An Integrated Workflow for Design and Implementation of Security Modules

Exemplified on the Trusted Platform Module Specification Level 2 Version 1.2, Revision 116

Kapeundl Martin

An Integrated Workflow for Design and Implementation of Security Modules

Exemplified on the Trusted Platform Module Specification
Level 2 Version 1.2, Revision 116

Master's Thesis

at

Graz University of Technology

submitted by

Martin Kapeundl

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

2. April 2013

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

Co-Advisor: Dipl.-Ing. Martin Pirker

Ein integrierender Arbeitsablauf für das Design und die Implementierung von Sicherheits Modulen

Am Beispiel der Trusted Platform Module Spezifikation
Level 2 Version 1.2, Revision 116

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Martin Kapeundl

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

2. April 2013

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter:	Univ.-Prof. M.Sc. Ph.D. Roderick Bloem
Mitbetreuender Assistent:	Dipl.-Ing. Martin Pirker

Abstract

Today's digital world is highly dependent on security enhanced systems to strengthen protection against data theft and misuse. A hardware security module can offer such systems, amongst other things, strong encryption, authentication and authorization as well as encryption key management.

The underlying specification for a security module plays a crucial role in its lengthy and extensive development process. It is a valuable document needed by specification designers as well as implementers to review design decisions and steer the advancement of future developments. During the evolution of a specification, many thoughts and considerations are created, discarded and even lost. This information might become helpful to fully understand concepts and design decisions, which is especially important in the complex and sensitive field of computer security.

A vague specification can be the source of various problems: From incompatible implementations to false assumptions which may ultimately lead to security problems. Therefore it is indispensable that a specification is as precise and easy to understand as possible. The more people are able to understand a specification, the more it is likely that problems in the design are discovered (early) and corrected. Especially for complex and large specifications it is highly desirable to document changes for comprehension purposes and to spot not only serious but also annoying flaws like typing errors. Those mistakes may very well become the origin of dangerous security problems later in the implementation.

The Trusted Platform Module (TPM) specification, which provides trusted functions to a higher level software stack, is used as an example on how to model such an integrated design process. We introduce a domain-specific language, called `FTPM` (*Formal TPM*), to describe TPM commands. With a TPM specification written in `FTPM` we can perform simple type checks and automatically generate different outputs. Due to the *automatic* creation it is guaranteed that the underlying specification and derived outputs evolve synchronously. We show that a TPM specification drafted in `FTPM` is easily readable for implementers and specification designers and inconsistencies can be detect quickly. Through a version control system, the development history is preserved as well which can be a valuable source of information for editors and implementers of the TPM specification.

`FTPM` is by no means a complete solution for different kinds of technical specifications or even for different types of security modules. We present a workflow which stretches from the description of TPM commands and structures in `FTPM` to the *automatic* creation of various outputs. We exemplify this workflow solely on the specification for TPM. Therefore, `FTPM` is specific to the operations described for Trusted Platform Modules.

Kurzfassung

Unsere digitale Welt ist abhängig von speziellen Systemen mit erweiterter Sicherheitsarchitektur um unsere Daten gegen Diebstahl und Missbrauch zu schützen. Ein Hardware-Sicherheits-Modul kann diesen Systemen unter anderem starke Verschlüsselung, Authentifizierung und Authorisierung sowie die Verwaltung von kryptographischen Schlüsseln zur Verfügung stellen.

Die für die Entwicklung dieser Hardware zugrunde liegende Spezifikation spielt eine besonders wichtige Rolle im Entwicklungsprozess. Sie ist ein wertvolles Dokument, welches von Spezifikations-Designern und -Entwicklern zum Review verwendet und mit dessen Hilfe daher auch der weitere Entwicklungsprozess gesteuert werden kann. Während des Entstehungsprozesses einer Spezifikation entstehen viele Gedanken und Überlegungen, wobei manche verworfen werden und andere verloren gehen. Diese Überlegungen können jedoch hilfreich sein, eine Spezifikation vollends zu verstehen – vor allem im komplexen und sensitiven Bereich der Computer-Sicherheit.

Eine uneindeutige Spezifikation kann der Ursprung für verschiedenste Probleme sein: Angefangen von inkompatiblen Implementierungen bis hin zu falschen Annahmen, die im schlimmsten Fall zu Sicherheitslücken führen können. Daher steht außer Frage, dass eine Spezifikation so präzise und so leicht verständlich wie möglich sein muss. Je mehr Menschen eine Spezifikation verstehen desto eher (und schneller) werden Probleme im Design erkannt und korrigiert. Vor allem für große und komplexe Spezifikationen ist es höchst wünschenswert, dass alle Änderungen nachvollziehbar sind und nicht nur schwerwiegende, sondern auch kleinere Mängel wie Tippfehler, korrigiert werden. Denn selbst kleine Fehler können der Ursprung von schwerwiegenden Sicherheitslücken in der späteren Implementierung werden.

Die Spezifikation von Trusted Platform Modulen (TPMs), welche eine gesicherte und kryptographische Funktionalität für darauf aufbauende Software zur Verfügung stellt, dient dieser Arbeit als Beispiel wie ein solcher Entwicklungsprozess modelliert werden kann. Wir stellen eine domänenspezifische Sprache – genannt FTPM (*Formal TPM*) – vor, mit der TPM Funktionsaufrufe beschrieben werden können. Ausgehend von einer Spezifikation, die in `FTPM` verfasst wurde, werden einfache Typenüberprüfungen vorgenommen und verschiedene Ausgaben erzeugt. Durch die *automatische* Generierung ist garantiert, dass sich die Spezifikation und der erzeugte Quellcode synchron entwickeln. Wir zeigen, dass eine TPM Spezifikation die mit `FTPM` verfasst wurde, leicht lesbar ist und ein schnelles Auffinden von Inkonsistenzen ermöglicht. Durch die Verwendung einer Versionskontrolle kann zudem die gesamte Entwicklungsgeschichte für künftige Entwickler und Spezifikations-Designer zugänglich gemacht werden.

`FTPM` ist weder eine fertige Lösung für verschiedenste technische Spezifikationen noch für verschiedene Sicherheits-Hardware. Wir stellen einen Arbeitsablauf vor, der sich von der Beschreibung von TPM-Funktionsaufrufen and Datenstrukturen bis hin zur *automatischen* Erzeugung von Ausgaben erstreckt. Da dieser Arbeitsablauf beispielhaft an der Spezifikation für TPMs entwickelt wurde ist er auch ausschließlich auf deren Design und Funktionsumfang ausgelegt.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

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

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Acknowledgements

I want to thank my advisor, Roderick Bloem, as well as my Co-Advisor, Martin Pirker, for their support and advices during the quite longish process of getting this thesis done.

Martin Kapeundl
Graz, Austria, April 2013

Credits

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

- The thesis was written using Keith Andrews' skeleton thesis [3].
- Icons in figures originate from the Oxygen project [65].
- Icons in Figure 6.6 and Figure 6.8 originate from the Eclipse project [19].
- EBNF diagrams in Chapter 5 are created with the Railroad Diagram Editor [43]

Chapter 1

Introduction

1.1 Problem

A system which provides a certain level of security is not trivial. Nevertheless, today's digital world is dependent on systems with enhanced security to protect our digital identity as well as our data against theft, misuse and modification. One approach to protect our digital data is the usage of a dedicated hardware chip that provides cryptographic functions which operate on (not necessarily securely) stored data. This dedicated hardware is called a *secured crypto-processor* and makes it significantly harder for malicious outsiders to read, modify or destroy data unauthorized and thus can greatly strengthen the security of a system. Examples for such chips are *smartcards*, *hardware security modules* (HSM) or the *Trusted Platform Module* (TPM).

Of course, these systems are exposed to attacks which they must withstand. The attacks can be roughly classified into attacks against the hardware or attacks against the software. The security feature of the hardware is to guarantee that data stored securely onto the device can never be accessed or leave the chip unintentionally and unprotected. As attacks against the hardware are not only more complicated but also need physical presence for execution, a secure design of the software is equally important. Most certainly, bugs on any layer, both the hard- and the software, may compromise the security of the data.

The Trusted Platform Module (TPM) can be viewed as a special kind of a HSM, as it is permanently integrated onto a computer's motherboard and cannot be removed. The capabilities of a TPM include a cryptographic coprocessor, which supports protected storage for sensitive data, integrity measurement of the host platforms state as well as reporting this state to a third party for verification purposes.

Today, TPMs are installed in a significant percentage of newly built laptops, PCs and smartphones. It is estimated that TPM chips are deployed in about 500 million devices worldwide [60]. These chips follow the specification from the Trusted Computing Group (TCG) [59], which describes the design [56], data structures [57] and available commands [58] on around 700 pages in total.

Unfortunately, the currently published TPM specification has several defects. Gürgens et al. [23] found several errors, inconsistencies and inaccurate descriptions in some parts of the specification. Although their comments lead to improvements in the following specification revisions, similar defects can still be found in the current version of the specification. Examples for problems in the currently available specification are presented in Chapter 4 (Figure 4.4).

Another drawback of the current TPM specification is that no public reference implementation exists. Therefore, the feedback cycle between specification consumers and specification designers is interrupted. This cycle is valuable to detect and correct shortcomings and problems early. Without feedback it is more likely that those problems end up in a published revision of the specification.

Improved readability of the specification document also aids the security of the chip. Important pieces of the specification—for example command authentication information—should be emphasized to

ease comprehension. Particularly, a user-friendly navigation between different parts of the specification aids the review for readers. Currently, it is cumbersome to navigate through the hundreds of pages of specification text, as the specification in its prevailing form does not provide any connection (hyperlinks) between data structures and commands at all.

It is eminent that a clean specification that was thoroughly analyzed by specification designers and implementers benefits the security of the chip. The specification is vital in the development cycle and needs to be as precise as possible to allow designers and implementers to detect errors early. Any ambiguities in the specification can lead to potential errors in the resulting implementation. These ambiguities may ultimately lead to problems like incompatibility between different implementations, misinterpretation by implementers or security issues.

1.2 Proposed solution

This thesis proposes a new development cycle for the future development and maintenance of the specification for TPM chips based on the current available version (Level 2 Version 1.2, Revision 116). The core component of the workflow is a domain-specific language (DSL), named `FTPM` (Formal TPM), in which TPM commands and structures can be described accurately.

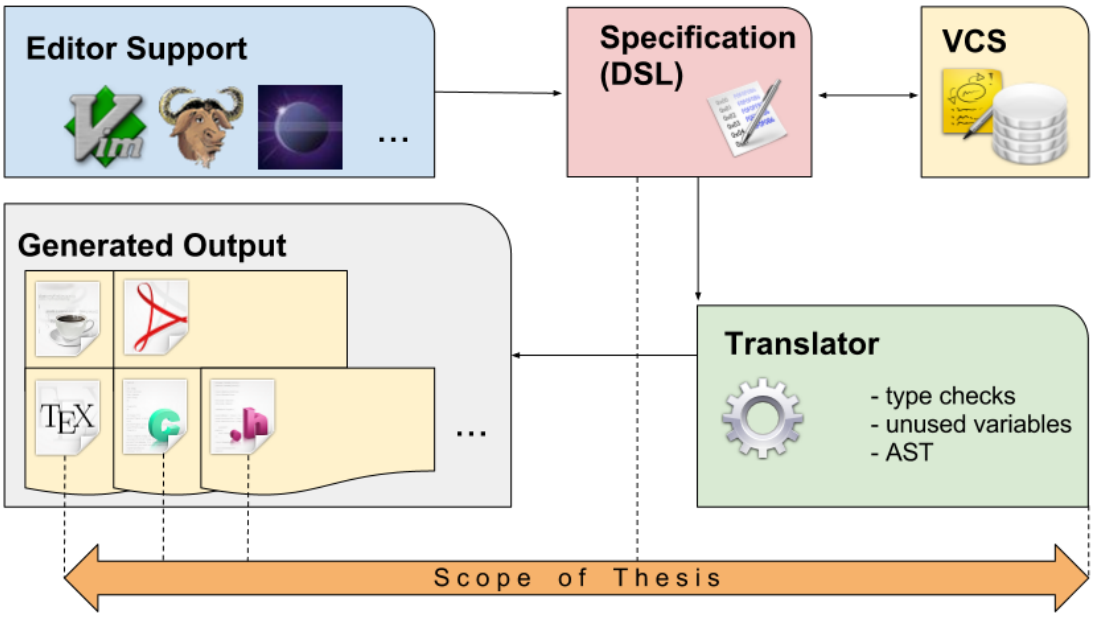


Figure 1.1: Based on a domain-specific language (DSL) to describe TPM commands, a translator parses and checks the specification source and directly generates various different outputs. A version control system (VCS) records and ensures access to the project history. The parts this thesis concentrates on are marked by dashed lines.

The workflow is depicted in Figure 1.1 and consists of the aforementioned domain-specific language, a translator, and a version control system (VCS). The translator reads and parses the specification, performs type checks on the input and *automatically* produces outputs in various desired formats. We show that this translator produces documentation as well as source and header files for the `C` programming language directly from a specification written in `FTPM`. We furthermore focus on the flexibility of the whole process. While additional checks performed by the parser or different output formats, for example in Extensible Markup Language (`XML`) or `Java` source code, can be added quite easily to `FTPM`, we solely focus on a first working prototype.

With `FTPM` we propose the following interaction between specification designer and implementer: First, the specification is described in the `FTPM` language. However, certain features and details of

the specification cannot be described with FTPM. For example, there are no language constructs for cryptographic routines built into the language. Consequently, code cannot be generated for these details. However, needed data structures and the essential execution steps of a TPM command can be described. This description is subsequently translated to different output formats.

If a command requires a change during the development process, source code modified by an implementer would be overwritten. We solve this problem with a simple separation of implementation details from the basic command structure: We factor out implementation details into separate functions (see Figure 1.2). Then, a implementer can use the generated code and needs to add the required functionality to the functions stubs. This way, changes and details added by the implementer are not overwritten if code is automatically re-generated with the FTPM prototype.

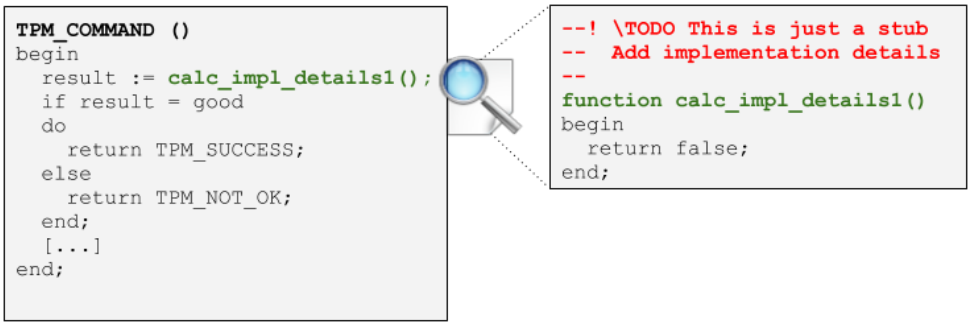


Figure 1.2: Stub functions are the entry point for implementers to add implementation details that are not described with FTPM by the specification designer.

This prototype consists of a parser, which allows not only early detection of typing mistakes but consequently eliminates errors in the resulting documentation as well. The *automatic* translation of FTPM source files in different output formats leads to a closely coupled design process and most importantly to a coherent documentation. Furthermore, a specification written in FTPM can provide a base for a reference implementation through the means of automatic code generation. Important areas of the specification, like authentication information, can be easily outlined and emphasized in FTPM, which improves readability and aids human comprehension.

The history of the design process—another possible factor to fully understand a specification—is stored by a Version Control System (VCS). Therefore, the thoughts and considerations, which lead to a specific revision, are preserved for future implementers or specification designers. A VCS offers tools to visualize changes, browse the history of the development and thus contributes to discover errors and mistakes which may be introduced unintentionally.

FTPM is specifically developed for the problem of describing functions for a TPM. The workflow is demonstrated on a selection of TPM commands, as a full implementation of all available commands and structures is out of scope for a first prototype. Furthermore, it is not the goal to develop a general purpose specification description language. FTPM strives for a clean syntax and omission of unnecessary language constructs to keep the language as small as possible.

Moreover, this thesis does not addresses problems at the hardware layer of security modules but rather focuses on improving the creation and maintenance of the underlying specification in order to minimize implementation errors. Increasing the quality of a specification empowers more people to evaluate design decisions, which is especially beneficial in the area of computer security.

Additionally, this thesis shows that inconsistencies in the TPM specification are easily detectable with FTPM. The automatic creation of output, like source code or documentation, leads to increased quality and may therefore avoid possible security issues. We show that TPM commands and structures written in FTPM result in a coherent, up-to-date and well-structured documentation. Moreover, it is possible to derive a reference implementation from it instantly, which can serve as a base for a real implementation.

With FTPM, the specification is available in an *executable* form. This allows a quick adaption to changing constraints. Likewise, it is possible to create additional tools and utilities that operate on FTPM. This flexibility is demonstrated by our prototype with the enforcement of constraints found in the TPM specification. These constraints are enforced by the compiler and thus automatically applied throughout the whole specification. Moreover, as an example of additional useful tools we demonstrate a small utility which counts the number of bytes a TPM command consumes. This information may be important on devices with limited memory, like embedded systems, but primarily demonstrates the gained flexibility of an *executable specification*.

1.3 Summary

We propose to write a specification for TPMs with a domain specific language. This language is the core of a workflow that aims to simplify and and improve the creation and maintenance of this specification. Thus, errors and mistakes in the specification can be detected early which ultimately aids the security of the chip.

In summary our approach addresses the following problems:

- Access to the complete history of the development process and to the thoughts and considerations through a VCS. Although there is a section that lists the changes between different versions of the specification in the documentation this section is naturally not complete and has to be maintained explicitly. A VCS provides specification designers and implementers access to the *whole* history at any time and keeps track of changes *automatically*.

- Validation of data structures and expressions used in TPM commands. The parser discovers syntactic and semantic errors in the specification ranging from simple typing mistakes to the usage of unexpected data-structures or -components. The early detection of these errors aids the discovery of potential serious problems.

- Strict enforcement of rules a properly implemented TPM modules has to adhere to. The specification for TPMs is large and complex. Therefore, it is beneficial to let the parser perform checks on data structures and commands and enforce rules which could otherwise easily be missed by an implementer.

- Creation of a interface for implementers of the specification. While it is not possible nor intended to automatically generate the source code for a complete implementation with every needed detail, a solid base for a real implementation can be provided. Implementation details are separated from fundamental execution steps and can be added by implementers later. Therefore, changes introduced to the specification do not interfere with implementation details supplemented by the implementer.

- Automatic generation of output (code, documentation) that evolves synchronously to the specification. Small and profound changes in the specification are immediately reflected in the resulting output and enforced by the parser. Especially important to implementers is an up-to-date and consistent documentation. Moreover, an improved structure of the documentation eases the navigation through the large specification and benefits comprehension.

- Increased flexibility because the specification is available in an *executable* format. It is easy to generate different outputs from a specification written with FTPM. Moreover, additional tools can give more insight to the specification if needed. For example, the byte consumption of TPM commands can be interesting for embedded devices.

1.4 Chapter Overview

The next chapter, Chapter 2, familiarizes the reader of this thesis with concepts of cryptography as well as Trusted Computing and TPMs. Chapter 3 gives an overview of different approaches towards a precise and up-to-date description of technical specifications. In Chapter 4 we demonstrate how a specification for TPMs for selected commands and structures can be written with `FTPM`. We outline improvements and detected inconsistencies together with shortcomings of the current implementation. Then, we discuss details of `FTPM` language thoroughly in Chapter 5. Chapter 6 presents the technical details of the chosen implementation as well as the possible workflow between specification designer and implementer. Finally, Chapter 7 discusses some ideas for future work and possible research fields for improvements of the current implementation. Moreover, the next version of specification for TPMs and needed adaptions on `FTPM` are briefly discussed.

Chapter 2

Background

The following chapter introduces basic building blocks that are needed to design a domain-specific language for the specification of Trusted Platform Modules (TPMs). After a short motivation, we explain the basic primitives and protocols for cryptography. Among these primitives are symmetric and asymmetric (public-key) encryption and hashing functions. Furthermore, important algorithms, like RSA (Rives, Shamir, Adleman) [46] and SHA-1 (Secure Hash Algorithm) [51] are briefly presented. Then, we discuss elemental concepts of Trusted Computing and especially TPMs. Next, we give a design overview and the general usage of TPMs. Finally, we present the authorization mechanism for executing TPM commands and gaining access to objects held by a TPM.

2.1 Cryptography

The art of cryptography is used by humans probably for thousands of years. Davies [13] mentions evidence of some kind of cryptography back to the earliest forms of writing. One motivation for cryptography is the desire to make information available to only a certain group of people and hide it securely from others. This could be sensitive data, like credit card account information or health records, but of course also private communication with friends and family.

The usage of new applications, especially new ways for communication and transactions in the world wide web—for example *e-money*, *e-banking* or *e-voting*—require an additional layer of protection against attacks. Examples for possible attacks are the unauthorized access, usage, modification, inspection or destruction of digitized data. Cryptography can be used for protection, as it can enable the following desirable properties [48]:

- **Confidentiality**
  An attacker should not be able to disclose private data.
- **Integrity**
  An attacker should not be able to modify data unnoticed.
- **Authentication**
  An attacker should not be able to masquerade as someone else.
- **Nonrepudiation**
  A sender should not be able to falsely deny transmission later on.

Encryption, Signatures, Hash Functions and Cryptographic Keys are some of the building blocks of cryptography to enforce these properties and are briefly introduced in the following sections.

2.1.1 Encryption, Decryption and Keys

Encryption (E) is the process to generate a ciphertext (C) out of a message (M). A ciphertext is a disguised form of the original message, which is unreadable to a human or a computer. Decryption (D) denotes the inverse process: The original message is recovered back from its encrypted form.
Formally, this process can be written as

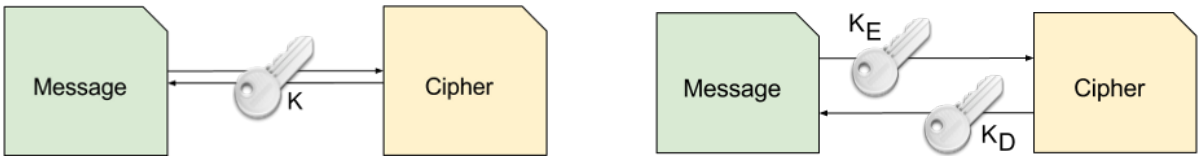
$$E_K(M) = C \quad \text{and}$$
$$D_K(C) = M \quad \text{as well as}$$
$$D_K(E_K(M)) = M$$

The cryptographic algorithm is based on a mathematical function (or key K), which is used for encryption (K_E) and decryption (K_D). If different keys are used for encryption and decryption, the former equation can be written as:

$$E_{K_E}(M) = C \quad \text{and}$$
$$D_{K_D}(C) = M \quad \text{as well as}$$
$$D_{K_D}(E_{K_E}(M)) = M$$

The value of the message M as well as the cipher C is dependent on the key K: The encryption of a message with two different keys results in dissimilar ciphertexts.

Figure 2.1 illustrates symmetric and asymmetric algorithms. A **Symmetric Algorithm** uses either the same key for both operations—encryption and decryption—or the respective keys can be derived from each other. This algorithm is visualized in Figure 2.1a, where a single key enciphers and decrypts a message. Figure 2.1b depicts an **Asymmetric** or **Public Key Algorithm** where different keys are used for encryption (K_E) and decryption (K_D).



(a) Symmetric algorithm **(b)** Asymmetric algorithm

Figure 2.1: Exemplary scheme of Symmetric (a) and Asymmetric (b) Algorithms. A single key is used for encryption and decryption in the Symmetric Algorithm while an Asymmetric (Public Key) Algorithm uses different keys.

2.1.2 Symmetric Algorithms

Symmetric algorithms either use the same key for en- and decryption or the keys can be derived from each other. Both, the sender (S), who encrypts a message, and the receiver (R), who wants to decrypt it, must agree on a key in order to establish a secure communication. This key has to remain secret at all times or otherwise anyone who gains knowledge of the key is able to encrypt and decrypt messages as well.

A symmetric algorithm provides much higher processing speeds due to less computational needs than an asymmetric algorithm. However, there exist at least two major disadvantages with symmetric algorithms, which concern the distribution and management of cryptographic keys:

- **Key Distribution**
  An additional secure channel for key distribution is needed to avoid eavesdropping of an attacker. If an attacker is able to get hold of the key he can not only decrypt all past and future messages encrypted with this key but also masquerade as a valid user.

- **Key Management**
  Usually, a separate key is needed for each participant of a secure communication. In the case of n people communicating $n * (n-1)/2$ keys ($O(n^2)$) are required [67]. Schneier [48] illustrates this by an example: A group of 100 users would need nearly 5000 different keys.

2.1.3 Asymmetric Algorithms

Asymmetric or public-key algorithms use different keys for encryption and decryption operations. The decryption key can not be calculated from the encryption key (in reasonable time with today's computers) and has to remain secret. However, the encryption key can be made public – hence the term *public-key* algorithm, as only the corresponding decryption key is be able to perform decryption of ciphered messages.

The relation between the keys is established by a *trapdoor function*. A trapdoor function denotes a function which is easy to compute in one direction but difficult to inverse without additional information (the *trapdoor*). Without the trapdoor it is impossible to derive a private key from a given public key and consequently no other party can gain knowledge of a private key by access to a public key.

Asymmetric algorithms do not need an additional secure channel for key distribution, as the sender simply encrypts data with the public key of the intended receiver. Therefore, the key management problem from symmetric algorithms mentioned in 2.1.2 is greatly reduced. For n users only $2n$ keys are needed, which amounts to a cost of $O(n)$ in terms of needed keys [67]. However, in order to verify that a public key belongs to a certain entity and to establish trust between communication parties some sort of *public-key infrastructure* (PKI) is common. A PKI is capable, among other things, to manage identities and is used for the distribution of keys. More information about PKIs can be found in [1], risks and problems of PKIs are summarized in [15].

Unfortunately, the performance of public key algorithms is rather slow compared to symmetric algorithms. Schneier [48] says, that public-key algorithms are generally at least 1000 times slower than symmetric algorithms. Consequently, public key algorithms are often used to exchange symmetric keys securely, which then perform the actual encryption or decryption operation. This approach is usually denoted by the term *Hybrid Algorithm* and is schematically shown in Figure 2.2. The sender has to perform the following steps in order to send an encrypted message to a receiver with a hybrid algorithm:

1. The sender creates a (symmetric) session key K_S.
2. A message M is encrypted with K_S, which results in the ciphertext C.
3. K_S is encrypted by the public key of the intended receiver (K_{pub}).
4. C and K_S, which is now protected by K_{pub}, is sent to the receiver.

The receiver has to perform the following tasks in order to decrypt the message successfully:

1. The receiver is the only one who knows the private key (K_{priv}), which belongs to the public key (K_{pub}) that was used is protect the symmetric key (K_S).
2. K_{priv} is used to extract the symmetric session key K_S.
3. The ciphertext C can be decrypted with K_S, which results in the original message M.

As the encrypted key is usually a lot smaller than the message M, this decryption process is quite fast. After gaining the symmetric key, the performance advantage to decrypt the message with a fast symmetric key can be utilized.

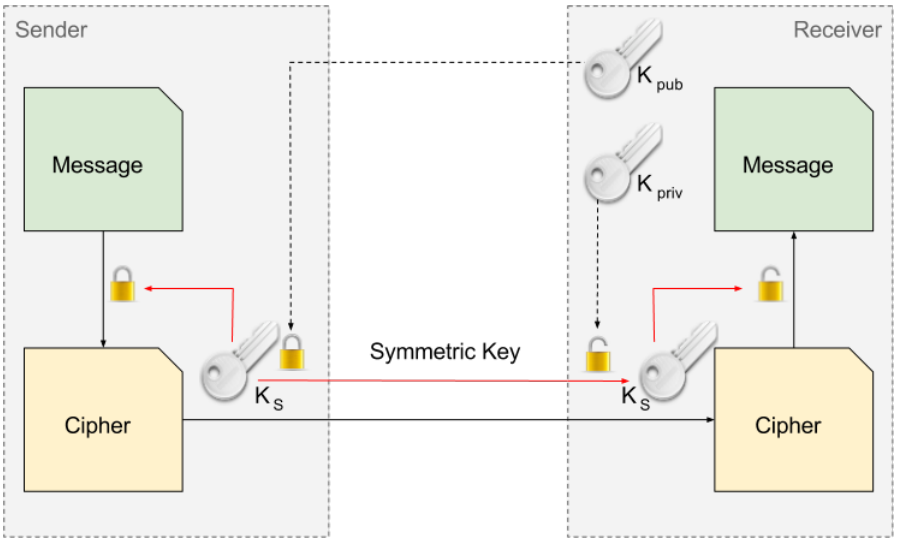


Figure 2.2: Hybrid Algorithm: A sender encrypts a message with a symmetric key. The symmetric key is encrypted with the public key of the intented receiver. The encrypted message and the encrypted key are sent to the receiver, who first decrypts the symmetric key with the private portion of his asymmetric key pair and subsequently can decrypt the message with the symmetric key.

2.1.4 Hash Functions

A hash function ($hash$) operates on input of arbitrary length (pre-image, P) and converts it to an output of fixed length (hash value or *digest*, h). This process is visualized by Figure 2.3 and can be written as:

$$h = hash(P)$$

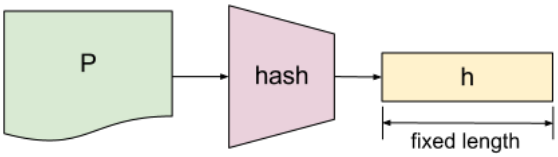


Figure 2.3: Hash function

As the hash value is usually smaller than the pre-image, it can be used to verify pre-images more efficiently: The hash value of a pre-image can attest a certain level of assurance that a given pre-image is the same as the real pre-image.

A one-way hash function denotes a function, where it is easy to compute a hash value but computationally infeasible to generate the pre-image of a given hash value. In cryptography one-way hash functions have to fulfill the following properties:

1. **Pre-image resistance**
   Given a digest, it is computationally infeasible to find another message which results to the same digest.

2. **Second pre-image resistance**
   Given a message, it is computationally infeasible to find another message which results to the same digest.

3. **Collision resistance**
   It should be computationally infeasible to find two messages, which result to the same digest.

The output of one-way hash functions is not dependent on the input. A change of a single bit in the pre-image changes about half the bits in the resulting hash value (avalanche property) [48]. These properties are important in the area of cryptography to increase the efficiency of encryption and decryption operations and to detect modifications on messages. Furthermore, hash functions can be used for authentication mechanisms, which will be discussed next.

Hash Based Message Authentication

A Message Authentication Code (*MAC*) denotes a one-way hash function with an additional (secret) key:

$$h_{mac} = hash_K(P)$$

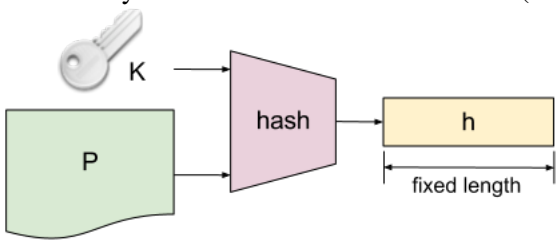


Figure 2.4: Hash Based Message Authentication

Upon creation of the hash value, a secret key is added to the pre-image, as shown schematically in Figure 2.4. Therefore, verification of the hash-value is only possible with the knowledge of the secret key K.

2.1.5 Random Number Generation

Randomness is an important concept and has many uses in cryptography. Two common cases where random values are needed, are key generation and *Nonces* (*N*umber-used-*once* [36]) in authentication protocols [16]. Stallings [50] defines ideal random values as unpredictable, statistically random and with uniform distribution.

In order to computationally generate random numbers used for cryptography, mainly two different approaches exist: **True Random Number Generators** and **Pseudo Random Number Generators**. *True* randomness denotes unpredictability even with unlimited computing resources while *pseudo* randomness tries to achieve unpredictability for limited computational power.

True randomness can be obtained by harvesting input from an entropy source. Sources for random values can be collected from physical processes like surrounding noise, key strokes or radioactive decay [50, 54]. As those values are not necessarily random, the gathered values are usually examined and post-processed to eliminate any pattern in the result. A possible example for a post-processor is a one-way hash function, like SHA-1 (see Section 2.1.7).

Because of the needed post-processing the cost to gain true random values can be significant [24]. The achieved throughput of true random number generators is lower compared to pseudo random number generators, which propose a cheap, efficient alternative. Pseudo random numbers look random while only using a small amount of true randomness (*seed*) and mathematical algorithms to produce a larger amount of random values. If the seed is large enough, those random values are unpredictable with limited computational powers. Therefore, SHA-1 or cryptographic algorithms can be seen as pseudo random number generators, too. See [31] for a discussion of different approaches to design random number generators.

2.1.6 Digital signatures

Signatures on (hardcopy) documents are used as a proof of authenticity, ownership or agreements. Digital signatures transfer those properties to digital documents or data with the use of encryption keys. While it is possible to use symmetric encryption for digital signatures ("arbitrated digital signatures" [48]), usually public-key schemes are used as they are more efficient in regard to the key management of the involved parties (see Section 2.1.2).

The basic protocol for digital signatures using public-key algorithms can be formally stated as:

$$\begin{aligned} \text{Signature:} \quad & H = hash(M) \\ & S = K_{priv}(H) \\ \text{Verification:} \quad & H = hash(M) == K_{pub}(S) \end{aligned}$$

A digital signature uses the *private* key from the sender. An encryption operation would use the public key of the receiver instead (see Section 2.1.3). To circumvent performance issues on long documents, the sender can use a one-way hash function (see Section 2.1.4) to reduce the number of bytes to sign. For verification of the signature, the receiver first decrypts the signature with the public key of the sender. As only one public key should be able to decrypt the message meaningful, the sender is bound to the document with his private key. Then, the resulting hash value is used for the verification of the signed message content: Only if a newly computed hash matches over the digest decrypted from the signature, the signed message is unchanged. The utilization of digital signatures is depicted in Figure 2.5.

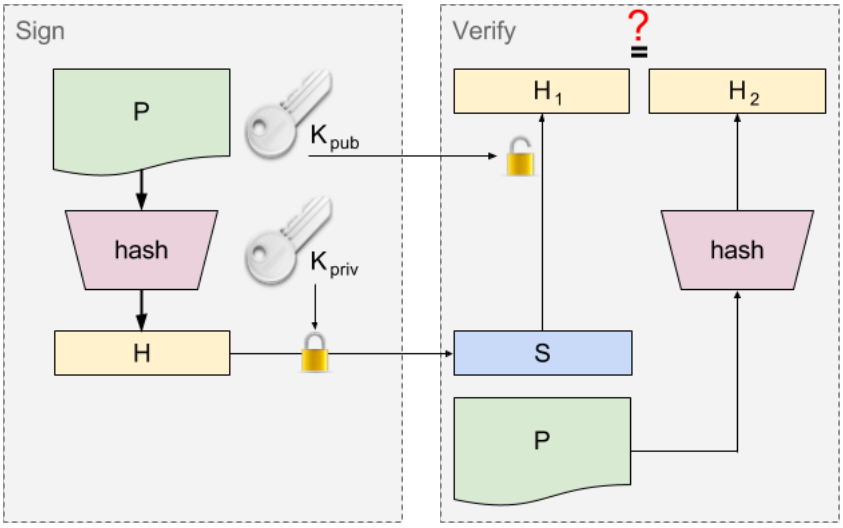


Figure 2.5: Signing and verification process with a public-key scheme. The sender encrypts the hash of a message with a private key (Signature, S) and sends it along the message to the receiver. The receiver decrypts the signature with the public key. This results in the hash digest computed by the sender. If this hash is of the same value as the hash of the message, the verification of the signature is successful.

2.1.7 Cryptographic Algorithms used in Trusted Platform Modules

The following section gives a brief overview of important algorithms and concepts that are relevant for Trusted Computing and Trusted Platform Modules (TPM). First, we discuss the RSA algorithm. This algorithm is widely used for encryption, decryption, signing and verification operations. Next, we present SHA-1 hashes, which are used in the authentication and authorization mechanism in TPMs. Finally, we introduce the concept of *Key Wrapping*, which is used to establish a trusted key hierarchy.

RSA

The RSA algorithm [46] is an public-key algorithm and named after its inventors Ronald L. Rivest, Adi Shamir and Leonard Adleman. It can be used for encryption and decryption as well as digital signatures. The algorithm depends on the *factoring problem*: It is presumed that it is hard to decompose a number that is the product of two large distinct prime numbers, back into the original factors.

The RSA algorithm works as follows [48]: First, two random large prime numbers p and q are chosen. For security purposes p and q should have the same length. For example a RSA key of 1024 bits length should use primes with a length of approximately 512 bits each [47].

$$n = pq$$

n will be used for the modulus of the public and private key. Then, a random encryption key e is chosen, where e and $(p-1)(q-1)$ are relatively prime. That is:

$$\begin{aligned} &\varphi(n) = (p-1)(q-1) \\ &\quad e \equiv 1 < e < \varphi(n) \qquad\qquad and \qquad\qquad gcd(e,\ \varphi(n)) = 1 \end{aligned}$$

φ denotes Euler's totient function, which counts the number of positive co-primes to n. The decryption key is computed with the extended Euclidean algorithm:

$$d \equiv e^{-1}\ (mod\ (\varphi(n))$$

The public key is formed by the numbers e and n, d is the private key. The encryption (c) of a message m is performed by:

$$c \equiv m^e\ (mod\, n) \qquad\qquad (0 \leq m < n)$$

If m is larger than n the message has to be divided into blocks where each block is smaller than n. The original message m is decrypted from c with:

$$m \equiv c^d$$

Regarding the above mentioned factoring problem, which is responsible for the security of RSA, it is presumed that the factor of n is needed to calculate m from c. This would allow to decipher the original message. Currently, no algorithm is known to solve this problem efficiently. However, a mathematical proof that the factor of n is even needed is outstanding [48].

SHA-1

SHA-1 is an implementation of the Secure Hash Algorithm (SHA) and belongs to a "family" of hash algorithms published as a Federal Information Processing Standard (FIPS-180) [51]. Another hash implementation described in this standard is referred to as SHA-2, which consists of the SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 algorithms. These algorithms differ in security strength and the data size which is used during the hash process. It is planned that future FIPS publications include a SHA-3 algorithm, which uses a different structure than the other algorithms of FIPS-180 [10].

SHA-1 produces digests of messages with 160 bits, which can be used for data integrity checks (see Section 2.1.4). Message digests are an important property in the creation and verification of digital signatures (see Section 2.1.6) and authentication.

Today, SHA-1 provides computational security for several years. In theory, 2^{80} operations are needed to brute-force attack a digest of 160-bit size. A collision on hash digests, as mentionend in Section 2.1.4,

targets hash operations by trying to find two distinct documents which result to the same hash digest. In 2005 a paper was published which discovered that collisions in SHA-1 are possible in much less than 2^{80} operations [66]: Further improvements on this attack reduced the complexity to find a collision down to 2^{62} operations [9]. In 2012 a attack was published which claims to reduce the amount of needed SHA-1 operations down to $2^{57.5}$ [53].

Key Wrapping

The goal of *Key Wrapping* is to protect a (sensitive) key in untrusted environments. The key to be protected may either be symmetric or asymmetric. For a symmetric algorithm the whole key (K_{sym}) has to be protected whereas only the private portion of the key pair (K_{priv}) for an asymmetric algorithm needs to be guarded. The result of a private key (K_{sym} or K_{priv}) encrypted with another (private) key is called wrapped key. The wrapped key is not usable without proper decryption of its corresponding *parent key* first.

It is possible to generate a whole tree hierarchy of dependent keys, where every decryption of one key K_n needs decryption by its parent key K_{n-1}:

$$\begin{aligned} \text{Wrap:} \quad & K_{wrap} = E(K_n, K_{n-1}) \\ \text{Unwrap:} \quad & K_n = D(K_{wrap}, K_{n-1}) \end{aligned}$$

Inside this tree key material is encapsulated. $E(K_n, K_{n-1})$ denotes the encryption of K_n with a key K_{n-1} while $D(K_{wrap}, K_{n-1})$ denotes the inverse process: The decryption of K_{wrap} with the key K_{n-1}. The topmost key of a tree is called *root-key* (K_R). This key is not protected by any other key, therefore another mechanism is needed to ensure protection.

In a sense a hybrid algorithm, which was briefly presented in Section 2.1.3, demonstrates a key wrapping strategy: A symmetric key is protected by the private portion of a public-key algorithm. This way, the fast symmetric key can be transmitted to the intended receiver over an untrusted medium. More generally this strategy can be applied when the storage of sensitive keying material on public readable media is needed.

For example, a TPM provides protected storage (shielded location, see Section 2.2.1). Access to this location is granted only after successful authorization. However, the storage capabilities on a TPM are limited. Therefore, key wrapping is used to securely store keys *outside* a TPM and load keys on demand prior usage. Moreover, the shielded location of a TPM can be utilized for protection of the root key.

2.1.8 Summary

This section presented basic building blocks of cryptography. After a short explanation of encryption and decryption operations, symmetric, asymmetric as well as hybrid algorithms were introduced. We outlined the practical usage of hybrid algorithms, which combine the speed of symmetric encryption with the more beneficial key management of asymmetric algorithms.

Furthermore, primitives used in cryptography, like hash functions and hash based message authentication, which are used to generate and verify fingerprints of digital data, were presented. Next, we discussed the importance of randomness and random number generation.

Finally we pointed out specific algorithms and important concepts, like digital signatures, RSA, SHA-1 and Key Wrapping strategies. Those concepts are core building blocks used in Trusted Computing and Trusted Platform Modules, which will be presented in the next section.

2.2 Trusted Platform Module Basics

After the cryptographic background of Section 2.1, the following section will give a quick overview of Trusted Computing and the role of Trusted Platform Modules (TPM). A TPM is the core technology around the concept of the domain-specific language `FTPM`. Therefore, the principle design of a TPM as well as its typical use cases are discussed. Furthermore, a more detailed look on command authorization mechanisms, key management and storage concepts is given. The following sections will provide a solid base for understanding `FTPM` and the design decisions leading to it.

2.2.1 Trusted Computing and Trusted Platform Modules

A Trusted Platform Module (TPM) is a low-cost mass-market chip which is permanently integrated onto a computing device and thus cannot be removed. It is platform-agnostic and a core component of Trusted Computing, which is a strategy to create more robust and secure systems with specialized hardware. Trusted Computing and TPMs are concepts invented by the Trusted Computing Group (TCG) [59]. The TCG is a not-for-profit organization that aims to define and promote the requirements for hard- and software needed for Trusted Computing. Per definition of the TCG, a trusted system is one "that behaves in the expected manner for a particular purpose" [55]. This means that a system can be trusted if it reliably reacts similar to previous events.

TPMs are integrated in most newly built computers. A TPM can be used for the creation and secure storage of cryptographic keys and thus provides a hardware root of trust. Furthermore, a TPM can be used to protect the platform against attacks on its integrity by storing measurement values. These values are used to detect unauthorized modifications. If a system is unmodified and not changed from one specific, known state, it can be trusted to work in known boundaries and thus satisfy the TCG's definition of a trusted system given above.

The current version of TPMs (version 1.2) uses the following cryptographic algorithms: RSA (see Section 2.1.7), SHA-1 (see Section 2.1.7) and HMAC (see Section 2.1.4). A TPM provides a Hardware Random Number Generator (see Section 2.1.5) as well as volatile and non-volatile memory. The functionality of a TPM is based on the following cornerstones:

- **Cryptographic Keys and Key Hierarchy** are used for encryption and signing arbitrary data;
- **Measurement** is used to record the state of a system;
- **Storage** is used to (securely) store keying material as well as measured values;
- **Attestation** is used to report the state of system to a third party.

Each of these concepts will be briefly discussed in separate sections.

Cryptographic Keys and Key Hierarchy

Before a TPM can be used, it is necessary to take ownership of it. As the module is typically disabled by default, it has to be enabled in the host computer's Basic Input Output System (BIOS) software. Next, a password is created and stored in a shielded location, to which unauthorized access is prevented by hardware mechanisms. This password generates the **Storage Root Key** (SRK), which is the base anchor of trust for all operations that either need authorization. The Storage Root Key is a RSA key pair (see Section 2.1.7) and protects all other derived keys inside the TPM. This means that if the TPM gets reset (which deletes the SRK), all dependent encrypted data is lost. The private portion of the SRK is not allowed to leave the TPM.

The **Endorsement Key** (EK), which is also an RSA key pair, is usually created and stored inside the TPM by the manufacturer during production time of the TPM . Its private portion is not allowed to leave the TPM, either. If the manufacturer included a digitally signed certificate, the EK can be used to verify genuineness of a specific TPM and thus can be used for remote authentication.

However, the direct use of an EK for unique identification of a TPM imposes a privacy problem: All other transactions made by the user become linkable and thus compromise the anonymity of a user. Therefore, an **Attestation Identity Key** (AIK) is used, which is an RSA key pair generated inside a TPM by the owner and thus unknown to third parties. The AIK is derived by the EK. As mentioned above, the private part of the EK is not allowed to leave the TPM. Therefore, it is possible to attest authenticity of a TPM with the AIK.

In order to vouch genuineness of a TPM without revealing *which* TPM is used to cope with the aforementioned privacy problem, the TCG proposes two solutions: First, a trusted third-party Privacy Certification Authority (Privacy CA), which manages identity information and checks the validity of an EK, can be used. However, the maintenance and abiding of trust to such an authority can be problematic, as the CA has to be involved in every transaction. Consequently, high availability of the CA is needed. Furthermore, the anonymity is now dependent on the CA but it cannot be excluded that the CA and the remote application, which requests authentication, collude and thus the same privacy problem continues to exist.

Therefore, in version 1.2 of the TPM specification, the TCG introduced the *direct anonymous attestation* (DAA) protocol. The DAA is a zero-knowledge protocol, which means that one party verifies a statement to another party without revealing any additional information. In case of DAA this means that not a certificate (EK, AIK) is used for authentication but rather a cryptographic proof that the owner uses a genuine TPM. The DAA protocol has the following entities: a DAA Issuer (e.g. manufacturer, IT administrator, ...), a DAA Signer (TPM) and an external party, who acts as a DAA Verifier. The protocol works a follows: First, the a certificate for a genuine TPM is issued by the DAA Issuer to a DAA Signer. With this certificate two things are proven:

1. The owner generated a signature which was approved by the DAA Issuer and
2. The DAA Signer is in possession of this certificate

With the possession of the certificate from the DAA Issuer, the DAA Verifier is now able to confirm the validity of the TPM. Implementation details of the DAA protocol can be found in [7].

The key material used by TPMs uses the RSA algorithm (see Section 2.1.7). This algorithm is capable of encryption, decryption, creation of digital signatures as well as verification. RSA keys are utilized by a TPM mainly in four different ways, namely as:

- **Signing Keys**
  Signing Keys are used to create digital signatures of data. The TPM enforces a *separation of duty* by preventing the use of signing keys for encryption operations and vice versa.
- **Storage Keys**
  Storage Keys are used to encrypt other keys to establish a key hierarchy (see Section 2.1.7).
- **Binding Keys**
  Binding keys are used to *bind* or *seal* data: *Binding* denotes the encryption of data with a specific storage key. *Sealing* means encryption of data with an additional *state-constraint*: The TPM has to be in a certain state in order to decrypt the data sucessfully.
- **Legacy Keys**
  Different keys usually have different modes of operation. Although it is possible to use legacy keys, which can be used for every operation, they weaken the aforementioned *separation of duty*. Therefore, the usage of legacy keys is deprecated.

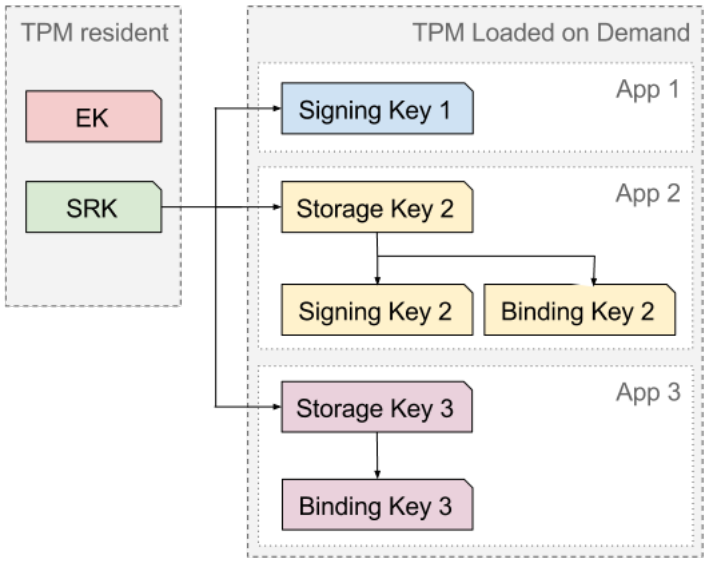


Figure 2.6: Example of a possible key hierarchy inside a TPM. Different applications can—depending on given rights—use different kind of keys. All keys are ultimately dependent on the SRK. The EK is used to attest that a TPM is from a certain manufacturer.

The last two remaining key types are **Identity Keys** and **AuthChange Keys**. An example for an identity key is the AIK, whose role in the remote authentication was briefly discussed above. AuthChange Keys were omitted in this enumeration because of their limited use in special TPM commands.

The SRK acts as a *Root of Trust* with regards to a key hierarchy: Every key created, loaded and executed inside a TPM is dependent on this key. An example key hierarchy is depicted in Figure 2.6, where different keys are utilized for different applications and use cases. However, the root anchor of trust for these keys is always the SRK, which is stored securely inside the TPM.

Storage

A TPM features shielded locations for cryptographic keys as well as unshielded locations for insensitive data. As storage inside a TPM is limited, it is possible to load needed data, such as keys, on demand from unsecured storage outside the TPM. Key wrapping mechanisms, which are discussed in Section 2.1.7, are used to protect sensitive data.

The storage area for measurement values inside the TPM is denoted by the term *Platform Configuration Register* (PCR). A PCR is an internal register capable to store a 160 bit digest – which is exactly the size of a SHA-1 hash (see Section 2.1.7). A TPM (version 1.2) has at least 24 PCRs, where the first 16 registers are static and the last eight are dynamic. Static registers are reset at boot time while dynamic registers are used by special mechanisms – such as *dynamic roots of trust* [22]. A PCR can always be read directly, but can never be directly modified. PCRs are an integral part to form a *Chain of Trust* (see Section 2.2.1).

Measurement

The configuration of a trusted system is measured and stored. On each startup of the system, these measurements are performed in sequence and thus form a *Chain of Trust*. A *Chain of Trust* in regard to measurement values means that for each validation of a value *m* a successful validation of the preceding measurement value *m-1* is required. The adherence of this sequence offers increased security because one single missing validation will result in the break of the *Chain of Trust* and thus allows to take

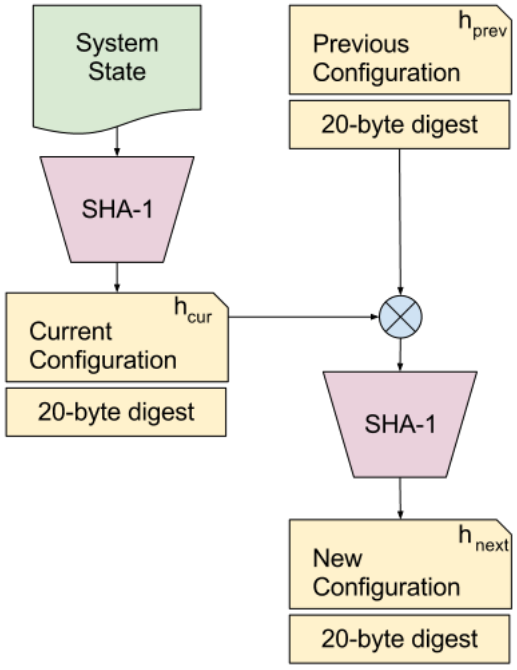


Figure 2.7: Extending a previously recorded digest to reflect an intentional system modification.

countermeasures, like aborting the startup routine. Hash functions, which were briefly discussed in Section 2.1.4, are used to calculate digests for those measurements.

An administrative action is necessary if a change in the *Chain of Trust* is needed—for example due to an intentional change in the hard- or software configuration. Stored measured values from a TPM cannot be directly overwritten but have to be *extended*. This *Extend* operation roughly works as follows: A digest of the new system configuration is calculated (h_{cur}). This digest is bitwise concatenated with the previous recorded digest of a former system configuration (h_{prev}). The resulting digest is hashed again to produce the final digest h_{next}. This digest now does not only reflect the current configuration but also implicitly keeps a history of all previous configurations as well. This concept is illustrated in Figure 2.7 and formally denoted by:

$$h_{next} = hash(h_{cur}||h_{prev})$$

Of course, the system has to keep an ordered list of individual measurements from old configurations (h_{prev}) in order to allow validation of new configuration measurements. The comparison of measurement values cannot be performed on the same machine because of obvious reasons: If a system is compromised there is no reason to trust neither reported measurements nor the software initiating measurements. Therefore, remote attestation is used to verify the state of a system to a remote party. This remote party can attest that a system is in a good, known state by comparing digitally signed measurement values it received from the TPM to locally stored reference measurement values.

Attestation

The term *Attestation* denotes the process to prove that a system is in a trusted state. This is done by performing the following three steps:

1. Measurement of configuration values (Chain of trust)
2. Reporting of measurements to a third party

3. Verification of the received measurement values by the third party

Measurement values are stored in PCRs inside the TPM (see Section 2.2.1). These values get digitally signed by an Attestation Identity Key (AIK, see Section 2.2.1), which proves that the signed statement originates from a hardware TPM. With this information the third party can compare received with expected measurement values and thus verify the state of the system.

Summary

This section presented the core functionality a TPM can provide for Trusted Computing. First, we discussed the concept of the key hierarchy as well as possible key types of a TPM. Next, we presented the role of measurements to sustain trust and how to extend this trust. Finally, we introduced two other important concepts, namely storage and attestation. In summary, a TPM can act as the Root of Trust for three properties:

- **Root of Trust for measurement**
  With the help of PCR records and current measurements, the state of a platform can be verified. The history of updated configurations is traceable in the PCR registers.
- **Root of Trust for reporting**
  The state of the platform can be reported to third parties to assure integrity and authenticity of the platform.
- **Root of Trust for storage**
  Sensitive data, like key material, is stored securely inside the TPM. Specific keys are not allowed to leave the TPM at all or only in encrypted form. Therefore, data encrypted with the help of a TPM cannot be recovered without it.

2.2.2 TPM Command mechanism

An Application Programming Interface (API) exposes the functionality of a TPM to an application. This API, named TCG Software Stack Specification (TSS), is defined by the available commands a TPM supports [58] as well as the needed data structures [57] for it.

Most TPM commands exposed by the TSS-API require proper authorization, which we will discuss in the next section. Then, we introduce the concept of *Rolling Nonces*, which is heavily used in command authorization. Finally, we present a quick overview of the common message structure of TPM commands.

Command Authorization

Most available TPM commands need proper authorization before execution, as they usually access or operate on data securely stored inside the TPM. There are different ways to establish authorization defined by the TCG, the two most important one being the *Object-Independent Authorization Protocol* (OIAP) and the *Object-Specific Authorization Protocol* (OSAP).

As opposed to the Object-Specific Authorization Protocol, the Object-Independent Authorization Protocol does not bind authorization to specific TPM entities and thus allows different commands to read or modify almost all objects held by the TPM. Furthermore, authorized communication sessions initiated by the Object-Independent Authorization Protocol remain open until explicitly closed by the calling application.

Object-Specific Authorization uses a certain usage secret to validate access to requested entities. This usage secret can be a password or a PIN-code. OSAP sessions end automatically under the following conditions:

- The entity for which authorization was given has been unloaded from the TPM,
- The usage-secret specific to the requested entity has been modified,
- Authorization data of keys or data stored inside the TPM has been modified, or
- The command for which the OSAP was initiated has failed.

Other authorization protocols defined by the TCG include the *Delegate-Specific Authorization Protocol* (DSAP), the *AuthData Insertion Protocol* (ADIP), the *AuthData Change Protocol* (ADCP) and the *Asymmetric Authorization Change Protocol* (AACP) [56, Chap. 13]

Rolling Nonces

Rolling Nonces are used to determine freshness of commands sent between a higher-level software stack and the TPM. Only parties that have knowledge about a shared secret – the *Nonce* (number used once [36]) – are empowered to execute a command. These Nonces are used to avoid *replay* attacks, which are used by attackers to gain more knowledge about a system. Replay attacks are carried out by intercepting and resending messages to the involved parties (in this case the TPM or TSS) again. Those attacks can lead to unwanted access to platform resources, if no countermeasures are installed. A general introduction to replay attacks can be found in [21].

The Rolling Nonce is a random 20 byte value. Every TPM command which needs authorization produces a new Nonce pair: A `nonceEven`, which is associated to the commands the TPM sends to the software stack (TSS) and a `nonceOdd`, which is associated to the commands the TSS sends to the TPM. These Nonces are used in the command authorization digest. Therefore, only the TSS that requested authorization and the TPM that answered the requests, know each others Nonce values. No other party should be able to calculate the accurate authorization digest and thus replay attacks can be avoided.

Figure 2.8 demonstrate the use of Rolling Nonces in the command authorization process by showing an excerpt of the `TPM_Sign` command, which is used to sign arbitrary data. The first step consists of the TSS requesting authorization for future commands by sending an `TPM_OIAP` (Object Independent Authorization Protocol) request to the TPM. This causes the TPM to generate and store a `nonceEven`, which is sent back to the calling TSS.

Next, the TSS generates a `nonceOdd` and uses both values, the `nonceEven` from the TPM and the generated `nonceOdd`, to calculate an authorization digest (`auth digest`). This digest, as well as the `nonceOdd` are sent to the TPM in the following request – the `TPM_Sign` command. The TPM uses its stored `nonceEven` and the received `nonceOdd` to calculate the `auth digest` itself.

If the calculated `auth digest` matches the received `auth digest`, the TPM knows that the TSS is the one which requested authorization with the `TPM_OIAP` command before, as only the TSS has knowledge of the `nonceEven` produced by the TPM. Figure 2.8 omits the response of the TPM, which would include the generation of a new `nonceOdd` and the recreation of a new `auth digest`. With these values, the TSS can verify that it communicates with the intended TPM simply by calculating and comparing the `auth digest` as well.

TPM Command Messages

Every TPM command sent to or from the TPM has a common message structure. This message structure consists of a header and a payload. The header includes information of the actual command to be executed as well as authorization information. The payload contains data dependent on the current TPM command.

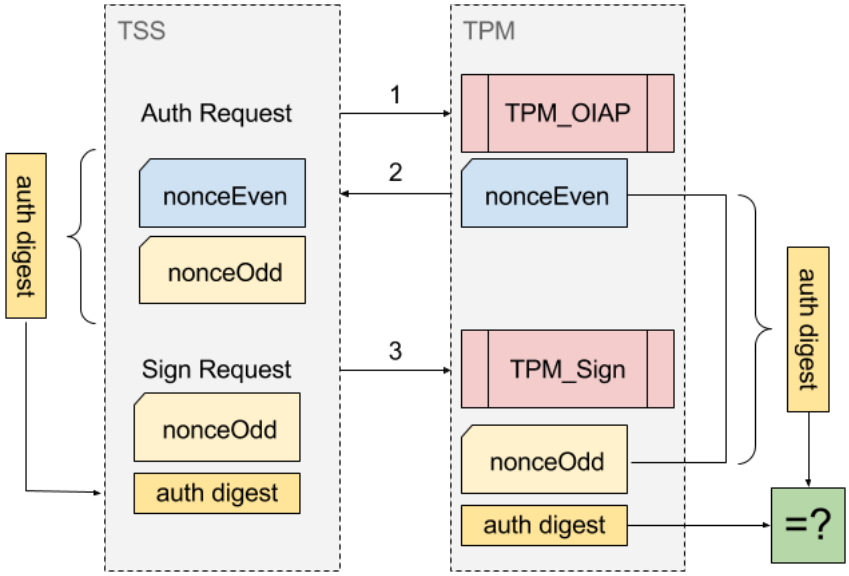


Figure 2.8: The use of Rolling Nonces in the interdependent authentication of the TPM and the TSS. First, the TSS requests authorization of the TPM with the use of the `TPM_OIAP` command. The TPM produces a `nonceEven` value and sends it back to the TSS. The TSS stores the received `nonceEven` and produces a `nonceOdd` value. Both values, `nonceEven` and `nonceOdd` are used in the calculation of the authorization digest by the TSS. This digest and the created `nonceOdd` value are send to the TPM with the next command, in this case `TPM_Sign`. To verify proper authentication of the TSS, the TPM uses the received `nonceOdd` and the stored `nonceEven` value to calculate the authorization digest as well. If both, the calculated and the received authorization digest match, the TSS is authenticated against the TPM and the `TPM_Sign` command is executed.

An important information inside the message header is the type of authorization required, which can be one of three different levels: Non-authorized, Single-authorized and Dual-authorized. This information is important to determine the correct number of authorization blocks inside a TPM command. A Non-authorized commands does not reference any object held by the TPM. An example of a non-authorized command is `TPM_Startup`, which initializes a TPM. The `TPM_LoadKey` command is a Single-authorized command, where a single cryptographic key gets loaded into the TPM for further usage. An example for a Dual-authorized command is the `TPM_ChangeAuth` command, which modifies authorization data of keys or data stored inside the TPM.

Every available authorization block has to be successfully validated before any action is performed. A failed authorization results in a Non-authorized message returning the failure to the calling application.

Summary

The TCG specifies different authorization protocols which validate access to resources held by the TPM. The most important ones are the Object-Independent and the Object-Specific Authorization Protocol.

Every authorization protocol heavily uses Rolling Nonces in the authorization process between a software stack (TSS) and a TPM. These Nonces are used to verify that posted commands are fresh and not replayed by an attacker in order to gain deeper knowledge of the system.

Every TPM command features a common message structure, which among other information contains the number of needed authorizations. Every single authorization block has to be successfully validated before an action is carried out, either by the TPM or the TSS.

Chapter 3

Related Work

The following chapter presents different existing solutions for the creation of software with a focus on an unambiguous and up-to-date documentation. First, we introduce dedicated specification languages on the example of the *Vienna Development Method* and the *Z-Notation*.
Next, we present literate programming, a program paradigm which emphasizes that programs are not only interpreted by compilers but also need to be understandable to humans. Therefore, literate programming especially targets the documentation process.
Then, we present two existing tools for automatic documentation generation—namely `Doxygen` and `Javadoc`—followed by a short motivation for language orientated modeling with the help of domain-specific languages in the next section.
All discussed approaches share the intention to create correct and maintainable programs, either through a formally correct and provable specification, a tight coupling between specification text and a resulting implementation or special focus on a concise representation. Moreover, all sections of this chapter use a common example as an illustration of the respective approaches.

3.1 Formal specification languages

A specification provides a solid base for subsequent correct implementations. As mentioned in [2], the former European Space Agency software engineering standards [17] highlight the importance of the *software requirements* (SR) phase in the life-cycle of software:

> "The SR phase is the analysis phase of a software project. A vital part of the analysis activity is the construction of a model describing what the software has to do, and not how to do it."

Any ambiguities in a specification can lead to potential severe errors in the resulting implementation. Certainly, it is beneficial to find potential problems early. This can be achieved by applying special tools and concepts during the SR phase. Simple testing for confirmation of the desired behavior—especially for critical or highly complex systems—might not be enough. Formal methods can provide such a proof of robustness to a certain degree through mathematical models and rigorous checks. As pointed out by [2], such a mathematical approach seems like an obvious choice because almost all other fields of engineering use mathematical models as well, for example Mechanical or Electrical Engineering.

Formal methods can be applied at different levels, beginning at the specification phase (*Formal specification*), through the actual development (*Formal developement and formal verification*), up to *Theorem provers*. A formal specification is very generally described in [35] as an

> "Expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy."

This description names four important concepts of a formal specification, particularly:

- **Formal**
  A formal specification consists of a certain syntax which allows precise descriptions of the problem domain (semantics). Additionally, a formal specification also needs "rules for inferring useful information from the specification (the proof theory)" [35].

- **Abstraction**
  The process of developing complex systems is usually an iterative process: Starting at a high-level description of the desired goals, this description is refined until it is specific enough that a system, which satisfies those goals, can be implemented. The finding of a well-balanced abstraction, which is neither too specific nor too vague, is difficult and requires experience and a deep understanding of the problem domain.

- **System**
  The system covers the problem domain of interest, for example a software or hardware specification. Knowledge about this domain is not only fundamental in order to guide the development and refinement of a specification but also to actually state what the problem is about.

- **Properties**
  Properties refer to the ultimate goals the specification strives to achieve. Among the goals are functional requirements and non-function requirements (performance, security, etc.). Complex systems can have a large amount of such properties to consider. Therefore, it is favorable to structure the specification in units which are interconnected through *structuring relationships* (specialization, aggregation, instantiation, enrichment, use, etc.)

A formal specification can increase the level of quality for documentation as well as the resulting implementation through the usage of these concepts. During the formalization different questions which challenge the problem and the chosen approach are raised and therefore can lead to new and better approaches. Furthermore, a formal specification describes the approach precisely and does not leave room for interpretations and ambiguities like a natural language might do. Subsequently, a formal specification allows automated processing with additional tools, like theorem provers, algorithmic model checking and generation of counterexamples.

The creation of a specification for non-trivial system is difficult and formal methods are not a widely adopted approach. Lamsweerde [35] states that formal specifications currently suffer from similar problems early programming languages had to face: Minor adoption because of complexity. The evolution of programming languages lead to better abstractions away from very low-level, machine-based instructions up to different programming models and paradigms (object-oriented, imperative, functional or logic paradigm [38]).

3.1.1 Formal specifications with VDM-SL and Z-Notation

Two of the most well known specification languages are the *Vienna Development Method Specification Language* (VMD-SL), which is part of the *Vienna Development Method* (VDM) and the *Z-Notation* language. VDM was originally developed by IBM in the 1980s to be used for programming language description and compiler design. It is a collection of techniques to model, specify and design computer-based systems. The Z-Notation was established by Jean-Raymond Abrial in the 1980s. It also targets the creation of precise specifications for computer systems.

VDM-SL and Z-Notation share a common ground: Both languages are based on mathematical sets, relations and predicate expressions. VMD-SL uses concepts found in programming languages and therefore is more similar to programming languages, while the primarily focus of Z-Notation lies in descrip-

tion of states. VDM-SL is standardized by the British Standards Institution (BSI) [8] and the International Organization for Standardization (ISO) [29]. The Z-Notation is standardized by the ISO as well [28].

3.1.2 Example

This section illustrates a formal specification of a stack with VDM. Many more examples of specifications written in VDM or Z-Notation can be found in [5], differences are illustrated for example in [25].

An informal description of the functionality of a stack is as follows: A stack is a data structure storing elements in last in, first out (LIFO) order. Therefore, two operations are necessary: `push`, to put an element on top of the stack, and `pop`, which removes the topmost element from the stack.
Additional operations are `peek`, which returns the value of the topmost element on the stack but does not remove it, and `isempty`, which tests if the stack currently has stored any elements. The function `init` just initializes a new stack – that means it may allocate needed resources.

In [30], a complete formal specification of this description, enhanced with allowed operations and error states can be found. Listing 3.1 shows a property-based approach with VDM-SL:

```
init:              -> Stack
push:    N x Stack -> Stack
top:     Stack     -> Stack
remove:  Stack     -> (N U ERROR)
isempty: Stack     -> Boolean
```

```
top(init())              = ERROR
top(push(N,Stack))       = N
remove(init())           = init()
remove(push(N, Stack))   = Stack
isempty(init())          = true
isempty(push(N, Stack)) = false
```

Listing 3.1: Function signatures (left) and semantics (right) of a stack in VDM-SL. `N` indicates an element (natural numbers in this case), `U` indicates set union.

Listing 3.1 shows which operations an implementation has to provide: `init` for initializing a stack, the helper function `isempty` and the essential `push` and `pop` (which is expressed in this example by the combination of the functions `top` and `remove`). Furthermore, Listing 3.1 shows constraints on the defined operations as well. Trying to `peek` an element of an empty stack yields an error because of an underflow condition: It is not possible to return a natural number from an empty stack. The third constraint avoids introducing another error state by extending `remove` to ignore empty stacks.

With the help of these constraints, room for (false) assumptions can be restricted and thus ambiguities be minimized. Moreover, this specification leaves out unnecessary implementation details, like how values are stored internally (array, linked list) and thus provides a good abstraction.

Formalizing such a specification not only enforces deep considerations about the system to be implemented and consequently leads to a cleaner implementation but also can be used as a reference point for proof construction. A proof shows, that an implementation works correctly (satisfies the specification) for all valid inputs. Such a proof is not possible with a specification in a natural language.

3.1.3 Advantages and Disadvantages

The process to formalize a specification raises questions regarding the system, its properties and states. These questions are valuable to gain a deep understanding and refine the specification to become concise and as unambiguous as possible. Mathematical models can act as a base to prove the compliance or noncompliance of an implementation to a specification as well as inconsistencies in the specification itself. This provides a tremendous advantage, especially in complex systems, to catch errors early.

Unfortunately, the creation of a formal specification is highly complex and thus time-consuming and expensive. Correctness is difficult to prove and comprehension of the resulting models is a demanding

process even for specialists [18]. Therefore, it is currently not feasible to model huge specifications with formal methods but rather apply them on small and critical parts of the specification.

The high complexity is the main barrier for applying formal methods on the over 700 pages of specification for TPM chips [56, 57, 58]. Although rigorous checks are extremely beneficial for detecting errors, formal methods do not help to broaden the comprehension of design principles to a larger group of people due to the aforementioned complexity.

3.2 Literate programming

Another approach to combine documentation and source code is defined through *Literate programming* – a programming paradigm characterized by Donald Knuth in the 1970s [32]:

> "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

Literate programming emphasizes the readability and therefore comprehension of computer programs. It focuses to explain what a program does for human readers, by giving the programmer the freedom to compose parts of a program differently to what a compiler would require. Simultaneously, literate programs can automatically rearrange the code prior to compiling to allow correct parsing. Thereby, and by choosing expressive variable names, a program should become like literature for the reader.

3.2.1 Literate programming tools – `CWEB` and `noweb`

Originally, literate programming was introduced with the `WEB` [32] system. `WEB` consists of two tools: A documentation language – TEX[1] – and `PASCAL`[2] as a programming language. Those two tools are inherent in literate programming: From one literate source it is possible to extract documentation (*weave*) and executable source code (*tangle*). As mentioned before, one strength of literate programming is to eliminate the strict structure a compiler might need (e.g. variable declaration in C89/ANSI C). This helps humans, who read the documentation, to understand the intention quicker. *Weaving* and *tangling* are usually performed by some soft of *preprocessor*, which extracts marked information in different order for one or the other command.

`WEB` or its successors, like `CWEB`[3] (which uses `C/C++` instead of `Pascal`) or `FWEB` (which can handle `C/C++`, `Fortran`, `RATFOR` and to some extent TEX), generate source code in a specific language. `Haskell` has support for literate programming directly built-in[4]. However, there exist also language agnostic literate programming tools, for example `noweb`[5]. While those tools are independent of the target language, they usually have to sacrifice some features Knuth considered part of literate programming (*pretty printing* of documentation and source code, i.e. using several fonts and indentation rules), because those features are highly language-dependent. However, some of the features (like *pretty printing*) can be applied subsequently by the use of different, specialized tools on the generated output as well.

The goal of literate programming is to provide better documentation in order to allow more people to understand the intent of a computer program with the help of a consistent and complete documentation. Knuth states that a competent programmer needs two tools: A typesetting language like TEX for documentation and a programming language for a concise implementation. A language like TEX or LATEX[6]

[1] http://www.tug.org/
[2] ISO 7185:1990
[3] http://www-cs-faculty.stanford.edu/ uno/cweb.html
[4] http://www.haskell.org/haskellwiki/Haddock
[5] http://www.cs.tufts.edu/ nr/noweb
[6] http://www.latex-project.org/

allows high-quality typesetting of (mainly) technical and scientific documents. By using both tools simultaneously, a complete and consistent documentation can be developed. Changes made to the program logic should (directly) lead to changes in the resulting documentation. Naturally, it still depends on the programmer to provide meaningful documentation — or any documentation at all.

3.2.2 Example

As in Section 3.1.2, a small example is used to illustrate literate programming. For comparison, a part of the same data structure—a stack—will be presented. The example is written in `noweb` notation and the complete listing of the example can be found in Appendix A.1. Please refer to Section 3.1.2 for a general explanation of a stack.

```
@ \section{Introduction}
This is a (minimal) implementation of a stack [...]
\begin{enumerate}
 \item [[pop]] \\ Removes the topmost element from the stack.
\end{enumerate}

<<pop.c>>=
int pop(STACK **head)
{
        if (empty(*head)) { error("Error: stack underflow"); }

        STACK *top = *head;
        int value = top->data;
        *head = top->next;
        free(top);
        return value;
}
```

Listing 3.2: Excerpt of the implementation of a stack using literate programming.

As shown in the Listing 3.2, a literate program in `noweb` contains the implementation source code along with the documentation. Documentation chunks are marked with `@` signs on the first column until either another documentation chunk or code chunk, marked with `<<NAME>>` is found. The chunks can be crossed-referenced, reordered and can be amended by adding information for a named chunk later in the file.

The documentation consists of pure LaTeX, which empowers the programmer to use the rich typesetting features. Therefore, a nicely formatted, cross-referenced documentation can be generated automatically at no additional cost for the user. The documentation also automatically provides links to keywords and function names and automatically creates an index. Moreover, `noweb` can output documentation directly in LaTeX and HTML.

3.2.3 Advantages and Disadvantages

With literate programming, a programmer has a tool to provide professional, structured documentation alongside a concrete implementation. Changes introduced to the implementation can be documented in the same source file and thus are more likely to get written down. With the use of specialized documentation systems, like LaTeX, high-quality, easy to navigate documentation can be generated. This aids comprehension of complex systems and thus reduces ambiguities.

Literate programming is still not a widely adopted approach in programming. Knuth cites Jon Bentley that one reason for the relatively rare spread of literate programming might be that the percentage of people good at programming is not necessarily good at writing [4]. While the usage of LaTeX provides

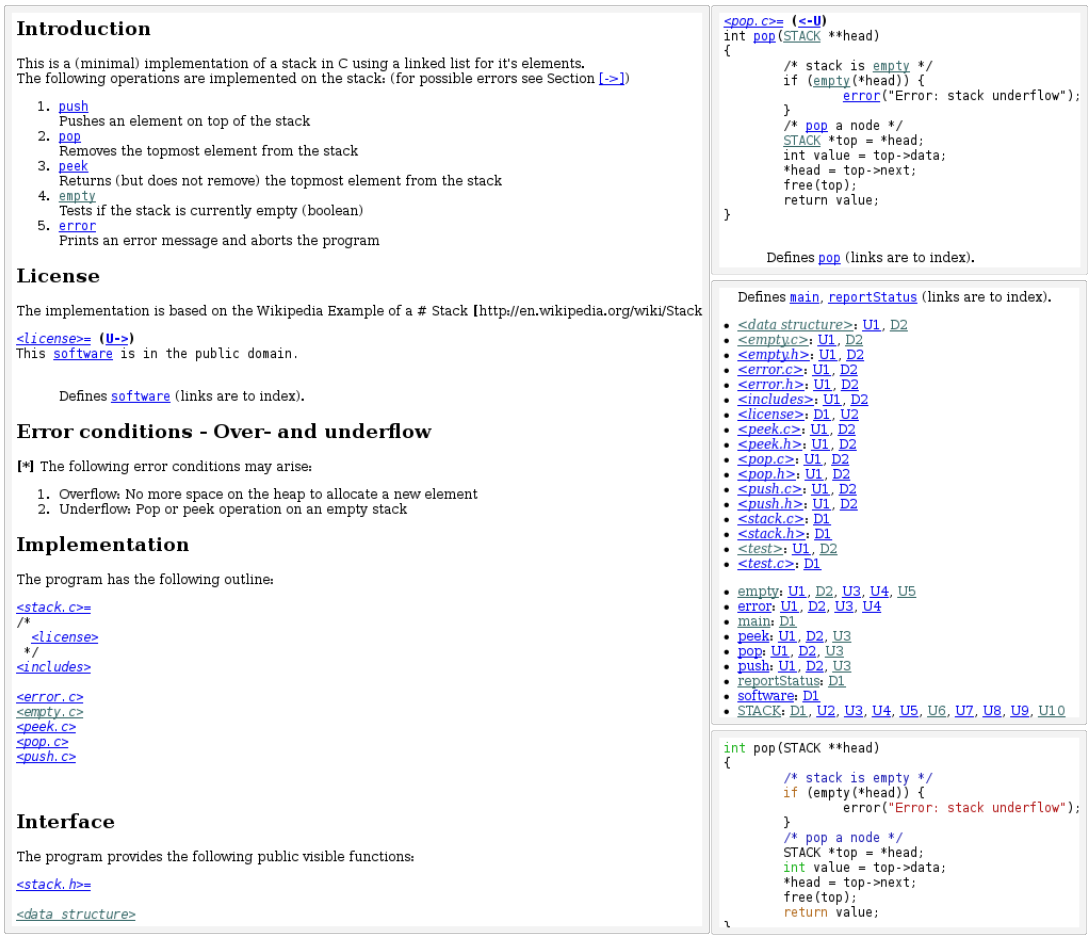


Figure 3.1: HTML-Documentation generation using `noweb` with the input from Listing 3.2. This is a combined screenshot, which shows some of the features of `noweb`: The right bottom column is the generated `C` source code – all the other fields are bits from the generated documentation: General descriptive text, implementation source as well as automatically created index of available keywords.

great flexibility, its syntax also disturbs the reading of the literate source file. Additionally, there are not many tools to aid development of literate programming to this date.

Another problem lies in the two-fold step to produce executable source code: Debugging of problems becomes more complicated. Either source code has to be transferred into the literate source after debugging a problem or the literate source file is directly used. If it is used directly, the programming chunks of the literate source have to be extracted and compiled afterward. Therefore, line numbers in error messages from the compiler may not be correct.

The concept to combine documentation and source code in one file and then automatically generate different output is definitely a good approach to keep an implementation in sync with its documentation. Many programming languages took this approach and use it actively—for example `Java` and `Javadoc` or the aforementioned `Haskell` language. This approach will also be used extensively in `FTPM`. `FTPM` strives to hide implementation details with a specialized syntax to allow a concise and easy to write description of the needed functionality, even for non-programmers or people without a background in a typesetting language like LaTeX.

3.3 Documentation generators

One key point of literate programming (see Section 3.2) is the tight coupling between documentation and implementation. The combination of documentation and source code should finally lead to a better and more complete documentation. Changes introduced to an implementation are more likely documented, if the adaption of the documentation is either near to the actual change of the implementation or at least in the same file.

Documentation generators use this principle of a tight coupling between implementation and documentation as well. The difference to literate programming is that their main focus lies not on a *literate* documentation, where documentation is arranged primarily for human readers. They rather use the available commenting system from the target language, annotated with macros and commands to structure the documentation. Therefore, documentation of code segments follows the rules of the used compiler and the syntax of the target language. Then, documentation can be generated by the use of specialized tools, which transform the enhanced comments into structured documentation.

Due to the fact that documentation generators use the comments from the target language, they automatically work with the respective available (specialized) editors. Modern editors can also provide additional features, like *folding*, which hide comments or a *dot-lookup* mechanism to provide a list of suitable commands while typing.

3.3.1 Doxygen and Javadoc

As with literate programming, documentation generators can be divided into language dependent and language agnostic tools. An example of a language agnostic tool is `ROBODoc`[7]. `ROBODoc` can be used for the documentation generation of all languages which supports comments. It uses a preprocessor to extract specially formatted comments and create documentation from it.

Examples of language dependent tools are `Javadoc`[8] – especially written for generating API Documentation in HTML for the `Java` language – and `Doxygen`[9]. `Doxygen` primarily targets "`C++`, `C`, `Java`, `Objective-C`, `Python`, `IDL` (Corba and Microsoft flavors), `Fortran`, `VHDL`, `PHP`, `C#`, and to some extent `D`".

Both tools – `Javadoc` and `Doxygen` – work similar: They scan source code comments for special keywords which drive the automated generation of documentation. This keywords can be used to provide documentation in different levels of detail (for example a brief description followed by an extensive explanation) as well as other useful features, like version information and the date of the last change. Additionally, this meta-data can be provided by an Version Control System (VCS), which automatically keeps this information up to date.

3.3.2 Example

The following section presents the stack example from the previous sections (see Section 3.1.2 and Section 3.2.2) to be extended with `Doxygen` annotations. `Doxygen` allows to use a `Javadoc` syntax for it's markup, which is shown here. The complete listing of this example can be found in Appendix A.2.

This example shows an excerpt of an informational header as well as the signature of a `pop` function. The header consists of a brief introduction followed by a more detailed explanation (abbreviated, for a full listing see Appendix A.2 or Figure 3.2). With relatively few and easy commands (`@brief`, `@param`, `@return`) a high-quality documentation in various different output formats, as shown in Figure 3.2, can be created.

[7] http://rfsber.home.xs4all.nl/Robo/robodoc.html
[8] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[9] www.doxygen.org/

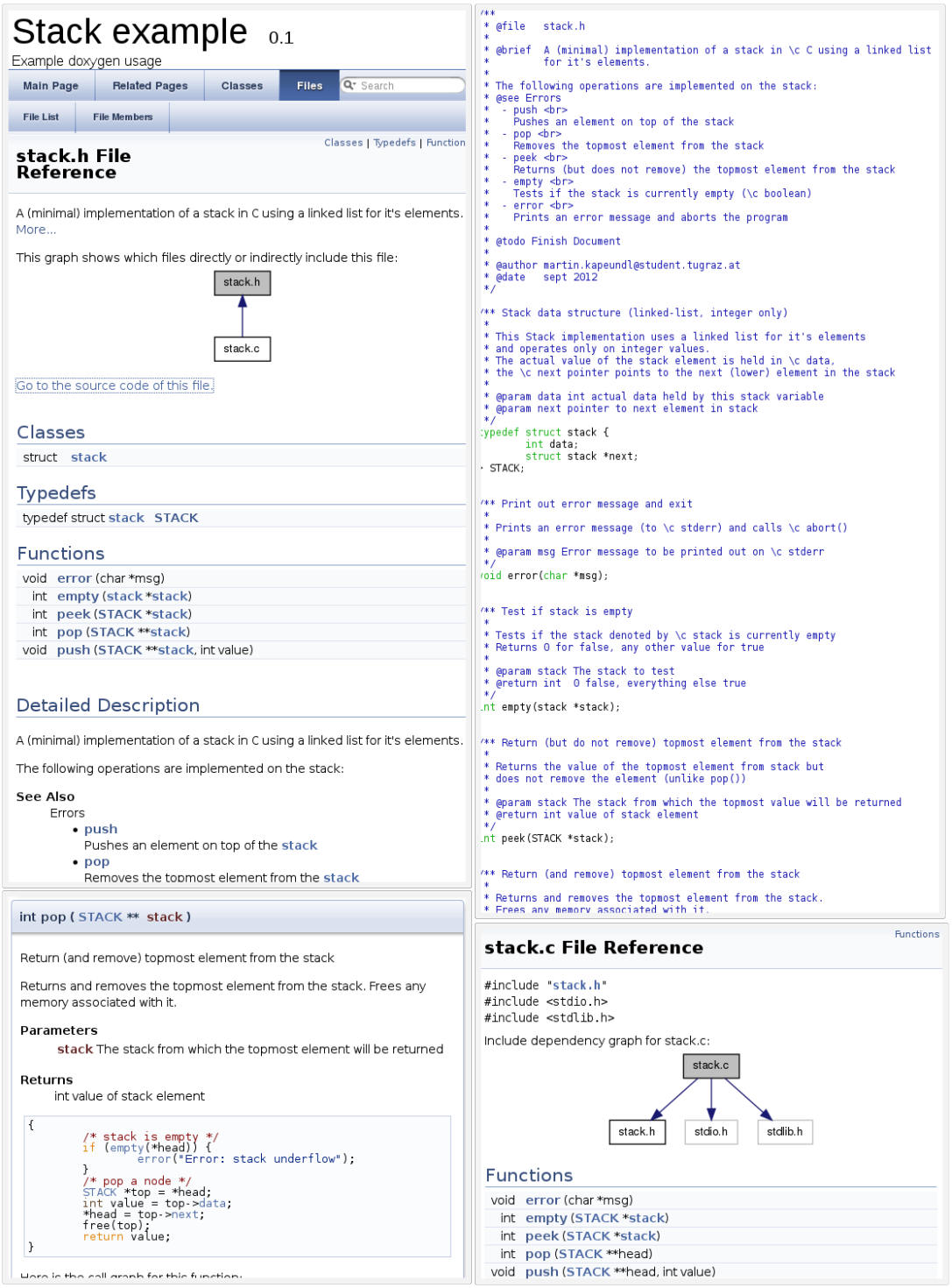


Figure 3.2: HTML-Documentation generation using `Doxygen` with the input from Listing 3.3. A combined screenshot, which shows some of the features of `Doxygen`: Based on the annotated comments of the `C` source shown in the top right corner, various different documentation and inheritance diagrams are created.

```
/**
 * @file   stack.h
 *
 * @brief  A (minimal) implementation of a stack in \c C
 *         using a linked list for it's elements.
 * The following operations are implemented on the stack:
 *  - pop <br> Removes the topmost element from the stack
 */

/** Return (and remove) topmost element from the stack
 *
 * Returns and removes the topmost element from the stack.
 * Frees any memory associated with it.
 *
 * @param stack The stack from which the topmost element will be returned
 * @return int value of stack element
 */
int pop(STACK **stack);
```

Listing 3.3: Excerpt of a C-source file annotated with Doxygen commands.

3.3.3 Advantages and Disadvantages

The concept to annotate source code comments is useful and widely adopted. Large software projects, like `Java`[10], `Qt`[11] or `KDE`[12] use a documentation generator to generate high-quality documentation for their systems successfully. The features of `Doxygen` and `Javadoc` are extensive, ranging from easy to navigate documentation with hyperlinks to automatic creation of inheritance diagrams.

However, we find that a language specific documentation generator combined with an actual implementation is not be the best choice for the creation of a specification of TPM chips [56, 57, 58]. `FTPM` strives to leave out any unnecessary implementation details: Essential information should be clearly visible and not clobbered by a complete source listing. Therefore, `FTPM` combines the approach of annotated comments for documentation with a focus on a concise syntax developed for describing the problem domain of TPM chips. The approach of designing dedicated languages for specific problems is presented in the next section.

3.4 Language oriented programming

Creating specialized tools, which concentrate on one thing and and one thing only, is a a common approach to solve problems elegantly. This idea is especially widespread in the long tradition of `Unix`[13] programs. Tools like `awk`[14] were specifically created to extract textual data from large files easily. After extraction, the elements can subsequently passed to other (specialized) tools, like `sed`[15] for further processing.

One of the authors of `awk`, Brian Kernighan[16], refers to this tools (or programming languages) as *little languages*: These are languages, which are usually used in a narrow domain, can be crafted to match a problem well and ideally allow to write code which is understandable, easy to maintain and able

[10] http://www.oracle.com/technetwork/java/index.html
[11] http://qt.digia.com/
[12] http://www.kde.org
[13] http://www.unix.org/
[14] http://cm.bell-labs.com/cm/cs/awkbook/index.html
[15] http://www.gnu.org/software/sed/
[16] http://www.cs.princeton.edu/ bwk/

solve a problem efficiently. Eric S. Raymond[17] expressed the "Rule of Generalization" in his book *The art of Unix programming* [44], which says:

> "Avoid hand-hacking; write programs to write programs when you can."

Raymond refers to details as a source of errors and delays because they tend to be overlooked or misinterpreted by humans. Therefore, good abstractions can lead to better (and more correct) specifications. The usage of automatic code generation is preferable in almost every situation, especially if it "raises the level of abstraction". This is the case, when "the specification language for the generator is simpler than the generated code, and the code doesn't have to be hand-hacked afterwards".

Such highly specified languages, are called domain-specific languages (DSL). As opposed to *General programming languages* (GPL), like `Java` or `C`, they focus on specific problem domains. `SQL`[18] `XML`[19] or aforementioned `awk` can be considered as famous examples of successful DSLs. As noted by Brian Kernighan, many (once originally) domain-specific languages are extended with (supposedly) useful features [37]. Thereby, the difference to a general purpose language subsequently vanishes and thus the DSL may lose its specific focus.

3.4.1 Domain-specific languages

Deursen et al. [14] outlines several different use cases of well known DSLs and provides a good definition for a DSL, too:

> "A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

A DSL can either be a standalone tool or it may be embedded into a host language (*embedded* or *internal* domain-specific language). For example, in [26] a DSL embedded into the `Haskell` programming language is described, which reduces the overhead of the creation of a complete new language. An internal DSL has the full power of its base language available and additionally there is no need to create a parser or compiler. Furthermore, there is no *symbolic barrier* [20], that is the DSL not being well integrated with a surrounding development environment. An example for a good integration in the development environment would be syntax highlighting in a code editor or the availability of integrated documentation tools like `Javadoc` (see Section Section 3.3). However, even if the syntax of a DSL is recognized and enhanced by the development environment it is not necessarily interpreted correctly. For instance, while syntax highlighting might be available for some language concepts it is not available for other, more specialized constructs. Or the available documentation annotation is not completely adequate for the currently chosen implementation.

In contrast to an internal DSL the syntax of an external DSL is not limited by a base language but only on the capability to build a adequate parser, compiler or interpreter. The liberty to freely select a suitable syntax for arbitrary problems comes at the price of the additional effort to build a parser, compiler or interpreter, too. This effort is not to be underestimated but its complexity can be reduced if specialized tools, like a *compiler compiler* or *parser generator* is used. Examples for well known parser generators are `ANTLR`[20] or `Bison`[21]. As the whole process of interpreting, compiling or translating the DSL source into other output is in the hand of the DSL designer, it can be customized even further.

[17] http://www.catb.org/ esr/
[18] ISO/IEC 9075-1:2011
[19] http://www.w3.org/TR/REC-xml/
[20] http://www.antlr.org/
[21] http://www.gnu.org/software/bison/

Through the representation of the DSL as an abstract syntax tree, different evaluations at runtime become possible.

As identified by Sprinkle et al. [49], the creation of a DSL *may* be expedient if either enough characteristics of the problem domain demand a DSL or if a single characteristic is significant enough. Among those characteristics *"repetitive elements or patterns"* or *"use by a domain expert"* fit very well in the problem domain to describe TPM commands. For example, almost every TPM command needs authorization before it can be used. This authorization process always follows the same pattern. Another example is the message structure of TPM commands: Every command consists of $0..n$ parameters for input as well as $0..n$ parameters for output. This fact can be used to not only visually outline the representation of input and output parameters in the `FTPM` source to allow fast recognition by readers, but also to automatically create marshalling (see Section 4.4) code responsible to compose and decompose input as well as output parameters in the actual executable code.

Another risk in the use of DSLs is denoted by "language cacophony" [20]: The mixture of different languages make it hard for people to use them. As languages are hard to learn it is often easier to use just one language as opposed to the mixture of different languages to reach the goal of accurately describing a certain problem. This risk can be avoided if the created DSL is both, simple to understand and use and yet powerful enough to solve the problems in the particular domain. Naturally, this is not an easy goal and usually needs experience and probably several attempts until this is achieved.

Often, a DSL is targeted at non- or lay-programmers, which are people who would not consider themselves as programmers but their work is very close to actual programming. The term "lay-programmer" is introduced in [20] and exemplified by users of working on spreadsheets. If the usage of a DSL has no immediate gain for the people using it or it is too complicated, it will get rejected. This is not only true for lay-programmers but of course for professional programmers as well. A DSL should enable its users to write and review code more productive. Moreover, a DSL should always be considered as part of a greater process. It will most likely not replace other used technologies but augment the available tools.

3.4.2 Example

The example of the previous sections (see Section 3.1.2, Section 3.2.2 and Section 3.3.2) is slightly modified to better suit as an illustration for a DSL. This time, not the functionality but a specific (fictional) implementation of a stack is described.

Listing 3.4, which is inspired by [20], uses an XML representation to describe certain properties of the stack. We previously mentioned that `XML` can be considered a DSL itself. While `XML` is readable by computers as well as humans, the creation of `XML` documents is not very user friendly because of its verbosity and thus prone to typing mistakes.

```
<?xml version="1.0" encoding="UTF-8" ?>
<data type="stack" values="int" name="foo" capacity="100"
      comment="A stack of Integers. Can hold up to 100 elements">
  <value>1</value>
  <value>42</value>
  <value>21</value>
</data>
```

Listing 3.4: XML representation of a stack.

Listing 3.5 depicts the same functionality as Listing 3.4. However, this time yet another (fictional) DSL is used to describe the exact same properties. We argue that the second representation is much more readable to humans and not as prone to typing mistakes because of the reduced verbosity. Listing 3.5 is more fluent and key properties—like the name of the stack, its capacity or its type—are more visible

than in Listing 3.4. Additionally, the source from listing Listing 3.5 can be easily translated to the XML representation of Listing 3.4.

```
foo is Stack of Integer:
  comment  = "A stack of Integers. Can hold up to 100 elements"
  capacity = 100
  values   = 1, 42, 21
```

Listing 3.5: DSL representation of a stack.

3.4.3 Advantages and Disadvantages

As there exist different general programming languages with specific strengths, there is definitely a place for small domain-specific languages for certain problem areas. Carefully designed domain-specific languages can match a particular problem domain very well and thus improve the creation, maintenance and readability of programs. Moreover, a concise domain-specific language can also reduce the complexity of challenging problems. If the problem domain has (enough) significant characteristics, the creation and usage of a domain-specific language can have tremendous advantages. However, the creation of a domain-specific language (external or internal) is not an easy task. Deep understanding of the targeted problem domain is required and expertise in the creation of programming languages is advantageous.

Internal DSLs facilitate the infrastructure of a host language, which is extended by special constructs according to the capabilities of the base language. Therefore, not only the compiler or interpreter of the host language can be used, but also available additional tools like syntax highlighting or documentation generators. However, the possible syntax of internal DSLs is restricted by the available syntax of its base language.

External DSLs offer the freedom of arbitrary syntax constructs tailored at very specific use case of the problem domain. At the same time, their creation needs more effort than the creation of an internal DSL, as a compiler, interpreter or translator has to be built, too.

No matter if the DSL is targeted at non-programmers or professionals, it will only succeed if it provides an additional benefit. Learning another language is always difficult, therefore the DSL has to be concise and focused on the problems it strives to solve [20].

3.5 Summary

This chapter presented an overview of existing approaches on how to improve a technical specification. First, we introduced dedicated specification languages, which use formal methods to create a mathematical model of the problem domain and describe its properties accurately. Subsequently, different levels of rigorous checks are possible to reduce errors and inconsistencies. However, as the utilization of formal methods is not trivial it might be suited better at selected, small portions or critical components rather than a large specification.

Then, we presented literate programming, which argues that both, a typesetting language for a structured and professional documentation and a programming language for a concise implementation, is needed. Literate programming provides a mechanism to rearrange parts of the source code in such ways, that they become more readable to humans. It is argued, that the tight coupling between source code and specification text leads to a more accurate description and up-to-date specification.

Literate programming may not have gained a widespread use, but certainly influenced the creation of documentation generators, which we presented by the example of Doxygen and Javadoc in the next section. Documentation generators use annotated source to generate professional documentation.

As with literate programming, the documentation and implementation is very tight coupled and therefore more likely to be kept up-to-date.

Finally, we introduced domain-specific modeling and domain-specific languages. These are little languages specifically tailored at particular problems and thus can reduce the complexity in the creation, maintenance and review of programs. However, the complexity to create a suitable domain-specific language is high and requires deep understanding of the targeted problem domain. If a good abstraction of the underlying problems can be found, a domain-specific language can—possibly combined with selected features of documentation generators—provide a useful tool for professional programmers as well as domain experts.

Chapter 4

Exemplified Implementation

The following chapter shows acquired experiences we made with a prototype FTPM implementation of selected TPM commands. First, we give an overview of the implemented commands and data structures. Then, we show how we realized the implementation of these commands in FTPM. We present the complete process of translating FTPM to C source code and automatically generate documentation suitable to be processed by the Doxygen program in detail. We use the opportunity to point out experiences and give insights to areas of FTPM which need future improvements. Moreover, we demonstrate the flexibility of having an executable specification available with a small utility, which outputs a summary of used bytes for TPM commands.

4.1 Selected TPM commands and structures

We chose to implement a subset of the available TPM commands in FTPM as a first proof of concept because a complete implementation of all available TPM commands is out of scope for this thesis. This chapter uses parts of the following commands to show the capabilities of FTPM and how the results look like:

TPM_OIAP is a TPM command to initialize authorization of objects (Object Independent Authorization Protocol). Most TPM commands and objects need authorization prior usage. The OIAP protocol is used by many TPM commands to achieve proper authorization.

TPM_PCRREAD returns the 20-byte digest value of a Platform Configuration Register (PCR). Stored measurement values are an integral portion in the attestation of trust and are stored in PCRs.

TPM_EXTEND updates the 20-byte digest value of a PCR to reflect configuration changes.

4.1.1 TPM_OIAP

Most TPM commands and objects need proper authorization before access to its entities, such as keys, is granted. Proper authorization is established by initiating an authorization session, which creates Nonces. These Nonces are used to authenticate the owner of TPM objects and to authorize its usage. Furthermore, they are important to prevent replay attacks. For a more detailed discussion of the available authorization mechanisms as well as implementation details refer to Chapter 2.2.2.

The TPM_OIAP is an authorization command that provides an authorization that is independent of subsequent commands. After a successful authorization with TPM_OIAP any object held by a TPM is accessible. The command is designed for efficiency, as one successful authorization is sufficient for

multiple subsequent commands. The authorization session is persistent until a TPM command fails or the upper software stack explicitly closes the session (with the TPM_Terminate_Handle command).

The TPM_OIAP command does not have any incoming parameters. However, it produces the first nonceEven, a random value of 20 bytes, and an authorization handle (4 bytes), which are used for subsequent authorization, as outgoing parameters. As the number of concurrent authorization sessions is limited, the TPM_OIAP command may fail, which is indicated by the return value TPM_RESOURCES. Otherwise, the authorization was successful and thus the value TPM_SUCCESS is returned.

4.1.2 TPM_PCRREAD

The TPM_PCRREAD command is related to the storage capabilities of a TPM. Measurement values are stored in Platform Configuration Registers (PCRs) inside a TPM. These values are needed for remote attestation of the expected system configuration (see Chapter 2.2.1). With the TPM_PCRREAD command the stored 20-byte digest value of a previous measurement is exported to the calling application. There is no authorization necessary for an application to read the digest values stored in the registers.

The command takes an integer value of the number for the desired PCR as input and reports back the stored digest to the application as output. If an invalid PCR is selected, the command fails with the TPM_BADINDEX return code.

4.1.3 TPM_EXTEND

TPM_EXTEND is also a storage related command. It is the only command for modification of stored digest values in PCRs. The extend algorithm, which includes past digests in the generation of current configuration values is explained in detail in Chapter 2.2.1. The TPM_EXTEND command does not need authorization prior usage.

As input parameters the command takes an index number for the PCR to extend as well as a new digest. If the selected PCR is valid the TPM concatenates the new digest with the digest available in the specified PCR. The newly created SHA-1 (see Chapter 2.1.7) digest is subsequently stored in the PCR and returned to the calling application.

4.1.4 Data structures

Naturally, the TPM commands operate on data structures. These structures are specified in the TPM Specification Part 2 [57]. There are common data structures used by all TPM commands, and structures which are needed only by the respective command. Among the common data structures are shorthand type definitions for various integer types (e.g. UINT8, which is used for an unsigned integer of 8 bits size) or the declaration of various return values (e.g. SUCCESS, FAILURE, etc).

Additional data structures used by the TPM_OIAP commands are nonces and handles. A handle is 32 bit number pointing to a resource internal to the TPM. The TPM uses the handle to unambiguously identify an requested entity. A nonce is a digest of 20 bytes length. The storage related commands, TPM_PCRREAD and TPM_EXTEND operate mainly on digests. In essence, these digest do not differ to nonces: They are an array of 20 bytes. and Section 6.3.5).

4.2 Implemention in FTPM

This section exemplifies a prototype implementation of the selected commands in FTPM. The commands are shown in Listing 4.1 to Listing 4.7. After each code listing, we will highlight important features and discuss identified weak points. Furthermore, we immediately show achieved results. We state future improvements for a roadmap to the next versions of FTPM.

4.2.1 FTPM Program Header

The original specification is divided into different parts: *Design* [56], *Structures* [57] and *Commands* [58]. These documents reflect different facets of the specification. We find that the overall structure is well suited to introduce the TPM specification to the reader. However, the chosen presentation lacks usability because different parts of the specification do not cross reference to each other. Therefore, the implementation process can become cumbersome: The implementation workflow is heavily disturbed if hundred of pages of specification text have to be flicked through in order to reach information about a certain data structure. This is especially inconvenient during the implementation of a complex TPM command.

As a consequence, one design goal of FTPM is to provide an improved usability in the navigation and visual presentation of the TPM specification. The latter is achieved with a coherent layout while automatic hyperlinks between different specification parts enhance the first. This way, fast movement throughout the specification becomes possible. Currently, we concentrated our efforts on a better integration of structures and commands. Future versions of the FTPM prototype would need to focus on the integration of the *Design* document in the workflow, too. Every TPM command automatically links to its needed data structures. Moreover, data structures cross reference to any used types as well. Therefore, it becomes much easier to look up the composition of complicated data structures.

Listing 4.1 shows the typical header of a command or structure implemented in FTPM. The first two lines are taken from the TPM_OIAP command and are comments that hold meta information for the documentation generation. This information is important to the automatic documentation generation as it indicates to which part and chapter of the TPM documentation the subsequent statements belong. For the example in Listing 4.1, the FTPM prototype therefore puts the documentation information gained from the TPM_OIAP command in a chapter called *Authorization Sessions* during the actual documentation generation process.

```
1 --!file commands
2 --!chapter 18 Authorization Sessions :: 18.01 TPM OIAP
3
4 include "../../structures/2_basic_definitions/2_2_defines.ftpm"
5 include "../../structures/4_types/4_4_handles.ftpm"
6 include "../../structures/5_basic_structures/5_5_tpm_nonce.ftpm"
7
8 include "helper/oiap_helper_stubs.ftpm"
```

Listing 4.1: Meta-comments and included files for the TPM_OIAP comment. Due to the meta-comments, the documentation is generated in a chapter with the title *Authorization Sessions* in the *Commands* part of the specification.

Then, the next few lines of Listing 4.1 show the inclusion of needed data structures, namely basic definitions (integer values, result codes, etc) on line 4, handles (pointer to internal resources of a TPM) on line 5 and the data structure for Nonces on line 6 (see Chapter 2.2.2 and Chapter 4.1.1). Currently, Line 8 includes another file, which this time is no structure but a helper file for the actual implementation of the TPM_OIAP command. This helper is used in order to not clutter the ordinal implementation with implementation details. Moreover, this separation is used to provide an interface between specification designers and implementers. Implementation details that cannot be modeled with FTPM can be added in these functions. Additionally, changes brought to the specification do not interfere with custom code from an implementer during code generation if details and principal command structure are separated like this.

Listing 4.2 shows part of the TPM_NONCE structure, which is included on line 6 from Listing 4.1. Again, the meta-comments on line 1 and 2 are used in the documentation process and are responsible

```
--!file structures
--!chapter 05 Basic Structures :: 05.05 TPM Nonce

include "../2_basic_definitions/2_2_defines.ftpm"

--! The number of bytes for a TPM_NONCE
constant TPM_NONCE_SIZE is 20;

struct TPM_NONCE is (
  --! This SHALL be the 20 bytes of random data.
  --  When created by the TPM the value MUST be the next 20 bytes from the RNG.
  nonce BYTE[TPM_NONCE_SIZE]
);
```

Listing 4.2: The TPM nonce structure in FTPM. A nonce is a random sequence of 20 bytes.

that documentation for this structure is generated in the *Structures* chapter. More precisely, the documentation ends up in a sub-chapter named *TPM Nonce* which is part of a larger chapter entitled *Basic Structures*. Listing 4.2 also needs the basic definitions, which were included in Listing 4.1 earlier. FTPM automatically ignores this instruction in order to prevent multiple definition of the same structures. FTPM takes care internally to not include a file more than once (see Chapter 6.3.2 for details).

Generated C source and header files

The FTPM prototype automatically produces compilable C source code and header files. Of course, the generation of C source implies that it is also possible to solely generate headers files from an FTPM input. This is useful in an early stage of the development of a TPM specification to outline the design of the modules. Every generated source contains the token "ftpm" in the filename to indicate that this file has been automatically generated by the FTPM translator.

Code generation is executed recursively. This means, that every included file to one FTPM source is automatically processed by the FTPM compiler as well. Subsequently, code (and documentation) is generated for every dependent file, which guarantees an up-to-date code base.

```
#ifndef FTPM_5_5_TPM_NONCE_FTPM_H
#define FTPM_5_5_TPM_NONCE_FTPM_H

#include "2_2_defines_ftpm.h"

#define TPM_NONCE_SIZE 20

typedef struct {
        BYTE nonce[TPM_NONCE_SIZE];
} TPM_NONCE;

#endif // FTPM_5_5_TPM_NONCE_FTPM_H
```

Listing 4.3: A generated header file for the TPM_NONCE structure. Macro guards (#ifndef..#define..#endif) are included for every header file generated by FTPM.

The include mechanism of FTPM, which is shown in Listing 4.1, is very similar to the include mechanism in the C programming language. If headers from the standard C library are needed, for example for data types or functions, they are pulled in automatically during the generation of code. As an example,

the generated header for Nonces is shown in Listing 4.3. A macro guard (*#include guard*) is generated for every header file to prevent multiple inclusion in the resulting C code. The actual declaration of a Nonce is straightforward: it consists of a struct which contains an array of size TPM_NONCE_SIZE of the datatype BYTE. The datatype BYTE is defined in 2_2_defines.h as an alias for the primitive datatype unsigned integer 8bit type in FTPM, which maps to uint8_t in the generated C source code.

Generated documentation

Figure 4.1 depicts the resulting documentation from the code snippet of Listing 4.1. With just a few lines of FTPM, a documentation is available to specification implementers, which has the following features:

- **Consistent navigation**
  On top of every documentation page resides a navigation bar (1) which provides fast access to the different parts of the specification. Moreover, Doxygen generates a search-field on the top right of the documentation page.

- **Structure of the documentation**
  Every TPM command or structure is placed inside the chapter denoted by the meta-comments (2). Therefore, related comments and structures can be grouped together like in the original specification.

- **Cross referenced sections** The resulting documentation is deeply cross-referenced. An informational paragraph on top of the page quickly shows, which data structures are needed for the depicted command (3). Every data type is cross referenced until it resolves to one of the primitive data types in FTPM (see Chapter 5.2.3).

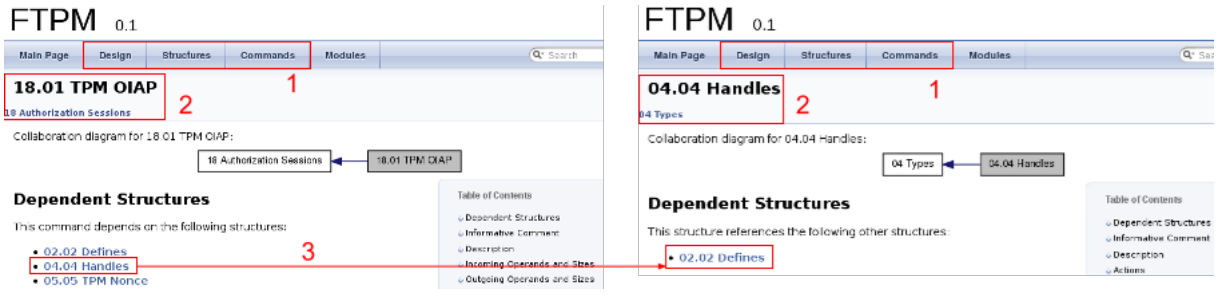


Figure 4.1: Doxygen documentation generated from meta-comments in FTPM and include files. The main navigation bar (1) provides links to the different parts of the specification. Commands and Structures are organized in chapters (2) of their respective parts. All parts are cross referenced with each other to allow easy navigation (3).

Advantages

Every dependent file to a FTPM source is automatically and recursively re-generated during processing. Therefore, the specification is always up-to-date. The biggest advantage of the meta-comments in FTPM and the list of include files is the resulting documentation. A logical and consistent structure allows fast navigation throughout the different parts of the specification. Moreover, every dependent structure is listed for the reader and references to its declaration exist. In the resulting C code the FTPM prototype automatically generates *include guards* to prevent multiple inclusion. Resulting source files are marked with the token “ftpm” to indicate that these source files were automatically generated by the FTPM prototype.

Further improvements

Currently, the available meta-comments allow the creation of chapters with sub-chapters. For a more complex (deeper nested) layout the comment parser of the FTPM prototype, especially the meta-comments, has to be adapted.

The present form of the include mechanism in FTPM is rudimentary. Currently, there is no abstraction to the physical files on the hard-disk. Furthermore, a concise notation for include files would shorten the include paths and thus remove a possible source for typing errors. The classification of include files in modules, like packages in Java, could improve readability through logical grouping. Furthermore, this approach would remove the necessity of meta-comments at all and thus provide greater flexibility.

4.2.2 Informative and Descriptive Comments

After the inclusion of needed structures and helper files informative and descriptive comments of the implemented TPM ordinal or structure may follow. Examples for such comments are shown in Listing 4.4, which uses information provided from the TPM_OIAP and TPM_PCRREAD commands.

```
--!info
--\warning No informative
--         comment set

--!desc
--\warning No descriptive
--         comment set
```

```
--!info The TPM_PCRRead operation provides
--      non-cryptographic reporting of
--      the contents of a named PCR.

--!desc The TPM_PCRRead operation returns the
--      current contents of the named register
--      to the caller.
```

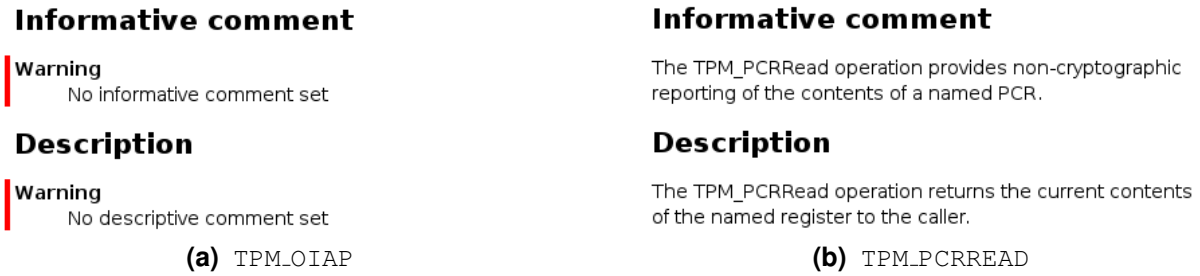
Listing 4.4: Informative and descriptive comments for the TPM_OIAP (left) and TPM_PCRREAD (right) commands: If no informative or descriptive documentation text exits, the specification designer can use a Doxygen markup (\warning) to (visually) emphasize in the generated documentation output that this section is not finished.

In Listing 4.4 the TPM_OIAP command. which is taken from the official TPM specification, does not have an informative or descriptive note in the corresponding chapter but rather has a general discussion of the the authorization mechanism in a dedicated chapter. Therefore, the implementation of TPM_OIAP in FTPM neither has an informative nor a descriptive comment. For consistency the informative and descriptive comments are included and annotated with the Doxygen keyword \warning, which visually marks these sections in the generated documentation. This is a beneficial reminder for specification designers to include information during the process of creating or changing a specification version. The TPM_PCRREAD command does provide informative and descriptive texts, which are rendered in dedicated sections. The result of Listing 4.4 is shown in Figure 4.2.

Advantages

The informative and descriptive comments are essential parts in the TPM specification. They provide important information about the overall design, used data types and features of a TPM command. Therefore, this information is one of the primary sources for implementers of the TPM specification.

If the documentation for the specification is automatically generated a consistent layout on every page is guaranteed. This consistence is useful because the reader of the specification knows what to expect in each chapter and does not get surprised. If a section is still incomplete, the reader is informed with specially marked regions. The highlighted regions immediately give hints about the status and simultaneously act as a reminder for specification designers to update or complete the provided information.



(a) TPM_OIAP **(b)** TPM_PCRREAD

Figure 4.2: Examples of informative and descriptive comments in the generated Doxygen documentation: In (a) the sections are specifically marked through the Doxygen keyword \warning. This keyword results in regions entitled "Warning" which are visually marked with a red bar on the left side to indicate missing information.
A complete informative and descriptive comment section is shown in (b) for the TPM_PCREAD command. All comments are generated from the FTPM source of Listing 4.4.

Further improvements

With the current FTPM prototype, different type of comments (informative, descriptive, actions) can be described easily. However, there is no notation to link between chapters directly in the comments. This is useful to supply the reader with additional, related information. The TPM_OIAP command shown in Listing 4.4 is an example where such an improvement would be beneficial, as the authorization mechanism of TPMs is described in a dedicated chapter.

4.2.3 TPM Command declaration

The next listing, Listing 4.5 shows the incoming and outgoing parameter block of the TPM_OIAP (left) and TPM_EXTEND ordinal (right). Although the TPM_OIAP command does not have any visible incoming parameters in the implementation of FTPM, the generated documentation automatically adds the parameter which are common to every TPM command. These additional parameters consist of a number, which specifies the authorization type of the request (TPM_TAG), the size of the passed incoming parameters as well as an identification number for the ordinal (TPM_COMMAND_CODE). The omission of these common parameters contributes to an uncluttered representation of parameters in FTPM and allows the reader to focus on the parameters that are actually important for the implementation.

These common parameters are automatically included for the outgoing parameter as well, followed by the other outgoing parameters distinct to the respective commands: authHandle and nonceEven for TPM_OIAP and outDigest for TPM_EXTEND. As a TPM may be used concurrently by multiple sessions, the authHandle is used to identify a specific authorization session. The parameter nonceEven holds the random bytes generated by the TPM_OIAP command while the modified digest after a successful execution of TPM_EXTEND is stored in outDigest. With this notation and omission of common parameters, the list of incoming and outgoing arguments of a TPM command is kept as short as possible.

Generated C source and header files

The FTPM prototype generates the header file shown in Listing 4.6 for the TPM command from Listing 4.5. Every TPM command has to return an enumeration of type TPM_RESULT. Incoming parameters are passed by name while outgoing parameter are passed by reference. The generated TPM command in C has the same parameters as the implementation in FTPM. This means that parameters common to every TPM command are stripped from the generated code as well. However, at this stage these parameters are

```
ordinal TPM_OIAP

  incoming:

  outgoing:
    --! Handle that TPM creates that
    --  points to the authorization
    --  state.
    authHandle TPM_AUTHHANDLE,

    --! Nonce generated by TPM and
    --  associated with [the] (sic)
    --  session.
    nonceEven  TPM_NONCE
```

```
ordinal TPM_Extend

  incoming:
    --! The PCR to be updated
    pcrNum  TPM_PCRINDEX,

    --! The 160 bit value representing
    --  the event to be recorded
    inDigest TPM_DIGEST

  outgoing:
    --! The PCR value after execution
    --  of the command.
    outDigest TPM_PCRVALUE
```

Listing 4.5: Incoming and Outgoing parameter block for the TPM_OIAP (left) and TPM_EXTEND (right) commands. Even if no incoming parameter is present, the incoming block has to be specified in FTPM for consistency. Parameter blocks in FTPM leave out common parameters that are shared by every TPM command. These shared parameter are generated automatically and thus contribute to a clean notation for the specification of parameter in FTPM.

not necessary anymore. As parameters to or from a TPM are passed as a binary stream, a special code to translate the stream into named incoming parameters and back from named outgoing parameters into a binary stream for the upper software stack is necessary (*marshalling*).

It is possible to automatically create such marshalling code with FTPM during code generation. This code would use the first 10 bytes of the passed parameter stream to detect which TPM command has been requested. Then, based on the command name, it would split the binary stream into the necessary parameters, accordingly. However, the implementation of the FTPM prototype does not provide any marshalling code at this stage of development.

```
TPM_RESULT TPM_Extend(
        /* incoming */
        TPM_PCRINDEX pcrNum,      // The PCR to be updated
        TPM_DIGEST inDigest,      // The 160 bit value representing the event
                                  // to be recorded

        /* outgoing */
        TPM_PCRVALUE *outDigest   // The PCR value after execution of the
                                  // command.
);
```

Listing 4.6: A part of a header file automatically generated by FTPM for the TPM_EXTEND command. Incoming parameters are passed by value whereas outgoing parameters are passed by reference.

Generated documentation

The documentation generated for incoming and outgoing parameters by FTPM resembles the look from the original specification but with several improvements: First, the types of the parameters link back to their declaration. This is a vast improvement in the navigation especially if the number of pages for a complex specification is as large as with the TPM specification. With a few clicks on the generated hyperlinks all relevant data is at hand for implementers and specification designers. Figure 4.3 shows

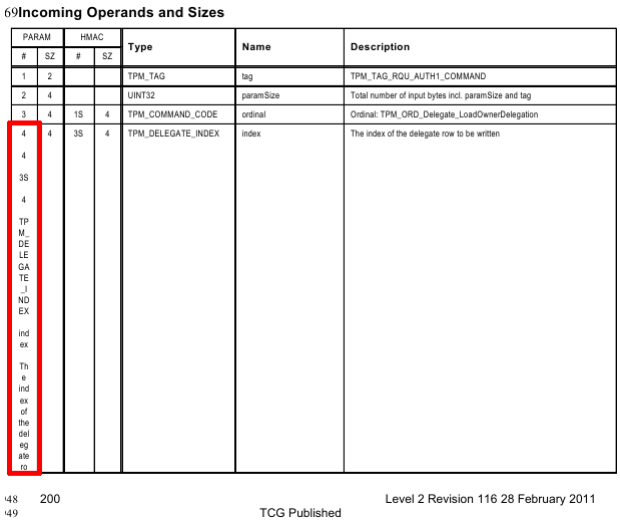
the generated documentation for the TPM_OIAP command from Listing 4.5 with a possible navigation through the documents marked with red arrows: A click on the parameter type TPM_NONCE in the *TPM Commands* section of the documentation leads to the *Structures* section where the type for TPM_NONCE is defined. There, all relevant information about this type is shown. As a TPM_NONCE consists of an array of another user-defined type (BYTE) another link leads to the definition of BYTE, which is defined in yet another different section of the *Structures* section of the documentation. As the official TPM specification does not have hyperlinks between related chapters, a navigation like this through the different documents can become quite tedious.

Figure 4.3: Generated documentation output for the TPM_OIAP command. The red arrows show a possible navigation through the documentation, as types in the parameter table of TPM commands are links to the respective type definition in the TPM structure file.

Another improvement is the consistency in the visual representation which is also achieved through the automatic generation. For example, the layout of the incoming and outgoing parameter is guaranteed to be always in the exact same sequence: Common parameter information (parameter number and size), HMAC information (number and size), parameter type, name and a description. If this representation ever needs to change, a single modification in the underlying template responsible for output generation is all that is necessary to automatically adapt every affected source file. Because of the automatic generation of documentation it is also guaranteed that every chapter follows the same structure and is coherently formatted—from headlines to paragraphs and emphasis on certain elements (e.g. informative, descriptive and action comments). As the current available documentation for TPM modules is handcrafted it does contain small mistakes. FTPM cannot prevent typing mistakes but it helps the editors to provide a coherent representation with minimal effort and thus avoid mistakes as shown in Figure 4.4 from the current TPM specification.

Advantages

The presentation of incoming and outgoing parameters in FTPM is decisive. Unimportant parameters are left out. With the use of dedicated blocks it becomes immediately clear for the reader, if a parameter is either incoming or outgoing. The resulting C code uses this information as well. Comments indicate the differentiation between incoming and outgoing parameters. Because of the automatic generation, this distinction is consistent throughout every generated command.



69 **Incoming Operands and Sizes**

PARAM #	PARAM SZ	HMAC #	HMAC SZ	Type	Name	Description
1	2			TPM_TAG	tag	TPM_TAG_RQU_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of input bytes incl. paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Ordinal: TPM_ORD_Delegate_LoadOwnerDelegation
4	4	3S	4	TPM_DELEGATE_INDEX	index	The index of the delegate row to be written

48 200 Level 2 Revision 116 28 February 2011
49 TCG Published

Figure 4.4: A layout error, which can be found in the current TPM specification (version 1.2, revision 116) is marked with a red rectangle: The properties of last incoming parameter from the command `TPM_DELEGATE_LoadOwnerDelegation` is wrongly written in a single column instead of the usual layout.

The generated documentation tremendously eases the navigation through the different parts of the specification because the type of every parameter is cross-referenced to its declaration. Moreover, every aliased type links back to its original declaration. Thereby, implementers can browse through *all* relevant places of the TPM specification for a single command. Moreover, the documentation is consistent, both in structure and visual presentation.

Further improvements

The marshalling code for the `C` programming language, which parses the incoming or outgoing parameter stream into named parameters could be generated by `FTPM` as well. Currently, there already exists the possibility to generate additional, pre-defined helper functions during the automated generation of code with `StringTemplate`. Functions to output debug statements are examples of installed helper functions. A simple `Makefile` is another one, which is also auto-generated by `FTPM`. Therefore, this additional output needs to be extended to include marshalling code for each TPM command. Moreover, the marshalling code could also be generated automatically from the TPM commands: The command name can be used as enumeration value and the overall size of the incoming and outgoing parameters is (mostly) directly contained in the parameters itself.

4.2.4 `FTPM` Program Body

The texts for informative and descriptive comments (see Figure 4.2) ranges from small notes to multiple lines or pages. These texts usually give an overview of the purpose as well as background information in the informative statement. Moreover, the original TPM specification provides concrete directives for TPM commands in a dedicated *Action* section. The TCG defines every text besides the informative text to be normative. Furthermore, they may contain several keywords, like “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT” etc, which are to be interpreted according to RFC-2119 [6].

These keywords are often used in both, informative and normative, comments throughout the specification. Moreover, these keywords are well suited to enforce custom rules during the execution of `FTPM`, too. For example, on page 118 of Part 2 of the TPM Main Specification [57] the following instruction with regards to return codes is given:

> "When a command succeeds, the TPM **MUST** return TPM_SUCCESS. When a command fails, the TPM **MUST** return a legal error code."

This means that every command either has to return TPM_SUCCESS, or in case of an error it must return some other specified value of the enumeration TPM_RESULT. This is actually a rule which is checked by FTPM. If an ordinal does not provide a result code, the parser emits the following warning:

```
[ERROR] foo.ftpm (15): the ordinal 'TPM_FOO' MUST have a return value of type
                       `TPM_RESULT'
```

If the result code is not known to FTPM it is not legal. This means that it has not been declared as an valid enumeration value and therefore the following warning is printed:

```
[ERROR] foo.ftpm (15): 'NOT_A_VALID_RET_CODE' not found in 'TPM_RESULT.
    NOT_A_VALID_RET_CODE'
```

Return codes are an example for a reasonable enforcement of custom rules by the FTPM compiler. More of these rules are spread throughout all parts of the TPM specification, and thus it is quite easy for implementers to unintentionally disregard a rule. The FTPM compiler is a tool for implementers and specification designers to enforce abidance of these rules.

Listing 4.7 shows the main implementation body of TPM_OIAP in FTPM. The command has essentially just two directives: First, an authHandle has to be acquired. Then, the first Nonce (nonceEven) used in any subsequent authorization during the session needs to be created.

Every comment inside an ordinal is automatically an *action* comment (see Chapter 5.2.1), which are primarily used in the documentation generation. After several of such *action* comments (line 2 to 14) the implementation tries to acquire a valid authorization handle via a call to the custom written function setAuthHandle. This function is defined in the helper file oiap_helper_stubs.ftpm, which was included at the top of the source listing (Line 8 in Listing 4.1). If the TPM does not have any authorization handles left, the return code TPM_RESOURCES is returned. Otherwise, the implementation proceeds to the second tasks of creating a new Nonce. Line 22 again calls a custom function (setNonceEven) for this purpose. This function is also defined in the same helper file. The possibility to use custom function calls in FTPM allows the implementer of a TPM command to cleanly outline the main purpose of the command. With the custom function calls setAuthHandle and setNonceEven the command is not cluttered with implementation details and thus the main intent is perceivable more quickly.

The command ends with another action comment (line 24 to 27). This comment is marked with the Doxygen keyword \warning as the official specification is ambiguous at this point. In the generated documentation this statement is visually marked and can be reviewed by the editors to improve the provided information. As mentioned above, the last statement in every ordinal has to be a return statement, which is shown on line 28. It indicates either a successful execution of the command or specifies a failure code (line 17).

It is not the goal nor possible for FTPM to include every implementation detail. At some point, the implementation effort is not feasible anymore. The helper function setNonceEven, shown in Listing 4.8, is an example, where FTPM stops to provide support for the creation of implementation details. While it would certainly be possible to create a nice syntactical element for the generation of random numbers in FTPM, which are needed for a real Nonce, the abstraction level ends here and the function is marked as a stub. However, the function still returns useful values for testing purposes: A Nonce, which solely contains values of 0xA5 is generated. This is a default *initialization vector* which is specified by the TCG and can be used by implementers of the TPM specification to test an implementations with known values before it is released. They are a valuable mechanism to compare computed digests or other values to spot errors in the implementation and thus a good starting point for a real implementation from source code generated by FTPM.

```
1  begin
2      --! 1. The TPM_OIAP command allows the creation of an authorization
3      --     session handle and the tracking of the handle by the TPM.
4      --     The TPM generates the handle and nonce.
5
6      --! 2. The TPM has an internal limit as to the number of handles that
7      --     may be open at one time, so the request for a new handle may
8      --     fail if there is insufficient space available.
9
10     --! 3. Internally the TPM will do the following:
11     --     * TPM allocates space to save handle, protocol identification,
12     --       both nonces and any other information the TPM needs to manage
13     --       the session.
14     --     * TPM generates authHandle and nonceEven, returns these to caller
15     if setAuthHandle(authHandle) == TPM_HANDLE_STATUS.TPM_INVALID_HANDLE
16     do
17         return TPM_RESULT.TPM_RESOURCES;
18     end;
19
20     --! 4. On each subsequent use of the OIAP session the TPM MUST generate
21     --     a new nonceEven value.
22     setNonceEven(nonceEven);
23
24     --! 5. When TPM_OIAP is wrapped in an encrypted transport session, no
25     --     input or output parameters are encrypted.
26     --\warning What are the consequences of this statement?
27     --         Is this really an **ACTION** Statement?
28     return TPM_RESULT.TPM_SUCCESS;
29 end;
```

Listing 4.7: Ordinal implementation of `TPM_OIAP`. The listing shows the ordinal body, which consists of multiple *action* comments (label 1 to 5), a conditional (line 15) and a call to a helper function (line 22).

Generated C source and header files

The `FTPM` statements from Listing 4.7 can directly translated to valid `C`. `FTPM` provides static type checks in order to reveal ambiguities in the TPM specification. For example, temporary helper variables or parameter occasionally have the same name in the original specification. As the parameters described in the specification are passed as a stream of bytes to the commands this is not a real problem. However, self-explanatory variable names help comprehension and avoid confusion. With the `FTPM` prototype, the parser is able to pinpoint multiple uses of the same names and point in the relevant lines in the source file.

Listing 4.9 shows the translation of the helper function `setNonceEven` from Listing 4.8. Every argument to a function is passed as a pointer. `FTPM` takes care of handling pointer notation internally, as shown in Listing 4.9: The assignment operation inside the loop (line 7) correctly uses `->` instead of a dot to access structure pointers. Likewise a dot is rendered for locally defined structures in the resulting `C` source code. Moreover, the type of the index variable (`i`) for the loop is generated automatically, depending on the size of the loop. As the size of a Nonce evaluates to 20 (bytes) an `uint_8_t` is adequate for the loop index.

As mentioned before, `FTPM` generates output for every dependent input source (include files) recursively. Per default, it writes `C` source code and header files and a simple `Makefile`, which can be used to test the generated source on syntax errors or warnings with a `C` compiler. Errors or warnings from static type checks during the execution of `FTPM` are printed to the standard output and consist of the filename and the line to which the error or warning belongs. An example for such an output is shown

```
--! \todo just a stub - please provide a proper implementation
function setNonceEven(nonceEven TPM_NONCE)
begin
    --! \todo: randomize nonce - this is an initialization vector
    for i in 0 to TPM_NONCE_SIZE
    do
        nonceEven.nonce[i] := 0xA5;
    end;
end;
```

Listing 4.8: Implementation of the helper function `setNonceEven` for the command `TPM_OIAP`. The real implementation would use random values for the Nonce, the implementation in `FTPM` provides the initialization vector specified by the TCG.

```
1 void setNonceEven(TPM_NONCE *nonceEven )
2 {
3     /* \todo: randomize nonce - this is an initialization vector */
4     for (uint8_t i = 0; i < TPM_NONCE_SIZE; i++) {
5             nonceEven->nonce[i] = 0xA5;
6     }
7 }
```

Listing 4.9: Automatic handling of pointers in the generated `C` source. The data structure for `nonceEven` is passed as a pointer (line 1), the assignment (line 5) reacts accordingly. The index variable `i` in the for-loop (line 4) is automatically created as well.

in Listing 4.10. The first example shows a warning which is emitted if an unsigned integer variable gets assigned a negative value. The second example demonstrates how the `FTPM` prototype reacts if an undefined variable (`aUndef`) is used. `FTPM` does not stop the processing of input source if an error is detected but continues the parsing of the source code as long as possible. Hence, as many potential problems as possible are shown to the implementer at once.

```
1 aError unsigned integer 8bit := -32;
2 aUndef := "foobar";
```

```
1 [ERROR] demo.ftpm (1): aError, -32 have incompatible types in
2         'aError unsigned integer 8bit [...]'
3 [ERROR] demo.ftpm (2): Cannot resolve 'aUndef'
4 [ERROR] demo.ftpm (2): Cannot assign 'string' to 'void'
```

Listing 4.10: Example for warnings emitted by `FTPM`: Assignment of incompatible data types (line 1) and assignment to a variable, which has not been declared previously.

Generated documentation

Action comments from the `FTPM` source are extracted for documentation generation. They are the last component for a complete documentation of either a TPM structure of command. As with *Informative* and *Descriptive* commands, there is currently no notation to link between related chapters within the *Action* comments. `FTPM` automatically arranges the *Action* comments as a numbered list, which describes the necessary steps a TPM command has to carry out in sequence. Line 11 and 14 of Listing 4.7 result in a nested list depicted in Figure 4.5. If an action comment starts with a star character (`*`), a sub-list is

automatically created by the translator. The identified ambiguity in the original specification mentioned above is visually marked with a red bar on the left side and the keyword “Warning”.

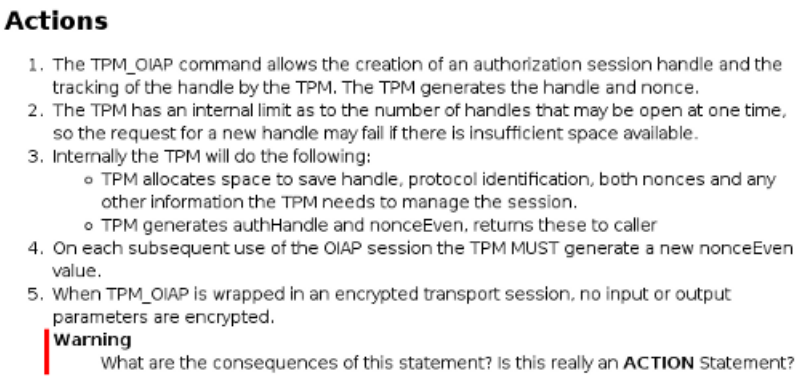


Figure 4.5: Example of action comments of the `TPM_OIAP` command. The comments are structured as an enumerated list. A nested list is created, if the comment starts with a star character and can be seen at directive three. `Doxygen` keywords, as the warning shown in step 5, can be used to visually mark sections in the output.

Advantages

Static type checks provide a valuable source of information for specification designers. `FTPM` identifies misused or undeclared variables or parameters and gives hints if a variable or parameter is declared multiple times. In the original TPM specification the multiple declaration of helper variables and similar parameter names occur. Moreover, it is possible to enforce rules which are contained in the informative, descriptive or action commands throughout the specification. This approach asserts the compliance to the defined rules in every implemented command or structure.

Further improvements

Currently, only a few custom rules which enforce program logic are implemented. As the number of implemented commands progresses, the number of possible checks, which will then be subsequently applied to all TPM commands and structures, increases.

Other areas of improvement consist of more built-in language features. For example, the existing `FTPM` prototype has no syntactical element to set a single value for a whole array. All it takes would be an extension in the main grammar file and an a renderer for `C` source code, which maps the assignment to a call to `memset`, `memcpy` or similar existing function calls. Additionally, a static type check rule would be advantageous, in order to prevent the assignment of incompatible types for arrays. As with custom static rules, beneficial language features can be added if more TPM commands and structures are implemented.

4.3 Gained flexibility through automatic output generation

A single run of the `FTPM` prototype produces multiple outputs concurrently. For the `C` programming language we generate source code and header files as well as additional files (helper functions and a `Makefile`) automatically. Because every dependent source file is processed by the `FTPM` compiler it is possible to guarantee that the output is always up-to-date. The compiler cares that even multiple edits on different parts of the specification will still result in a working implementation. Moreover, the generated output can be changed simply. The templates used for code generation are plain text files, which can be edited independently to the `FTPM` prototype.

The FTPM language hides complex language elements, such as pointers or memory management from the reader. These are language-dependent details that, in our opinion, should not be included in a technical specification. At the same time, it is possible to enforce the abidance to defined rules in the specification. With the conventional approach of a manual implementation, the abidance to these rules is under the responsibility of the implementer. Given the sheer size of the TPM specification an automatic validation is unarguably a better approach.

The approach taken by FTPM to produce documentation for TPM modules and structures is similar to the code generation above. We generate Markdown files which are subsequently processed by the Doxygen documentation generator. Markdown is a plain-text format, which eases the creation of structured text with an easy to use syntax. As with the automatic generation of C source code, documentation for one FTPM input source recursively generates the documentation of every source this input source depends on. With this approach, the documentation of all involved objects is guaranteed to be always up to date.

One documentation run generates multiple helper files. First, introductory texts for each of the three main parts of TPM are generated: Design, Structures and Commands. Then, various configuration files for Doxygen are generated:

- The main Doxygen configuration, which is adapted to contain needed configuration options for markdown and the FTPM output.
- A Doxygen layout file, which describes the layout of the generated documentation and contains links for easy navigation of the three parts of the official TPM specification (Design, Structures, Commands)
- A Cascading Style Sheet (CSS) file, which is used to adapt the visual representation of the generated documentation to closely mimic the official available specification.

Again, each of this files is available in a template file (see Chapter 6). The templates are plain text files and thus ease modifications and furthermore guarantee a consistent visual presentation of the documentation. Moreover, because of tightly cross-referencing related parts of the documentation, we achieve a vast improvement in the possible navigation. With a few clicks on the generated hyperlinks all relevant data is at hand for implementers and specification designers.

As the documentation of FTPM is automatically generated it is guaranteed to be up-to-date with the source. The current workflow to generate the documentation consists of specification designers and editors. The editors format and publish what the specification designers write. During the reformatting of the layout errors like in Figure 4.4 can be introduced. However, it is also possible that parts of the documentation contain old and obsolete information. If the documentation is generated directly from the input source it is guaranteed to always contain the latest information.

In combination with Doxygen incomplete or missing elements can be indicated easily. These areas are marked with keywords such as \warning or \todo and immediately convey the reader that this section is not ready yet or has known problems. An example of such indication is shown in Figure 4.6. At the very beginning of the action statements the specification designer gives a hint that this command is not completed yet. Furthermore, a note advises future improvements: A link to the relevant section where the important concept of *locality* is explained. This would definitely help to better understand difficult sections and therefore the specification as a whole.

A complete history of all changes made to the specification, as well as the thoughts and considerations leading to it, is possible if a version control system (VCS) is used for its storage. As FTPM sources are plain text files they are well suited to be kept in a VCS. Together with descriptive log messages the commits in the VCS provide a history of the development process of the TPM specification, which is valuable for specification designers and implementers. Currently, the TPM specification provides a *Change History* and a *Questions Section* at the start of each of the main specification parts. The history

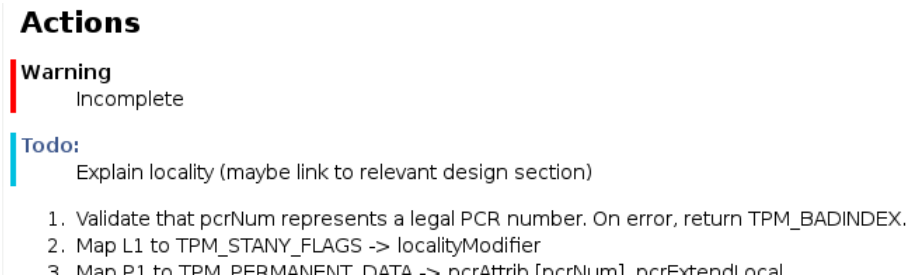


Figure 4.6: With the special keywords `\warning` and `\todo`, which are available in `Doxygen`, areas in the documentation are easily marked as incomplete or as work-in-progress.

lists differences from previous specification revisions. The section dedicated to questions keeps a history of design decisions. In the introduction to the TPM design, the editors explicitly state the importance of the development history [56]:

> "The question section keeps track of questions throughout the development of the specification and hence can have information that is no longer current or moot. The purpose of the questions is to track the history of various decisions in the specification to allow those following behind to gain some insight into the committees thinking on various points."

This is, among the storing and distribution facilities, the exact purpose of a VCS. A VCS aids in the automatic creation of the *Change History* as well as to provide an access point to the *complete* history including the design decisions written as log messages.

4.4 Bytesize information for a TPM command

As a demonstration for a convenient tool, which becomes easily realizable because the TPM specification is available in an executable representation with `FTPM`, we implemented a small utility. This utility shows the number of bytes a TPM command *at least* consumes. The utility walks the abstract syntax tree and gathers information of how many bytes a certain ordinal block consumes. Then, it prints a summary of the values to standard output, as shown in Listing 4.11. This can be used for example to quickly determine the I/O buffer sizes needed for TPM commands. This is advantageous particularly for devices with limited memory, for example embedded devices.

```
Ordinal:     TPM_OIAP          Ordinal:  TPM_PCRREAD          Ordinal:   TPM_EXTEND
---------------------          ---------------------          ---------------------
Total:      45  Bytes          Total:      44  Bytes          Total:      89  Bytes
---------------------          ---------------------          ---------------------
Incoming:   10  Bytes          Incoming:   14  Bytes          Incoming:   34  Bytes
Outgoing:   34  Bytes          Outgoing:   30  Bytes          Outgoing:   30  Bytes
Body:        1  Byte           Body:        0  Bytes          Body:       25  Bytes
```

Listing 4.11: Byte-count utility demonstrated on the `TPM_OIAP` (left), `TPM_PCRREAD` (middle) and `TPM_EXTEND` (right) commands: It prints the number of accumulated bytes for input and output parameters as well as any bytes used inside ordinals or specified helper functions.

The amount of bytes accumulated for the entry *body* consists of temporary variables declared in either the ordinal itself or any specified helper functions. For the `TPM_OIAP` command, the single byte shown in Listing 4.11 comes from the index variable `i` (an `unsigned integer 8bit`) in the `for` loop declared on the stack from Listing 4.8.

Although the TPM command TPM_OIAP does not have any input parameters, the byte count utility incorporates the size of the common parameter header, which consists of 10 bytes: two bytes for the type of authorization for the command and respectively four bytes for the overall size of the passed parameter as well as the TPM command code. As these parameters are common to every TPM command, implementations written in FTPM do not explicitly include these parameters (see Section 4.2.3). However, an actual implementation needs to set and include these parameters as well and therefore they are considered in the byte-count utility, too.

4.5 Summary

This chapter demonstrated the usage of FTPM. We provided a selection of three commands to show how a specification for TPM components can be written with FTPM. We showed advantages as well as shortcomings of the current FTPM prototype in detail. Moreover, we discussed the different generated outputs for the selected commands. The output consists of executable C source code and header files, documentation in Markdown format prepared for processing with Doxygen as well as a custom tool to print information about used bytes of TPM commands.

We showed that the automatic generation of executable source helps to bootstrap the implementation process of the TPM specification by providing a solid base. Moreover, the level of detail and level of abstraction with FTPM is easily adaptable. For example, we decided that a good level of abstraction for our implementation is the declaration of initialization vectors rather than real random values for Nonces. Furthermore, we demonstrated our approach to separate implementation details from the specification into (stub) functions. This strategy creates a clean interface between designer and implementer because changes to the specification will not overwrite custom source code added by implementers during the automatic code generation. Additionally, ambiguities like using the same name for variable and parameter names or incompatible assignments are detected by the FTPM compiler early. Most importantly, FTPM automatically enforces the abidance to defined rules in every command or structure and thus helps an implementer to comply.

Next, we outlined the advantages of generation the documentation from within FTPM. Most importantly the improved navigation, which greatly helps to navigate hundred of pages of complex specification text. Furthermore, a consistent yet easy to change layout and formatting is achieved by the automatic generation of documentation as well. Combined with a version control system the complete history of changes and considerations is available for specification designers and implementers.

Finally, an executable specification can also be used to implement small tools and utilities quickly. The *Byte-Counter* utility can be used to estimate the amount of needed bytes each TPM command consumes. This utility is useful to get a list of commands which need adaption if the available memory is limited.

Chapter 5

Language elements

This chapter introduces syntax and semantics of the FTPM language in detail. While we will not cover all constructs available in FTPM, we discuss the most important ones. The complete grammar is listed in Appendix B for reference. We introduce parts of the underlying grammar with figures in the Extended Backus-Naur Form (EBNF) [27]. Furthermore, we will use short code snippets to exemplify the language.

The structure of this chapter follows the composition of grammar for the FTPM language: After a broad overview of the principle composition of a FTPM input source, we introduce variable declarations. Then, the next logical breakdown consists of TPM commands, helper functions and blocks in general. Finally, the smallest language unit we present are statements, which are responsible to drive program logic.

5.1 General

The notation we use for the EBNF grammar in this chapter is the same as used in ANTLR, which consists of colon as a delimiter for rule names on the left and (non-) terminal symbols on the right. Vertical bars (|) represent alternatives and question marks (?) optional (one or zero) elements. A plus sign (+) indicates one or many and a star (⋆) zero or many repetitions of the preceding symbol. Additionally, rules can be grouped by parentheses. Names written in uppercase are lexer rules, while lowercase names are parser rules.

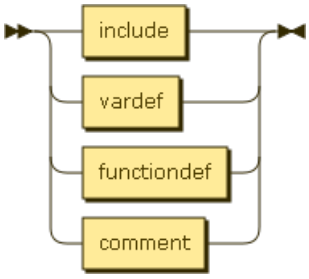
At the topmost level, a valid FTPM source is either empty, consists of (possibly multiple) declarations and exactly one TPM command. A TPM command is called *ordinal* in the TPM specification:


```
ftpm : declaration⋆ ordinal?
```

While FTPM allows a combined notation of declarations and an ordinal definition, it is recommended to separate them into different files. This does not only aid comprehension by keeping the source files minimal but also allows to replicate the present anatomy of the TPM documentation with the FTPM prototype.

5.2 Declaration

A declaration in FTPM is either a comment, the include keyword, a variable or a function definition. All declarations and statements in FTPM end with a semicolon.




```
declaration : comment | include | vardecl | funcdef;
```

5.2.1 Comment

In many programming languages, comments are disregarded by the parser. However, in the FTPM language comments are an important mechanism as they drive large parts of the documentation generation. A comment is started with two dashes and an exclamation mark (--!). Everything after this sequence of characters is considered a comment until the end of the line. Comments which span over multiple lines start the subsequent lines with two dashes (--) only.

```
comment : COMMENT_FIRST COMMENT_CONSECUTIVE*

COMMENT_SIGN :       '--';
COMMENT_FIRST :      COMMENT_SIGN '!' ~('\n'|'\r')* '\r'? '\n';
COMMENT_CONSECUTIVE: COMMENT_SIGN  ~('\n'|'\r')* '\r'? '\n';
```

A comment can be optionally followed by a keyword, which is either file, chapter, info, or desc. These keywords denote the *class* of a comment, which will be used in the output generation phase (mostly documentation generation). All comment classes can be declared multiple times in one FTPM input source. However, for the file and the chapter comments all but the last occurrences will be ignored.

```
COMMENT_CLASS: 'file' | 'chapter' | 'info' | 'desc';
```

The file comment denotes to which of the three sections from the TPM specification the subsequent statements belong. Possible values are *design*, *structures* and *commands*. This comment sets the name of the section per FTPM input source.

```
COMMENT_PART: 'design' | 'structures' | 'commands';
```

The chapter keyword servers an similar purpose than the file keyword before. It denotes the name of the chapter inside the respective section to which the following statements belong. Furthermore, it can also name the current section if the text contains two colons (::). If no name is explicitly stated, the filename of the FTPM input source is used.

An example comment, which illustrates the usage of the file and chapter classes is shown in Listing 5.1

```
1 --!file    commands
2 --!chapter 13 Cryptographic Structures :: 13.05 TPM Sign
```

Listing 5.1: Example of file and chapter comments used for documentation generation.

These comments will instruct the documentation generation to place any following statements in a sub-chapter named *13.05 TPM Sign* in the chapter *13 Cryptographic Structures*. These chapters will end in the document *Commands*, which is Part 3 of the current TPM documentation.

`info` and `desc` comments are used in the documentation generation as well. The current TPM specification distinguishes between informative comments and normative statements. The `info` keyword marks the comment as informative, while `desc` marks a description comment, which belongs to the normative statements (see Listing 5.2). Both comment texts are labeled and visually outlined in the resulting documentation output.

```
1 --!info This is an informative comment
2 --!desc This is a descriptive comment (normative)
```

Listing 5.2: Informative and Descriptive comments provide a general information.

The remaining documentation class is automatically assumed if no comment keyword is explicitly stated and denoted as an *action* comment. *Actions* are normative statements and describe the sequence of necessary steps a TPM command has to execute (Listing 5.3).

```
1 --! 1. The TPM validates the AuthData to use the key
2 --     pointed to by keyHandle.
3 --! 2. If the areaToSignSize is 0 the TPM returns TPM_BAD_PARAMETER.
```

Listing 5.3: Action comments are normative and describe necessary execution steps.

The list of currently available classes is kept minimal at purpose to reproduce the prevailing form of the TPM specification. However, it is little effort to extend this list and include other keywords to highlight different aspects of the specification in the future. With `Doxygen` as documentation generator, it is furthermore possible to include special `Doxygen` keywords[1] to format the generated documentation text. Examples for useful keywords are `@bug`, `@todo`, `@deprecated` or `@warning`. These keywords will visually label sections which need special attention, too.

Action comments are automatically formatted as a numbered list of normative instructions. The numbering of the list is handled automatically. Currently, `FTPM` supports one level of indent to describe individual tasks. Nested lists are created, if the comment starts with a star (*) character (Listing 5.4).

```
1 --! 3. Internally the TPM will do the following:
2 --    * TPM allocates space to save handle, protocol identification, both
3 --      nonces and any other information the TPM needs to manage the session.
4 --    * TPM generates authHandle and nonceEven, returns these to caller
```

Listing 5.4: Action comments are rendered as numbered lists in the resulting documentation. A star character (*) is used to generate nested lists.

5.2.2 Include Keyword

The `include` keyword (see Listing 5.5) allows the structuring of `FTPM` source into different files or modules. The processing of the `include` mechanism happens during the lexer phase. If the lexer finds an `include` keyword the content of the denoted file is imported and the lexer continues. Implicitly

[1] http://www.stack.nl/~dimitri/doxygen/commands.html

the lexer includes a file only once to avoid parsing errors which may result because of the multiple inclusion of the same file. This approach to prevent multiple inclusions is common in many programming languages as well, for example `Objective-C`. Implementation details about the inclusion mechanism are discussed in Chapter 6.3.2.

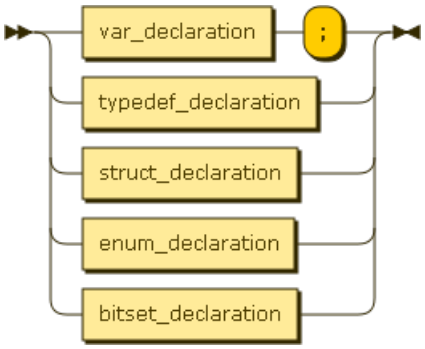
```
include "structures/02_basic/02_defines.ftpm"
```

Listing 5.5: The `include`-keyword imports different sources into the current file.

5.2.3 Variables and Types

Variables in `FTPM` can be declared in two different ways. The first possibility is to statically specify a type. This approach is used in the declaration of structures used by the TPM. The other approach is to declare temporary variables. These variables automatically derive their type from the assigned (static) variable.

A static type for variables in `FTPM` is one of the primitive types, which is either an `integer` of various sizes, a `boolean` or a `string`. Moreover, `FTPM` allows the definition of constants (`constant`) and aliases to other types with the `typedef`-keyword. `Structures`, `Enumerations` and `Bitsets` are composed types available in `FTPM`.




```
vardef : var_declaration ';'
       | typedef_declaration
       | struct_declaration
       | enum_declaration
       | bitset_declaration
```

Names and identifiers in `FTPM` must start with a letter or an underscore, followed by zero or more letters, numbers or underscores. Optional square brackets (`[]`) indicate an array. If the array does not contain a size specifier it is assumed to be dynamic.

```
var_declaration : ID vartype (':=' expression)?
                | ID vartype '[' array_capacity? ']' (':' expression)?;

ID : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
```

Primitive types

Among the primitive types of `FTPM` are strings, which are identified by the keyword `string`. String values can contain any character and have to be enclosed in double quotes. Boolean values are denoted by the keyword `boolean` and accept the usual literal values of `true` and `false`. Strings and

Type	Description
`string`	Sequence of characters inside quotes
`boolean`	Boolean values, either *true* or *false*
`signed integer 8bit`	Numbers range from -128 to 127
`unsigned integer 8bit`	0 to 255
`signed integer 16bit`	-32.768 to 32.767
`unsigned integer 16bit`	0 to 65.535
`signed integer 32bit`	$-(2^{31})$ to $2^{31}-1$
`unsigned integer 32bit`	0 to $2^{32}-1$
`signed integer 64bit`	$-(2^{63})$ to $2^{63}-1$
`unsigned integer 64bit`	0 to $2^{64}-1$

Table 5.1: Primitive datatypes in `FTPM`.

Boolean values have supportive character for the specification designer as they are not needed for actual commands but rather accompanying tasks like the output of debug information.

The main type of `FTPM` are integers. There exist eight different integer types which differ in their available bit-size: 8 bit, 16 bit, 32 bit and 64 bit. The signedness of an integer type is indicated by the mandatory keyword `signed` or `unsigned`. Table 5.1 gives an overview of the primitive types available in `FTPM`.

The syntax for declaring primitive types of `FTPM` is shown in Listing 5.6:

```
unsigned integer 8bit flag;                signed integer 64bit big_counter;
boolean debug := true;                     unsigned integer 32bit pcrSelect[20];
```

Listing 5.6: Example declarations with primitive types.

Aliases for existing types are assigned via the `typedef` keyword. Moreover, the keyword `constant` is used to create a numerical constant. After both keywords follows the name of the alias or the numerical value, which is specified next (Listing 5.7). It is not required that an alias points to a primitive type in `FTPM`. It may also point to already defined aliases.

```
typedef UINT8 is unsigned integer 8bit;          typedef FOO is UINT8;
constant TPM_NUM_PCR is 16;
```

Listing 5.7: Aliases to existing types are created with `typedef`. A numerical constant is declared by the keyword `constant`.

Composed types

`FTPM` also provides composed types, namely *Enumerations*, *Structures* and *Bitsets*. Enumerations are a collection of elements and denoted by the keyword `enumeration`. After the name and the type of the enumeration follows a list of enumeration values. These values are used as named constants The enumeration list is enclosed by brackets and separated by commas. An optional initial value for enumeration values can be specified, too. The values for enumeration members are a sequence of integers.

This sequence is either determined by the previous member and subsequently increased by one for the current member, or an explicit initialization value is given. If no value is specified for the first element it defaults to zero (Listing 5.8). Currently, the FTPM prototype does not check if the assigned values of an enumeration are different to each other.

```
enum TPM_RESULT is UINT32 (
    TPM_SUCCESS,                --! value is 0
    TPM_AUTHFAIL,               --! value is 1
    TPM_BADTAG := 30,           --! value is 30
    TPM_IOERROR                 --! value is 31

);
```

Listing 5.8: An enumeration in FTPM. Initialization values for enumeration members are optional. If omitted, the value is increased by one from the previous value. The first value in an enumeration defaults to 0.

A structure is a collection (record) of multiple, possibly different, data types. The declaration consists of the keyword struct, a name for the structure and a list of structure members. Each member is separated by commas (Listing 5.9).

```
struct TPM_STRUCT_VER is (
    major    BYTE,
    minor    BYTE,
    revMajor BYTE,
    revMinor BYTE
);
```

Listing 5.9: Structures are a composed type with several structure members separated by commas.

The last available composed type in FTPM is a bitset. A bitset can be considered a structure that stores bits. Each member of the bitset stores a certain number of bits. A bitset uses a similar notation to enumerations and structures: the keyword bitset followed by a name and type information. After an opening bracket the members of the bitset are declared. Again, the separating character between bitset-members is a comma. Furthermore, the members of a bitset can contain an optional assignment with a constant, numerical expression. This numerical expression specifies the number of bits the member seizes (Listing 5.10).

```
bitset TPM_BITSET is UINT16 (
    member1 := 1,     --! member1 holds 1 bit
    member2 := 3      --! member2 holds 3 bits
);
```

Listing 5.10: A bitset is a structure with an optional number of reserved bits for the member.

Access to members of enumerations, structures or bitsets is established by the use of a dot operator (.) followed by the member name. Listing 5.11 exemplifies the usage of members from composed types in FTPM.

```
--! Enumeration
return TPM_RESULT.TPM_SUCCESS;

--! Structure
TPM_STRUCT_VER version;
version.major := 1;
version.minor := 2;
```

Listing 5.11: The dot operator is used to access members of composed variables.

Implicit types

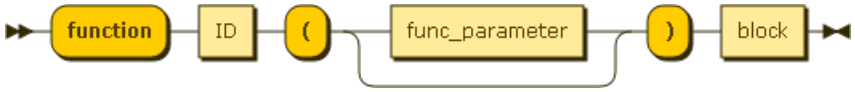
Many TPM ordinals do need several helper variables, which temporarily hold the result of an expression or computation. In order to ease the declaration for such helpers, `FTPM` provides a special syntax to create such dynamically typed variables: A temporary variable is defined through the keyword `var` followed by a name and an assignment. This assignment sets the type and simultaneously initializes the variable (Listing 5.12).

```
var R1 := TPM_STRUCT_VER.major;    --! R1 is of type TPM_STRUCT_VER.major
```

Listing 5.12: Temporary variables use the type of the assigned variable implicitly.

5.2.4 Function Definition

The purpose of functions is to modularize program components. They encapsulate a specific part of the program (TPM command) and thus foster the readability. Syntactically, a function consists of a name, optional parameters and zero or more return values, followed by a statement-block (see Section 5.4).




```
functiondef : 'function' ID '(' func_parameter? ')' block
```

The code in Listing 5.13 shows an example of a function used as a helper for the initialization of a PCR value in the `TPM_PCRRead` command.

```
function setPCRValue(pcrIndex  TPM_PCRINDEX, outDigest TPM_PCRVALUE)
begin
    --! set the compliance vector for internal tests
    outDigest.digest[0] := 0x15;   outDigest.digest[1] := 0x8f;
end;
```

Listing 5.13: A function is used to create modular, easier to read programs.

As mentioned before, functions and ordinals have the same set of statements which are allowed to appear inside them. This is not surprising, as an ordinal can be viewed as a specialized function, which returns exactly one value (of type `TPM_RESULT`) and can only be declared once per `FTPM` file. Furthermore, ordinals use a special notation to declare incoming and outgoing parameters, while functions do not have this distinction of parameter types.

5.3 Ordinals

TPM commands are called *ordinals*. Each ordinal has a name and certain input and output parameters (see Chapter 2.2.2). An exemplary specification of an ordinal in FTPM is shown in Listing 5.14.

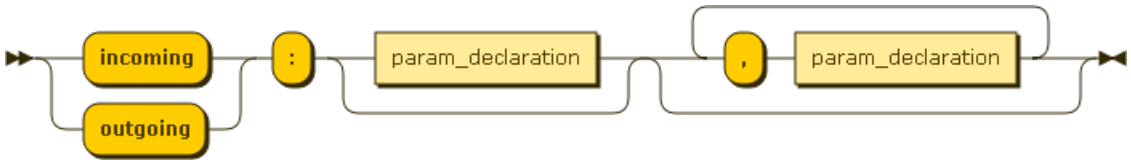

```
ordinal : 'ordinal' ID parameter_block block
```

5.3.1 Incoming and Outgoing Parameters

Parameters in ordinals are either incoming (passed from the upper software stack to the ordinal) or outgoing (passed from the ordinal to the upper software stack). Every TPM command has a common set of incoming and outgoing parameters, which are passed to *every* ordinal. The default incoming parameters are an *authorization tag*, the *parameter size* of all incoming parameters and a *command code*. These three parameters are also among the default outgoing parameters in addition to a *command return code*.

In order to remove typing mistakes and reduce the number of visible parameters, FTPM leaves out these common parameters. The information of these parameters is implicitly contained in the ordinal written in the FTPM language: The authorization tag can be derived by the number of authorization blocks (see Section 5.4.1). The parameter size depends on the number of other incoming or outgoing parameters, the command code *is* the name of the TPM command and the return code is available in the abstract syntax tree as it will be forced to exist by the parser of FTPM.

For fast recognition of incoming and outgoing parameter we chose to introduce blocks: One block for incoming parameter denoted by the keyword `incoming` and another block for outgoing parameter which is distinguished by the keyword `outgoing`. These blocks are mandatory even if no incoming or outgoing parameters exist.




```
parameter_block : ('incoming' | 'outgoing') ':'
                    param_declaration? ( ',' param_declaration )*
```

After the parameter blocks follows a block of statements (see Section 5.4). This block is fenced by `begin` and `end;` keywords and a mandatory return statement (see Section 5.5), which indicates the result type of an ordinal.

5.4 Blocks

Functions and Ordinals consist of at least one block, whereby multiple, nested blocks are possible. FTPM does not use brackets to enclose blocks but rather marks the begin and end of a block with the keywords `begin` and `end` respectively. As described in Section 6.3.4, FTPM uses static scoping. This means, that the visibility of variables is limited to current and to possibly nested, *inner* blocks. Moreover, a variable defined in an *inner* block masks a variable with the same name defined in an *outer* block.

```
ordinal TPM_Extend

    incoming:
        pcrNum      TPM_PCRINDEX,
        inDigest    TPM_DIGEST

    outgoing:
        outDigest   TPM_PCRVALUE

begin
    return TPM_RESULT.TPM_SUCCESS;
end;
```

Listing 5.14: An ordinal consists of name, incoming and outgoing parameters and at least one block of statements, which is terminated by a return instruction. The parameters are grouped in categories which allows quick recognition. Parameters common to every TPM command are left out as they can be derived from the abstract syntax tree. Every ordinal has at least one return statement, which indicates the result.

5.4.1 Authorization Blocks

Most TPM commands need authorization before they can be executed (see Chapter 2.2.2). `FTPM` uses a special notation to describe and perform such an authorization. An `auth`-block provides authorization information in a visual appealing representation. It helps readers to quickly gather the most important information: *Who?* (object) needs authorization with *what?* (secret) under *what?* circumstances (required). Therefore an authorization block in `FTPM` consists of the following elements: A keyword `auth`, followed by three properties: `.object`, `.secret` and `.required`. The leading dot is used to distinguish the properties from variable assignments. An authorization block is finalized by the keyword `end` and a semicolon, like ordinary blocks (see Listing 5.15).


```
auth_block : 'auth'
                '.' auth_object ';'
                '.' auth_secret ';'
                '.' auth_required ';'
             'end' ';'
```

```
auth
    .object   := key
    .secret   := key.usageAuth;
    .required := true;
end;
```

Listing 5.15: An authorization block quickly summarizes the properties for a successful authorization: The key object with a certain usage secret. Additionally, it is indicated if the authorization must be executed at all times by the keyword `.required`.

The same authorization information is of course available in the current TPM documentation. However, the authorization information in the documentation is not as quickly recognizable to the reader as with the notation used in `FTPM`. In the current TPM specification, the three properties (*object*, *secret*, and *required*) are scattered in the incoming parameter table and in the *action* section. In `FTPM`, the the same information is represented in a dedicated, concise syntax. Figure 5.1 shows a typical authorization information of the original TPM specification. Compared to the syntax of `FTPM` in Listing 5.15

the reader of the specification has to collect the relevant information from three different places. The incoming parameter `TPM_TAG` and the first action-comment (1) inform the reader that the command requires authorization prior usage. Next, the incoming parameter list has to be scanned to find the relevant key-object (2) for which authorization is required. Finally, the usage secret for the authorization is to be found in yet a different part of the incoming parameter section: In the description column for `TPM_AUTHDATA` (3).

1331 **Incoming Operands and Sizes**

PARAM #	PARAM SZ	HMAC #	HMAC SZ	Type	Name	Description
1	2			TPM_TAG	tag	TPM_TAG_RQU_AUTH1_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_Seal.
4	4			TPM_KEY_HANDLE	keyHandle	Handle of a loaded key that can perform seal operations.
5	20	2S	20	TPM_ENCAUTH	encAuth	The encrypted AuthData for the sealed data.
6	4	3S	4	UINT32	pcrInfoSize	The size of the pcrInfo parameter. If 0 there are no PCR registers in use
7	<>	4S	<>	TPM_PCR_INFO	pcrInfo	The PCR selection information. The caller MAY use TPM_PCR_INFO_LONG.
8	4	5S	4	UINT32	inDataSize	The size of the inData parameter
9	<>	6S	<>	BYTE[]	inData	The data to be sealed to the platform and any specified PCRs
10	4			TPM_AUTHHANDLE	authHandle	The authorization session handle used for keyHandle authorization. Must be an OSAP session for this command.
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by TPM to cover inputs
11	20	3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
12	1	4H1	1	BOOL	continueAuthSession	Ignored
13	20			TPM_AUTHDATA	pubAuth	The authorization session digest for inputs and keyHandle. HMAC key: key.usageAuth.

1335 **Actions**

13361. Validate the authorization to use the key pointed to by keyHandle

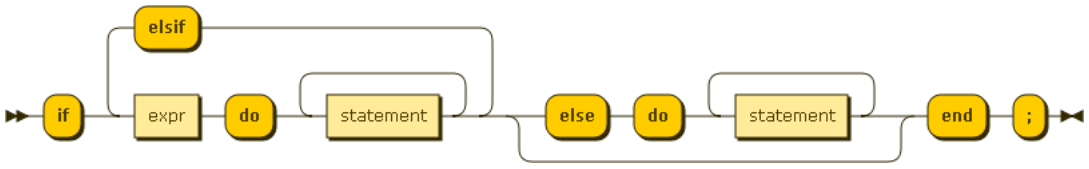
Figure 5.1: Authorization information in the original version of the TPM specification. This example is taken from the `TPM_Seal` and illustrates the different locations for needed information regarding the authorization of the command.

5.5 Statements

Various different types of statements can be declared inside a block. Next to variable and temporarily variable definitions (see Section 5.2.3) `FTPM` provides control structures, loops and expressions.

5.5.1 Control structures

`if` statements control the program flow. In `FTPM` they have they following structure:




```
if_statement :  ifstat elsifstat* elsestat? 'end' ';'
ifstat       :  'if' expr 'do' statement+
elsifstat    :  'elsif' expr 'do' statement+
elsestat     :  'else' 'do' statement+
```

This means, that similar to block statements, `FTPM` abstains from brackets but uses a written keyword to mark the begin of a block called `do`. As with expressions `FTPM` strives to create readable programs

by preferring verbose statements over a short-hand notation. The keyword `end` is reused from ordinary blocks to keep the available syntax elements short.

5.5.2 Loops

Currently, only one notation for loops is present in `FTPM`. This comes from the fact that while loops are most certainly needed for other TPM commands, they are not necessary for the currently implemented subset of TPM commands. If needed, multiple variations of this basic loop can be quickly added to the `FTPM` language.

A simple loop is defined as:


```
for_loop : 'for' ID 'in' expression 'to' expression
           'do'
               statement+
           'end' ';'
```

The loop declares a variable (`ID`). The type of this variable is automatically derived from the subsequent expressions. For example, consider the following loop:

```
1 for idx in 1 to 10
2 do
3   myArr[idx] := 0x00;
4 end;
```

The variable `idx` is visible in the block marked by the keywords `do` and `end`. Additionally, the variable `idx` is of type `unsigned integer 8bit`, as the expression happens to be in that range.

5.5.3 Operators and Expressions

The operators of `FTPM` and their precedence are shown in Table 5.2. Each of the following subsections shows brief usage examples of these operators. The higher an operator is listed in this table, the higher is its precedence to operators listed in rows below it. For example, this means that multiplicative operations are executed before additive operations. Some operators allow an alternative notation. For example, the unary negation operator can be written either as *!* or as the English Word *not*. Both notations provide the same functionality to negate an boolean expression and can be used interchangeable. This approach to allow different notations is also used for example in the `Ruby` programming language[2]. Listing 5.16 compares the available notations of the unary negation operator. A boolean result from a function call (`isValid()`) is negated. The left side of the example uses the shorthand notation, which is common in many programming languages. The right side uses the alternative notation. This notation produce more fluent code and thus aids readability.

```
1 if !isValidPCRIndex(pcrIndex)
2 do
3   [...]
4 end;
```

```
1 if not isValidPCRIndex(pcrIndex)
2 do
3   [...]
4 end;
```

Listing 5.16: Example of an unary negation operator: Depicted on the left is the shorthand notation while the right side uses the English word *not* to achieve the same result.

[2] http://www.ruby-lang.org

Description	Operator
unary postfix	() [] .
unary prefix	-expr !expr (alternative: not expr) ˜expr
multiplicative	* / %
additive	+ -
shift	<< >>
relational	>= > <= <
equality	== !=
bitwise AND	&
bitwise OR	|
bitwise XOR	^
logical AND	&& (alternative: and)
logical OR	|| (alternative: or)
conditional	expr1 ? expr2 : expr3
assignment	:=

Table 5.2: Operators and their precedence.

Arithmetic Operators

Table 5.3 lists the arithmetic operators defined in `FTPM`. An exemplified usage of the operators is shown

Operator	Meaning
+	Add
-	Subtract
-expr	Unary minus
*	Multiply
/	Divide
%	Modulus

Table 5.3: Arithmetic operators.

in Listing 5.17. The behavior is the same as in other programming languages and described in the table.

Equality and Relational Operators

`FTPM` provides the operators from Table 5.4 to test for equality and relation.

Again, the relational and equality operators work as in other programming languages (see Listing 5.18).

```
foo unsigned integer 8bit := 6;
bar unsigned integer 8bit := 2;

foo + bar       --! equals 8
foo - bar       --! equals 4
-foo            --! equals -6
foo * bar       --! equals 12
foo / bar       --! equals 3
foo % bar       --! equals 0
```

Listing 5.17: Exemplified usage of arithmetic operators.

Operator	Meaning
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Table 5.4: Equality and relational operators.

Assignment operator

FTPM uses := as the assignment operator. We decided for the mathematical notation of := as opposed to a single assignment character (=). This way, the assignment is more visually distinct to the equality operator (==). Some assignments are shown in Listing 5.19.

Bitwise operators

Bitwise operators are common tools to describe low-level operations. Currently, FTPM supports the notation for bitwise operations shown in Table 5.5. Listing 5.20 illustrates the usage of bitwise operators.

Other Operators

Table 5.6 presents the remaining operators. A function call is represented by a pair of brackets. Optional arguments to function calls are specified inside the brackets. Access to elements of an array is represented

```
foo unsigned integer 8bit := 6;
bar unsigned integer 8bit := 2;

foo == bar       --! evalutes to boolean false
foo != bar       --! evalutes to boolean true
foo > bar        --! evalutes to boolean true
foo < bar        --! evalutes to boolean false
foo >= bar       --! evalutes to boolean false
foo <= bar       --! evalutes to boolean false
```

Listing 5.18: Exemplified usage of relational and equality operators.

```
foo unsigned integer 8bit := 6;
bar unsigned integer 8bit := 2;

var tmp := foo + bar;            --! creates variable named `tmp' (of type
                                 --  unsigned  integer 8bit)  and assigns it the
                                 --  value of 8

result unsigned integer 8bit;
result := ((foo * bar) + 1) % 2; --! the variable result gets assigned the value 1

```

Listing 5.19: Exemplified usage of the assignment and arithmetic operators.

Operator	Meaning
&	AND
|	OR
^	XOR
~	Unary bitwise complement
<<	Shift left
>>	Shift right

Table 5.5: Bitwise and Shift operators.

by square brackets. Arrays in FTPM are indexed from 0. Therefore the last element of an array arr with n elements is accessed by $arr[n-1]$. Members of structures, enumerations or bitsets is established by a dot. The Ternary Operator is a shorthand notation for an $if..else$ control flow. Listing 5.21 exemplifies the usage of these operators.

5.6 Summary

This chapter introduced important elements of the FTPM language. First, we presented the general structure of a FTPM file which consists of the definition of multiple structures and at most one ordinal. While not enforced by the grammar we recommend the separation of structures and TPM ordinals to increase readability.

Then, we detailed the include and comment mechanism and discussed the structure of variable declarations along with the possible types. We presented the notation to declare primitive and composed types as well as alias to existing types or numerical constants. Next, we introduced the syntax for ordi-

Operator	Meaning
()	Function call
[]	Array access operator
expr1 ? expr2 : expr3	Ternary operator;executes expr2 if expr1 is *true* or else expr3 is executed
.	Member access to Enumerations, Structures or Bitsets

Table 5.6: Other operators.

```
constant bitmask is 0x0f;
foo unsigned integer 8bit := 0x22;

if((foo & bitmask) == 0x02)     --! AND
do [...]

if((foo & ~bitmask) == 0x20)    --! AND NOT
do [...]

if((foo | bitmask) == 0x2f)     --! OR
do [...]

if((foo ^ bitmask) == 0x2d)     --! XOR
do [...]

if((foo << 4) == 0x220)         --! Shift left
do [...]

if((foo >> 4) == 0x02)          --! Shift right
do [...]
```

Listing 5.20: Bitwise and Shift operators.

```
arr signed integer 8bit[30];            --! array of 30 8-bit integers
myArray[0] := 0x00;                     --! assign 0x00 to array element on
                                        --  position 0
myArr[29]  := funcCall(myArr[0]);       --! call function `funcCall' with the
                                        --  element on position 0

--! simple enumeration specifing two values
enumeration MY_ENUM is unsigned integer 8bit (
  VAL0 := 0x00,
  VAL1 := 0x01
);

myArr[1] := MY_ENUM.VAL1;               --! assign VAL1 (0x00) to myArr[1]
```

Listing 5.21: Example of function call, array and composed structure access operators.

nals and functions as well as incoming and outgoing parameters. We showed our approach to represent authorization information, which is readable and concise.

Finally, we demonstrated how blocks, statements and expressions make the smallest logical unit of the `FTPM` language. `FTPM` provides arithmetic operators and control structures like if-statements and loops, which are similar to most programming languages.

Chapter 6

Tools, Design and Implementation

The following chapter explains the design and implementation of the `FTPM` prototype in detail. First, we present a general overview of `FTPM` and for what specific problems it was designed. Next, we give a generic overview of the selected methods and tools used in the implementation. Then, we provide an detailed insight into each phase an input source code traverses when it is processed by the `FTPM` executable. Those phases consist of details from parsing `FTPM` input source, to implemented static type checks, up to the *automatic* generation of various output formats. Moreover, we discuss the interaction between the most important components from the implementation of the `FTPM` prototype.

6.1 General overview of `FTPM`

A TPM is a dedicated hardware security chip with specialized features for cryptographic purposes (see Chapter 2.2). It has a cryptographic co-processor, which is used in encryption, decryption and signing operations. It also has key-management and storage facilities, which are needed to load different types of key material as well as to store sensitive and insensitive data inside the TPM. Furthermore, a TPM is capable to store measurements, which then are subsequently used to build a chain-of-trust. These measurements can be reported to a third party, which is then able to verify the state of the system by comparing the reported measurement values to previously stored values. With this process, called remote attestation, a certain level of trust can be assured.

These key features of a TPM do provide a certain level of security. However, history has proven that every system, no matter what level of security it provides, is prone to some attacks (known or unknown). Game consoles, like the *XBOX* or *Playstation*, smartphones like the *iPhone* or *HDCP*, which is a protection scheme to prevent creation of digital copies audio and video material, are examples where the producer made special efforts to prevent consumers to use the devices in any other way than intended. However, bugs in the implementation could eventually be utilized to break the security of the devices and allowed the installation of custom software or hardware modifications. An overview of devices, the deployed security and the time span until the security was broken as well how it was broken can be found in [11].

When looking at hardware security chips in general, there exist the following four primary attacks, as identified by Kömmerling et al. [33], which naturally apply to TPM chips as well:

- **Microprobing techniques** operate directly on the chip surface for observation, manipulation and interference;
- **Eavesdropping techniques** monitor connections and electromagnetic radiation;
- **Fault generation techniques** “use abnormal environmental conditions to generate malfunctions in the processor that provide additional access”; and

- **Software attacks** "exploit security vulnerabilities found in the protocols, cryptographic algorithms, or their implementation".

This classification can be further simplified to attacks against the hardware and attacks against the software. The hardware has to guarantee that data stored securely on the device can never be accessed or leave the chip unintentionally or unprotected. Countermeasures against physical modification, eavesdropping and fault generation techniques must be provided. The software side provides access to data and functions stored on the hardware, therefore its design is at least as crucial as the hardware design itself. Especially as the software layer is a much more reachable target than the more complicated hardware attack vectors. Bugs on both layers render the security features of the chip most certainly useless.

We show that a domain-specific language (`FTPM`, Formal TPM) is well suited to create and maintain the specification of Trusted Platform Modules [56, 57, 58]. The more people are able to read and understand the specification, the more likely design errors and bugs can be found and removed which ultimately aids the security of the device. It enables specification writers and programmers to compose data structures and TPM commands consistently and prevents syntax errors early. The clean syntax of `FTPM` accelerates comprehension of complicated and security-relevant specification text. An automated translator can produce various different outputs directly from specification text written in `FTPM`. This automatic output generation can, among others, produce high-quality documentation. A well-prepared documentation is an important cornerstone for identification of security problems and hence crucial in the feedback cycle between specification designers and implementers of TPM modules.

The approach of a domain-specific language does not only allow a direct and automatic generation of various output but also a specifically tailored syntax for highly complex procedures, which fosters comprehension. The maintenance of complex software is one of the most effort-consuming activities in the whole software-life-cycle [52]. While maintenance is responsible for up to 80% of the cost of software, a significant percentage (up to 50-60%) is used for proper comprehension of the software [12]. Naturally, understanding software and its functionality is a prerequisite for designing a system with robust, enhanced security.

We use `FTPM` to implement a subset of the TPM specification, as the complete specification is out of scope for this thesis due to its extensiveness. Furthermore, we provide a translator for `FTPM` source to the `C` programming language. We summarized the advantages of automatic code generation through a domain-specific language shortly in Chapter 3.4. Moreover, `FTPM` is capable to produce high-quality documentation in various output formats. Currently, this is achieved by preparing `FTPM` source files for the `Doxygen` documentation generator (see Chapter 3.3). However, other output generation is easily realizable as well because the specification is available in an *executable* form. Therefore, `FTPM` is extensible to support the creation of different documentation formats like `XML` or `DocBook`[1]. Additionally, programming languages other than `C` may be convenient to construct rapid prototypes to test the functionality of a TPM command. The extended flexibility of having an executable specification is demonstrated by a *byte-counter*, which sums the amount of bytes by analyzing the TPM command signature and used temporary variables. This might be useful for devices where memory is limited, for example embedded systems.

6.2 Tools

The `FTPM` prototype consists of a domain-specific language which allows the description of TPM commands, which are called *ordinals* in the official TPM specification, as well as data structures. A parser reads `FTPM` source files and performs checks on syntax and semantics. Afterward, a translator takes the parsed `FTPM` source and produces different outputs. We implement a translator to the `C` programming language as well as documentation in various different output formats (`HTML`, LaTeX, etc). The translator

[1] http://docbook.org

to the C programming language is written from scratch while we facilitate the powerful capabilities of Doxygen for documentation generation (see Chapter 3.3). Each step in this process is described in more detail in the following sections.

6.2.1 ANTLR

The FTPM language is implemented as an external domain-specific language. This means that an external parser for the syntax of FTPM is needed. While the flexibility of an external domain-specific language is advantageous, the creation of the needed tools (parser, compiler, interpreter, translator etc.) is extensive (see Chapter 3.4). ANTLR is a *parser generator*, which is a tool that aids in the creation for building a parser, compiler or interpreter for a formal language, written by Terence Parr[2].

ANTLR offers a special notation (see Section 6.3.3) to easily create grammars for parsers and lexers as well as the creation of an abstract syntax tree (AST). This tree is the starting point for custom written checks on syntax and semantic of FTPM input files. Furthermore, a set of APIs (application programming interfaces) is provided to interact, modify and query the AST at different stages during parsing. For example a part of the AST of the FTPM language for the TPM_PCRead command is shown in Figure 6.1.

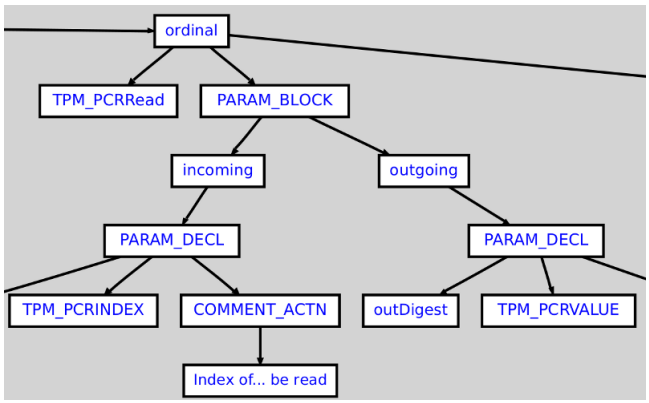


Figure 6.1: Example Abstract Syntax Tree which shows part of a parsed TPM_PCRRead command using the FTPM language.

ANTLR can process context-free grammars, written in the Extended Backus-Naur Form (EBNF) [27]. It is a *recursive descent*, *top-down*, $LL(*)$ parser. That means, it parses input from **L**eft-to-right, constructs a **L**eftmost derivation and per default is not restricted in the lookahead of input tokens (*). For a detailed discussion on ANTLR and $LL(*)$ see [42, 40].

While ANTLR does support a variety of different target languages[3], we chose to implement the parser for the FTPM prototype in Java, as this is also the language ANTLR itself is written in and thus seems to provide the most stability and features.

6.2.2 StringTemplate

The final processing step in the FTPM prototype consists of output generation. A translator walks the AST, which was constructed, inspected and modified in previous stages during parsing, and emits various different output. We use StringTemplate[4] as template engine for most of our output creation. A template engine allows the creation of structured, reusable text without the need to recompile the program upon textual modifications. StringTemplate is written by Terrence Parr as well and therefore has a tight integration to ANTLR. It has a similar syntax for the creation of templates to what ANTLR uses for the creation of abstract syntax trees. Furthermore, it enforces a strict separation of model and view, where no program logic (model) can be used inside the output templates (view). This strict separation allows the complete creation of various different outputs with a single-point-of-change: the template. Moreover, StringTemplate features inheritance between templates. This means, that a specialized

[2] http://www.cs.usfca.edu/~parrt
[3] http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets
[4] http://www.stringtemplate.org/

template can inherit common parts of a more generic template and thus eases the creation of new custom templates.

Automatic output generation

For the creation of output in the `C` programming language a specialized template, which directly translates `FTPM` to `C` source code, is used. A simple `Makefile`, which provides a mechanism to quickly create executable code, and helper functions in a separate files, are generated automatically as well by the translator. For documentation output the first step consists of translating `FTPM` source to `Markdown`[5] syntax. `Markdown` is a plain text format, which provides basic formatting commands and is especially easy to read and to easy write. The next stage generates configuration files, which allow subsequent runs of `Doxygen` (see Chapter 3.3) to further process the generated output. `Doxygen` creates pleasant looking and well structured output in `HTML` as well as LaTeX.

By facilitating the power of `Doxygen` and `Markdown`, specification writers can produce coherent, high-quality output easily, as shown in Figure 6.2.

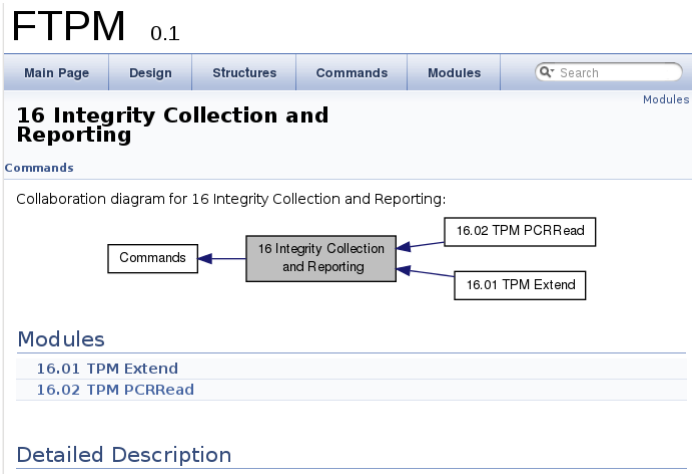


Figure 6.2: Output produced after `FTPM` input was translated to multiple `Markdown` files which then got processed by `Doxygen`.

6.2.3 Other utilities

The `FTPM` prototype is written in `Java`. As stated above, it uses `ANTLR` (Version 3.4) for the construction of a parser and `StringTemplate` (Version 4.0) for the creation of various different outputs: `C` source code, a simple `Makefile`, `Markdown` files for documentation, `Doxygen` configuration files, etc. In order to use `Doxygen` for the automatic creation of documentation, at least version 1.8.2 has to be installed. Moreover, we use `Apache Ant`[6] as a build tool for the `FTPM` executable and `Apache Commons CLI`[7] for parsing command line options passed to `FTPM`.

6.3 Implementation

This section presents the technical implementation of the `FTPM` prototype. The implementation can be roughly divided into six phases, as shown in Figure 6.3. First, an `FTPM` input file (alongside with the

[5] http://daringfireball.net/projects/markdown/
[6] http://ant.apache.org
[7] http://commons.apache.org/cli

desired options) is passed to the program. The input source is read into memory and broken into tokens by the lexer. Then, the tokens are arranged by the parser to form an abstract syntax tree (AST). This AST is the starting point for up to three remaining phases: symbol table generation, static type checks and (optional) output emission. Each of these phases walk the AST generated by the parser. Depending on the current phase the walker gathers information about AST nodes, modifies the underlying tokens or generates output. Next, we discuss each phase of the implementation in detail. The main grammar as well as an overview of the directory structure can be found in Appendix B.

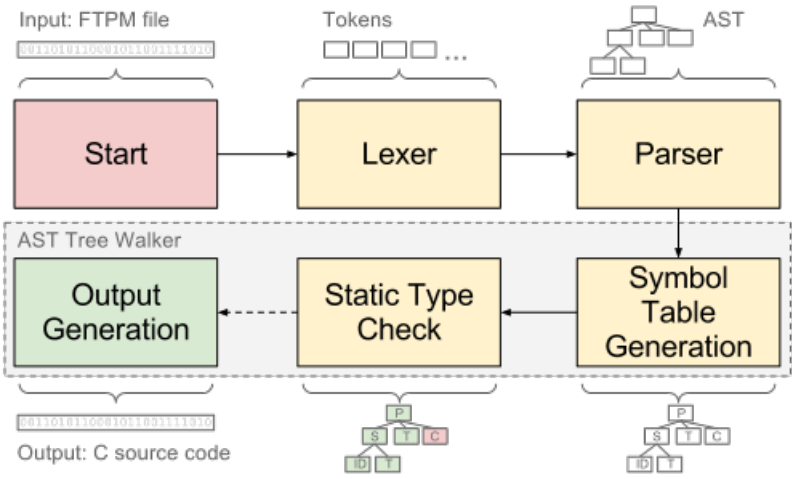


Figure 6.3: The phases a `FTPM` input file traveres: After reading from the file system, the input source gets tokenized by the lexer. The tokens are then arranged to an abstract syntax tree (AST) by the parser. Next, the AST is traversed multiple times for different duties: Symbol generation, static type checking and finally automatic output generation.

6.3.1 Phase 1: Start

The starting point of our prototype is the processing command-line options. Currently, the `FTPM` program accepts a variety of options for different output generation, as shown in Listing 6.1

```
usage: java -jar FTPM.jar file1.ftpm [file2.ftpm ...] [--bytes]
       [-d <arg>] [--debug] [-h] [-o <directory>] [-s <directory>]
       [-t <target>] [-v]

    --bytes                          print used bytes for an ordinal
 -d,--documentation <arg>            available documentation generation
                                     targets: doxygen
    --debug                          print various debug information
 -h,--help                           show help
 -o,--output-dir <directory>         where to write generated output files
 -s,--spec-directory <directory>     parse all ftpm-files in this directory
 -t,--target <target>                available code generation targets: c,h
 -v,--version                        show version info and exit
```

Listing 6.1: Available command-line options of `FTPM`.

The only mandatory argument for the `FTPM` executable is a `FTPM` input source: At least one input file has to be specified. Multiple `FTPM` sources may be specified either by explicitly passing them to the `FTPM` executable or by using the option `--spec-directory`. This option processes all files ending with `.ftpm` in the given directory.

The supported output emitters are controlled by the options `--bytes`, `--target` and `--documentation`. The option `--bytes` prints the number of bytes an ordinal consumes. For this purpose, the size of each parameter passed to an ordinal (input or output) as well as the sum of bytes used by temporary variables is accumulated.

The `--target` and `--documentation` options control the other available output generation, namely source code and documentation. Currently available code targets are `C` source (`c`) and header (`h`) files. The creation of `C` sources automatically causes the creation of header files as well. For documentation generation the only available option for now is `doxygen`. If this target is specified, the `FTPM` prototype will generate various `Markdown` files as well as `Doxygen` configuration files. Subsequent runs of the `doxygen` command on these generated files produces documentation in `HTML` and LaTeX.

The `--debug` option causes `FTPM` to be more verbose while processing the input files. Information about the symbol table (see Section 6.3.4) and static type checks (see Section 6.3.5) are printed to the standard output. Moreover, the final AST is saved in the `dot` format[8], which is used to describe graphs and can subsequently be viewed with external programs.

6.3.2 Phase 2: Lexer

After all command line options are processed, the program reads the specified `FTPM` input source into memory. Then, a lexer divides the input stream into defined tokens (see Appendix B). For the lexer and parser of `FTPM` an ANTLR *combined grammar* is used. A combined grammar contains the definition for the lexer as well as the parser in one file and can be found in `Ftpm.g`.

If the `FTPM` file contains an `include`-keyword (see Chapter 5.2.2) the lexer first checks if it already processed the specified file. If it was processed by the lexer earlier, it is simply ignored. Otherwise, the currently processed file is placed on a stack and the lexer starts processing the new, to-be-included file. When the end of a file is reached by the lexer, any previously saved `FTPM` file is retrieved from the stack and the lexing continues until the stack is empty and no other file is to be processed.

This mechanism prevents multiple inclusion and consequently multiple definition of the same variables or functions. However, this also implies that circular dependencies between different modules (mutual recursion) are not possible with `FTPM`.

Currently, the `FTPM` prototype is case-sensitive. However, a future optimization to `FTPM` is to remove this constraint to ease the creation of `FTPM` source files regardless of preferences in notation. As there is no option to automatically generate a case-insensitive lexer for in `ANTLR` in the used version (3.4), this could be achieved by either specifying all keywords case insensitive directly in the lexer or by extending the class `ANTLRFileStream` to treat everything as either upper or lower case[9].

6.3.3 Phase 3: Parser

The parser is fed with tokens produced by the lexer from the previous phase (see 6.3.2). The primary purpose of this phase is to construct an abstract syntax tree (AST). The AST is the central starting point for all subsequent analysis (symbol table management and static type checking) and output emission runs.

The parser can be found in `Ftpm.g`. This is an `ANTLR` grammar file and uses a special notation for the construction of an AST (see Figure 6.4): The symbol `->` explicitly denotes the start of a tree construction rule ("rewrite rule"). The character `^` labels the root node, which, in this case, is an *imaginary token* named `VAR_DECL`.

Imaginary tokens are defined at the top of the file `Ftpm.g` and used to construct AST nodes with tokens not found by the lexer. The usage of imaginary tokens aids to differentiate between similar

[8] http://www.graphviz.org/doc/info/lang.html

[9] http://www.antlr.org/wiki/pages/viewpage.action?pageId=1782

syntactical constructs or specifically mark a subtree inside the AST. Moreover, grammars in subsequent phases become easier to read and write, as imaginary tokens provide a clear reference.

```
i UINT16 := 4 + 2;
```

```
var_declaration
  : ID vartype (ASSIGN expression)?
    -> ^(VAR_DECL ID vartype expression?)
```

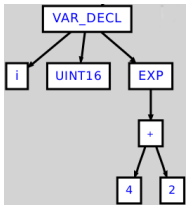


Figure 6.4: FTPM source for a variable declaration (top-left) and the corresponding ANTLR grammar (bottom-left). During processing the parser constructs the AST node shown on the right: VAR_DECL, an imaginary token, is the root node for variable declarations. The subtree for a variable declaration starts with a name, followed by a type and an optional expression, which holds the initial value for the variable.

An alternative notation for AST construction, which is used mainly in the expression rules in `Ftpm.g`, is shown in Figure 6.5: Instead of the `->` symbol, the AST is constructed *inside* the grammar rule with the `^` operator. While this notation is shorter, neither a reordering nor an explicit inclusion of *imaginary* tokens is possible.

The resulting AST from Figure 6.5 also shows the usage of another imaginary token (`EXP`), which was created by a grammar rule prior to `logical_or_expression`. The `EXP` node is used to clearly group expression nodes. A subsequent tree-walker (see Section 6.3.4 or Section 6.3.5) then not has to differentiate between possible expression types again but simply looks for `EXP` nodes to recognize the start of an expression.

```
if (i == 0 or i > 10)
do
  [...]
end;
```

```
logical_or_expression
    :   logical_and_expression (OR^ logical_and_expression)*
    ;
```

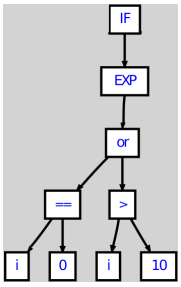


Figure 6.5: A different notation for AST construction using ANTLR: Instead of the `->` operator, the symbol `^` is used *inside* the grammar rule (next to the keyword *OR*) to denote the root node. The FTPM source on the top left is, amongst other rules, processed by the grammar rule `logical_or_expression` (bottom left). The resulting AST is shown on the right side: A root node labeled `IF`, followed by an imaginary node labeled `EXP` followed by the actual expression nodes.

6.3.4 Phase 4: Symbol table management

After the construction of an AST (see Section 6.3.3), all the following phases *walk* the nodes of the tree and perform different actions. The first tree-walking phase consists of generating a symbol table, which will be used for static type checks of the next phase (Section 6.3.5). ANTLR has built-in support to construct tree-walkers from grammars. The tree-walker grammar for symbol table management is defined in `SymTabTreeWalker.g`.

With regards to symbol table management, FTPM uses *static scoping*, which means that each block provides its own scope of visibility. Consequently this means that variables defined in a deeper scope mask variables with the same name from an outer scope. Moreover, a variable defined in an outer scope cannot access variables which are defined in an inner scope but the opposite is possible.

Static scoping can be found in many general purpose programming languages, like C or Java. Another approach would be *dynamic scoping*. In dynamic scoping variables are looked up in the *calling* stack if not found locally. Common Lisp is an example for a programming language which uses dynamic scoping.

While dynamic scoping is more flexible than static scoping, FTPM uses the latter for several reasons. First, we find that statically scoped programs are easier to read than an equivalent dynamic scoped program. As the source for a complete TPM specification can get quite large, dynamic scoping could be the source of hard-to-find problems: If, for example, a submodule of a TPM ordinal wrongly alters the value of a variable defined in some other ordinal. Moreover, we perform several static type checks with FTPM, which would be more difficult to implement with dynamic scoping. Furthermore, FTPM is syntactically closer to languages with static scoping. As one goal of FTPM is to directly generate C source code, static scoping seems like a natural approach as well.

For the actual Symbol-Table Management the classes GlobalScope and LocalScope, which are defined in the Java package ftpm.symtab, are used. Both classes implement the interface Scope (Scope.java), which has a name, an (optional) enclosing scope and methods to define and resolve symbols (see Figure 6.6). Because of the possible enclosing scope, recursive lookup of symbol defined in outer scopes is easily achievable. Common functionality for both concrete classes is provided by the abstract base class BaseScope, which is defined in the same Java package.

The concrete symbols are all defined in the java package ftpm.symtab. The design of the classes for symbol table management and static type checks is taken from [41]. A basic overview of the available classes as well as their hierachy is shown in Figure 6.6. Each Symbol is defined in a Scope and has a Type. Some symbols, namely EnumSymbol, StructSymbol, BitsetSymbol, FunctionSymbol and OrdinalSymbol derive from the class ScopedSymbol. ScopedSymbol is the base class for symbols which provide their own scope for its members. The Type of a class is either a built-in type (BuiltInTypeSymbol), thus signed or unsigned integers, strings or booleans (see Chapter 5.2.3), or a user defined type (TypeDefSymbol or UserDefSymbol).

As stated above, this phase consists of walking the AST and simultaneously create symbols. The created symbols are stored inside the AST nodes (FtpmAst.java). Additionally each symbol has a reference to the its corresponding AST node. This way, access to information that either class holds can be easily provided if needed.

In many implementations of programming languages, comments are disregarded by the lexer and consequently not processed by the parser. However, vital functionality for the automatic generation of documentation in FTPM is contained inside the comments. Therefore, comments in FTPM are treated like normal tokens and are stored as AST nodes. After the creation of symbols, another tree-walker (FtpmCommentsTreeWalker.g) is used to assign possible comments to their corresponding symbols. If the found comment can be assigned to a symbol it is stored inside the base symbol class (Symbol.java). If the comment belongs to the whole file, it is stored in a separate structure (MetaComment.java). For an overview of different comment types see Chapter 5.2.1.

6.3.5 Phase 5: Static type checks

The next phase consists of a tree-walker, FtpmTypeCheckTreeWalker.g, which performs several checks on the available AST nodes. The actual implementation of static checks reside in the class SymbolTable (SymbolTable.java). The result type of static type checks on expression, for example binary operations like and, is stored in the root expression node (see Figure 6.5). This way, the

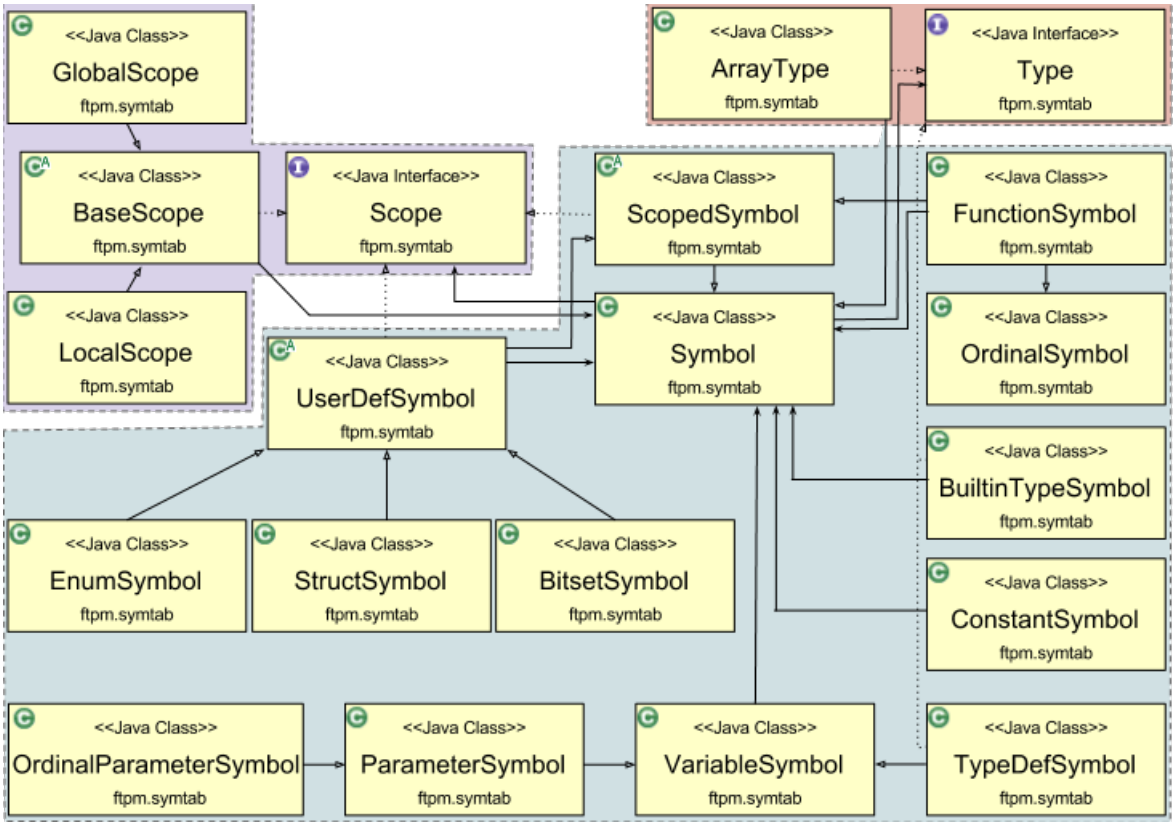


Figure 6.6: UML Diagram of the available classes related to symbol table management in `FTPM`. The colored background indicates different groups of classes: Classes related to scopes have a purple background. A red background denotes classes which hold type information. The green background block shows classes which store various information related to symbols.

result of expressions is passed upwards through the tree and available for subsequent type checks.

In order to allow the computation of static type checks on different, but compatible types, `FTPM` internally *promotes* (casts) certain types if possible. The calculation of a promotion is performed in `SymTabTypeMgmt`. This type promotion eases subsequent checks, which depend on equal types. `FTPM` currently carries out only simple type promotion, or more precisely promotion of integer types without loosing information. Information is not lost, if a *narrow* type is promoted to a *wider* type, for example promoting an `unsigned integer 8bit` to `unsigned integer 16bit`. However, the reverse is not feasible. Consider the following example:

```
1 foo unsigned integer 8bit;
2 foo := -1 - 4294967295;
```

First a variable with the name `foo` is declared with the type `unsigned integer 8bit`. This type is used for unsigned (non-negative) numbers, ranging from 0 to 255. Then, in the assignment the number `-1`, which internally is of type `signed integer 8bit` and the number `4294967295`, which is of type `unsigend integer 32bit` are declared. In order to check, if the assignment to the variable `foo` is feasible, the following type promotions are calculated by `FTPM`: Both numbers, `-1` and `4294967295`, get promoted to a type of `signed integer 64bit` because the result of the expression happens to be in this number range. Then, the AST node for the arithmetic expression (`-1 - 4294967295`) is assigned this computed type. Subsequently, the static type checker verifies if the types (`unsigned integer 8bit` and `signed integer 64bit`) are compatible, which clearly they aren't. Therefore, this check results in the warning:

```
[WARN] foobar.ftpm (52): Cannot assign 'signed integer 64bit' to 'unsigned integer
    8bit'
```

Currently, failed checks performed in this phase do not terminate the execution of the program. Only a warning is emitted, which shows the failed check as well as the respective filename and line number. Moreover, the currently available type checks in the `FTPM` prototype should be improved and extended, as currently only the most basic checks are implemented. Especially the constraints found in the TPM specification are suited to be implemented as proper type checks. For example the definition of Boolean types, which are defined as unsigned integer of 8-bit size in section 2.2.2 of Part 2 of the TPM Main Specification, provides the following comment [57]:

> "Boolean incoming parameter values other than 0x00 and 0x01 have an implementation specific interpretation. The TPM SHOULD return TPM_BAD_PARAMETER."

If the `FTPM` prototype enforces a check like the above it would automatically be enforced in *every* implemented TPM command.

6.3.6 Phase 6: Output generation

The final phase of `FTPM` consists of the *automatic* generation of various different outputs. The sequence for output creation is depicted in Figure 6.7: After the initial phases, which processed the input—starting with the lexer, which produced tokens from the given input stream (see Section 6.3.2), followed by the parser, which created an AST (see Section 6.3.3) that subsequently got traversed by various tree-walkers (see Section 6.3.4 and Section 6.3.5)—the `FTPM` prototype is ready to produce output.

Overview

An `OutputFactory` (`OutputFactory.java`) knows which kind of outputs are available. The parser queries the available outputs from the factory and requests the creation of an `Output` object, which is implemented as an abstract class. An `Output` can use one (or more) templates or use alternative ways for output generation (e.g. process the raw AST nodes from the parser itself). The output creation uses a decorator design pattern: Each output has to implement a `write` method, which returns a `String`. The `OutputFactory` simply calls the write method and decides whether to write the returned `String` to the standard output or to a specified file. Additionally, each output can specify *dependent* output object, which should be created as well. Examples for such dependent objects are Makefiles or `Doxygen` configuration files and.

The classes involved in the output generation are shown in more detail in with the UML diagram in Figure 6.8. The class `Parser` (`Parser.java`) is the main processing point of `FTPM`. Output related classes are bundled in and below the directory `src/ftpm/output`. Every output target either implements the interface `OutputGenerationTargetTemplate` or the more generalized interface `OutputGenerationTarget`. The interface `OutputGenerationTarget` is usesful for classes which produce their output without a template. It is used mainly for debugging classes, which print tokens or debug information to the standard output (`OutputLexerTokens.java` or `OutputStaticTypes.java`), or write the abstract syntax tree in the `dot` format to a file (`OutputASTDot.java`) in order to analyze the generated AST with an external viewer. The class `OutputBytesize` is not used for debugging purposes but rather to print the bytes an TPM command requires to the standard output.

The interface `OutputGenerationTargetTemplate` does not differ very much from `OutputGenerationTarget` mentioned above. It does only add two methods to the interface: `getTemplateDirName()` and `getTemplateFileName()`. These methods are used by `OutputTemplate` to

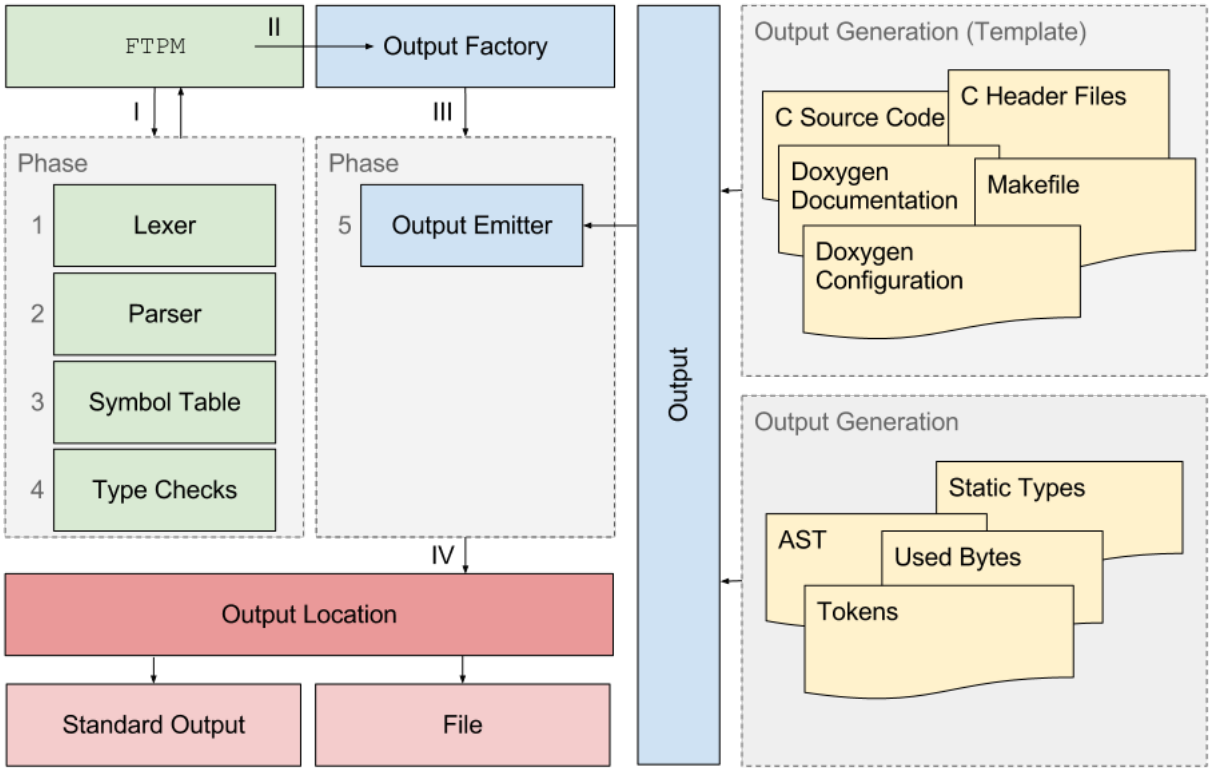


Figure 6.7: Basic schema of output generation in `FTPM`: After the initial phases (1-4, green background), output can be generated through the use of an *Output Factory* (blue). *Output* objects either use templates or directly process the AST nodes from the parser. Then, output is written to an *Output Location* (red background), which can be the *Standard Output* or a specified *File*.

load and process a specified `StringTemplate` file. Code and documentation generation are examples for this classes of output targets, which can be found in `OutputCode_C.java`, `OutputCode_H.java` and `OutputDocDoxygen.java`.

Templates

The actual processing of a `StringTemplate` closely mimics any other tree-walking with `ANTLR`. However, the `->` notation, which was used to build AST nodes in the parser, now calls `StringTemplate` definitions. These definitions are the foundation for output emission using templates. Moreover, `FTPM` uses the available inheritance mechanism of `StringTemplate`. A new template does not have to provide definitions for all available template calls but rather overrides only the ones needed by this specific template. As the default action in the top-level template does not emit any output at all, specific templates can be kept short. For example, the template for `C` header files does not need to implement the template for a function body as it is merely interested in the function declaration and thus is much shorter than the template for `C` source code.

The grammar for template aided output generation can be found in `FtpmOutput.g`. Output emission can be controlled by two kinds of *renderers*. The first kind of renderer is passed to the tree grammar and decides if a certain attribute is passed to the template at all. Currently, the class `SameSourceTemplateRenderer` (`SameSourceTemplateRenderer.java`), which implements the interface `TemplateRenderer`, is used for this task. As the name suggests, this class only emits output if the current attribute in the AST comes from the currently processed file. This way, the file structure from the input is transferred to the output: A file `A.ftpm`, that includes `B.ftpm`,

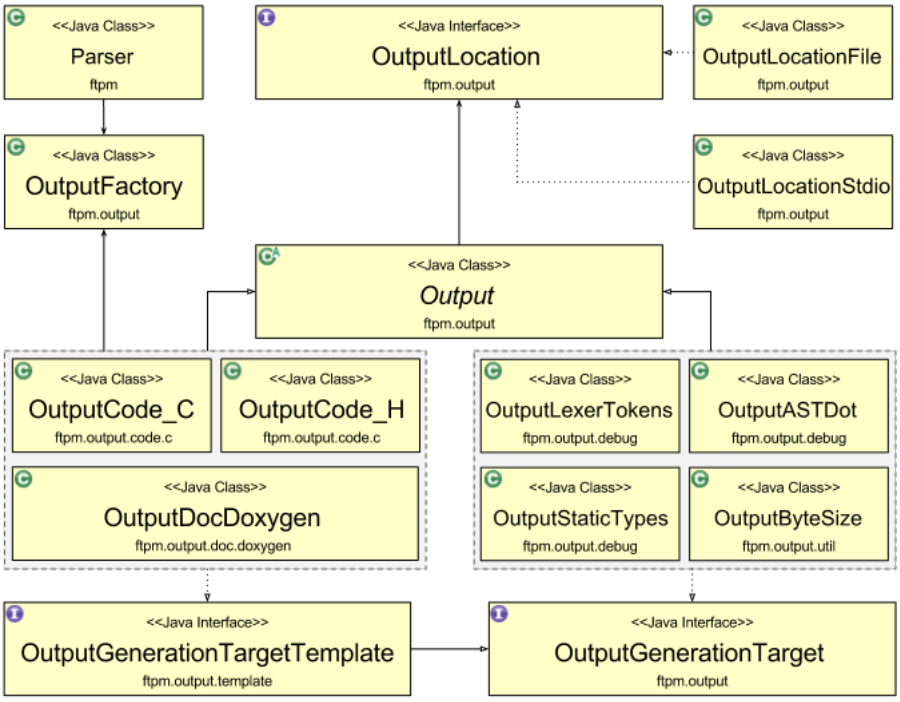


Figure 6.8: Excerpt of the classes responsible for various output generation: An OutputFactory governs the available output emitters. Emitters either use templates, like Documentation (Doxygen) or Code (C source and header files) or process the AST directly (OutputStaticTypes, OutputByteSize, ...).
Every Output implements the interface OutputGenerationTarget, which provides a write method. Upon creation of the output, this method gets called on the provided Output channel (StdOut or a File).

will produce the files `A.c` and `B.c` and not a single file which combines both input sources. However, with this strategy, the AST has to be walked for every input file separately, even when most of the input is discarded and only a small portion of the AST is used for output emission. Therefore, a future optimization of `FTPM` is to walk the AST only once and generate output for different files simultaneously.

The second kind of *renderer* is responsible for the representation of AST nodes inside the template. `StringTemplate` does not allow logic inside the template as it would break the model-view-controller pattern [45, 34], which is an underlying design principle of the engine [39]. However, a template may use a renderer to control how the output looks like. An example for such a renderer can be found in `BuiltInTypeSymbolRenderer.java`, which maps `FTPM` variable type definitions to equivalent `C` type definitions (see Listing 6.2) and additionally cares that needed header file (in this case `stdint.h` and `stdbool.h`) are included in the final output as well. This renderer is installed for

```
unsigned integer  8bit -> uint8_t
unsigned integer 16bit -> uint16_t
unsigned integer 32bit -> uint32_t
unsigned integer 64bit -> uint64_t
boolean                -> bool
```

```
signed integer  8bit -> int8_t
signed integer 16bit -> int16_t
signed integer 32bit -> int32_t
signed integer 64bit -> int64_t
string               -> char*
```

Listing 6.2: `StringTemplate` renderer, which provides a mapping between variable types available in `FTPM` and available in `C`.

the class `BuiltInTypeSymbol`. The template passes every object of this class to the renderer, which cares for the correct mapping.

Example

This chapter concludes with a short but complete example of translating a code snippet in FTPM to valid C source code. We will illustrate the output mechanism described above and show intermediate results and important components. The example is depicted from Listing 6.3 to Listing 6.6.

First, Listing 6.3 declares a variable in FTPM with the name i. This variable is of the type unsigned integer 16bit, which is a built-in type of FTPM (see Chapter 5.2.3). Additionally, the variable is also initialized with the expression 4 + 2.

```
1 i unsigned integer 16bit := 4 + 2;
```

Listing 6.3: Declaration of a variable in FTPM.

This statement is first processed by the lexer and parser resulting in an AST representation. Then this AST is traversed by tree-walkers, which build a symbol table (FtpmSymTabTreeWalker.g), process comments (FtpmCommentsTreeWalker.g) and perform static type checks (FtpmStaticTypeChecker.g). Then, a final tree-walker is responsible for output generation.

Listing 6.4 shows an excerpt of the rule in the tree grammar of FtpmOutput.g, which processes variable declarations. The rule states that a variable declaration in FTPM is made up of an imaginary root node VAR_DECL, which was inserted by the parser during AST creation. Under this root node follows an identifier, a type and an optional expression. The template definition inside the rule, which is also called var_declaration, is only called if the method renderAttribute returns true (Disambiguating Semantic Predicate). This method is responsible for the rendering of AST nodes from the same file only, which was mentioned above. If the nodes come from the currently processed file, the template definition var_declaration is called. If not, the second -> operator matches, which purposely discards the AST nodes.

```
1 var_declaration
2     :   ^(VAR_DECL i=ID t+=vartype (v+=expression_root)?)
3         ->  { renderAttribute($i) }? var_declaration(name={$i}, type={$t},
              opt_value={$v})
4         -> // empty on purpose
```

Listing 6.4: Tree grammar rule in FtpmOutput.g for a variable declaration

The template definition for var_declaration is shown in Listing 6.5. It is a simple StringTemplate definition which has three input parameters: a name, a type and an optional value. The rule arranges this three parameters to valid C, which prints the type of variable first and then its name. If a value is passed to the rule as well, an additional assignment operator followed by the value is printed. The statement is finalized by a semicolon.

```
1 var_declaration(name, type, opt_value) ::= <<
2 $type$ $name$$if(opt_value)$ = $opt_value$$endif$;
3 >>
```

Listing 6.5: StringTemplate definition for the creation of C output for a variable declaration in FTPM.

The final output is shown in Listing 6.6. As in this case the object of type happens to be of type BuiltInTypeSymbol, the corresponding BuiltInTypeSymbolRenderer is implicitly called

by the template. This renderer maps the type `unsigned integer 16bit` to the type `uint16_t` (see Listing 6.2). Moreover, the needed header file for this type, `stdint.h`, is included in the final output as well.

```
#include <stdint.h>

uint16_t i = 4 + 2;
```

Listing 6.6: Example output in `C` of the variable declaration in `FTPM` from Listing 6.3

Every output class extends the abstract class `Output`. Through this class an *output location* can be set. Upon calling the method `Output.write()` the returned string is either written to standard output or to a specified file. Moreover, an output class can specify dependent output targets, which will be written subsequently as well. In order to not circularly write the same files multiple times, the `Output` class keeps track of the files that were already written. Through the design described above, the creation of output for `FTPM` is reduced to the lines of code depicted in Listing 6.7.

```
// target:    OutputCode_C.IDENTIFIER,  OutputDocDoxygen.IDENTIFIER, ...
// inputfile: the ftpm source file
// nodes:     AST produced by the parser
Output o = outFactory.createOutput(target, inputfile, nodes);
o.setOutputLocation(new OutputLocationFile());
o.write();
```

Listing 6.7: Java call in `Parser.java` to create an output of type *target* with *nodes* processed by the parser for the file named *inputfile*.

An excerpt of the method, which is called by the `write` method from Listing 6.7 is shown in Listing 6.8. First, the corresponding template file gets loaded by `TemplateLoader`. Afterward, a renderer—`BuiltInTypeSymbolRenderer`—is registered to render objects of type `BuiltInTypeSymbol` in the loaded template. Finally, the class `OutputTemplate` walks the AST nodes produced by the parser with the tree-grammar `FtpmOutput.g` and applies the passed template. The result is returned to the caller, which may write it to a file or print it to the standard output.

```
public String generate()
{
    StringTemplateGroup template = new TemplateLoader(TEMPLATE_DIR_NAME,
        TEMPLATE_FILE_NAME, TemplateLoader.Category.CODE).getTemplate();

    BuiltInTypeSymbolRenderer typeRenderer = new BuiltInTypeSymbolRenderer();
    template.registerRenderer(BuiltInTypeSymbol.class, typeRenderer);

    // apply template
    TemplateRenderer templRenderer = new SameSourceTemplateRenderer(
        getInputSourceName());
    OutputTemplate ot = new OutputTemplate(nodes, template, templRenderer);
    return ot.applyTemplate().toString();
}
```

Listing 6.8: Excerpt of the Java call in `OutputCode_C.java` to apply a template to AST nodes.

6.4 Summary

This chapter presented the technical details of the implementation of the FTPM prototype. After a general overview of FTPM and used libraries and tools we presented the implementation of the FTPM program in detail. We thoroughly discussed each phase a FTPM input source traverses, which consists of processing command-line options, lexical analysis and parsing the created tokens. After the creation of a symbol table several type checks are performed on the created abstract syntax tree. Finally, the FTPM input is translated to various different outputs. We implemented FTPM as a statically typed language because of possible unintentional side-effects and affinity to the C language because of the chosen output target. Furthermore, we showed how the *automatic* generation of various output formats is implemented. Finally, we illustrated the whole process with all discussed phases on a simple but complete example of generating C source code out of FTPM.

Chapter 7

Outlook – TPM 2.0

The TCG already started to work on the next major version of the TPM specification: TPM 2.0. This chapter outlines a few notable design decisions of the upcoming version. We discuss where FTPM can be used directly with the next version of the TPM specification and where the specification differs and thus has to be adapted. Next, we propose ideas for improving FTPM that arose during the development of FTPM to describe our vision of an optimized workflow for the creation of the next TPM specification.

7.1 Next version of the TPM specification

The TPM specification continues to evolve. Currently, the Trusted Computing Group works on the release of the next version of the specification. A public review draft for this next specification has been posted in October 2012[1], which will be the base for the TPM specification version 2. This TPM specification consists of Part 1: Architecture [61], Part 2: Structures [62], Part 3: Commands [63] and Part 4: Supporting Routines [64].

Again, the complete specification is comprehensive. It consists of about 1400 pages spread over all four parts. In order to completely understand the specification, it is necessary to understand all four parts. Internally many things will change with this new specification version. Besides logical adjustments, like an interface to use cryptographic algorithms other than SHA-1 and RSA or reworked authorization methods, the TCG explicitly states that automatic processing of the specification text is encouraged:

> "The information in this document is formatted so that it may be converted to standard computer language formats by an automated process. The purpose of this automated process is to minimize the transcription errors that often occur during the conversion process."

Moreover, the next version of the TPM specification includes a reference implementation in C. While this is definitively a great help for implementers it also introduces a new level of complexity to the specification: Source code *inside* the specification text. The choice for the programming language C to demonstrate the implementation of data structures and commands for a hardware module seems obvious. However, C itself is a low-level, *general purpose* programming language. Memory allocation and pointer arithmetic are common operations in C and interfere with the readability of the specification. Besides the complexity of the TPM specification, the additional complexity of the program language demands close attention and great care by the implementer. Moreover, if a different language for an implementation of TPMs is needed, a reference implementation in C is not directly usable.

FTPM on the other hand tries to provide a lean language for describing TPM commands and structures. It strives to draw attention to important parts of the specification (e.g. authorization, parameter

[1] http://www.trustedcomputinggroup.org/resources/trusted_platform_module_specifications_in_public_review

blocks) and hide complex operations with a specialized syntax at the same time. The changes introduced to the next version of the specification for TPMs mean that FTPM needs adaption to reflect these changes. However, as we see it, the next version of the specification of TPM can also be realized in FTPM. Moreover, automatic processing *is* already the corner stone of FTPM and thus highly appropriate. Additionally, our approach of an executable TPM specification hides complex details of the implementation, which are now part of the specification text (e.g. memory allocation and pointer arithmetic).

7.2 Ideas for Future Work

FTPM is able to produce arbitrary output. Currently, the translator of FTPM is able to produce C source and header files as well as documentation with Doxygen. The documentation mimics the currently available specification structure but additionally improves the usability with hyperlinks between different parts. Chapter 4 already mentioned possible improvements in few areas:

- A better notation for include files, which reduces the probability of typing errors and eliminates the need for *meta comments*. This optimization results in greater flexibility in the documentation generation, too.
- A notation to create links between related chapters in comments of FTPM. A more tightly referenced specification alleviates navigation. This aids comprehension of a complicated specification as the reader is pointed to all related information at once.
- Automatic generation of marshalling code. The information to create code which translates a binary stream sent by the upper software stack to a TPM into parameters for a specific code and back is already contained in the current notation of FTPM. However, the marshalling code is not implemented in our FTPM prototype, yet.

Another improvement, which was not mentioned in the previous chapter, concerns the documentation generation. If hyperlinks are not only generated for data types in the parameter blocks but also automatically for every named variable in the action comments of the specification text, related information is even more connected. As the symbol table is available to the translator, an obvious solution to this problem is to try to look up every single comment text in it. If an entry is found, a link to the definition of the type is generated.

As mentioned in Chapter 3.4.1, the creation of a domain-specific language is complex and time consuming. Therefore, this thesis chose only a few commands and structures of the TPM specification for a first prototype. However, in our opinion the most important thing FTPM needs is extensive testing: More TPM commands need to be implemented in FTPM in order to detect possible shortcomings of the language that we are not aware of. It is evident that FTPM cannot be regarded to be finished without a complete implementation of all commands. Until then, it is uncertain, if FTPM in its current form is really adequate to create a specification for TPMs.

Of course, the FTPM language itself can be further refined to better suit the specification design process. Currently, there exist some rough edges, like the include mechanism mentioned in Chapter 4.2. However, the language itself evolves with the implementation of more commands and with more extensive testing. Most importantly, the more commands are implemented, the more custom rules, which are found throughout the specification text, can be enforced by the FTPM compiler. If the abidance to this rules is checked by a computer program it is easier for an implementer to comply.

Furthermore, additional tools to aid the development of TPM commands are useful. Besides the generation of marshalling code, good editor support is definitely a must to reach adaption of FTPM. While not strictly necessary, syntax highlighting or code completion are a tremendous relief for programmers. A file to get rudimentary syntax highlighting for the *VIM* editor can be found in Appendix C.1.

It is also imaginable to elaborate on the workflow associated with the creation of a specification for TPMs: An specialized editor provides an environment well suited to create TPM commands. This editor ideally provides syntax-highlighting and code completion. The execution of the `FTPM` tool informs the creator of found problems. When the implementer is satisfied with the command, structure or text, the source code is transferred into a Version Control System. Then, the editor is obliged to enter a commit message, which ideally describes the conducted changes in detail. When the specification reaches a certain maturity a revision or major version can be created, *directly* from the VCS. This way, a complete workflow, along with an exhaustive history, is made available to implementers and specification designers.

7.3 Summary

This chapter gave a short overview of the next version of the TPM specification. The upcoming version has put effort into providing a document which can be automatically processed in order to avoid transcription errors. Coincidentally, this was one motivation for the creation of `FTPM`. The upcoming version of the TPM specification provides a reference implementation in `C`. While a reference implementation is an important help for implementers, it also introduces a new level of complexity through the use of a low-level, general purpose programming language. The complexity of the `C` programming language hides important parts of the specification and is not very useful for implementers who target a different language. `FTPM` minimized the complexity with a specialized syntax. There are no pointers or memory allocation necessary. It emphasizes important parts of a command, like incoming and outgoing parameters or authorization information. Moreover, different outputs directly from `FTPM` are realized easily.

Next, we proposed future improvements to `FTPM`. We pointed out that currently the language cannot be considered stable or feature complete as long as only a small subset of TPM commands is implemented. Further testing is necessary to uncover problematic areas in the language. Moreover, we proposed areas which need further work. We mentioned improvements for the include mechanism, tighter integration of related chapters and support by tools outside of `FTPM` (e.g. an editor with syntax highlighting). Finally, we also gave our vision of an optimized workflow for the creation of a specification text, which is centered around `FTPM`.

Chapter 8

Concluding Remarks

In this work we presented FTPM, which is a domain-specific language specifically tailored to describe TPM commands and data structures. In Chapter 1 we first introduced to the problem and subsequently proposed our solution. We mentioned problems like inconsistencies, poor visual representation or that no reference implementation of TPMs is publicly available. This is especially problematic as TPMs are widespread available devices and aim to provide a certain level of security. Then, we proposed a new tool for writing and maintaining the specification of TPMs: FTPM, which strives for a concise syntax that emphasizes important elements and hides unnecessary complexity at the same time. We wanted to create different output *automatically* and *directly* from a specification written in FTPM.

Chapter 2 provided background information on the problem area. We gave an overview of the purpose of a TPM, its capabilities and the used underlying cryptographic routines. The main topics presented in cryptography were encryption, decryption and digital signatures. We discussed advantages and disadvantages of symmetric and asymmetric algorithms and outlined the importance of (true) random number generation for cryptography as a whole and especially for a dedicated hardware chip that is responsible to manage trust. After an introduction to hash functions we discussed the used cryptographic algorithms currently deployed in a TPM. Furthermore, we mentioned the basic cornerstones of a TPM: Cryptographic Keys, Measurement, Storage and Attestation. Moreover, we discussed the authorization mechanism and the concept of Rolling Nonces, which are widely used concepts in TPMs.

The next chapter, Chapter 3, presented related technologies in the problem area. We first covered dedicated specification languages, like VDM-SL or Z-Notation. While formal specification languages provide a solid base to prove compliance or noncompliance they are also difficult in such a complex scenario like the TPM specification. Moreover, the resulting mathematical models are often understood by specialists in contrast to our stated goal of making the specification more readable. Then, we discussed *Literal programming*, which tries to combine a written specification with executable source code. Documentation generators like Doxygen or Javadoc, which we discussed next, follow a similar approach where specification text and source code is physically close together. The benefit of such an approach is that it is arguably more likely that code and specification text evolve synchronously. Finally, we discussed the usage of domain-specific languages. We stated that the specification of TPMs is well suited for a domain-specific language because of *"repetitive elements or patterns"* can be found and it will be *"used by a domain expert"* [49]. We illustrated each of the approaches discussed in the chapter by an example as well as discussed advantages and disadvantages.

Then, Chapter 4 presented the capabilities of FTPM on real examples. We provided an exemplified implementation for three TPM commands: TPM_OIAP, TPM_PCRREAd and TPM_EXTEND. We showed the input in the FTPM language and its results: C source code and documentation suitable to be processed by Doxygen. We highlighted the advantages of our approach, which can be summarized as follows:

- Generation of code and documentation, which ensures that these parts of the specification evolve synchronously and are always up-to-date. Specification designers and implementers can (re-) create the complete documentation and large parts of C source code and header files automatically, on demand and at any time (even during the development process). Access to the specification to as many people as possible increases the chance to detect design errors and security flaws.
- Validation of data structures and expressions used in TPM commands, which ensures that changes to one part of the specification do not accidentally break the specification in another part unnoticed. The parser can report the unexpected use of data types to the specification designer.
- Enforcement of a semantic policy throughout the specification. Given the size of the TPM specification (either version 1.2 or 2.0) it is easy for implementers to accidentally disregard stated rules. If the rules are monitored by the parser it is possible to warn the user upon violation.

We also showed the improvements we could achieve to the documentation if it is automatically generated by FTPM: Cross-referencing related chapters results in a fast navigation through the hundreds of pages of specification text. We find this a valuable addition and used it extensively during the creation of the FTPM prototype: As soon as the needed data structures for our sample commands were available in FTPM we rather used the generated documentation than the official specification to look up needed information.

Because the FTPM language is written in plain text it is very suitable to be stored in a version control system (VCS). A VCS provides access to the complete history of the specification and to the thoughts and motivations during the development process. This may be an important source to understand the intentions but can also be used to automatically generate a chapter that includes changes from one version of the specification to the next.

The next chapters, Chapter 6 and Chapter 5 gave a complete overview of how our solution works. Chapter 5 introduced the FTPM language. We discussed the most important syntactical constructs and outlined the thoughts that led to FTPM in its current form. Furthermore, this chapter detailed special constructs in FTPM that in our opinion improve the presentation of the specification. Examples are the notation for needed incoming and outgoing parameters or authorization information. Another example is the omission of unneeded parameters in TPM commands that are common to every command. Furthermore, we demonstrated syntactic elements in the FTPM language that aim to group related information and subsequently ease recognition. We presented the notation for authorization blocks as an example for such elements.

In Chapter 6 we gave insight to the used technologies that were used in our prototype implementation. We introduced used tools and concepts, like ANTLR as parser generator and templates for structured output generation with StringTemplate. Furthermore, we detailed the different processing phases that are necessary to translate an input FTPM source to final outputs like source code and documentation. We presented the most important classes involved in processing of the syntax tree as well as relationships between them. Then, we also showed extensively how we implemented our approach to emit structured text and how the template mechanism works.

Finally, Chapter 7 gave an outlook on the next version of the specification of TPM, which is currently created. As a first draft of the specification became available during the creation of this thesis we shortly discussed the major changes this next version will introduce. Furthermore, we briefly contemplated which modifications to FTPM are necessary in order to become fully functional for this next TPM specification version. Moreover, we also stated disadvantages and shortcomings of the current FTPM prototype. We summarized our proposed changes to FTPM and recapitulated our suggested solution from Chapter 1: A complete workflow dedicated to the creation and modification of the TPM specification.

Appendix A

Source Code Listings for Examples presented in Related Work

This appendix provides a full listing of the presented example implementation of a stack used with *Literate Programming* (see Chapter 3.2.2) and `Doxygen` (see Chapter 3.3.1).

A.1 Literate programming using `noweb`

If `noweb` is installed and the code listing is saved in a file called `stack.nw`, the following simplistic `Makefile` can be used to extract documentation (in LaTeXor html) as well as `C` source and header files.

```
doc-html:
        noweave -filter l2h -index -autodefs c -html stack.nw > stack.html

doc-latex:
        noweave -index -latex stack.nw > stack.tex

code:
        notangle -Rstack.h stack.nw > stack.h
        notangle -Rstack.c stack.nw > stack.c
        notangle -Rtest.c stack.nw > test.c
```

```
@ \section{Introduction}
This is a (minimal) implementation of a stack
in \textsf C using a linked list for it's elements. \\
The following operations are implemented on the stack:
(for possible errors see Section \ref{secterr})
\begin{enumerate}
 \item [[push]] \\
   Pushes an element on top of the stack \\
 \item [[pop]] \\
   Removes the topmost element from the stack \\
 \item [[peek]] \\
   Returns (but does not remove) the topmost element from the stack \\
 \item [[empty]] \\
   Tests if the stack is currently empty (boolean)
 \item [[error]] \\
   Prints an error message and aborts the program
\end{enumerate}

@
\section{License}
```

```
The implementation is based on the Wikipedia Example of a #
Stack\footnote{http://en.wikipedia.org/wiki/Stack\_(abstract\_data\_type)}

<<license>>=
This software is in the public domain.

@
\section{Error conditions - Over- and underflow}
\label{secterr}
The following error conditions may arise:
\begin{enumerate}
 \item Overflow: No more space on the heap to allocate a new element
 \item Underflow: Pop or peek operation on an empty stack
\end{enumerate}


@
\section{Implementation}
The program has the following outline:
<<stack.c>>=
/*
  <<license>>
 */
<<includes>>

<<error.c>>
<<empty.c>>
<<peek.c>>
<<pop.c>>
<<push.c>>


@
\section{Interface}
The program provides the following public visible functions:
<<stack.h>>=

<<data structure>>

<<error.h>>
<<empty.h>>
<<peek.h>>
<<pop.h>>
<<push.h>>


@
\section{Test}
The following shows example usage:
<<test.c>>=
<<test>>




<<data structure>>=
typedef struct stack {
  int data;
  struct stack *next;
} STACK;


```

```
<<error.h>>=
/* Print out error message and exit */
void error(char *msg);

<<empty.h>>=
/* Test if STACK 'stack' is empty */
int empty(STACK *stack);

<<peek.h>>=
/* Return (but do not remove) topmost element from STACK 'stack' */
int peek(STACK *stack);

<<push.h>>=
/* Push element 'value' on top of STACK 'stack' */
void push(STACK **head, int value);

<<pop.h>>=
/* Return (and remove) topmost element from STACK 'stack' */
int pop(STACK **head);



<<includes>>=
#include "stack.h"

#include <stdio.h>
#include <stdlib.h>

<<error.c>>=
void error(char *msg)
{
  fprintf(stderr, "Error: %s\n", msg);
  abort();
}

<<empty.c>>=
int empty(STACK *stack)
{
  return stack == NULL;
}

<<peek.c>>=
int peek(STACK *stack)
{
  return stack->data;
}

<<push.c>>=
void push(STACK **head, int value)
{
  /* create a new node */
  STACK *node = malloc(sizeof(STACK));

  if (node == NULL){
    error("Error: no space available for node");
  }

  /* initialize node */
  node->data = value;
  node->next = empty(*head) ? NULL : *head; /* insert new head if any */
  *head = node;
}
```

```
145
146 <<pop.c>>=
147 int pop(STACK **head)
148 {
149   /* stack is empty */
150   if (empty(*head)) {
151     error("Error: stack underflow");
152   }
153   /* pop a node */
154   STACK *top = *head;
155   int value = top->data;
156   *head = top->next;
157   free(top);
158   return value;
159 }
160
161 <<test>>=
162 #include "stack.h"
163 #include <stdio.h>
164
165 static void reportStatus(STACK *stack)
166 {
167   fprintf(stdout, "Stack empty now? %s\n", empty(stack) ? "Yes" : "No");
168 }
169
170 int main(int argc, char **argv)
171 {
172   STACK *stack;
173
174   reportStatus(stack);
175
176   fprintf(stdout, "Pushing '1' on stack.\n");
177   push(&stack, 1);
178   reportStatus(stack);
179
180   fprintf(stdout, "Peeking stack: %d\n", peek(stack));
181   reportStatus(stack);
182
183   fprintf(stdout, "Poping Stack: %d\n",  pop(&stack));
184   reportStatus(stack);
185
186   return 0;
187 }
```

A.2 Documentation generator: `Doxygen`

After `Doxygen` is installed, a configuration file has to be created with

```
doxygen -g
```

Optionally, this configuration file can be adapted. Subsequent runs of

```
doxygen
```

will result in outputs similar to Figure 3.2

```
1 /**
2  * @file   stack.h
3  *
```

```
4   * @brief  A (minimal) implementation of a stack in \c C using a linked list
5   *         for it's elements.
6   *
7   * The following operations are implemented on the stack:
8   * @see Errors
9   *  - push <br>
10  *    Pushes an element on top of the stack
11  *  - pop <br>
12  *    Removes the topmost element from the stack
13  *  - peek <br>
14  *    Returns (but does not remove) the topmost element from the stack
15  *  - empty <br>
16  *    Tests if the stack is currently empty (\c boolean)
17  *  - error <br>
18  *    Prints an error message and aborts the program
19  *
20  * @todo Finish Document
21  *
22  * @author martin.kapeundl@student.tugraz.at
23  * @date   sept 2012
24  */
25
26 /** Stack data structure (linked-list, integer only)
27  *
28  * This Stack implementation uses a linked list for it's elements
29  * and operates only on integer values.
30  * The actual value of the stack element is held in \c data,
31  * the \c next pointer points to the next (lower) element in the stack
32  *
33  * @param data int actual data held by this stack variable
34  * @param next pointer to next element in stack
35  */
36 typedef struct stack {
37         int data;
38         struct stack *next;
39 } STACK;
40
41
42 /** Print out error message and exit
43  *
44  * Prints an error message (to \c stderr) and calls \c abort()
45  *
46  * @param msg Error message to be printed out on \c stderr
47  */
48 void error(char *msg);
49
50
51 /** Test if stack is empty
52  *
53  * Tests if the stack denoted by \c stack is currently empty
54  * Returns 0 for false, any other value for true
55  *
56  * @param stack The stack to test
57  * @return int  0 false, everything else true
58  */
59 int empty(stack *stack);
60
61
62 /** Return (but do not remove) topmost element from the stack
63  *
64  * Returns the value of the topmost element from stack but
65  * does not remove the element (unlike pop())
```

```
66   *
67   * @param stack The stack from which the topmost value will be returned
68   * @return int value of stack element
69   */
70  int peek(STACK *stack);
71
72
73  /** Return (and remove) topmost element from the stack
74   *
75   * Returns and removes the topmost element from the stack.
76   * Frees any memory associated with it.
77   *
78   * @param stack The stack from which the topmost element will be returned
79   * @return int value of stack element
80   */
81  int pop(STACK **stack);
82
83
84  /** Push element \c value on top of the stack
85   *
86   * Pushes an element with value \c value on top of the stack
87   * Allocates memory for a new element.
88   *
89   * @param stack The stack onto which the element will be pushed
90   * @param value The value to push onto the stack
91   */
92  void push(STACK **stack, int value);
93
```

Appendix B

FTPM Language

This appendix first gives a short overview of the java project. Then, the main ANTLR grammar files for the FTPM language (see Chapter 5) is given.

B.1 Directory structure

The FTPM program is organized into several directories. The names of the top-level directories in the program are:

antlr-generated contains java classes for the lexer, parser and tree-walkers. These files are automatically generated by ANLTR and should not be edited directly.

lib contains third party libraries as Java Archive (jar) files. Currently, ANTLR and Commons CLI are used.

src contains all Java classes needed for FTPM. The main method can be found in Ftpm.java, the starting point for the lexer, parser and tree-walkers is in Parser.java. Inside the src folder, the logic is organized into several sub directories: ast (AST construction), output (several classes for various output generation), parser (ANTLR grammar files), symtab (symbol and symbol table management) and util (helper classes and utilities).

templates contains StringTemplate files for each available output generation. The templates are organized into sub-directories for code- and documentation-generation, respectively.

B.2 Grammar

This section shows a simplified ANTLR grammar for FTPM. We removed comments as well as helper functions in order to highlight the main aspects of the grammar. For the same reason, tree construction rules are omitted as well. First, the parser is printed. Lowercase names indicate parser rules while uppercase names are used for Lexer rules. After the parser the complete lexer grammar is printed as well. Again, unnecessary implementation details and comments have been removed.

B.2.1 Parser

```
1 parse
2     :   (declaration)* (ordinal)?
3     ;
```

```
declaration
    :   include
    |   vardef
    |   functiondef
    |   comment
    ;

include
    :   INCLUDE
    ;

comment
    :   COMMENT_FIRST COMMENT_CONSECUTIVE*
    ;


// variable definitions
vardef
    :   var_declaration SCOLON -> var_declaration
    |   typedef_declaration
    |   struct_declaration
    |   enum_declaration
    |   bitset_declaration
    ;

// Variable declaration
var_declaration
    :   ID vartype (ASSIGN expression)?
    |   ID vartype OBRACKET array_capacity? CBRACKET (ASSIGN expression)?
    ;

array_capacity
    :   (a=INT | a=ID)
    ;

vartype
    :   primitive_type
    |   STRUCT ID
    |   BITSET ID
    |   ID
    ;

primitive_type
    :   INTEGER
    |   BOOLEAN
    |   STRING
    ;

// Typedef
typedef_declaration
    :    TYPEDEF ID IS vartype SCOLON
    |    TYPEDEF ID IS vartype OBRACKET array_capacity? CBRACKET SCOLON
    |   (TYPEDEF | CONSTANT) ID IS INT SCOLON
    ;

// Struct
struct_declaration
    :   STRUCT ID IS OPAREN struct_declaration_list? CPAREN SCOLON
    ;
```

```
struct_declaration_list
    :   struct_member (COMMA struct_member)* ->  struct_member+
    ;

struct_member
    :       comment* ID vartype
    |   comment* ID vartype OBRACKET array_capacity? CBRACKET
    ;

// Enum
enum_declaration
    :   ENUM ID IS vartype OPAREN enumerator_list CPAREN SCOLON
    ;

enumerator_list
    :   enumerator (COMMA enumerator)*
    ;

enumerator
    :   comment* ID (ASSIGN INT)?
    ;

// Bitset
bitset_declaration
    :   BITSET ID IS vartype OPAREN bitset_declaration_list CPAREN SCOLON
    ;

bitset_declaration_list
    :   bitset_declarator (COMMA bitset_declarator)*
    ;

bitset_declarator
    :   comment* ID (ASSIGN INT)?
    ;

// Function defintions
functiondef
    :   FUNCTION ID OPAREN func_parameter? CPAREN block
    ;

func_parameter
    :   param_declaration (COMMA param_declaration)*
    ;

// Ordinals
ordinal
    :   ORDINAL ID parameter_block block
    ;

// Parameter
parameter_block
    :   incoming_parameter_block
        outgoing_parameter_block
    ;

incoming_parameter_block
    :   INCOMING_BEGIN COLON param_declaration? (COMMA param_declaration)*
    ;

outgoing_parameter_block
    :   OUTGOING_BEGIN COLON param_declaration? (COMMA param_declaration)*
    ;
```

```
128 param_declaration
129     :   comment* ID vartype
130     |   comment* ID vartype OBRACKET CBRACKET
131     ;
132
133 block
134     :   BEGIN statement* END SCOLON
135     ;
136
137 // Statements
138 statement
139     :   auth_block
140     |   block
141     |   var_declaration SCOLON
142     |   tmp_var_declaration SCOLON
143     |   if_statement
144     |   loop
145     |   builtin_functions SCOLON
146     |   stmt_expressions
147     |   return_stmt
148     |   comment
149     ;
150
151 stmt_expressions
152     :   lhs ASSIGN expression SCOLON
153     |   postfix_expression SCOLON
154     ;
155
156 auth_block
157     : AUTH
158         DOT auth_object   SCOLON
159         DOT auth_secret   SCOLON
160         DOT auth_required SCOLON
161       END SCOLON
162     ;
163
164 auth_object
165     :   { input.LT(1).getText().equals("object")}?
166         'object' ASSIGN expression
167     ;
168
169 auth_secret
170     :   { input.LT(1).getText().equals("secret")}?
171         'secret' ASSIGN expression
172     ;
173
174 auth_required
175     :   { input.LT(1).getText().equals("required")}?
176         'required' ASSIGN expression
177     ;
178
179 return_stmt
180     : RETURN expression? (COMMA expression)* SCOLON
181     ;
182
183 tmp_var_declaration
184     :   TMP_VAR a=ID ASSIGN b=ID(DOT ID)*
185     |   TMP_VAR a=ID ASSIGN b=ID OBRACKET expr CBRACKET
186     |   TMP_VAR a=ID ASSIGN b=ID OPAREN c+=expression_list CPAREN
187     ;
188
189
```

```
// expressions
lhs
    :   postfix_expression
    ;

expression
    :   expr
    ;

// expression without EXP root node
expr
    :   conditional_expression
    ;

conditional_expression
    :   ( logical_or_expression)
        ( QMARK expression COLON conditional_expression
    ;

logical_or_expression
    :   logical_and_expression ( (OR | OR_SHORT) logical_and_expression)*
    ;

logical_and_expression
    :   inclusive_or_expression ( (AND | AND_SHORT) inclusive_or_expression)*
    ;

inclusive_or_expression
        : exclusive_or_expression (BOR exclusive_or_expression)*
        ;

exclusive_or_expression
        : and_expression (BXOR and_expression)*
        ;

and_expression
        : equality_expression (BAND equality_expression)*
        ;

equality_expression
    :   relational_expression ((EQUALS | NEQUALS) relational_expression)*
    ;

relational_expression
    :   shift_expression ((LT | GT | LTEQUALS | GTEQUALS) additive_expression)*
    ;

shift_expression
        : additive_expression ((LSHIFT|RSHIFT) additive_expression)*
        ;

additive_expression
    :   multiplicative_expression ((ADD | SUBTRACT) multiplicative_expression)*
    ;

multiplicative_expression
    :   power_expression ((MULTIPLY | DIVIDE | MODULUS) power_expression)*
    ;

power_expression
    :   unary_expression (POW unary_expression)*
    ;
```

```
252 unary_expression
253     :   op=SUBTRACT unary_expression
254     |   (op=EXCL | op=EXCL_SHORT) unary_expression
255     |   op=BNOT unary_expression
256     |   postfix_expression
257     ;
258
259 postfix_expression
260     :   primary
261           (
262             (  r=OPAREN expression_list CPAREN
263             |  r=OBRACKET expr CBRACKET
264             )
265            |  r=DOT ID
266           )*
267     ;
268
269 primary
270     :   ID
271     |   INT
272     |   CHARS
273     |   TRUE
274     |   FALSE
275     |   OPAREN expr CPAREN
276     ;
277
278 // Language constructs: if, loop,...
279 if_statement
280     :   ifstat elsifstat* elsestat? END SCOLON
281     ;
282
283 ifstat
284     :   IF expression DO statement+
285     ;
286
287 elsifstat
288     :   ELSIF expression DO statement+
289     ;
290
291 elsestat
292     :   ELSE DO statement+
293     ;
294
295 loop
296     :   for_loop
297     ;
298
299 for_loop
300     :   FOR ID IN i=expression TO e=expression DO statement+ END SCOLON
301     ;
302
303 builtin_functions
304     :   debug_functions
305     ;
306
307 debug_functions
308     :   PRINTLN OPAREN expression? CPAREN
309     ;
310
311 expression_list
312     :   expression (COMMA expression)*
313     ;
```

B.2.2 Lexer

```
// LEXER
PRINTLN           : 'println';
PRINT             : 'print';
ASSERT            : 'assert';
INCLUDE           : 'include' (WS)? CHARS;
FUNCTION          : 'function';
ORDINAL           : 'ordinal';
IS                : 'is';
IF                : 'if';
DO                : 'do';
ELSIF             : 'elsif';
ELSE              : 'else';
FOR               : 'for';
IN                : 'in';
TO                : 'to';
LOOP              : 'loop';
TRUE              : 'true';
FALSE             : 'false';

// Datatypes
TYPEDEF           : 'typedef';
CONSTANT          : 'constant';
ENUM              : 'enum';
STRUCT            : 'struct';
BITSET            : 'bitset';
STRING            : 'string';
BOOLEAN           : 'boolean';

fragment MODIFIER : 'signed' | 'unsigned';
fragment BITSIZE  : '8bit' | '16bit' | '32bit' | '64bit';

INTEGER           : MODIFIER (WS)+ 'integer' (WS)+ BITSIZE;

INCOMING_BEGIN    : 'incoming';
OUTGOING_BEGIN    : 'outgoing';

BEGIN             : 'begin';
END               : 'end';

OR                : 'or';
OR_SHORT          : '||';
AND               : 'and';
AND_SHORT         : '&&';

BNOT              : '~';
BOR               : '|';
BXOR              : '^';
BAND              : '&';
LSHIFT            : '<<';
RSHIFT            : '>>';

EQUALS            : '==';
NEQUALS           : '!=';
GTEQUALS          : '>=';
LTEQUALS          : '<=';

POW               : '^^';
EXCL              : 'not';
EXCL_SHORT        : '!';
GT                : '>';
```

```
LT                  : '<';
ADD                 : '+';
SUBTRACT            : '-';
MULTIPLY            : '*';
DIVIDE              : '/';
MODULUS             : '%';
OBRACKET            : '[';
CBRACKET            : ']';
OPAREN              : '(';
CPAREN              : ')';
SCOLON              : ';';
ASSIGN              : ':=';
COMMA               : ',';
QMARK               : '?';
COLON               : ':';
RETURN              : 'return';
TMP_VAR             : 'var';
DOT                 : '.';
AUTH                : 'auth';

// Fragments
fragment DIGIT     : '0'..'9';
fragment INTNR     : DIGIT+;
fragment HEX_DIGIT : (DIGIT | 'a'..'f' | 'A'..'F');
fragment HEXNR     : ('0x'|'0X') (HEX_DIGIT)+;

// Datatypes
INT                 : ('-')? (INTNR | HEXNR);

CHARS               : '"'  (~('"' | '\\')  | '\\' .)* '"'
                    |  '\'' (~('\'' | '\\') | '\\' .)* '\'';

//IDENTIFIER
ID  :   ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;

// COMMENTS
fragment COMMENT_SIGN:      '--';
fragment COMMENT_EXCL_MARK: '!';

COMMENT_FIRST
  :     COMMENT_SIGN COMMENT_EXCL_MARK a=~('\n'|'\r')* '\r'? '\n'
  ;

COMMENT_CONSECUTIVE
  :     COMMENT_SIGN ~('\n'|'\r')* '\r'? '\n'
  ;

// WHITESPACE
WS:   ( ' ' | '\t' | '\r' | '\n' | '\u000C');
```

Appendix C

Support Files

This appendix lists supportive files used for the development of FTPM files. We list syntax and indent files for the VIM editor, which can be used as a base for other editors to provide correct syntax highlighting and indentation. The complete grammar of FTPM can be found in Appendix B.

C.1 VIM

C.1.1 VIM Indent

The following is a minimal configuration file which provides correct indentation for the creation of FTPM source code with the VIM editor.

```
" Vim indent file
" Language: Ftpm
" Author:   Martin Kapeundl <martin.kapeundl@student.tugraz.at>

" Only load this indent file when no other was loaded.
if exists("b:did_indent")
    finish
endif
let b:did_indent = 1

setlocal shiftwidth=2
setlocal tabstop=2
setlocal indentexpr=GetFtpmIndent()
setlocal autoindent
setlocal comments=:--
setlocal ignorecase

" indent for these keywords (ignore case)
setlocal indentkeys+==~ordinal,=~incoming,=~outgoing,~=begin,=~if,=~do,=~else,=~
    elsif,=~end,=~auth

" Make comments that cover multiple lines have
" start with comment sign at the beginning of each new line
setlocal fo=croq

" do not indent include directives
let s:no_indent = '^\s*\<\(include\)\>'

" Only define the function once.
if exists("*GetFtpmIndent")
    finish
```

```
endif

function! GetFtpmIndent()
    " current line number
    let lnum = v:lnum
    " At start of file use zero indent
    if lnum == 1
        return 0
    endif

    " get current and previous line
    let curline = getline(lnum)
    let prevline_num = prevnonblank(lnum-1)
    let prevline=getline(prevline_num)
    let ind=indent(prevline_num)

    " do not indent include directives
    if curline =~s:no_indent
        return 0
    endif

    " Add a shiftwidth to statements following ordinal, incoming,...
    if prevline =~? '^\s*\<\(ordinal\|incoming\|outgoing\|begin\|do\|else\|auth\)
        \>'
        let ind = ind + &sw
    endif

    " Subtract a shiftwidth from else, elsif,end
    if curline =~? '^\s*\(else\|elsif\|end\|begin\|outgoing\)\>'
        let ind = ind - &sw
    endif

    return ind
endfunction
```

C.1.2 VIM Syntax

The following file provides basic syntax highlighting for constructs written in the FTPM language with the VIM editor.

```
" Minimal VIM syntax file
" for the 'FTPM' Language
" Author: Martin Kapeundl
" Date:   August 2011

if exists("b.current_syntax")
    finish
endif

syn case ignore


syn match include1 "include \".*\""

syn region comment1 display oneline start='--' end='$'
            \ contains=commentkeywords1,@Spell

syn keyword commentkeywords1 TODO FIXXME
syn match   commentkeywords1 "!\(file | chapter\|desc\|info\|document\)"

```

```
syn keyword type1 typedef enum struct bitset string signed var
syn keyword type1 unsigned integer 8bit 16bit 32bit 64bit
syn keyword type1 boolean true false constant

syn keyword statement1 println print assert function is
syn keyword statement1 ordinal if do elsif else for in to loop
syn keyword statement1 incoming outgoing begin end return while
syn keyword statement1 not and auth
syn match statement1   "\.\(object\|secret\|required\)"

syn region string1 start='"' end='"'

hi def link comment1            Comment
hi def link commentkeywords1    Todo
hi def link type1               Type
hi def link statement1          Statement
hi def link include1            PreProc
hi def link string1             Constant
```

Appendix D

TPM commands implemented in FTPM

This appendix lists the TPM commands we demonstrated in Chapter 4, which consists of TPM_OIAP, TPM_PCRREAD and TPM_EXTEND. An overview of the purpose and functionality of these commands can be found in Chapter 4.1. We list only the TPM commands and defined helper functions but omit needed TPM structures due to the lack of space. We first list the main command source which is then followed by the helper implementations.

D.1 TPM_OIAP

```
--!file commands
--!chapter 18 Authorization Sessions :: 18.01 TPM OIAP

include "../../structures/2_basic_definitions/2_2_defines.ftpm"
include "../../structures/4_types/4_4_handles.ftpm"
include "../../structures/5_basic_structures/5_5_tpm_nonce.ftpm"

include "helper/oiap_helper_stubs.ftpm"

--!info
--\warning No informative comment set

--!desc
--\warning No descriptive comment set

ordinal TPM_OIAP

    incoming:

    outgoing:
        --! Handle that TPM creates that points to the authorization state.
        authHandle TPM_AUTHHANDLE,

        --! Nonce generated by TPM and associated with session.
        nonceEven  TPM_NONCE

begin
    --! 1. The TPM_OIAP command allows the creation of an authorization session
    --     handle and the tracking of the handle by the TPM. The TPM generates
    --     the handle and nonce.

    --! 2. The TPM has an internal limit as to the number of handles that may be
    --     open at one time, so the request for a new handle may fail if there
    --     is insufficient space available.
```

```
35
36      --! 3. Internally the TPM will do the following:
37      --    * TPM allocates space to save handle, protocol identification, both
38      --      nonces and any other information the TPM needs to manage the session.
39      --    * TPM generates authHandle and nonceEven, returns these to caller
40      if setAuthHandle(authHandle) == TPM_HANDLE_STATUS.TPM_INVALID_HANDLE
41      do
42          return TPM_RESULT.TPM_RESOURCES;
43      end;
44
45      --! 4. On each subsequent use of the OIAP session the TPM MUST generate a
46      --     new nonceEven value.
47      setNonceEven(nonceEven);
48
49      --! 5. When TPM_OIAP is wrapped in an encrypted transport session, no input
50      --     or output parameters are encrypted.
51      --\warning What are the consequences of this statement?
52      --         Is this really an **ACTION** Statement?
53      return TPM_RESULT.TPM_SUCCESS;
54  end;
```

```
1   include "../../../structures/2_basic_definitions/2_2_defines.ftpm"
2   include "../../../structures/4_types/4_4_handles.ftpm"
3   include "../../../structures/5_basic_structures/5_5_tpm_nonce.ftpm"
4
5   --! internal structures for a TPM
6   constant TPM_MAX_SESSION is 20;
7
8   --! \todo internally the TPM must keep track of the nonce values
9   curNonceEven TPM_NONCE;
10  curNonceOdd  TPM_NONCE;
11
12  --! \todo just a test stub
13  --!return @ftpm_get_random_bytes(newNonce);
14  function setAuthHandle(authHandle TPM_AUTHHANDLE)
15  begin
16      authHandle := 0x00;
17      return TPM_HANDLE_STATUS.TPM_VALID_HANDLE;
18  end;
19
20  --! \todo just a test stub
21  function setNonceEven(nonceEven TPM_NONCE)
22  begin
23    --! \todo: randomize nonce - this is an initialization vector
24    for i in 0 to TPM_NONCE_SIZE
25    do
26        nonceEven.nonce[i] := 0xA5;
27    end;
28  end;
```

D.2 TPM_PCRREAD

```
1   --!file commands
2   --!chapter 16 Integrity Collection and Reporting :: 16.02 TPM PCRRead
3
4   include "../../structures/2_basic_definitions/2_2_defines.ftpm"
5   include "../../structures/5_basic_structures/5_4_tpm_digest.ftpm"
6
```

```
include "helper/pcrread_helper_functions.ftpm"

--!info The TPM_PCRRead operation provides non-cryptographic reporting of
-- the contents of a named PCR.

--!desc The TPM_PCRRead operation returns the current contents of the
-- named register to the caller.

ordinal TPM_PCRRead

    incoming:
        --! Index of the PCR to be read
        pcrIndex  TPM_PCRINDEX

    outgoing:
        --! The current contents of the named PCR
        outDigest TPM_PCRVALUE
begin
    --! 1. Validate that pcrIndex represents a legal PCR number
    --  On Error, return TPM_BADINDEX
    if !isValidPCRIndex(pcrIndex)
    do
        return TPM_RESULT.TPM_BADINDEX;
    end;

    --! 2. Set outDigest to TPM_STCLEAR_DATA->PCR[pcrIndex]
    setPCRValue(pcrIndex, outDigest);

    --! 3. Return TPM_SUCCESS
    return TPM_RESULT.TPM_SUCCESS;
end;
```

```
include "../../../structures/2_basic_definitions/2_2_defines.ftpm"
include "../../../structures/5_basic_structures/5_4_tpm_digest.ftpm"

--! Defined in 7_4_tpm_permanent_data
constant TPM_NUM_PCR is 16;

--!TODO autogenerated stub
function isValidPCRIndex(idx TPM_PCRINDEX)
begin
    return idx <= TPM_NUM_PCR;
end;

--!TODO autogenerated stub
function setPCRValue(pcrIndex  TPM_PCRINDEX, outDigest TPM_PCRVALUE)
begin
    --! compliance vector
    outDigest.digest[0]  := 0x15; outDigest.digest[1]  := 0x8f;
    outDigest.digest[2]  := 0x1d; outDigest.digest[3]  := 0x6a;
    outDigest.digest[4]  := 0x35; outDigest.digest[5]  := 0x8f;
    outDigest.digest[6]  := 0x50; outDigest.digest[7]  := 0x51;
    outDigest.digest[8]  := 0x2a; outDigest.digest[9]  := 0x81;
    outDigest.digest[10] := 0x08; outDigest.digest[11] := 0xcf;
    outDigest.digest[12] := 0xe6; outDigest.digest[13] := 0xec;
    outDigest.digest[14] := 0xd0; outDigest.digest[15] := 0xf9;
    outDigest.digest[16] := 0x07; outDigest.digest[17] := 0xc5;
    outDigest.digest[18] := 0xc6; outDigest.digest[19] := 0x7c;
end;
```

D.3 TPM_EXTEND

```
--!file commands
--!chapter 16 Integrity Collection and Reporting :: 16.01 TPM Extend

include "../../structures/2_basic_definitions/2_2_defines.ftpm"
include "../../structures/5_basic_structures/5_4_tpm_digest.ftpm"
include "../../structures/7_internal_data_held_by_tpm/7_3_tpm_stany_flags.ftpm"

include "helper/extend_helper_functions.ftpm"

--!info This adds a new measurement to a PCR

--!desc Add a measurement value to a PCR


ordinal TPM_Extend

    incoming:
        --! The PCR to be updated
        pcrNum  TPM_PCRINDEX,

        --! The 160 bit value representing the event to be recorded
        inDigest TPM_DIGEST

    outgoing:
        --! The PCR value after execution of the command.
        outDigest TPM_PCRVALUE

begin
     --! 1. Validate that pcrNum represents a legal PCR number.
    --     On error, return TPM_BADINDEX.
    if pcrNum > TPM_NUM_PCR
    do
        return TPM_RESULT.TPM_BADINDEX;
    end;

    --! 2. Map L1 to TPM_STANY_FLAGS -> localityModifier
    var L1 := getLocalityModifier();

    --! 3. Map P1 to TPM_PERMANENT_DATA -> pcrAttrib [pcrNum]. pcrExtendLocal
    var P1 := getExtendLocal(pcrNum);

    --! 4. If, for the value of L1, the corresponding bit is not set in the bit
        map P1, return
    --     TPM_BAD_LOCALITY
    --@todo: not implemented
    if not isBitSet(L1, P1)
    do
        return TPM_RESULT.TPM_BAD_LOCALITY;
    end;

    --! 5. Create c1 by concatenating (TPM_STCLEAR_DATA -> PCR[pcrNum] || inDigest
        ).
    --     This takes the current PCR value and concatenates the inDigest
        parameter.
    h1 TPM_DIGEST;
    createAndStoreDigest(h1, pcrNum, inDigest);

    --! 6. Create h1 by performing a SHA-1 digest of c1.
    --! 7. Store h1 to TPM_STCLEAR_DATA -> PCR[pcrNum]
```

```
57
58
59      --! 8. If TPM_PERMANENT_FLAGS -> disable is TRUE or
60      --     TPM_STCLEAR_FLAGS -> deactivated  is TRUE
61      --       * Set outDigest to 20 bytes of 0x00
62      if isTPMPermanentFlagsDisabled() or isTPMStClearFlagsDisabled()
63      do
64          setOutDigest0(outDigest);
65      --!    Else
66      --       * Set outDigest to h1
67      else
68      do
69          setOutDigest(outDigest, h1);
70      end;
71
72      return TPM_RESULT.TPM_SUCCESS;
73  end;
```

```
1   include "../../../structures/2_basic_definitions/2_2_defines.ftpm"
2   include "../../../structures/5_basic_structures/5_4_tpm_digest.ftpm"
3   include "../../../structures/7_internal_data_held_by_tpm/7_3_tpm_stany_flags.ftpm"
4   include "../../../structures/7_internal_data_held_by_tpm/7_4_tpm_permanent_data.
        ftpm"
5   include "../../../structures/7_internal_data_held_by_tpm/7_5_tpm_stclear_data.ftpm
        "
6   include "../../../structures/8_pcr_structures/8_8_tpm_pcr_attributes.ftpm"
7
8   --! internal data held by tpm
9   my_tpm_stany_flags     TPM_STANY_FLAGS;
10  my_tpm_permanent_data  TPM_PERMANENT_DATA;
11  my_tpm_stclear_data    TPM_STCLEAR_DATA;
12
13  --!TODO autogenerated stub
14  function getLocalityModifier()
15  begin
16      return my_tpm_stany_flags.localityModifier;
17  end;
18
19  --!TODO autogenerated stub
20  function getExtendLocal(pcrNum TPM_PCRINDEX)
21  begin
22      return my_tpm_permanent_data.pcrAttrib[pcrNum].pcrExtendLocal;
23  end;
24
25  --!TODO autogenerated stub
26  function isBitSet(L1 UINT32, P1 BYTE)
27  begin
28      return true;
29  end;
30
31  --!TODO autogenerated stub
32  function createAndStoreDigest(h1 TPM_DIGEST, pcrNum TPM_PCRINDEX, inDigest
        TPM_DIGEST)
33  begin
34      h1 := my_tpm_stclear_data.PCR[pcrNum];
35  end;
36
37  --!TODO autogenerated stub
38  function isTPMPermanentFlagsDisabled()
39  begin
40      return true;
41  end;
```

```
42
43 --!TODO autogenerated stub
44 function isTPMStClearFlagsDisabled()
45 begin
46     return true;
47 end;
48
49 --!TODO autogenerated stub
50 function setOutDigest0(outDigest TPM_DIGEST)
51 begin
52     --! compliance vector
53     outDigest.digest[0]  := 0x00; outDigest.digest[1]  := 0x00;
54     outDigest.digest[2]  := 0x00; outDigest.digest[3]  := 0x00;
55     outDigest.digest[4]  := 0x00; outDigest.digest[5]  := 0x00;
56     outDigest.digest[6]  := 0x00; outDigest.digest[7]  := 0x00;
57     outDigest.digest[8]  := 0x00; outDigest.digest[9]  := 0x00;
58     outDigest.digest[10] := 0x00; outDigest.digest[11] := 0x00;
59     outDigest.digest[12] := 0x00; outDigest.digest[13] := 0x00;
60     outDigest.digest[14] := 0x00; outDigest.digest[15] := 0x00;
61     outDigest.digest[16] := 0x00; outDigest.digest[17] := 0x00;
62     outDigest.digest[18] := 0x00; outDigest.digest[19] := 0x00;
63 end;
64
65 --!TODO autogenerated stub
66 function setOutDigest(outDigest TPM_DIGEST, h1 TPM_DIGEST)
67 begin
68     --! compliance vector
69     outDigest.digest[0]  := h1.digest[0];  outDigest.digest[1]  := h1.digest[1];
70     outDigest.digest[2]  := h1.digest[2];  outDigest.digest[3]  := h1.digest[3];
71     outDigest.digest[4]  := h1.digest[4];  outDigest.digest[5]  := h1.digest[5];
72     outDigest.digest[6]  := h1.digest[6];  outDigest.digest[7]  := h1.digest[7];
73     outDigest.digest[8]  := h1.digest[8];  outDigest.digest[9]  := h1.digest[9];
74     outDigest.digest[10] := h1.digest[10]; outDigest.digest[11] := h1.digest[11];
75     outDigest.digest[12] := h1.digest[12]; outDigest.digest[13] := h1.digest[13];
76     outDigest.digest[14] := h1.digest[14]; outDigest.digest[15] := h1.digest[15];
77     outDigest.digest[16] := h1.digest[16]; outDigest.digest[17] := h1.digest[17];
78     outDigest.digest[18] := h1.digest[18]; outDigest.digest[19] := h1.digest[19];
79 end;
```

Bibliography

[1] Adams, C. and S. Lloyd [2003]. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Technology series, Sams. ISBN 9780672323911. http://books.google.at/books?id=ERSfUmmthMYC. (Cited on page 9.)

[2] Aichernig, Bernhard K. and Peter Lucas [1999]. *Formale Methoden in der Praxis*. In *Unterlagen zum 1. Österreichischen ISA-EUNET Workshop, Wien*. Österreichische Computer Gesellschaft, AK Software Qualität und Verläszlichkeit. (Cited on page 23.)

[3] Andrews, Keith [2011]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. http://ftp.iicm.edu/pub/keith/thesis/. (Cited on page ix.)

[4] Binstock, Andrew [2008]. Interview with Donald Knuth. http://www.informit.com/articles/article.aspx?p=1193856. (Cited on page 27.)

[5] Bjørner, Dines [Hrsg.] [1990]. *VDM and Z - formal methods in software development*. 3. International Symposium of VDM Europe. (Cited on page 25.)

[6] Bradner, S. [1997]. *Key words for use in RFCs to Indicate Requirement Levels*. http://tools.ietf.org/html/rfc2119. (Cited on page 46.)

[7] Brickell, Ernie, Jan Camenisch, and Liqun Chen [2004]. *Direct anonymous attestation*. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145. CCS '04, ACM, New York, NY, USA. ISBN 1581139616. doi:10.1145/1030083.1030103. http://doi.acm.org/10.1145/1030083.1030103. (Cited on page 16.)

[8] BSI [2012]. *British Standards Institution, BS ISO/IEC 13817-1:1996*. http://standardsdevelopment.bsigroup.com. (Cited on page 25.)

[9] Cannière, Christophe and Christian Rechberger [2006]. *Finding SHA-1 Characteristics: General Results and Applications*. In Lai, Xuejia and Kefei Chen (Editors), *Advances in Cryptology - ASIACRYPT 2006*, *Lecture Notes in Computer Science*, volume 4284, pages 1–20. Springer Berlin Heidelberg. ISBN 9783540494751. doi:10.1007/11935230_1. http://dx.doi.org/10.1007/11935230_1. (Cited on page 14.)

[10] Chang, Shu-jen, Ray Perlner, William E. Burr, Meltem Sönmez Turan, John M. Kelsey, Souradyuti Paul, and Lawrence E. Bassham [2012]. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf. (Cited on page 13.)

[11] Chaos Computer Club [2010]. *Console Hacking 2010*. Slides from the 27th Chaos Communication Congress, Accessed: 2013-04-15. (Cited on page 71.)

[12] Corbi, T. A. [1989]. *Program understanding: Challenge for the 1990s*. *IBM Systems Journal*, 28(2), pages 294 –306. ISSN 0018-8670. doi:10.1147/sj.282.0294. (Cited on page 72.)

[13] Davies, Donald [1997]. *A brief history of cryptography*. *Information Security Technical Report*, 2(2), pages 14 – 17. ISSN 1363-4127. doi:10.1016/S1363-4127(97)81323-4. http://www.sciencedirect.com/science/article/pii/S1363412797813234. (Cited on page 7.)

[14] Deursen, A. van, P. Klint, and J. Visser [2000]. *Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices*, 35(6), pages 26–36. http://www.st.ewi.tudelft.nl/~arie/papers/. (Cited on page 32.)

[15] Ellison, C. and B. Schneier [2000]. *Ten risks of PKI: What you're not being told about public key infrastructure*. *Comput Secur J*, 16(1), pages 1–7. (Cited on page 9.)

[16] Ellison, Carl [2007]. *Cryptographic Random Numbers*. *Draft P1363 Appendix E. http://www. clark. net/pub/cme P*, 1363. (Cited on page 11.)

[17] ESA [2012]. *ESA software engineering standards, Issue 2*. http://cisas.unipd.it/didactics/STS_school/Software_development/ESA_SW_engineering_standard_Issue2-050.pdf. (Cited on page 23.)

[18] Finney, K. [1996]. *Mathematical notation in formal specification: too difficult for the masses? Software Engineering, IEEE Transactions on*, 22(2), pages 158–159. ISSN 0098-5589. doi:10.1109/32.485225. (Cited on page 26.)

[19] Foundaton, The Eclipse [2012]. *Eclipse*. http://www.eclipse.org. (Cited on page ix.)

[20] Fowler, Martin [2005]. *Language Workbenches: The Killer-App for Domain Specific Languages?* online. http://martinfowler.com/articles/languageWorkbench.html. (Cited on pages 32, 33 and 34.)

[21] Gong, L. [1993]. *Variations on the themes of message freshness and replay-or the difficulty in devising formal methods to analyze cryptographic protocols*. In *Computer Security Foundations Workshop VI, 1993. Proceedings*, pages 131 –136. doi:10.1109/CSFW.1993.246633. (Cited on page 20.)

[22] Grawrock, David [2009]. *Dynamics of a Trusted Platform: A Building Block Approach*. 1st Edition. Intel Press. ISBN 1934053171, 9781934053171. (Cited on page 17.)

[23] Gürgens, Sigrid, Carsten Rudolph, Dirk Scheuermann, Marion Atts, and Rainer Plaga [2007]. *Security evaluation of scenarios based on the TCG's TPM specification*. In *In Proceedings of the European Symposium on Research in Computer (ESORICS*. doi:10.1007/9783540748359_29. (Cited on page 1.)

[24] Gyorfi, T., O. Cret, and A. Suciu [2009]. *High performance true random number generator based on FPGA block RAMs*. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. ISSN 1530-2075. doi:10.1109/IPDPS.2009.5161207. (Cited on page 11.)

[25] Hayes, Ian [1992]. *VDM and Z: A comparative case study*. *Formal Aspects of Computing*, 4, pages 76–99. ISSN 0934-5043. http://dx.doi.org/10.1007/BF01214957. 10.1007/BF01214957. (Cited on page 25.)

[26] Hudak, P. [1998]. *Modular domain specific languages and tools*. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134 –142. ISSN 1085-9098. doi:10.1109/ICSR1998685738. (Cited on page 32.)

[27] ISO [1996]. *International Organization for Standardization, Information technology – Syntactic metalanguage – Extended BNF*. http://www.iso.org. (Cited on pages 55 and 73.)

[28] ISO [2012]. *International Organization for Standardization, ISO/IEC 13568:2002*. http://www.iso.org. (Cited on page 25.)

[29] ISO [2012]. *International Organization for Standardization, ISO/IEC 13817-1:1996 ISO*. http://www.iso.org. (Cited on page 25.)

[30] Jones, Cliff B [1990]. *Systematic software development using VDM, 2nd Edition*. Prentice Hall International (UK) Ltd., The University, Manchester, England. (Cited on page 25.)

[31] Jun, Benjamin and Paul Kocher [1999]. *The Intel random number generator. Cryptography Research Inc. white paper*. (Cited on page 11.)

[32] Knuth, Donal E. [1983]. *Literate programming*. *The Computer Journal*, pages 97–111. (Cited on page 26.)

[33] Kömmerling, Oliver, Kuhn, and Markus G. [1999]. *Design principles for tamper-resistant smartcard processors*. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pages 2–2. WOST'99, USENIX Association, Berkeley, CA, USA. http://www.cl.cam.ac.uk/~mgk25/sc99-tamper.pdf. (Cited on page 71.)

[34] Krasner, G.E., S.T. Pope, et al. [1988]. *A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of object oriented programming*, 1(3), pages 26–49. (Cited on page 82.)

[35] Lamsweerde, Axel van [2000]. *Formal specification: a roadmap*. In *Proceedings of the Conference on The Future of Software Engineering*, pages 147–159. ICSE '00, ACM, New York, NY, USA. ISBN 1581132530. doi:10.1145/336512.336546. http://doi.acm.org/10.1145/336512.336546. (Cited on pages 23 and 24.)

[36] Needham, Roger M. and Michael D. Schroeder [1978]. *Using encryption for authentication in large networks of computers*. *Commun. ACM*, 21(12), pages 993–999. ISSN 0001-0782. doi:10.1145/359657.359659. http://doi.acm.org/10.1145/359657.359659. (Cited on pages 11 and 20.)

[37] Noren, Allen [2009]. Interview with Brian Kernighan. http://broadcast.oreilly.com/2009/04/an-interview-with-brian-kernig.html. (Cited on page 32.)

[38] Nørmark, Kurt [2010]. *Overview of the four main programming paradigms*. http://people.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html. Retrieved 30. September 2012. (Cited on page 24.)

[39] Parr, Terence J. [2004]. *Enforcing strict model-view separation in template engines*. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. WWW '04, ACM, New York, NY, USA. ISBN 158113844X. doi:10.1145/988672.988703. http://doi.acm.org/10.1145/988672.988703. (Cited on page 82.)

[40] Parr, Terence J. [2007]. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf. ISBN 0978739256. (Cited on page 73.)

[41] Parr, Terence J. [2009]. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st Edition. Pragmatic Bookshelf. ISBN 193435645X, 9781934356456. (Cited on page 78.)

[42] Parr, Terence J. and Russell W. Quong [1995]. *ANTLR: A predicated-LL(k) parser generator*. *Software: Practice and Experience*, 25(7), pages 789–810. ISSN 1097-024X. doi:10.1002/spe.4380250705. http://dx.doi.org/10.1002/spe.4380250705. (Cited on page 73.)

[43] Rademacher, Gunther [2012]. *Railroad Diagram Generator*. http://railroad.my28msec.com/rr/ui. (Cited on page ix.)

[44] Raymond, Eric S. [2003]. *The Art of UNIX Programming*. First Edition. Addison-Wesley. ISBN 9780131429017. http://www.catb.org/esr/writings/taoup/html. (Cited on page 32.)

[45] Reenskaug, T. M. H. [2007]. *The original MVC reports*. https://www.duo.uio.no/handle/123456789/9621. (Cited on page 82.)

[46] Rivest, R. L., A. Shamir, and L. Adleman [1978]. *A method for obtaining digital signatures and public-key cryptosystems*. *Commun. ACM*, 21(2), pages 120–126. ISSN 0001-0782. doi:10.1145/359340.359342. http://doi.acm.org/10.1145/359340.359342. (Cited on pages 7 and 13.)

[47] RSA Laboratories [2013]. *How large a key should be used in the RSA cryptosystem?* http://www.rsa.com/rsalabs/node.asp?id=2218. (Cited on page 13.)

[48] Schneier, Bruce [1995]. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0471117099. (Cited on pages 7, 9, 11, 12 and 13.)

[49] Sprinkle, Jonathan, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis [2009]. *What Kinds of Nails Need a Domain-Specific Hammer?* *IEEE Software*, 26(4), pages 15–18. ISSN 0740-7459. doi:10.1109/MS200992. http://www.dmst.aueb.gr/dds/pubs/jrnl/2009-IEEESW-DSLM/html/SMTS09.htm. Guest Editors' Introduction: Domain Specific Modelling. (Cited on pages 33 and 91.)

[50] Stallings, William [1999]. *Cryptography and network security (2nd ed.): principles and practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0138690170. (Cited on page 11.)

[51] Standards, Federal Information Processing [2012]. *Secure Hash Standard*. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. (Cited on pages 7 and 13.)

[52] Standish, Thomas A. [1984]. *An Essay on Software Reuse*. *Software Engineering, IEEE Transactions on*, SE-10(5), pages 494 –497. ISSN 0098-5589. doi:10.1109/TSE.1984.5010272. (Cited on page 72.)

[53] Stevens, Marc [2013]. *New collision attacks on SHA-1 based on optimal joint local-collision analysis*. (Cited on page 14.)

[54] Sunar, B., W.J. Martin, and D.R. Stinson [2007]. *A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks*. *Computers, IEEE Transactions on*, 56(1), pages 109 –119. ISSN 0018-9340. doi:10.1109/TC.2007.250627. (Cited on page 11.)

[55] TCG [2007]. *TCG Architecture Overview, Version 1.4*. http://www.trustedcomputinggroup.org/resources/. (Cited on page 15.)

[56] TCG [2011]. *TPM Specification Version 1.2, Part 1, Design Principles*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on pages 1, 20, 26, 31, 39, 52 and 72.)

[57] TCG [2011]. *TPM Specification Version 1.2, Part 2, Structures of the TPM*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on pages 1, 19, 26, 31, 38, 39, 46, 72 and 80.)

[58] TCG [2011]. *TPM Specification Version 1.2, Part 3, Commands*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on pages 1, 19, 26, 31, 39 and 72.)

[59] TCG [2011]. *Trusted Computing Group*. http://www.trustedcomputinggroup.org. (Cited on pages 1 and 15.)

[60] TCG [2012]. http://www.trustedcomputinggroup.org/community/2011/03/do_you_know_a_few_notes_on_trusted_computing_out_in_the_world. (Cited on page 1.)

[61] TCG v2 [2012]. *TPM Module Library Version 2, Part 1, Architecture*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on page 87.)

[62] TCG v2 [2012]. *TPM Module Library Version 2, Part 2, Structures*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on page 87.)

[63] TCG v2 [2012]. *TPM Module Library Version 2, Part 3, Commands*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on page 87.)

[64] TCG v2 [2012]. *TPM Module Library Version 2, Part 4, Supporting Routines*. http://www.trustedcomputinggroup.org/developers/developer_resources. (Cited on page 87.)

[65] The Oxygen Icon Team [2012]. *Oxygen Icon Theme*. http://www.oxygen-icons.org. (Cited on page ix.)

[66] Wang, Xiaoyun, YiqunLisa Yin, and Hongbo Yu [2005]. *Finding Collisions in the Full SHA-1*. In Shoup, Victor (Editor), *Advances in Cryptology - CRYPTO 2005*, *Lecture Notes in Computer Science*, volume 3621, pages 17–36. Springer Berlin Heidelberg. ISBN 9783540281146. doi:10.1007/11535218_2. http://dx.doi.org/10.1007/11535218_2. (Cited on page 14.)

[67] Young, William D. [2012]. *Foundations of Computer Security, Symmetric vs. Asymmetric Encryption*. "http://www.cs.utexas.edu/~byoung". Retrieved 07. January 2013. (Cited on page 9.)