities.
- Additional stress and long-time tests.

- Creation of RPMs and Debian packages to provide the emulator with the current Linux distributions.
- Porting the emulator to other operating systems such as FreeBSD/OpenBSD or Windows.

Jesus Molina from the University of Maryland is already working on porting the emulator to a PCI embedded system and will also take an active part in the further development of the emulator. His goal is the development of a free (hardware) TPM device as well as an appropriate BIOS. In addition, Jeff Kravitz from the IBM Watson Research Center is preparing a new TPM device driver package for IBM ThinkPads which will eliminate the known errors and include some additional utilities and example applications.

Appendix A

FIPS Tests

A.1 Statistical Random Number Generator Tests [FIPS140]

A single bit stream of 20,000 consecutive bits of output from the generator is subjected to each of the following tests. If any of the tests fail, then the module must enter an error state.

A.1.1 The Monobit Test

1. Count the number of ones in the 20,000-bit stream. Denote this quantity by X.

2. The test is passed if $9654 < X < 10346$.

A.1.2 The Poker Test

1. Divide the 20,000 bit stream into 5,000 contiguous 4-bit segments. Count and store the number of occurrences of each of the 16 possible 4-bit values. Denote $f(i)$ as the number of each 4 bit value i where $0 \leq i \leq 15$.

2. Evaluate the following: $X = \frac{16}{5000} \cdot \sum_{i=0}^{15} f(i)^2 - 5000$.

3. The test is passed if $1.03 < X < 57.4$.

A.1.3 The Runs Test

1. A run is defined as a maximal sequence of consecutive bits of either all ones or all zeros, which is part of the 20,000-bit stream. The occurrences of runs (for both consecutive zeros and consecutive ones) of all lengths (≥ 1) in the sample stream must be counted and stored.

2. The test is passed if the number of runs that occur (of lengths 1 through 6) is each within the corresponding interval specified below. This must hold for both the zeros and ones; that is, all 12 counts must lie in the specified interval. For the purpose of this test, runs greater than 6 are considered to be of length 6.

Length of Run	Required Interval
1	2267 - 2733
2	1079 - 1421
3	502 - 748
4	223 - 402
5	90 - 223
6+	90 - 223

A.1.4 The Long Runs Test

1. A long run is defined to be a run of length 34 or more (of either zeros or ones).
2. On the sample of 20,000 bits, the test is passed if there are NO long runs.

A.2 SHA-1 Test Vectors [FIPS180]

A.2.1 First Example

Message: ASCII string "abc"

Digest: `A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D`

A.2.2 Second Example

Message: ASCII string "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"

Digest: `84983E44 1C3BD26E BAAE4AA1 F95129E5 E54670F1`

A.2.3 Third Example

Message: ASCII string which consists of 1,000,000 repetitions of "a"

Digest: `34AA973C D4C4DAA4 F61EEB2B DBAD2731 6534016F`

A.3 HMAC Test Vectors [RFC2202]

A.3.1 Test Case 1

Key: `0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b`

Data: "Hi There"

Digest: `0xb617318655057264e28bc0b6fb378c8ef146be00`

A.3.2 Test Case 2

Key: "Jefe"

Data: "what do ya want for nothing?"

Digest: `0xeffcdf6ae5eb2fa2d27416d5f184df9c259a7c79`

A.3.3 Test Case 3

Key: `0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

Data: `0xdd` repeated 50 times

Digest: `0x125d7342b9ac11cd91a39af48aa17b4f63f175d3`

A.3.4 Test Case 4

Key: `0x0102030405060708090a0b0c0d0e0f10111213141516171819`

Data: `0xcd` repeated 50 times

Digest: `0x4c9007f4026250c6bc8414f9bf50c86c2d7235da`

A.3.5 Test Case 5

Key: `0x0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c`

Data: "Test With Truncation"

Digest: `0x4c1a03424b55e07fe7f27be1d58bb9324a9a5a04`

A.3.6 Test Case 6

Key: `0xaa` repeated 80 times

Data: "Test Using Larger Than Block-Size Key - Hash Key First"

Digest: `0xaa4ae5e15272d00e95705637ce8a3b55ed402112`

A.3.7 Test Case 7

Key: `0xaa` repeated 80 times

Data: "Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data"

Digest: `0xe8e99d0f45237d786d6bbaa7965c7808bbff1a91`

Appendix B

Source and Documentation Files

B.1 CD-ROM Content

`report.pdf`	Report
`presentation.pdf`	Slides of the presentation
`tpm_emulator-0.1.tar.gz`	TPM emulator package
`tddl-0.1.tar.gz`	TDDL package
`tpm_emulator-0.1/`	extracted TPM emulator package
`tddl-0.1/`	extracted TDDL package
`report/`	Latex source of the Report

B.2 TPM Emulator Package

`Makefile`	Main makefile
`tpm_commands.h`	TPM command specifications
`tpm_data.h`	Internal data specifications
`tpm_emulator.h`	Main header file
`tpm_handles.h`	Handles specification
`tpm_marshalling.h`	TPM marshaling specifications
`tpm_structures.h`	TPM data structures
`module.c`	Kernel interface
`tpm_*.c`	TPM command implementation (see section 4.13)
`crypto/kernel_wrapper.c`	GMP kernel wrapper
`crypto/Makefile`	Crypto makefile
`crypto/hmac.[ch]`	HMAC implementation
`crypto/rsa.[ch]`	RSA implementation
`crypto/sha1.[ch]`	SHA-1 implementation
`crypto/gmp/Makefile`	GNU MP makefile
`crypto/gmp/*.[ch]`	Ported GNU MP sources

B.3 TDDL Package

`Makefile`	TDDL Makefile
`tddl.c`	TDDL main source file
`tddl.h`	TDDL main header file
`test_tddl.c`	TDDL test file

Bibliography

[TPMPart1] **TPM Main Specification Part 1, Design Principles**
Specification version 1.2, Revision 62, October 2, 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part1_TPM_Structures.pdf

[TPMPart2] **TPM Main Specification Part 2, TPM Structures**
Specification version 1.2, Revision 62, October 2, 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part2_TPM_Structures.pdf

[TPMPart3] **TPM Main Specification Part 3, Commands**
Specification version 1.2, Revision 62, October 2, 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/tpmwg-mainrev62_Part3_TPM_Structures.pdf

[TPMArch] **TCG Specification Architecture Overview**
Specification version 1.2, April 28, 2004 (Work In Progress)
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf

[TPMNew] **TPM v1.2 Specification Changes**
Specification version 1.2, October 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/TPM_1_2_Changes_final.pdf

[TSS11] **TCG Software Stack Specification**
Specification version 1.1, August 20, 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/TSS_Version__1.1.pdf

[TPMPC] **TCG PC Specific Implementation Specification**
Specification version 1.1, August 18, 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/TCG_PCSpecificSpecification_v1_1.pdf

[TCGBG] **Trusted Computing Group Backgrounder**
May 2003
Trusted Computing Group, Incorporated
https://www.trustedcomputinggroup.org/downloads/TCG_Backgrounder.pdf

[TCPHP] **Trusted Computing Platforms: TCPA Technology in Context**
Siani Pearson (editor)
Pearson Education, 1st edition July 2002

[FIPS140] **FIPS PUB 140-1, Security Requirements for Cryptographic Modules**
US DOC/NBS, January, 1994
http://www.itl.nist.gov/fipspubs/fip140-1.htm

[FIPS180] **FIPS PUB 180-1, Secure Hash Standard**
US DOC/NBS, April, 1994
http://www.itl.nist.gov/fipspubs/fip180-1.htm

[RFC2104] **RFC 2104 - HMAC: Keyed-Hashing for Message Authentication**
http://www.ietf.org/rfc/rfc2104.txt

[RFC2202] **RFC 2202 - Test Cases for HMAC-MD5 and HMAC-SHA-1**
http://www.ietf.org/rfc/rfc2202.txt

[PKCS1] **PKCS #1: RSA Cryptography Standard**
RSA Laboratories, June 2002
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf

[GNUMP] **GNU Multiple Precision Arithmetic Library**
http://www.swox.com/gmp/

[KernelCS] **Linux Kernel Coding Style**
Part of the Linux Kernel Package: `Documentation/CodingStyle`
http://www.kernel.org/,
http://www.linuxjournal.com/article.php?sid=5780

[KRCS] **C Programming Language**
Brian W. Kernighan and Dennis Ritchie
Pearson Education, 2nd edition March 1988

[KernelMPG] **The Linux Kernel Module Programming Guide**
Peter Jay Salzman and Ori Pomerantz
http://www.faqs.org/docs/kernel/

[Kernel24] **Linux Kernel 2.4 Internals**
Tigran Aivazian
http://www.faqs.org/docs/kernel_2_4/lki.html

[KernelLDD] **Linux Device Drivers, 2nd Edition**
Alessandro Rubini and Jonathan Corbet
O'Reilly, 2nd edition June 2001
http://www.xml.com/ldd/chapter/book/index.html

[KernelPD] **Porting device drivers to the 2.6 kernel**
http://lwn.net/Articles/driver-porting/

[KernelHT] **How to NOT Write Kernel Drivers**
Arjanv van de Ven
http://people.redhat.com/arjanv/olspaper.pdf

[GPL] **GNU General Public License**
http://www.gnu.org/copyleft/gpl.html

[OSI] **Open Source Initiative**
http://www.opensource.org/licenses/

[IBMSW] **IBM TPM Device Driver and Example Code**
http://www.research.ibm.com/gsal/tcpa/

[IBMLJ] **Take Control of TCPA**
David Safford, Jeff Kravitz, and Leendert van Doorn
Linux Journal, issue 112, August 2003
http://www.linuxjournal.com/article.php?sid=6633

[IBMDS1] **The Need for TCPA**
David Safford
IBM Research, October 2002
http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf

[IBMDS2] **Clarifying Misinformation on TCPA**
David Safford
IBM Research, October 2002
http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf

[IBMIM] **Design and Implementation of a TCG-Based Integrity Measurement Architecture**
Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn
IBM Research, Thomas J. Watson Research Center, January 16, 2004
http://www.ece.cmu.edu/%7Eleendert/publications/rc23064.pdf

[IBMSL] **Analyzing Integrity Protection in the SELinux Example Policy**
Trent Jaeger, Reiner Sailer, and Xiaolan Zhang
IBM Research, 2003
http://www.research.ibm.com/people/s/sailer/publications/2003/usenix_security_03.pdf

[IBMTL] **tcgLinux - TPM-based Linux Run-time Attestation**
http://www.research.ibm.com/secure_systems_department/projects/tcglinux/

[HPDAA] **Direct Anonymous Attestation**
Ernie Brickell, Jan Camenisch, and Liqun Chen
HP Technical Report, May 18, 2004
http://www.hpl.hp.com/techreports/2004/HPL-2004-93.pdf

[DCTR] **Experimenting with TCPA/TCG Hardware**
John Marchesini, Sean Smith, Omen Wild, and Rich MacDonald
Department of Computer Science Dartmouth College
Technical Report TR2003-4761, December 15, 2003
http://www.cs.dartmouth.edu/%7Esws/papers/mswm03.pdf

[DCEF] **The Enforcer Project**
http://enforcer.sourceforge.net/

[SUTC1] **Improving End-user Security and Trustworthiness of TCG-Platforms**
Klaus Kursawe and Christian Stuble
Saarland University, September 29, 2003
http://krypt1.cs.uni-sb.de/download/papers/KurStu2003.pdf

[SUTC2] **Trusted Computing Platform Alliance Mythen, Wirklichkeit und Loesungswege**
Dirk Gunnewig, Ahmad-Reza Sadeghi, and Christian Stüble
Saarland University, 2003
http://krypt1.cs.uni-sb.de/download/papers/GuSaSt2003.pdf

[SUTC3] **Bridging the Gap between TCPA/Palladium and Personal Security**
Ahmad-Reza Sadeghi and Christian Stble
Saarland University, Technical Report, 2003
http://krypt1.cs.uni-sb.de/download/papers/SadStu2003.pdf

[TCBS] **Palladium and the TCPA**
Bruce Schneier
Crypto-Gram Newsletter, August 15, 2002
http://www.schneier.com/crypto-gram-0208.html#1

[TCRS] **Can you trust your computer?**
Richard Stallman
NewsForge Article, October 21, 2002
http://www.newsforge.com/business/02/10/21/1449250.shtml?tid=19

[ATTC1] **No TCPA Organisation**
http://www.notcpa.org/

[ATTC2] **Against TCPA**
http://www.antitcpa.com/

[IBMCC] **IBMs Antworten auf die TCPA-Fragen des Chaos Computer Club**
Christian Stüble and Gerald Himmelein
c't Magazine, issue 15, 2003, http://www.heise.de/ct/03/15/021/default.shtml