Design and Implementation of a Trusted RFID Reader

Master of Science Thesis
January 12th 2007

Examination Committee
Dr. ir. G. Karagiannis[1]
Andrea Soppera[2]
Dr. ir. A. Pras[1]

Author
S.F.G. Verdonkschot

[1] Chair Design and Analysis of Communication Systems, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands

[2] Networks Research Centre, Chief Technology Office, British Telecom, Adastral Park, Martlesham Heath, United Kingdom.

Abstract

Radio Frequency Identification (RFID) technology is an emerging technology that allows for the electronic tagging and wireless identification of objects using RFID tags. Its application offers great potential for optimizing business processes by improving efficiency and by possible attractive cost savings. But its deployment also raises significant consumer privacy issues. RFID tags may reveal private consumer data without the subject's knowledge or consent. The challenge is to provide privacy protection without raising tag production and management cost.

A new architecture for a trustworthy RFID reader has been proposed that can make RFID systems more privacy friendly. The RFID reader will contain a Trusted Platform Module (TPM). The TPM is a tamper-resistant hardware module designed to provide robust security capabilities like remote attestation and sealed storage. These capabilities combined with a newly designed reader software architecture can provide privacy policy compliant readers. The software architecture consists of a policy engine and an auditing agent for respectively enforcing and auditing the privacy policy of the trusted reader. The objective is to develop the proposed RFID architecture and to build an experimental implementation which will demonstrate its potential use within a specific business case scenario.

Acknowledgements

I would like to thank Andrea Soppera and Trevor Burbridge for granting me the opportunity to work with them on this assignment. My gratefulness also goes to all other colleagues at the Networks Research Centre. My stay at BT has been a very interesting and educational experience.

I also want to thank Georgios Karagiannis for guiding me with this thesis every step of the way and for being such a patient man.

I am also grateful to Lut Baten and Marja Verdonkschot for their assistance with the organizational and lingual aspects of the report.

I would like to thank my parents for providing me with all the opportunities one could only wish for.

Last but not least I want to thank my girlfriend Anke for always being there when things got tough and for putting up with my absence. I owe you big time!

Table of contents

Table of figures

Abbreviations

AIK	Attestation Identity Key
API	Application Programming Interface
BBB	Bios Boot Block
BIOS	Basic Input Output System
BT	British Telecom
CA	Certificate Authority
CTO	Chief Technology Office
EPC	Electronic Product Code
FSM	Finite State Machine
GRUB	Grand Unified Boot loader
GUI	Graphical User Interface
IP	Internet Protocol
IPL	Initial Program Loader
ITU	International Telecommunication Union
JNI	Java Native Interface
MAC	Media Access Control / Mandatory Access Control
MBR	Master Boot Record
NRC	Networks Research Centre
PC	Personal Computer
PCR	Platform Configuration Register
PDU	Protocol Data Unit
ROM	Read Only Memory
RFID	Radio Frequency Identification
SHA-1	Secure Hash Standard version 1
SDL	Specification and Description Language
SSH	Secure Shell
TCFL	Trusted Client for Linux
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TPM	Trusted Computing Group Platform Module
TSS	Trusted Computing Group Software Stack
UDP	User Datagram Protocol
UPC	Universal Product Code

Chapter 1: Introduction

1.1 Radio Frequency Identification

Radio frequency Identification (RFID) is in the broadest sense a technology that allows unique identification of objects by the use of radio signals [1]. If we use this general definition for RFID we can see that many RFID systems are already in use today, for example in access control cards for admission to buildings and cars, payment systems on toll roads and gas stations, the tracking of library books and with skiers using ski lifts.

While RFID is perceived by many as a newly developed technology, its roots can be traced back to as early as the 1940s [2]. During World War II the British Royal Air Force designed a system to identify their airplanes by the use of onboard transponders. These transponders allowed them to identify incoming airplanes as allies or enemies. When the onboard transponders received signals from radar stations on the ground, coded identification signals were sent back to these radar stations.

RFID works according to this same basic concept. A signal is sent by a reader to a transponder or tag, which wakes up and either reflects back a signal (passive system) or broadcasts a signal (active system) using its own power source, for example an onboard battery, back to the reader.

A passive RFID tag has the advantage of being smaller and having a longer lifespan as it does not contain a battery. The active tag on the other hand, has an increased range and offers increased reading reliability. The most important factor that separates these two types of tags is the cost. Active tags are at least a factor 10 more expensive than passive tags. Passive tags are now available in large quantities for 10 cents a piece.

The signal that an RFID tag transmits is usually a serial number of 64 or 96 bits. This serial number can be set by the manufacturer of the tag or be programmed by the end user depending on the type of tag that is used. Some tags only allow for one serial number to be written to the tag, which will then remain the same for the rest of its lifespan. Other tags allow unlimited updates of the serial number. For a complete and concise treatment of the technical aspects of RFID we refer the reader to [3].

EPCglobal [4] is a consortium that has defined standards for RFID tags and devices. It has also defined the Electronic Product Code (EPC) [4] which prescribes how the serial number of an EPC RFID tag should be composed and interpreted. The EPC code can be seen as an extension of the Universal Product Code (UPC) which is used by the well known bar code that has been in use in retail for product type identification since the 1970s. The EPC code improves on the UPC code by specifying not only a product type but also a manufacturer identifier and a unique serial number, allowing for unique identification of every single item.

Of course RFID is much more than just an advanced version of the bar code with increased scanning range. It is envisioned that RFID will have a big impact on consumers and businesses in all kinds of areas. Many possibilities still need to be conceived. Some possibilities are real time inventory, anti-counterfeiting protection of medical drugs and clothing, sensor equipped RFID tags that measure environmental

conditions, etc… The possibilities are endless. Unfortunately this does not come without a price.

1.2 Privacy Issues with RFID

RFID is often described as a new technology that allows machines to perceive. It allows for easy and accurate information gathering about the physical world. Is it possible that machines will perceive too much with RFID?

In the world we live in today we already have a difficult time keeping our personal information private. We have license plates on our cars, pay with credit cards and have loyalty cards for our favorite shops and supermarkets. But ultimately we can still control the amount of information we give out. We can use cash instead of bank cards, choose not to use loyalty cards, etc… With RFID this becomes a whole other story.

An RFID system with cheap, unprotected, passive tags poses some interesting problems. The passive tags respond to every RFID reader, not exclusively to the RFID readers intended. This system basically invites one to buy a reader and start reading everybody's tags. It should not be hard to read somebody's tags without even being noticed. RFID tags can have a considerable range up to a few meters, and the reader does not even have to be in line-of-sight. The tags themselves give no indication when they are being read. What if the customer is not even aware that RFID tags are present? It is practically impossible to protect privacy with such a system.

The second issue is that the tags will probably contain unique identifiers if they adhere to the EPC standard. These unique identifiers could be mapped onto specific objects or products, basically allowing one to read what items a person is carrying. Also profiles could be created of these "readable" individuals, which can be valuable information for companies. Even if the identifiers cannot be mapped onto specific items, they can still be used to track the individual physically.

It should be clear that RFID raises some serious issues with regards to information privacy. These issues should be addressed and solutions need to be devised before adoption of RFID on a wide scale can begin. It is the purpose of this thesis to present a possible privacy protection scheme which can alleviate some of the privacy issues that have been raised.

1.3 Thesis goal and research questions

In section 1.2 we have discussed some of the privacy issues that can arise when using unprotected RFID systems. A number of schemes have been developed in an attempt to combat these issues. An overview of these privacy protecting schemes is provided in section 2.1.

A new privacy protection scheme has been devised by BT's Networks Research Centre in corporation with the University of Berkeley. With this newly devised scheme, the privacy protection is provided by a specially designed "Trusted RFID Reader" as discussed in section 2.2 and 2.3. It is the goal of this thesis to investigate whether or not the concept of a trusted RFID reader can be turned into a working prototype that operates according to the set requirements.

To obtain a valid implementation of the trusted RFID reader, a number of research questions will need to be answered in this thesis:

1. What are the requirements for the trusted RFID reader?
2. How will the trusted RFID reader be designed, based on these requirements?
3. How is the design of the trusted RFID reader implemented?
4. Can it be shown that the implementation of the design indeed conforms to the set requirements?

1.4 Thesis structure

Chapter 2 renders an overview of the privacy protection schemes that are available for RFID and introduces the concept of the privacy protection scheme offered by the trusted RFID reader

Chapter 3 discusses the notion of trusted computing and the capabilities that it provides. The capabilities that are used by the trusted reader are looked at in more detail.

Chapter 4 shows the design of the trusted reader at different levels of abstraction using functional building block diagrams and finite state machine descriptions.

Chapter 5 shows how the design is implemented using UML class diagrams, sequence diagrams, API descriptions and pseudo code.

Chapter 6 provides evaluation results of test cases and experiments to show the correct operation of the trusted reader prototype.

Chapter 7 presents some conclusions and ideas for future work.

Chapter 2: Radio Frequency Identification

2.1 Proposed privacy protecting solutions

A number of privacy protection technologies have been devised by the RFID research community to address the privacy and security threats. We will provide a short overview of these solutions, followed by a description of the concept of a trusted reader as a possible privacy protecting solution for RFID.

2.1.1 The kill command

An easy method for safeguarding RFID tag information is to incorporate a kill command in the RFID tag. When the kill command is executed, the tag will become permanently disabled. As dead tags don't talk, this prevents any further dissemination of the RFID tag information. It could be imagined that the tags are killed at the point-of sale, e.g. at the register in a clothing store. The kill command needs to be protected by a password to prevent unauthorized parties from executing this command. If this is not well protected, theft is facilitated immensely. The kill command is a mandatory command for all class-0 and class-1 RFID tags that adhere to the specifications and standards produced by EPCglobal. A description of the kill command can be found in [5] and [6].

Of course the kill command is a very drastic measure and which entails some disadvantages. If the tag is disabled at the point-of-sale, the RFID tag cannot be used for post sale applications. For example washing machines that screen the clothes for the correct wash settings or fridges alerting us that a particular product is no longer fresh. The kill command also prevents the use of the RFID chip for product warranty and return or automatic recycling.

There are also situations where the kill command is not usable at all. E.g. library books or videos of a video rental shop. In this case the tag information needs to be maintained after the customer takes the items from the library or store.

2.1.2 Read passwords

Another possibility is to construct a tag that only replies to readers that present a reading password. Only if the password is correct, will the tag respond with the tag identifier. The problem with this solution is that the reading password itself is sent unencrypted from the reader to the tag, leaving the reading password vulnerable to eavesdropping readers. Once the reading password is obtained, the tag identifier can be queried like a normal RFID tag. The class-1 generation-2 tags implement the read password functionality. It is not required to use the read password functionality. It can be bypassed by setting the read password to zero. The read password is also referred to as the access password. More information can be found in [6].

2.1.3 Pseudonym scheme

The pseudonym scheme [7] is a simple system that works by giving a tag multiple tag identifiers or pseudonyms. The tag would then cycle through them each time it is read. This will make unauthorized checking more difficult as the adversary will not know which pseudonyms belong to the same tag. The tag's owner would have a list of all the pseudonyms so that the tag can be identified, whatever pseudonym is returned to the reader. This approach will also require more complex RFID tags than the

standard passive tags now available. These more complex tags will come at a high(er) price.

2.1.4 Blocker tag

The blocker tag [8] is a simple RFID device, similar to a normal passive RFID tag. The tag does however perform a different function from that of normal RFID tags. The blocker tag basically prevents the reading of certain tags by readers by tricking the RFID reader into thinking that billions of tags are present within its operating field. It does this by abusing the anti-collision protocol that is used in the EPCglobal protocol specification. It fakes the presence of RFID tags with all possible combinations of the tag identifier. If we assume that the identifier is 96 bits long, this means that 2 to the power of 96 fake tags are detected. This overwhelming number stalls the RFID reader and makes detection of the real RFID tags impossible.

It should be noted that the authors of the blocker tag paper make a distinction between public and private tags. The blocker tag would then only be active for private tags and not for public tags. A possible scenario is that product tags from a supermarket are marked private at the point-of-sale. If a blocker tag is then incorporated in the shopping bag, the private tags would be illegible to malicious readers that the consumer encounters on the way home. When the shopping bags are removed at home, the RFID tags are usable again for post-sale applications.

2.2 A different approach

The British Telecom Networks Research Centre has done a lot of research in the area of pervasive computing and in particular RFID. The centre published a paper in collaboration with the University of Berkeley about combining an RFID architecture with trusted computing equipped RFID readers [9]. This combination offers a different approach to protecting privacy compared to current solutions available. A lot of research is going on in the area of RFID privacy because it is believed to be the main obstacle preventing wide adoption of RFID.

If we look back at section 2.1 where we briefly looked at the proposed privacy protection solutions, we can see that they are all based on the same principle of protecting the RFID tag identifier by either preventing the actual read operation by killing the tag or using reading passwords or by obfuscating the actual detected tag identifiers. What all these solutions have in common is that they all operate by adding security measures on the tag itself usually requiring the use of semi-active or active tags. These tags have to be used instead of the cheaper passive tags because they have enough computational power to handle the additional security measures while the passive tags do not.

The trusted reader concept is different in the sense that the security is put on the side of the reader and not in the tag itself. This allows the (re)use of the much cheaper passive tags which represent the biggest share of RFID tags shipped to this date. This is a significant way to keep costs down when implementing a privacy protecting RFID architecture in a business considering that active tags can cost ten times the price of a passive tag. The use of passive tags also facilitates the transition from legacy RFID architectures to the proposed privacy protecting architecture should they already be in place. Note that also a combination of privacy protection schemes is possible to increase the security and privacy even further. E.g. a trusted reader combined with the pseudonym scheme.

2.3 The concept of a trusted reader

The main idea is to create a privacy friendly RFID environment by using RFID readers that have a user-definable privacy policy. This policy instructs the reader which tags can be read and which tags remain private. In a store example, the policy for a checkout RFID reader could be set to only read RFID tags of products that are sold in the shop and thus preventing the identification of earlier purchased products e.g. from another store in possession of the consumer.

Just the presence of these readers is not enough to guarantee that privacy is better safeguarded. There is need for a separate entity that is able to verify that all readers are in fact performing according to a predefined and a mutually agreed upon policy. It could be imagined to have a consumer organization that performs these kinds of audits as a service to companies. A company like BT could also perform this role. There is an incentive for businesses to engage in such a scheme as it directly increases the confidence of the customers that their privacy is not violated and thus adds value to the business.

This is the main idea conveyed in the earlier mentioned paper: “Privacy for RFID through trusted computing” by Molnar, Soppera and Wagner [9] and is the starting point for this thesis. The paper also provides us with a set of requirements and high-level designs of functional building blocks of the trusted reader which will be discussed in chapter 4.

Chapter 3: Trusted computing

3.1 The quest for security

The proliferation of always-on broadband internet access, the installation of publicly accessible Wi-Fi hotspots and the provisioning of 2.5 and 3G cellular data services on one hand and the proliferation of portable computers and devices like laptops, PDAs and smartphones on the other has created a situation in which hundreds of millions of devices are permanently interconnected to a (high speed) global network. This environment is an inviting playing ground for hackers, extortionists and thieves. All connected hosts and devices are begging to have their security scrutinized and possibly broken. Sensitive information can be retrieved or the infiltrated host can be used for the attack on other systems. This situation has significantly changed the requirements of the security of these connected systems and devices.

The apparition of destructive worms and viruses, the execution of denial-of-service attacks through botnets and the number of reported cases of identity theft has already proven that current systems are lacking in this regard. Client applications like virus and malware scanners and network devices like firewalls are able to avert some of these problems but cannot offer a complete solution. The Trusted Computing Group was formed to address these issues. It's interesting to note that security is getting more and more attention from manufacturers and consumers. Recently a new security feature has been implemented in CPUs by AMD and Intel that can complement the security features offered by trusted computing. AMD came up with the idea of the NX bit, short for No eXecute bit, which partitions the memory space into data and code segments. The separation into these segments largely prevents buffer overflow attacks, a popular way to breach system security.

3.2 Trusted Computing Group

The Trusted Computing Group [TCG] is a non-profit consortium that includes some of the major players in the hardware and software industry including AMD, Intel, IBM, HP, Sun, Toshiba and Microsoft. The consortium's mission statement is to produce and promote open specifications that enable trusted computing. Trusted computing is an umbrella of new security technologies that can aid in the protection against software attacks, the safe keeping of user data and digital identities and can support more secure user and platform authentication.

These open specifications describe interfaces of hardware and software building blocks which are applicable to multiple platforms and operating environments. The TCG envisions that these technologies can not only benefit servers and pc clients but can add real value to PDAs and next generation mobile phones as well.

3.3 Trusted Computing

Trusted computing is based on the notion of trust and the philosophy that trusted platforms can prove they are trustworthy instead of just having to assume that a platform can be trusted without any facts of evidence to support this. A trusted platform is able to guarantee that a specific platform is in a well known and healthy state and can irrefutably identify its user. If these facts are known of a system, then this system can be trustworthy enough to let it carry out specific functions or connect to it through a network and engage in transactions.

A trusted computing platform is defined by the TCG as a platform that is equipped with a Trusted Platform Module [TPM]. This TPM is a dedicated low cost hardware component that creates a Root of Trust. It is the foundation on which a trustworthy system can be built. The TPM implements a number of security functions that are explicitly required to be reliable and trustworthy to allow for a remote entity to trust the platform. All other necessary functionality is provided by complementary software solutions. The TPM is designed to be tamper resistant but currently cannot protect against hardware based attacks, e.g. the monitoring of the bus that connects the TPM to the motherboard. It is the intention of the TCG to have the TPM incorporated into the main CPU of future systems. This integration will make it very difficult for attackers to employ this method of attack in next generation TPMs.

3.4 Trusted Computing capabilities

The TPM's capabilities can be divided into four distinct categories

- Integrity measurement
- Cryptographic key services
- Protected storage
- Endorsement services

Integrity measurement is the process of measuring those parts of a system that are crucial for the trustworthiness of the system. If these measurements are accurate, an informed decision can be made about what the state a particular system is in and whether the system can be trusted or not. The TPM has a number of registers for the storage of these measurements. These registers are 160 bits large and are used to store SHA-1 [10] hashes of the hardware and software environment. Integrity measurement is covered in more detail in section 3.5.

The TPM supports *cryptographic key services* like asymmetric key generation (see section 4.3.1.7 of [11]), hardware performed encryption and decryption (see 4.3.1.8 of [11]), random number generation (see section 4.3.1.5 of [11]) and the digital signing of data (see 4.3.1.8 of [11]).

A very powerful feature of the TPM is *protected storage*. Protected storage provides an encryption service that not only depends on a particular encryption key but also on the software state of the platform. This means that the encrypted data will only be decrypted if the correct key is provided and if the current software state of the platform matches the stated software state at time of encryption. To bind encrypted data to integrity measurements is also called "Sealing" in trusted computing jargon. Protected storage is a feature that can protect sensitive information from threats like spyware [12] and viruses as the presence of these rouge software applications can be detected when the integrity measurements representing the software state of the platform are changed. As the software state no longer matches the software state at the time of encryption, access to the data will be denied.

There is another category of interesting services that the TPM can offer, the so called *endorsement services*. It is possible to create identity keys which can, with the help of a trusted third party, be used to prove the identity of a platform or a platform user. If such an identity key is used to encrypt data, the receiving party can unquestionably verify the sender. It should not be possible to forge this service without actually owning the identity key. These identities can prove very valuable for future e-

commerce applications as they are much more trustworthy than a simple username and password scheme.

The integrity measurement and the endorsement service support another powerful concept offered by trusted computing, the concept of *remote attestation*. This concept provides reliable reporting of the integrity measurements to local or remote entities and allows them to (remotely) verify the state of a platform. The integrity measurement that are sent, are signed by an identity key. The identity key can prove beyond any doubt that a particular platform is in a particular state because the identity key is linked to the endorsement key. Every TPM is shipped with a permanent and unique endorsement key that identifies uniquely that TPM. Only the trusted third party can link the identity key to the endorsement key so anonymity can be retained as many different identity keys can be created for a single TPM. Once the integrity measurements are obtained, they can be compared to the measurements of a known platform state. Based on this comparison the decision can be made whether or not to interact with the trusted platform at hand. Remote attestation will be covered in more detail in section 3.6.

3.5 Integrity measurement through a chain of trust

To achieve integrity measurement on a platform we need to construct a so called chain of trust. This chain is composed of different layers of software and hardware components that together form a platform with an operational operating system. When the platform is booted, the software components of the chain of trust will start loading one by one, starting from the lowest layer upwards, until the whole system is loaded. Each time a component is executed, it will measure the layer above by means of hashing. Only when this step is completed will the execution be turned over to the higher layer. The resulting measurements in the forms of hashes are stored inside dedicated platform configuration registers (PCRs) located in the TPM.

We depict a chain of trust composed of generalized software and hardware components present in TPM equipped platform. On this platform a particular operating system is installed and will be loaded during the booting process.

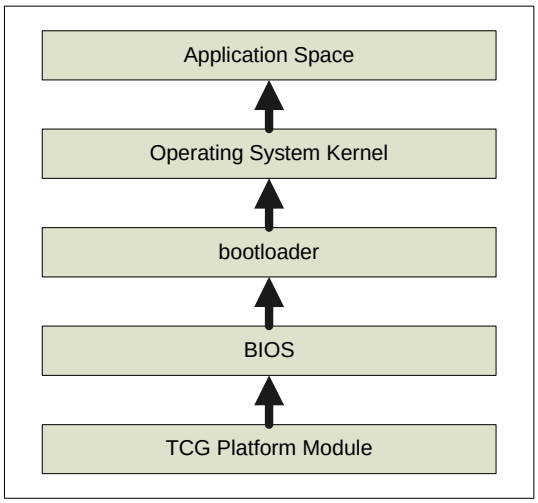


Figure 3-1: Chain of trust software stack

Each arrow in figure 3-1 symbolizes the execution of three distinct steps:

- measurement of the next layer through hashing (SHA-1)
- registration of these measurements inside the PCRs of the TPM
- handover of execution to the higher layer

The fact that the execution of the higher layer takes place after the verification process of that same layer should make it impossible for a higher layer to intervene with this verification process and forge the results. Because of this technique it should be impossible that altered components are executed without detection by the TPM. It is crucial that every chain in the chain of trust performs this verification task without error. If not, then the integrity measurements are worthless and cannot be used to make an assessment about the trustworthiness of the platform, which is the ultimate goal of integrity measurement.

The work done by the Trusted Computing Group has only produced standards at the lowest level of the chain of trust, in the form of a specification of the TPM programmers interface and at the highest level in the form of the specification of the TCG Software Stack (TSS). The TSS is a library that allows normal applications to make use of the functionalities offered by the TPM. The TCG have not defined anything for the intermediate layers, roughly put at the operating system level. This means that there is a gap between these layers which has been done on purpose. The TCG decided that it is the responsibility of operating system designers to fill this void and to give them the freedom for implementing their own architecture for the provisioning of trusted computing primitives for the operating system.

If we are to construct a trusted RFID reader or more precisely a trusted RFID reader application at the top of the chain of trust software stack, we will need to find trusted computing aware components for each layer depicted in figure 3-1. Fortunately a lot of research is conducted in this area and there are several published open-source components available that can offer some of the requested functionality. We will compare the different options available and discuss their inner workings and what functionality they can offer in section 5.4. For more information about the chain of trust and integrity measurement we kindly refer the reader to [13], the only abundant source of information on trusted computing concepts besides the actual specifications of the TCG.

3.6 Remote attestation

Remote attestation allows us to verify remotely the integrity measurements of a specific platform. As these obtained results can be verified to originate from the correct TPM and the results themselves cannot be forged, they provide enough information about a platform to be able to make a decision about whether or not the platform is trustworthy enough to perform some service or transaction. In this section we will briefly cover the steps that are taken by both parties during the remote attestation process.

The attestation process consists of three distinct phases, the integrity challenge, the integrity response and the integrity verification. The integrity challenge is a request by a remote party who wishes to know the state of a target platform. The remote party sends the integrity challenge to the attestable platform, followed by a nonce. A nonce is nothing more than a random string of bytes that will be added to the payload of the integrity response for increased security. The nonce is a means to prevent replay attacks, the resending of old integrity responses to new integrity challenges.

Once the integrity challenge is received by the attestable platform, the so called quote command is executed. This command generates a hash that represents the state of the entire platform. The hash is a composite hash of the values contained in the platform

configuration registers of the TPM. The resulting hash is concatenated by the supplied nonce and encrypted using an identity key available to the TPM. This encrypted data is sent back to the remote party as the integrity response.

The remote party can now start the integrity verification process. It first needs to check if the used identity key is in fact a genuine identity key originating from the intended TPM by contacting the appropriate third party certificate authority (CA). After decryption of the integrity response, the nonce needs to be compared to the nonce that was sent along with the integrity challenge. What remains now is the verification of the composite hash of the PCRs. The composite hash should be compared to a composite hash of a known system state. If there is a match, the state of the platform is known. Note that we were unable to use real identity keys for use with the quote command. We had to resort to ordinary signing keys. The reason for this is explained in section 5.5.3. A graphical representation of the remote attestation process is given in figure 3-2.

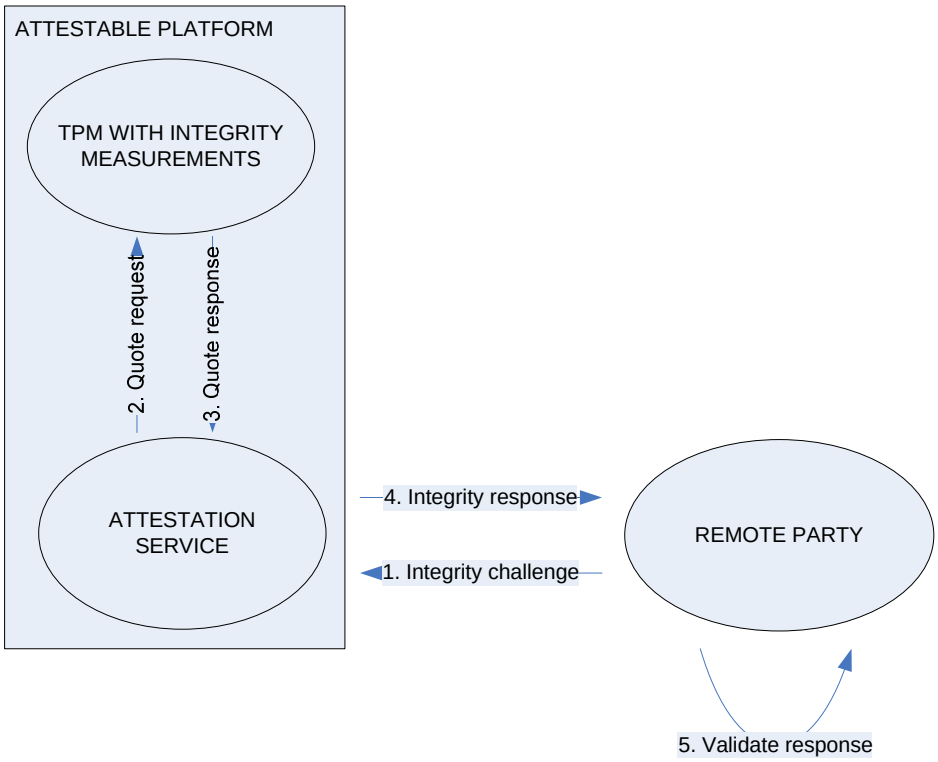


Figure 3-2: Remote Attestation

Chapter 4: Design of a secure RFID architecture

4.1 Introduction

We will first define the requirements for the trusted reader as presented in [9] and as discussed in section 2.2 and section 2.3 and use these as input for the design process. We will discuss the high level architecture design of the trusted reader given by the authors, followed by a description of the adaptations that have been made to this design and the justification for these changes.

We continue by defining more accurately the functions of the different building blocks and what interactions take place between these blocks. We will finish with functional designs at a lower abstraction level of the different functional blocks the trusted reader is composed of.

We will describe the algorithms and state machines using the Specification and Description Language (SDL). SDL is a specification language, specially defined by the ITU [14] for the specification of the behavior of reactive and distributed systems. A clarifying paper about using SDL to describe finite state machines can be found in [15].

Finally, the terminology used in this thesis will be defined to prevent interpretation mistakes. The “Trusted reader” is what we call the entire platform consisting of the external RFID reader, the personal computer with integrated trusted computing module and all the installed software on these two machines. The “Trusted reader application” is a part of the trusted reader. It is the piece of software that runs at the application level on top of the trusted computing aware operating system kernel and implements the functionality discussed in the rest of this chapter.

4.2 Project requirements

The requirements of the project need to be defined before an attempt at the construction or in this case an adaptation of a design can begin. Note that these requirements are formulated on the basis of [9] and discussions between Andrea Soppera, the RFID project leader, and myself. The goals set for the project are the following:

- The construction of a RFID reader that operates in a privacy-friendly manner. The reader needs to be equipped with a policy component that defines on the one hand which RFID tags are allowed to be identified and have its unique tag identifier passed on to the RFID infrastructure and on the other hand which RFID tags are considered to be private and should not be processed further. The active privacy policy should be easily updatable at run-time by the RFID reader owner. How these policies have to be defined is not yet specified. The specification of privacy policies will be covered in the design of the policy engine in section 4.5.2
- The construction of a reader that keeps records of the policies that have been enforced on the reader. These records should be remotely accessible so that they can be audited by a trusted third party. Because of this logging facility the reader won’t require full-time connection to the network infrastructure. This facility increases the flexibility of the reader and allows its application in environments where network connectivity is not constantly available, e.g.

during flight on an airplane. Obviously, it should not be possible to forge these records in any way.

- The construction of an RFID reader platform of which the integrity is measurable. Any form of tampering with the RFID reader platform should be locally and remotely detectable and when it is actually detected the reader should protect any secrets that might be stored on the reader to prevent the forging of the audit process. If the trusted reader is used in combination with tag encryption, the RFID reader should prevent the leakage of secrets used in the encryption/decryption of the RFID tags to protect the rest of the RFID infrastructure.
- The construction of a third party attestation client that allows remote verification of the trusted reader platform integrity and the retrieval and verification of the policy audit log of the trusted reader.

4.3 High level design of the trusted reader

4.3.1 Functional block diagram

A high level design of the reader is given and described in [9]. It is reproduced with minor adaptations in figure 4-1. The design shows the individual functional building blocks of the trusted reader, which are the reader core, the policy engine, consumer agent and the trusted platform module. The left side of the figure shows at which level the modules are envisioned for implementation in the reader platform. Figure 4-1 additionally shows the interactions of the trusted reader platform with external modules. The interactions are modeled by the arrows. The platform communicates with the RFID tags to read the tag identifiers; it passes the tag identifiers to the RFID middleware or some RFID application if allowed by the enforced privacy policy. The trusted reader also communicates with a possible third party for monitoring the platform integrity and for auditing the enforced privacy policies.

4.3.2 Functional block descriptions

The *reader core* is the central module of the reader and is positioned at the operating system level. It is responsible for interfacing with the trusted platform module and the RFID radio interface. Its main responsibility is twofold. Firstly, it makes sure the platform integrity is up to date in the TPM. Secondly, it ensures the monitoring of the launched applications, in this case the policy engine and the consumer agent.

The *policy engine* is the module that allows the reader to operate in a privacy friendly manner. The policy engine has a defined policy which tells the reader which tags are allowed to be read and which not. The policy in the policy engine is run-time updatable. The policy engine is also responsible for the storage of tag-reader secrets if needed for the decryption of encrypted RFID tags. These secrets are passed to the reader core when needed. The secrets are placed in the policy engine because this software component is more easily updatable than the reader core kernel.

The *consumer agent* is a logging component that allows third parties to actively monitor privacy policies. It interacts with both the policy engine and the reader core. It records the reading operations that have occurred and the policies that have been enabled. The consumer agent is also responsible for reporting the integrity of the system and can halt the system if it is compromised.

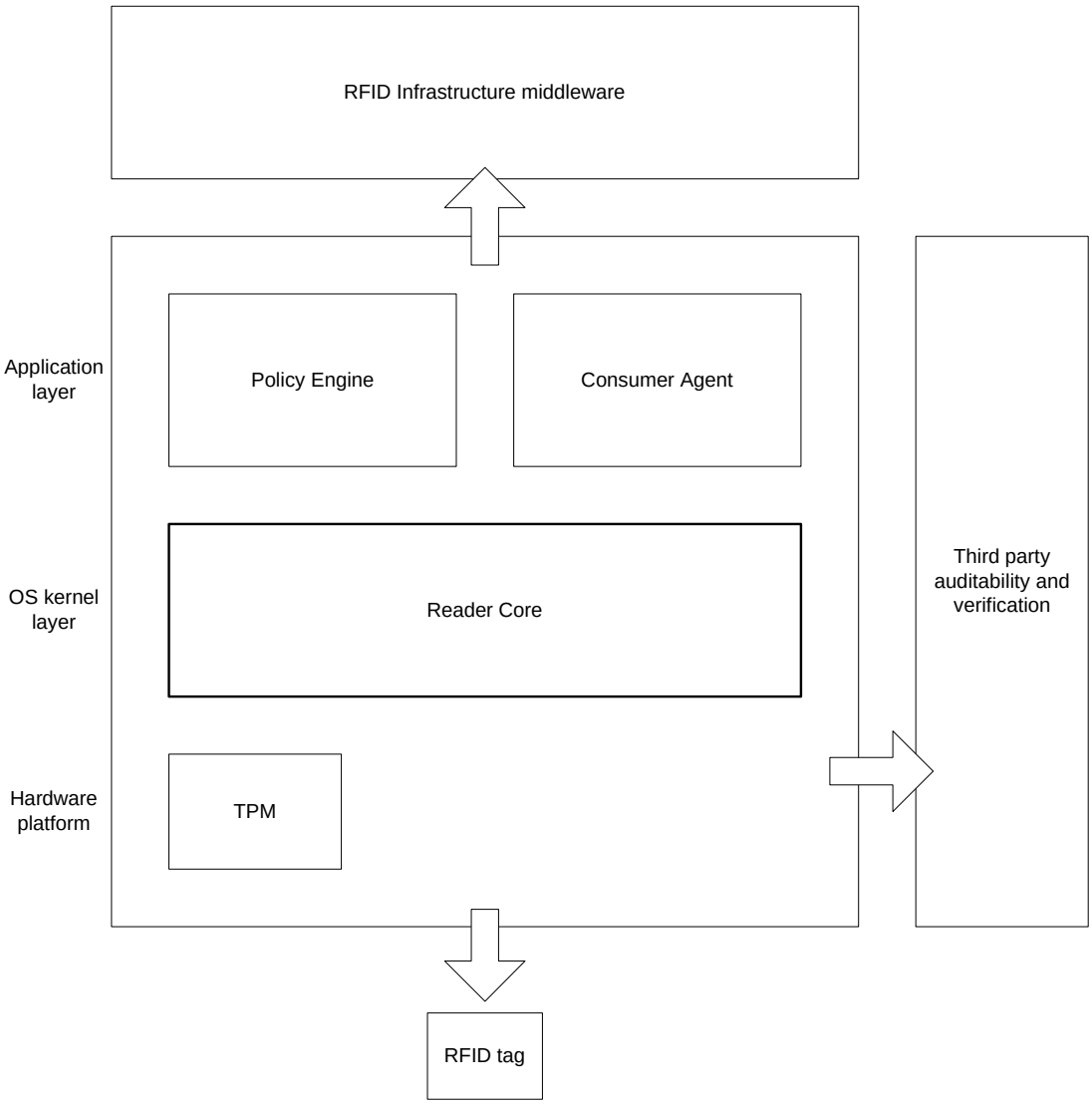


Figure 4-1: Functional block diagram of the trusted reader

4.4 Design modifications

4.4.1 Introduction

Before we discuss the design modifications made it is important to realize that "Privacy for RFID through trusted computing" by Molnar, Soppera and Wagner [9] was written with an RFID platform with an integrated TPM in mind. In our implementation, however, this is not the case. We use a platform with an integrated TPM that is connected to an external RFID reader. We regard this system as an integrated unit or black box but in practice this is not the case, which affects the design of the trusted reader platform.

4.4.2 Reader core

The reader core is defined to interface with the TPM and to make sure that the platform integrity is accurately recorded. It is also responsible for monitoring the applications that are run on top of the kernel. This is a generic description of a trusted

computing aware kernel that performs integrity measurement up to the application space level. In our design we will also make use of such a kernel. We compare two possible trusted computing aware kernels for inclusion in the implementation chapter, in section 5.4.3 and 5.4.4. Figure 4-2 shows an updated functional block diagram of the reader. It should be noted that we changed the name of the element at the OS kernel layer from reader core to trusted aware kernel and that we will redefine the functions of the reader core module in the next paragraph.

The reader core is also responsible for interfacing with the RFID radio interface at the operating system level. For an embedded system it is acceptable to incorporate core functionality in the embedded kernel. In our case the RFID radio interface is not integrated in the platform. To place our external reader software interface at the operating system level is not a logical choice as there are only application level libraries available to communicate with the external reader. Therefore in our design we redefine the reader core module as a software module at the application level that has the single responsibility to interface with the external RFID reader through supplied application level libraries by the RFID reader manufacturer. The reader core fulfills the role of a pass-through window for tag identifiers that are read from the RFID reader to the rest of the RFID architecture with the policy engine policing the individual tag identifiers. We use this pass-through module so that we can easily separate the business logic that is contained in the policy engine from this structural part of the application. By keeping the API of the policy engine backwards compatible we can easily modify the policy engine without having to make any changes to the rest of the trusted reader software modules.

4.4.3 Policy engine

The function of the policy engine in our design remains unchanged. We do have some remarks about the possibility to store secrets in the policy engine, however. The choice for storing secrets in this software module is an illogical design choice. In the requirements it is stated that the secrets should be protected in case the integrity of the platform is compromised. If the secrets are stored in a software module, then we cannot guarantee the safety of this data. It would be more logical to store the secrets using the sealed storage capabilities of the TPM. If the integrity of the system is compromised the TPM will not allow access to the secrets and hence the secrets are well protected. In our design we use the TPM to store the encryption key that is needed for the remote attestation of the platform.

4.4.4 Consumer agent

The consumer agent is the designated component for the auditing capabilities of the reader. The consumer agent keeps a log of all enforced privacy policies and reading operations but also supports remote retrieval of these logs and the integrity measurements of the trusted reader platform. In our design we split these two functionalities into two separate modules.

The first module, the consumer agent, takes care of the tamper proof logging of the policies on the reader. Note that it doesn’t log individual reading operations as we think that this will not scale very well on busy readers and because embedded systems have limited disk and memory space. We also believe that keeping logs of all reading operations does not add value to the auditing process. Moreover, this solution offers a more privacy friendly way of operation which is, after all, the main pursuit of this project.

The second module is the remote attestation module which allows remote parties to connect via a network connection to query the policy logs and the dynamically generated integrity measurements of the platform. This division seems logical as these are two distinct functionalities. Also the consumer agent is a module that is permanently active by way of monitoring the policy engine. The remote attestation module on the other hand is only invoked when a remote third party wishes to perform an audit.

In the description of the consumer agent it is also stated that the consumer agent can halt operation of the reader if the system is compromised. While it is surely possible to do that, the reasoning seems flawed. If the trusted reader platform is compromised we must assume that the attacker has administrator level access to the system. If this is the case the attacker can bypass any of the installed modules and run his own software to interface with the reader. There is nothing we can do to prevent that. The problem is that the trusted platform module is a passive security device. It measures and stores integrity data of the platform but is not designed to actively respond to any form of attack. This means that we cannot stop the attack but the TPM does allow us to detect the attack, possibly remotely and it keeps our secrets safe in the sealed storage.

We did include another form of integrity checking functionality into the consumer agent. The moment the trusted reader application is executed or every time the audit log is appended, it is first checked whether the encrypted signature still matches the stored log file. If it does not, the trusted reader software is immediately shut down as somebody might have tampered with the logs in an attempt to hide previously activated privacy policies.

4.4.5 Functional block diagram

If we update the functional block diagram shown in figure 4-1 with the modifications discussed we get a slightly different picture in figure 4-2. What is not shown in figure 4-2 is how the individual modules of the trusted reader interact with each other. These interactions are an important part of the design. We will define these interactions in section 4.4.6.

What is not covered at all by the functional block diagram is how the trusted computing chain of trust software stack is set up. It is mentioned that a trusted aware kernel is required at the OS kernel level in figure three. Merely having such a kernel is not enough to fulfill the requirement of platform integrity measurement. The integration of the different modules to construct the chain of trust is documented in section 5.4.

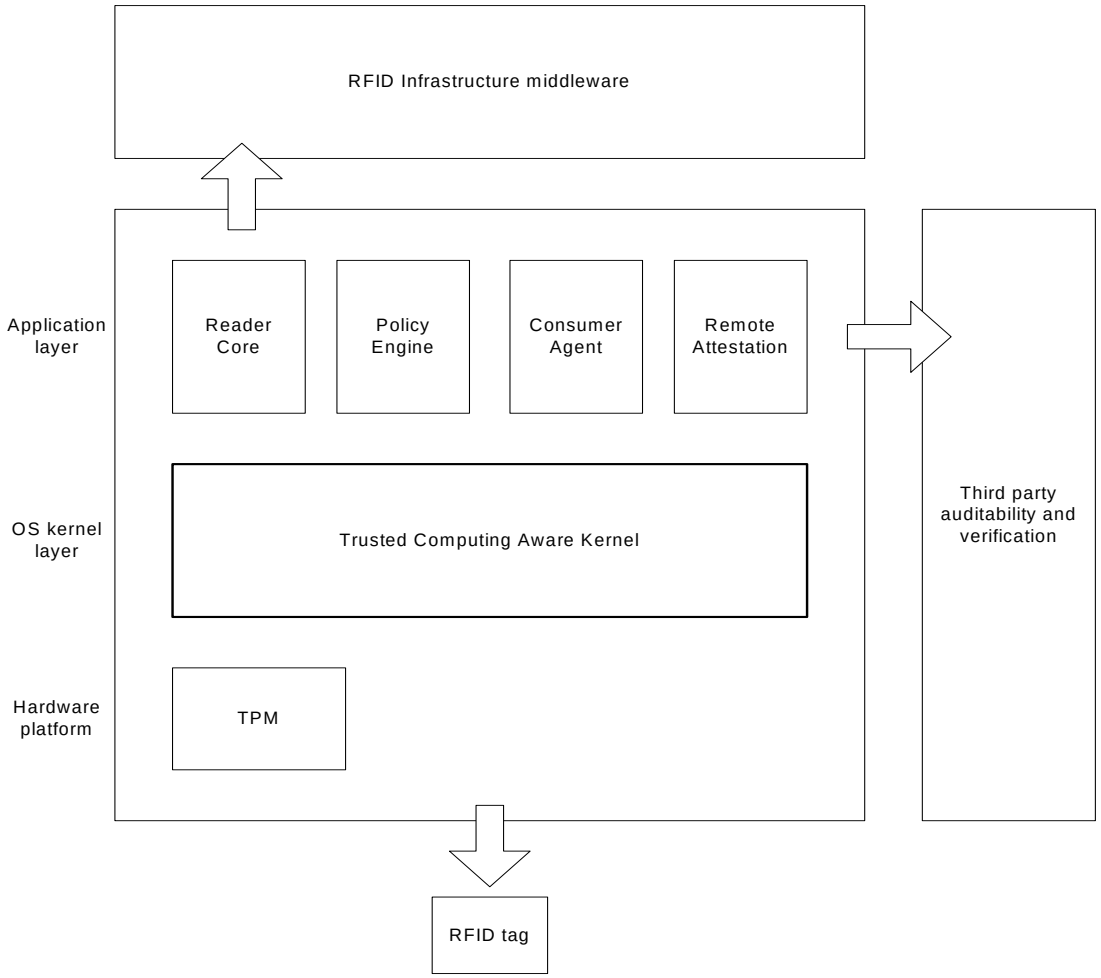


Figure 4-2: Updated functional block diagram of the trusted reader

4.4.6 Interaction diagram

4.4.6.1 Introduction

In this section we aim to define the interactions between the different modules that are defined at the application layer in figure 4-2. These interactions are important to fully understand how these modules cooperate and how the data flows through the system. We do this by providing an interaction diagram. This diagram models the interactions between the individual modules with connecting lines. We also offer descriptions of these interactions. The interaction diagram is shown in figure 4-3.

4.4.6.2 Diagram description

The interaction diagram models the software modules of the trusted reader as a single entity called the trusted reader software application. This application consists of the four modules defined at the application level in figure 4-2. The trusted reader software application interacts with four external modules.

- The *RFID application* is an application that can run locally on the reader or remotely as part of a bigger RFID infrastructure. The tags that are detected by the reader and cleared by the policy engine are sent to this application.

- The *Samsys RFID reader library* is a vendor supplied software library that allows easy integration of the RFID reader into new or existing applications.
- The *trusted computing software stack* is a software library that can provide applications access to trusted computing services.
- The *third party attestation client* is an application that can perform remote attestation with the trusted reader software application through a set up network connection.

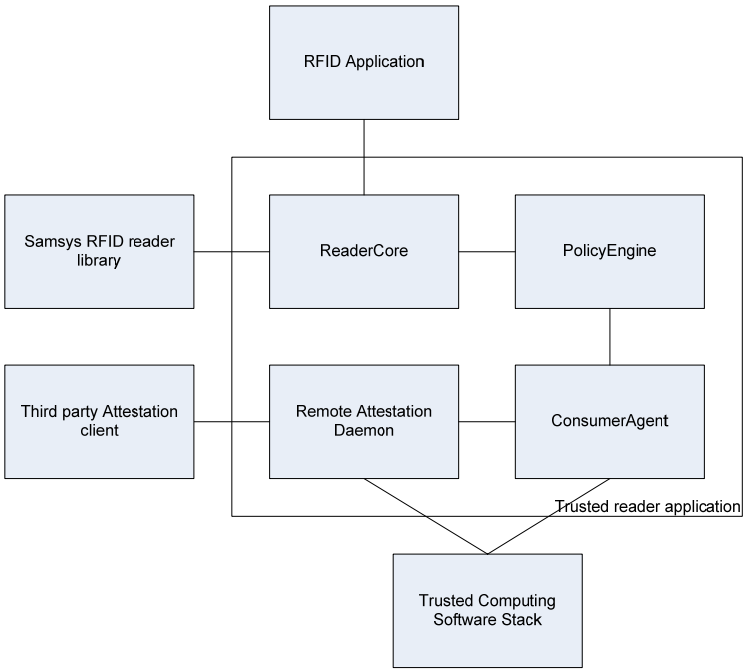


Figure 4-3: Interaction diagram

4.4.6.3 Diagram interactions

The reader core interacts with the Samsys RFID reader library to receive the tag identifiers that are detected by the RFID reader. The reader core also works together with the policy engine. The reader core queries the policy engine each time a tag is received. The policy engine decides for each tag whether or not it can be sent to the RFID application.

The policy engine always has an active policy set. The policy defines which tags are allowed to be passed to the RFID application and which tags have to remain private. Whenever the active policy changes the consumer agent is informed of this change and receives a description of the new policy.

The consumer agent is a logging component that keeps a record of all activated policies in the policy engine. The consumer agent interacts with the trusted computing software stack to make sure that tampering with the logfile is always detected.

The remote attestation daemon communicates with the third party attestation client through a network connection. The attestation client requests the policy logfile or integrity measurements of the reader platform. The remote attestation daemon interacts indirectly with the consumer agent by way of retrieving the policy logfile.

The remote attestation daemon also interacts with the trusted computing software stack to recover the integrity measurements of the platform from the TPM.

4.5 Lower level design

4.5.1 Reader core

In our design the reader core is a central module that interacts with numerous other modules. The reader core is used as a conduit for the passing of tag identifiers from the RFID reader up to the RFID application. The reader core also consults the policy engine for each tag that goes through the system to check if the RFID application is allowed to receive this information.

Separating the flow of the tag from the logic that is contained in the policy engine module allows a clean separation of concerns. It allows us to easily update the policy logic without having to update the reader core. The reader core is depicted in figure 4-4.

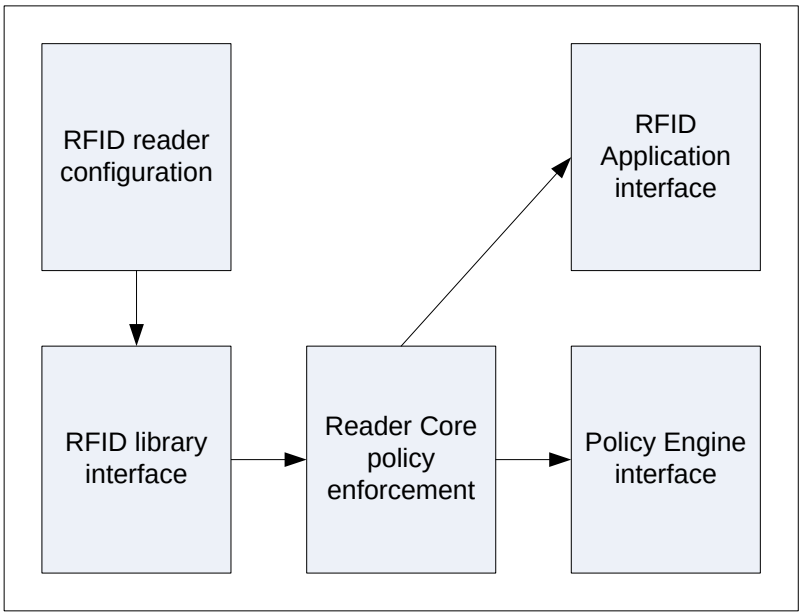


Figure 4-4: Functional block diagram of reader core

The *RFID library interface* is responsible for setting up a connection to the external RFID reader. Once the connection is established the RFID library interface will forward all the detected tag identifiers by the RFID reader to the reader core policy enforcement module.

RFID reader configuration is a module which uses the RFID library interface to configure the RFID reader. The RFID reader has a lot of configuration options. The reader can for example read a large amount of different kinds of tags but it can only read one tag type at the same time. This means that the RFID reader needs to be explicitly configured for the kind of tag used by the prototype.

For each tag that is delivered by the RFID interface to the *reader core policy enforcement*, the policy engine is queried through the *policy engine interface*. The policy engine is inquired whether or not the tag is allowed to be sent to the *RFID application interface* or if the tag is to be discarded according to the active privacy policy of the policy engine. The reader core policy enforcement module is accurately

defined in figure 4-5. SDL is used to describe the state machine behavior of the module.

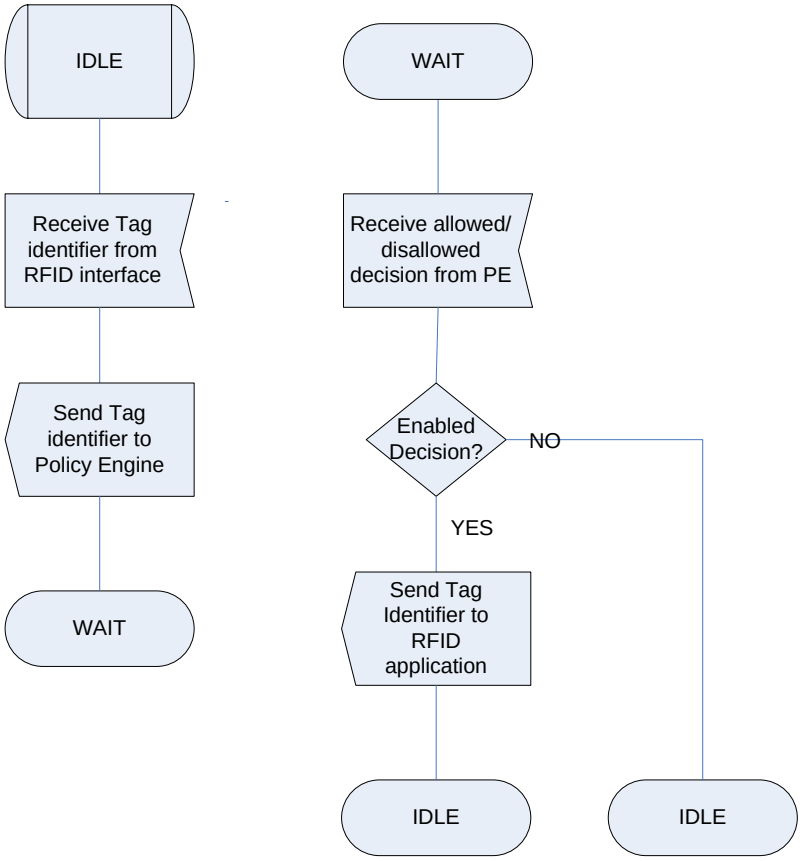


Figure 4-5: Reader core policy enforcement state machine

The reader core policy enforcement state machine begins in the IDLE state. It waits in the IDLE state until an incoming message is received from the RFID interface with a tag identifier.

Promptly the state machine forwards this tag identifier to the policy engine interface. The FSM is now in the WAIT state and it remains there until the policy engine interface returns the decision made by the policy engine whether to allow or disallow the tag information to be passed on to the RFID application.

If the policy engine returns an allowed decision the tag is forwarded the RFID application and the FSM returns to the IDLE state. If the policy engine returns a disallowed decision the FSM returns to the IDLE state immediately without sending the tag to the RFID application.

4.5.2 Policy engine

The policy engine module plays an important role in the trusted reader application. It is solely responsible for providing the privacy protection capabilities of the RFID reader. The policy engine checks if tags match the privacy policy. If the tags match

then they are allowed to be used by the RFID application. If the tags are not covered by the privacy policy they should be discarded.

When the policy engine is initialized it is normally provided with a specific policy. Either a new policy is defined by the RFID owner or the last active policy from the previous execution is loaded from disk. If the policy remains undefined, the policy engine will fall back to a default policy. The default policy decision can be configured to allow all tags or deny all tags. A functional block diagram of the policy engine is given in figure 4-6.

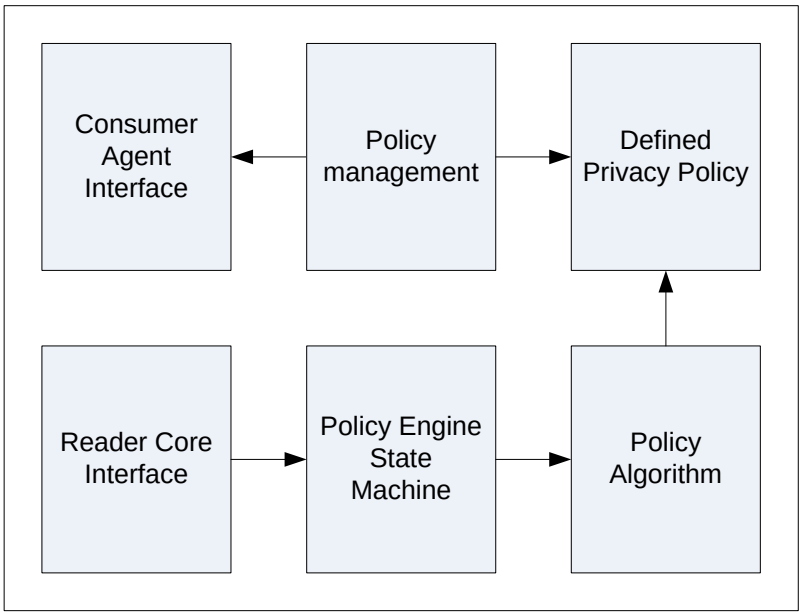


Figure 4-6: Functional block diagram of policy engine

The *policy engine state machine* receives queries from the reader core interface to decide whether or not the tags satisfy the privacy policy. The policy engine state machine executes the *policy algorithm* to make this decision. The policy algorithm traverses the *defined privacy policy* and returns the verdict to the policy engine state machine. The verdict is relayed to the reader core through the reader core interface. The policy engine state machine is depicted in figure 4-7 and the policy algorithm in figure 4-9.

The policy engine also contains a *policy management module*. This module manages the privacy policy of the policy engine. The policy management module allows the RFID owner to update the privacy policy at any time. When a new privacy policy is defined and activated the consumer agent is informed of this update through the *consumer agent interface*.

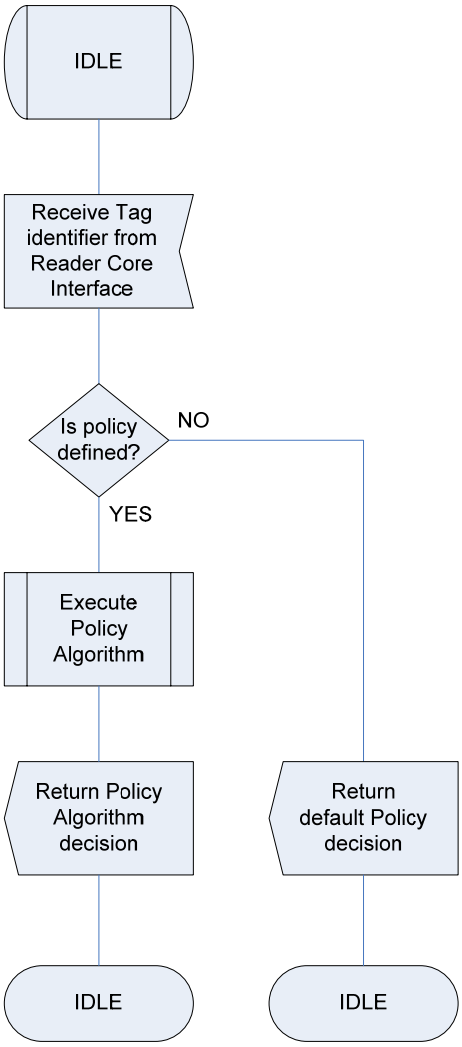


Figure 4-7: Policy engine state machine

The policy engine state machine begins in the IDLE state. It waits in the IDLE state until an incoming message is received from the reader core interface with a tag identifier.

If a policy is defined the policy algorithm is executed. The outcome of the algorithm is returned to the reader core through the reader core interface and the FSM returns to the IDLE state.

If no policy is defined the policy engine will not execute the policy algorithm. It will simply return the default policy decision to the reader and the FSM returns to the IDLE state.

An important part of the policy engine that has not been discussed in detail is how the actual policy is defined as the design of the policy algorithm is strongly dependent on the policy definition. To get a clear perspective we will first look at how RFID tag identifiers can be interpreted. Based on that knowledge we can move on to the policy definition and policy algorithm.

4.5.2.1 Tag Identifiers

An RFID tag identifier is not just a random unique number. The tag identifier is hierarchically structured similar to Media Access Control (MAC) addresses that are present in e.g. Ethernet cards. The tag identifier encodes a unique manufacturer identifier, a unique product identifier and a unique serial number.

Other RFID tag schemes defined, they encode different kinds of information for different industries or other purposes besides the one described here. These encoding schemes are defined by an organization called EPCglobal. The defined encoding schemes can be found in [16]. Note that at the time of implementation the earlier mentioned standard was not yet available for the type of tag that was used in our RFID prototype. Since the actual tags were also produced before the standard was completed the tag identifiers do not correspond to any of the encoding schemes defined in the new standard. We have defined our own simple encoding scheme to cope with this situation.

The tags that have been used in the prototype are Texas Instruments generation 2, class 1 tags. The tag identifier of these kind of tags is set upon manufacturing and is 96 bits long. The 96-bit identifier has been divided into five different sections to correspond to four specific fields for interpretation. One field is unused. We have excluded this unused field because of practical reasons with the tags used in the demonstration. See figure 4-8 for a graphical representation of the division of the tag identifier.

Figure 4-8: RFID tag identifier encoding scheme

The header (12 bits long) normally specifies which encoding scheme is used on the tag. Based on the chosen encoding scheme the rest of the bits can be interpreted correctly. In our case the header is just ignored because the value set in the header is unspecified. We will follow the encoding scheme as it has been specified here. It is clear that we encode a manufacturer identifier of 28 bits long, a product identifier of 24 bits long and a serial number of 24 bits long.

4.5.2.2 Policy definition

A policy is constructed by the definition of one or more policy rules. These policy rules allow the definition of an accurate privacy policy because the policy rules take into account the encoding scheme used in the tags. Four different types of policy rules have been defined. Each of these rules allows us to specify what values or ranges of values are acceptable for the different sections (read identifiers or serial number) of the tag identifier. There are four different kinds of policy rules that can be specified:

- Policyrule type 0 is mainly for testing purposes and if it is specified in the policy it allows all tags to be read.

- Policyrule type 1 allows the reading of tags where the manufacturer_id of the policyrule matches exactly to the manufactured_id of the tag.

- Policyrule type 2 defines a particular manufacturer_id and product_id which both need to be matched to the tag.

- Policyrule type 3 is the most detailed policyrule as it allows the definition of the manufacturer_id and the product_id but also a range of serial numbers, allowing specification of allowable tags up to individual items level.

This methodology for defining policies has been devised to support a prototype demonstration of a trusted RFID reader checkout scenario which will be presented later in the report in section 6.4. The policy scheme can easily be extended by defining additional policy rules which allow specification of ranges of identifiers for manufacturer and product identifiers, making it more suited for real-world applications. Additionally other encoding schemes can be supported by taking into account the newly defined encoding schemes now that the standard for the RFID tag identifiers is updated to incorporate the second generation RFID tags.

4.5.2.3 Policy algorithm

Now that policies can be defined using the different policy rules we need to design an algorithm that has to check if an identifier matches a defined policy or not. The definition of the policy rules allows for a simple algorithm to perform the policy enforcement. A definition of the policy algorithm is given in figure 4-9.

4.5.2.4 Policy algorithm FSM description

The policy algorithm FSM begins in the IDLE state. It waits for an incoming message with a tag identifier from the policy engine state machine. A Boolean variable named allowtag is initialized to false. The FSM is now in the GET_RULE state.

The algorithm is now ready to match the tag to the next policy rule of the privacy policy. Depending on the type of the policyrule different parts of the tag identifier are matched.

1. If the retrieved policyrule is of tagtype 0, the allowtag Boolean is set to true.
2. If the retrieved policyrule is of tagtype 1, the manufacturer identifier of the tag is compared to the manufacturer identifier of the policyrule. If the manufacturer identifier matches the tagallow Boolean is set to true.
3. If the retrieved policyrule is of tagtype 2, both the manufacturer identifier and the product identifier are compared. If both identifiers match the tagallow Boolean is set to true.
4. If the retrieved policyrule is of tagtype 3, the manufacturer identifier, product identifier and the product serial number are compared. If the manufacturer identifier and product identifier match and the product serial number falls in the range specified by the policyrule, the tagallow Boolean is set to true

If there are more policyrules the FSM goes back to state GET_RULE to process the next policyrule. If there are no more policyrules to process, the value of the allowtag Boolean is returned to the policy engine state machine and the policy algorithm FSM goes back to the IDLE state.

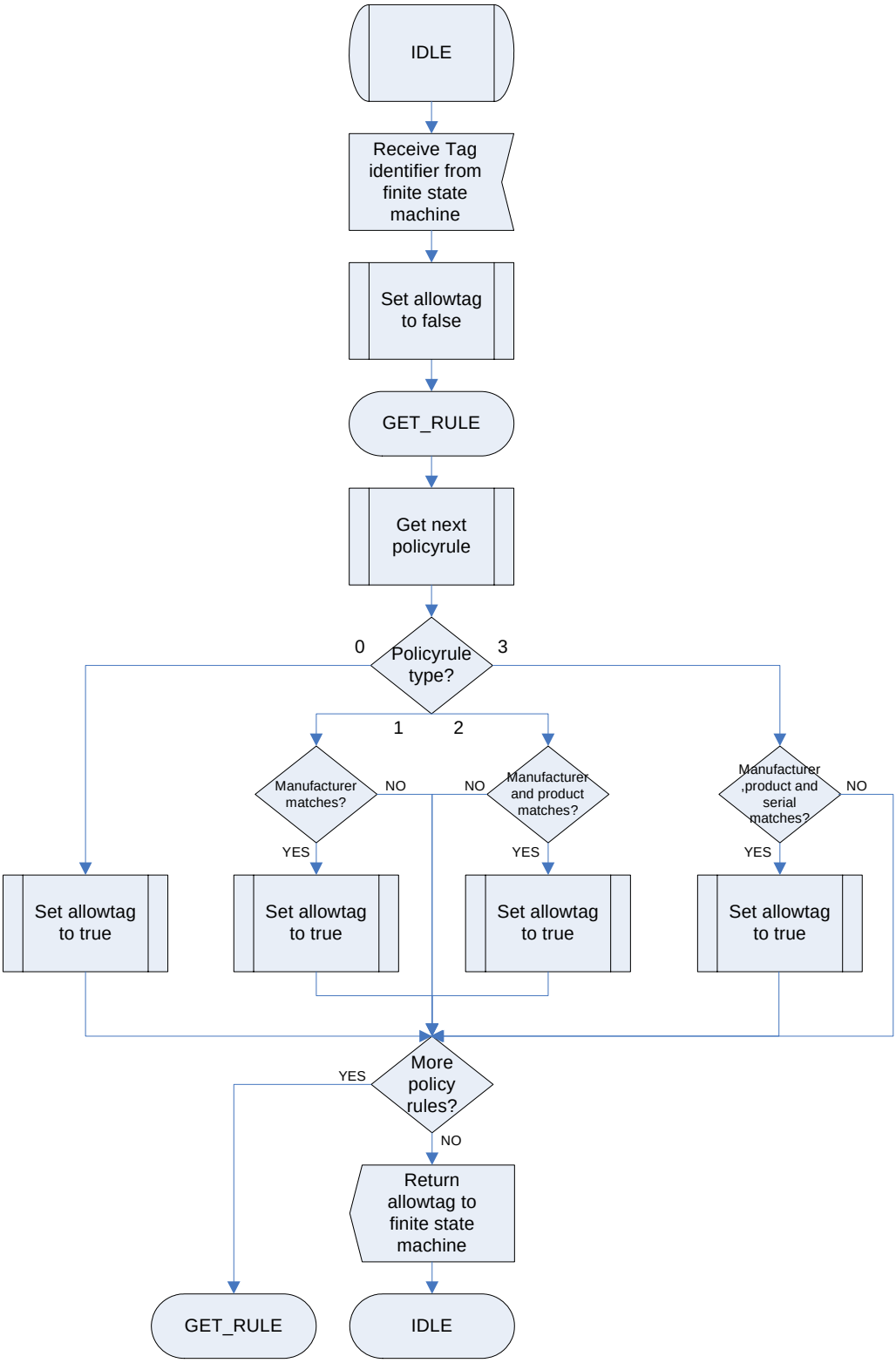


Figure 4-9: Policy algorithm

4.5.3 Consumer agent

The consumer agent is the logging module of the trusted reader software. The consumer agent is responsible for maintaining a log of all the privacy policies that are defined in the policy engine in a safe way. The consumer agent must be able to detect tampering of the log file to prevent undetected manipulation of the policy logs.

The consumer agent achieves tamper detection of the log by maintaining a signature. Every time a modification is made to the log, the signature is recalculated and updated accordingly. The signature is encrypted by an encryption key to protect the signature from modification. It is in fact the TPM that stores the key and handles the encryption. An overview of the different functional blocks of the consumer agent can be found in figure 4-10.

Note that not only the policies are logged. The consumer agent also logs every startup and shutdown of the reader software. This can aid in the detection of tampering where the integrity of the platform has not been compromised. A possible scenario might be that a reader is taken offline and subsequently brought online again running on different software. At a later time this is reversed. If a third party didn't execute remote attestation during this time, the attack might not be detected. The time stamping of startup and shutdown can help in detecting these kinds of attacks as it will show these time gaps.

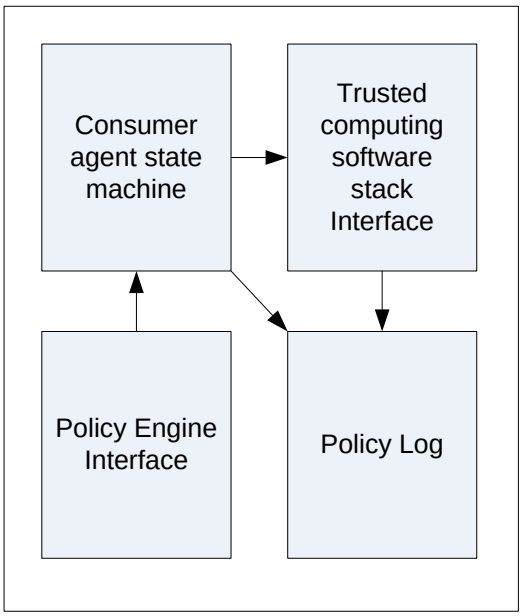


Figure 4-10: Functional block design of the policy engine

The *policy engine interface* signals the reader when the policy has been changed in the policy engine. The consumer agent state machine will use the *trusted computing software stack interface* to have the TSS check if the policy log matches its encrypted signature. If the signature is no longer valid the trusted reader software application will shut down immediately. If the signature is still valid the *policy log* is updated by the consumer agent state machine with the new privacy policy. After the update has been completed the consumer agent state machine immediately invokes the TSS

interface to update the signature of the policy log to match the updated policy log. The consumer agent state machine is formalized in figure 4-11.

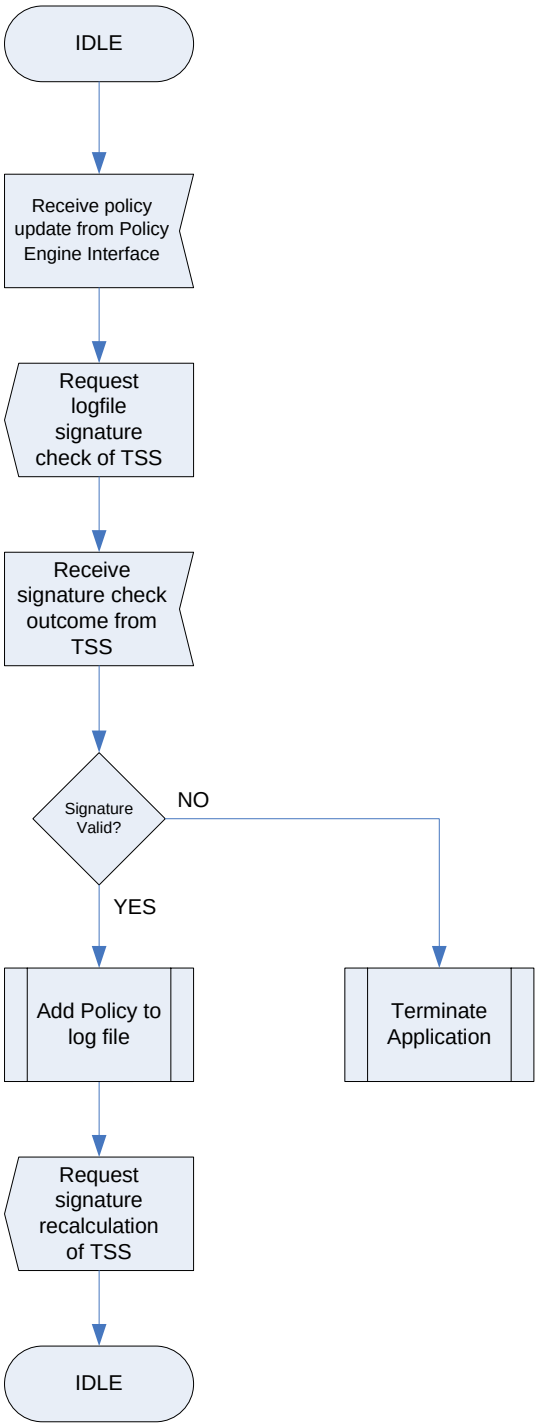


Figure 4-11: Consumer agent finite state machine

The starting state of the FSM is IDLE. The FSM waits for an incoming message from the policy engine interface containing the updated policy. Immediately after the message arrives, The FSM sends a message to the TSS interface to request validation of the policy log and signature. The TSS interface will respond with a message that contains the outcome of the verification. If the signature is no longer valid the trusted reader software application is terminated. If the signature is valid the FSM appends

the new privacy policy to the policy log file. The FSM sends a message to the TSS interface to request a recalculation of the signature to let the signature correspond to the updated policy log. The FSM goes back to the IDLE state.

4.5.4 Remote attestation

The remote attestation module is a network daemon that allows remote parties to connect to the trusted RFID reader over a network connection to:

- Retrieve a composite hash of the contents of the platform configuration registers to allow remote verification of the integrity of the reader hardware and software.

- Transfer the log files that contains a list of all policies that have been active on the reader combined with a hash signature for tamper detection.

An overview of the different functional blocks of the remote attestation module can be found in figure 4-12. Note that the policy log component in figure 4-12 is the same policy component as the policy component depicted in the functional block design of the policy engine in figure 4-10. The policy log models the actual log file and signature located on the hard disk.

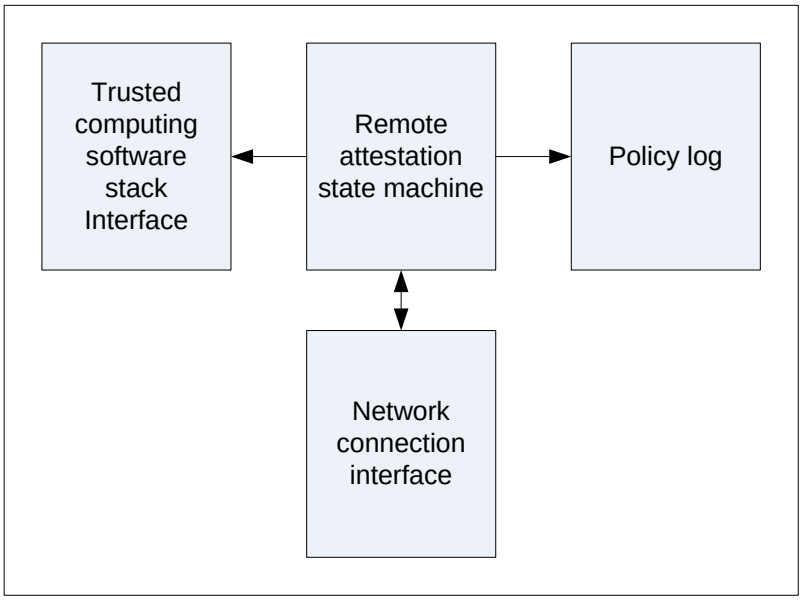


Figure 4-12: Functional block design remote attestation module

The remote attestation module is composed of four functional building blocks. The *network connection interface* waits for incoming connections from remote auditors and forwards any received requests that arrive via the established connection to the *remote attestation state machine*.

If a remote attestation request is received the remote attestation state machine will interact with the *trusted computing software stack interface* to retrieve the platform integrity measurements. These measurements are sent back to the third party through the established network connection via the network connection interface.

If the *policy log* is requested by the remote auditor the remote attestation state machine will retrieve the policy log and signature from the hard disk and send it back via the established connection through the network connection interface.

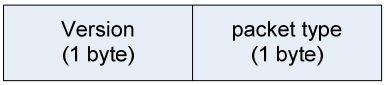
For the interaction between remote auditors and the network connection interface a network protocol has been devised. This protocol is described in section 4.5.4.1

4.5.4.1 Trusted reader network protocol

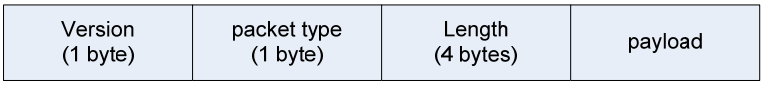
A simple request/response protocol has been defined to handle the data transfer between the remote attestation module and the remote attestation client. It is an application level protocol built on top of the reliable Transmission Control Protocol (TCP). TCP has been chosen as transport protocol because it offers in sequence error free data transfer. This is required as a single error introduced by the data transfer will invalidate the encrypted hashes and consequently makes the transferred data useless since its validity can no longer be ascertained.

Using TCP the data is sent across as a logical byte stream, allowing easy transfer of big chunks of data without the need for dividing the data in smaller blocks and reassembling them at the receiving side. This is useful as the log file can be of considerable size. If UDP had been chosen as the transport protocol, the design complexity of the protocol would be increased considerably as this functionality would then have to be built in the protocol itself. The protocol is defined by four PDUs:

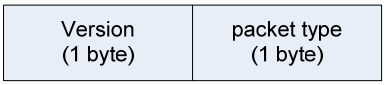
- **getQuoteRequest**: This PDU is sent by the remote attestation client to the remote attestation daemon to request the output of the integrity measurement of the platform. The PDU only contains two fields, a version field and a packet type field. The version field allows for easy changes or extensions to the protocol by defining new versions. Packets received with an unsupported version are immediately dropped by the protocol stack.



- **getQuoteResponse:** This PDU is sent by the daemon as a reply to the getQuoteRequest PDU and delivers the encrypted integrity measurements in its payload to the remote client. The length field defines the length of the payload expressed in bytes. The presence of the length field facilitates the decoding of the PDU at the receiving end.



- **getLogRequest:** This PDU is sent by the client to request transfer of the auditing logs. The structure of the PDU is similar to that of the getQuoteRequest PDU. Only the value of the packet type differs from the getLogRequest PDU.



- **getLogResponse:** This is the response of the daemon to the getLogRequest PDU. It transfers the audit log and the encrypted hash files in payload one and two respectively. The lengths of the payloads are again stored in the length fields for easier decoding. The length fields are placed in front of the payloads because the length fields have a fixed length. Putting fixed length fields is easier and faster to decode.

Version (1 byte)	packet type (1 byte)	length 1 (4 bytes)	length 2 (4 bytes)	payload 1	payload 2

4.5.4.2 Remote attestation state machine description

The remote attestation state machine is defined using SDL in figure 4-13.

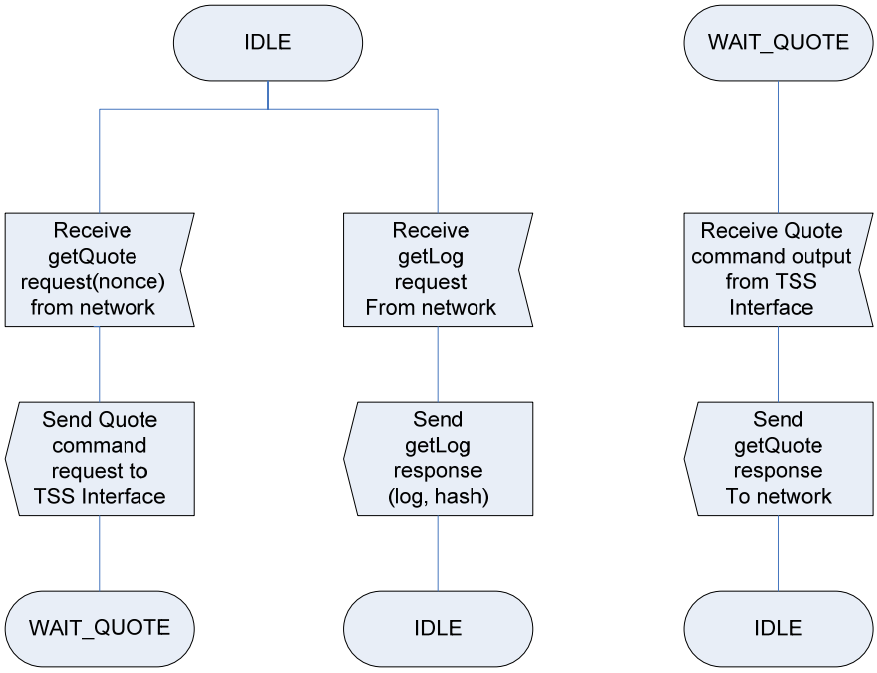


Figure 4-13: Remote attestation state machine

The FSM starts in the IDLE state. The FSM can now handle the reception of either a getQuoteRequest or a getLogRequest.

A getLogRequest can immediately be responded to by sending back a getLogResponse containing the log file of the consumer agent and the accompanying encrypted hash that are stored locally on the disk (policy log). The FSM returns to the IDLE state.

If a getQuoteRequest is received, the so called quote command is executed by the TSS interface. The FSM is in the WAIT_QUOTE state.

After the quote command is executed by the TSS the quote command output is sent back from the TSS to the FSM.

The FSM immediately sends this output back to the third party via the network connection. The FSM returns back to the IDLE state.

Chapter 5: Implementation of the trusted reader

5.1 Introduction

If we look back on chapter 4 it might seem that this thesis mainly consisted of the construction of the trusted reader application because it is the only element of which the design is discussed in detail. This chapter will show that this is an incorrect assumption.

In order to be able to satisfy the requirements with regards to integrity measurement of the trusted reader platform we needed to construct the aforementioned chain of trust. This chain of trust has been composed of separate open source software elements. The selection and integration of these different modules has been a major untertaking. This process will be described in this chapter.

We will also cover the software implementation of the trusted reader application by describing the static structures of the software in the form of class diagrams and API descriptions. We will also cover the more dynamic aspects by describing the interactions between the different objects using sequence diagrams. For a UML refresh we can recommend “UML for Java Programmers” by Robert C. Martin [17] which provides an excellent hands-on approach to modeling with UML.

A lot of the project time has been invested in learning and programming the trusted computing software stack. A lot of small helper applications have been implemented to get to know and test the capabilities of the TPM in practice. The know-how gained during this period has been put to use in the programming of the interactions between the trusted reader application and the trusted computing software stack. The results will be described in detail.

We will start the chapter with an overview of the hardware and software employed in the trusted reader platform and the motivations for these choices.

5.2 Hardware Inventory

At the start of the thesis project it had already been decided by the research group which hardware was to be used. One of the goals of the project was to construct a working prototype that is portable so that it can easily be used for demonstrations at workshops and conferences. As a result an IBM ThinkPad T42P laptop was selected.

The IBM T42P (see figure 5-1) is a modern laptop equipped with a fingerprint reader and an Atmel trusted platform module that implements the Trusted Computing Platform Alliance main specification version 1.1b [18], which was the last version available at that time. Recently a new version of the standard has been released (version 1.2) by the TCG. A clear overview of the available personal computers equipped with trusted platform modules can be found here [19].

Figure 5-1: IBM Thinkpad T42P equipped with an Atmel TPM

The second piece of hardware required for the project was an RFID reader. The reader needed to support a wide variety of tags allowing experimentation with the different tags with regards to range and reliability of read and write operations, interference by metallic packaging and so on. This interference caused real problems during the preparations for the workshop demonstration of the prototype. These problems are discussed further in section 6.4.1.

The reader also needed to have enough power to allow ranges up to five or six meters for effective demonstrations of the prototype. A Samsys MP9320 v2.8E reader (figure 5-2) has been selected. It is a high power RFID reader that allows a maximum of four RFID antennas to be connected. The reader is equipped with a serial port and an Ethernet interface. We use the Ethernet interface to connect the laptop to the reader in the prototype. A picture of a reader set up with four antennas, as used during the demonstration, can be found in section 6.4.1.

Figure 5-2: Samsys MP9320 v2.8e RFID reader

The reader comes with software libraries to allow easy integration of the reader into new applications. The libraries are available in the Java or .NET programming languages.

5.3 Software Inventory

At the beginning of the project it was already clear that the Linux operating system needed to be used. All available implementations of trusted computing aware components were part of or built on top of the Linux kernel.

Slackware Linux [20] has been selected as the Linux distribution of choice because of the experience available with this particular distribution. The distribution choice is not very critical because most open source modules used do not require a specific Linux distribution to function. The IBM Trusted Client for Linux (TCFL) [21] software module is an exception. It has been designed for exclusive use with the Red Hat Fedora Core [22] distribution. During the testing phase of TCFL and Enforcer [23][24] a dual boot situation was created to support the testing of TCFL. The IBM Trusted Client for Linux is discussed in section 5.4.4. Enforcer is covered in more detail in section 5.4.3.

A second main decision was the choice of the programming language to use for the implementation of the trusted reader application. There are a number of factors which we need to consider:

- As discussed, Linux is the operating system of choice on which the trusted reader application will be running.
- The trusted reader application will need to interface with the libraries of the RFID reader, which are available in Java or .NET.
- The trusted reader application will need to interface with the Trousers Trusted Computing Group Software Stack (TSS) [25] which is implemented as an application library using the C programming language. Trousers is presented in section 5.4.6.

Based on these restrictions we conceived three possible scenarios:

1. We develop the trusted reader application in the C programming language which allows easy integration with Trousers and use the Java native interface to interact with the Java RFID library.
2. We develop the trusted reader application in the Java programming language which allows easy use of the Java RFID library and use the Java native interface to interface with Trousers.
3. Write the trusted reader application in the C programming language which allows easy integration with Trousers and implement the protocol of the reader ourselves so that the Java RFID libraries need not be used at all.

We have opted for the second scenario for a number of reasons:

- There is a well supported Java development kit available for Linux combined with the Eclipse IDE [26]. The combination will form a good programming environment on the Linux platform. On the other hand, Vim [27] or Emacs [28] and GCC [29] offer a good C development environment as well.
- Java is considered to be a safer programming language than C but this is a tradeoff with execution speed. However we have speedy hardware and our application does not have timing critical requirements.
- Java has built-in support for easy construction of graphical user interfaces. We would need to find a good GUI toolkit to use with C otherwise.

- There is more programming experience with Java which should improve the design and speed of the implementation if this language is chosen.
- Opting for the third option would require implementing the reader protocol in C. The implementation would be a project on its own. Needless to say this would take too much time for little gain.

Note that there is an open source project that features an implementation of .NET for non Microsoft platforms called Mono [30], which is Spanish for monkey. It is not clear how mature the Mono project is at present although some larger applications are known to work using Mono on top of the Linux kernel.

5.4 Integration of the chain of trust software stack

In chapter three it has been discussed how the trusted platform module in combination with a chain of trust can be used for platform integrity measurement. So far we have only looked at the theoretical workings of this concept. In this chapter we will describe the actual implementation of the chain of trust and the issues that we faced using it in practice. We will traverse the chain of trust stack as displayed in figure 3-1 using a bottom-up approach. At each layer we will discuss what software elements we chose to integrate into the chain. If there are multiple options we provide a comparison between the options and motivate our selection.

5.4.1 BIOS and Trusted Platform Module

The first two elements of the stack are the TPM and the BIOS. These are hardware components available from multiple vendors. An IBM based solution (IBM ThinkPad T42 laptop with an Atmel TPM) has been chosen for this project because these components are known to be well supported by existing trusted computing software, which makes sense because IBM was one of the earliest adopters of Trusted Computing technology.

The BIOS is explicitly mentioned in figure 3-1 because it plays an active role in the secure bootstrap process. The BIOS contains a non-updatable portion called the BIOS Boot Block (BBB). This BBB contains executable code that is always executed first on the platform to prevent tampering by other running software.

The BBB is responsible for the measurement of the BIOS itself and consequent measurements of additional platform configuration data like motherboard configuration, option ROMs plus configuration and finally the master boot record (MBR). The MBR contains the first executable part of the boot loader. This is the moment that the execution is handed over from the BIOS to the boot loader. From this point on it is the responsibility of the boot loader to continue the chain of trust and pass it on to the operating system. We will discuss how the chain is continued by the trustedGRUB boot loader in the next paragraph.

The BIOS is also adapted to provide an interface to limited trusted computing functions. This interface is needed as it is not possible to communicate with the TPM directly in this stage of the boot process. The interface is very lightweight and offers only the minimal functionality required for the measurement of the environment. A detailed description of the BIOS interface and the specific implementation details for personal computers can be found in [31]. This interface is used by the trustedGRUB boot loader.

5.4.2 TrustedGRUB boot loader

The GRand Unified Boot loader (GRUB) [32] is a universal boot loader that can boot a large variety of operating systems, open source and proprietary like Linux, the various BSD derived operating systems and all windows installments. GRUB supports nifty features like a graphical user interface, network booting and built-in file system support.

TrustedGRUB [33] is a modified version of the GRUB boot loader provided by the Research Group for Applied Data Security at University of Bochum, Department of Electrical Engineering and information Sciences. They are working on the construction of a trustworthy computing platform called Perseus [34] which also implements the notion of secure booting with the help of trustedGRUB. The boot loader is used to measure itself, both the execution code and the configuration followed by the integrity checking of critical parts of the loadable operating system.

GRUB execution is divided into two stages. Stage one is merely a loader for stage two, the actual grub executable. This division is made because of the nature of the PC architecture. Stage one is a small piece of code that is stored in the master boot record of the first hard disk. The master boot record is defined to be only 512 bytes and is too small to fit everything in the first stage. Stage one is only able to load the first sector of stage two. This first sector upon execution loads in similar fashion the remaining part of stage two.

In trustedGRUB this daisy chain of stages has been adapted to continue the chain of trust. Stage one itself is already measured by the BIOS/TPM because of its presence in the master boot record. Stage one is modified to measure the first sector of stage two. The result of the SHA-1 hash of this sector is stored in PCR8. Modifying stage one has not been a simple task. Stage one is written in assembler and the trusted computing extensions require low-level interactions with the BIOS in a very primitive environment. The maximum size of stage one also proved to be a problem. To make enough room for the extra TCG functionality some code needed to be eliminated and consequently floppy drive support by trustedGRUB has been dropped.

The changes to the first sector of stage two are almost identical. The hash of the rest of stage two is calculated and stored in PCR9 followed by further execution of stage two. At this point trustedGRUB has been completely measured and loaded. TrustedGRUB is now ready to do what it does best, loading operating systems.

When an operation system is selected for booting trustedGRUB performs a hash of the kernel image and the boot parameters. The resulting hash is stored in PCR10. The fact that GRUB natively supports file systems at this point made the modifications fairly simple. Now that the measurements have been completed the kernel is booted. This is where the kernel takes over the responsibility for the chain of trust.

TrustedGRUB offers one more interesting function. It allows the specification of a so-called checkfile. In this checkfile one can list paths to certain files combined with their calculated SHA-1 hashes. The files listed in this checkfile are hashed and these hashes are stored in PCR11. The hash of the checkfile itself is also stored. TrustedGRUB can be configured to give a warning before booting if one of the values is off. Note that trustedGRUB does not prevent booting if the hashes in the checkfile do not match. It will only give a warning if so configured. These possible changes will

be reflected in the value of PCR11 and can be detected and handled at a later stage in the chain of trust.

Execution is now handed over to the operating system kernel. There are two open source projects that extend the stock Linux kernel with TCG capabilities named Enforcer and Trusted Client for Linux (TCFL). These two projects will be discussed and compared next.

5.4.3 Enforcer kernel

5.4.3.1 Introduction

Enforcer is a research project by the Department of Computer Science and the PKI lab of Dartmouth College. In the past they have done research in the area of secure coprocessors and their use in real-life applications. These coprocessors are dedicated crypto processors that can carry out cryptographic operations and are manufactured to be physically tamper resistant. The secure coprocessor can be considered as the predecessor of the Trusted Platform Module. Again IBM has been very active in this area and still offers these solutions today [35]. The Dartmouth people have written a paper [36] about hardening applications like a web server with the aid of an IBM 4758 [37][38][39] secure coprocessor. The Enforcer platform is the product of their research on trusted computing and is published in [23][24].

Dartmouth College has recently announced[40] collaboration with Sun Microsystems for contributing security features to Open Solaris, Sun's own open sourced UNIX kernel. These security features will be built on the work they have done with Enforcer. This shows that there is a definite interest of the industry in this particular research area and that practical implementations are not far away.

Its interesting to note that at the time research on Enforcer took place (mid 2003) the trusted computing hardware could already be easily obtained. IBM in particular was an early adopter and supporter of TCG technology and shipped desktops and laptops equipped with TPMs starting in January 2001[41]. Unfortunately, most of the sold chips remained unused since there was no supporting software. The people at Dartmouth took the challenge to find out exactly what this TPM could do using open source software and possibly provide a platform architecture that could be used by others to employ trusted computing in practice. The Enforcer was released on the open source development website sourceforge.net in August 2003.

At the time of development TCG Software Stacks (TSS) were not available so they had to program Enforcer using the hardware interface of the TPM instead of the much more friendly software interface of the TSS. Issues as big-endianness, complicated authentication sessions, horrendously written specifications and non-compliant TPM implementations surely presented a formidable challenge. Recently more trusted computing software stacks are being released [42]; Also Microsoft is working on integrating TPM support in the next installment of Windows, named Windows Vista. They call it the Next Generation Secure Computing Base (NGSCB) [43] or project codename "Palladium".

5.4.3.2 Architecture

Enforcer is a piece of software that enables integrity enforcement of a platform. Basically the system enforces a particular system state in the sense that it will not

allow access to or execution of newly added or modified files present in the file system since the last system state has been defined. This state is stored in a database as list of file paths and their hashes.

Enforcer comes in the form of a Linux kernel extension. It uses the Linux Security Modules (LSM) framework for interfacing with the kernel. The LSM framework is a general access-control framework that allows the creation of different security models to work without having to modify the main Linux kernel code. It basically offers a number of hooks that allows programmers to add additional logic before a decision is made about whether or not to allow access to kernel objects like tasks, files, network sockets, etc…

The Enforcer consists of two distinct parts. One part handles the initialization during the bootstrap and the other part is the runtime component that handles the integrity checking.

The idea behind Enforcer is simple. It has a signed database with file paths and hashes of these files. The files that are listed in its database are ones it needs to protect. Every time there is a file access on one of these files, e.g. a read, write or execute operation, a hash is performed on the file and is compared to the hash stored in the database. If the hash matches the one that is stored the access is allowed. If the hash does not match or is not present in the database it means that the file has been changed or newly added and is possibly malicious. The Enforcer can be configured to respond in a few ways. It can simply log the inconsistency and allow access, disallow access, halt the system or hash random data to the PCRs. This last option effectively protects any data placed in the sealed storage of the TPM. The main idea is that by locking down the file system in this way we get fine-grained control about what is able to run on top of our platform.

Enforcer checks files on more than just their hash. It can also store and check a file's user owner, group owner, file permissions, last modification time, the number of hard links to it and the size of the file.

This approach also has a disadvantage. If a system needs upgrading, e.g. a kernel upgrade or an update of our RFID application, then we need to reboot the system to a maintenance mode where the Enforcer is inactive. We can then proceed to upgrade the necessary components after which we need to regenerate our hash database and resign it with the proper key. It is the intention that this key is stored offsite, for example on a USB stick.

If any data is protected by sealed storage, it should be decrypted while the system is still in the normal active state and encrypted by the sealed storage again after the upgrade using the new PCRs. We remark that this is actually a shortcoming of the TPM design and not of the Enforcer.

The Enforcer allows us to enforce some additional things besides guarding the integrity of files.

- It can also bind certain files to specific applications. This way these files are only accessible by the defined applications when the Enforcer is active.

- Another offered functionality is to disallow the addition of new files to a directory of file system. This basically allows us to lock down a whole platform from tampering, even when an intruder has obtained administrative privileges.

Until now we have mostly covered the functionality of Enforcer but we have not discussed the way Enforcer continues the chain of trust. The Enforcer kernel behaves slightly different from trustedGRUB. The boot loader is passive in the sense that is does not prevent the loading of a modified kernel. Enforcer is different in this regard. Enforcer will check the PCRs and the integrity of its configuration files before starting up. If any of these components are tampered with, it will simply refuse to load and not boot up in enforcement mode. If the PCRs are ok but the configuration files are tainted, Enforcer will send a hash of random data to PCR #12 to make sure this is detectable through the PCR values. This arrangement makes sure that if PCR values zero to twelve match the reference values for a platform that Enforcer is active.

Since Enforcer allows us to specify exactly what is allowed to be executed in application space, we can remotely verify whether these restraints hold for the system at any given time simply by knowing if the Enforcer module is loaded or not. If it is detected that Enforcer is not active we must assume that the system is compromised as there are no restrictions on the application space anymore. If Enforcer is active we know for sure that only allowed applications are able to run on the system.

Note that when Enforcer is active, it also increases protection against misuse of our encryption keys. Encryption keys can be accessed by any application provided the PCRs match and the correct password is given. The password is a mere sequence of 20 bytes. The password could easily be retrieved as it is hard coded in a developed library that manages the communication between the trusted reader application and the TSS. When Enforcer is loaded, the PCRs are as they should be and we assume the password is easily retrievable. This situation leaves our keys open for the taking. Such an attack however, is still not possible as Enforcer will prevent the injection or execution of new and possibly rogue applications attempting to misuse our keys.

5.4.4 IBM Trusted Client for Linux (TCFL)

5.4.4.1 Introduction

From the start IBM has been a real driving force behind Trusted Computing. It is one of the major participants in the Trusted Computing Group and has it offered many of its laptop and desktop products with optional TPM modules far earlier than any of its competitors. IBM has also produced a lot of research material on the subject.

A first big step was the publication of an article in the Linux Journal in August 2003: “Take control of TCPA” [44] combined with the release of a TPM driver for Linux and a very basic companion TPM library along with some example programs. This release offered the first opportunity for non TCPA experts to play around with the TPM and get to know the Trusted Computing concepts from a practical point of view. Both TCFL and Enforcer reuse pieces from this library.

This TPM package was developed further and later superseded by the release of a specification compliant open source TSS called Trousers. Although Trousers is still in a beta stage, the software has proven stable enough for inclusion in our project. It

provides a (reasonably) friendly API to manufacture TPM aware applications. Trousers will be discussed later in section 5.4.7.

A lot of papers have been written by IBM research about using Trusted Computing primitives for the integrity measurement of platforms and applications.

- "Trusted Platform on demand (TPOD)" [45] discusses an architecture that allows remote integrity attestation of a platform and its software stack for the purpose of grid computing or secure hosting.
- The paper "Design and Implementation of a TCG-Based Integrity Measurement Architecture" [46] describes a similar architecture that maintains a list of fingerprints of executed content. This list is remotely attestable through the use of a PCR.

Using the ideas in these papers, the IBM researchers have created a software prototype Trusted Client for Linux. TCFL has been developed with a different goal in mind compared to Enforcer. TCFL was designed to protect desktop Linux clients from on-line and off-line attacks in a non invasive and transparent way. This approach differs from that of Enforcer. Enforcer is very invasive because it tries to completely lockdown a system. TCFL will allow execution of untrusted or altered applications as long as the integrity of the system is not attacked.

5.4.4.2 Architecture

TCFL also comes in the form of a Linux kernel extension using the LSM framework and consists of three modules. The Extended Verification Module (EVM) which handles the integrity checking, the Simple Linux Integrity Module (SLIM) that offers integrity containment and the Integrity Measurement Architecture module (IMA) which offers attestation of the system runtime.

5.4.4.2.1 Extended Verification Module

This module operates the same way as Enforcer does. It keeps a fingerprint in the form of a hash of all files on the platform. It stores this hash in the metadata of the file system. An encrypted signature of this metadata is calculated and stored as well to prevent tampering.

When a file is opened or executed, the hash is calculated and compared to the hash that is "on file". The result of this comparison is passed to the SLIM module and this module uses it in the decision whether to allow or disallow access. So EVM itself does not enforce anything. SLIM will take care of that.

It is possible to configure and store additional attributes for files. Things like version information could prove useful. A check could take place that the most current version of a certain application is used.

5.4.4.2.2 Simple Linux Integrity Module

SLIM is a simple Mandatory Access Control (MAC) module that offers a degree of flexibility compared to Enforcer. It divides processes and files into separate integrity classes. System processes get a higher integrity level than "risky" processes like network daemons. The same process takes place with files. System files are of a higher class than regular files. EVM is used for checking the hashes of these higher

integrity class files. If one of the attributes doesn't match the expected values, the file is considered tampered with and accessed as a lower integrity file.

What the system tries to create is a separation between the high and lower integrity classes. It enforces the following rules:

- It will allow low integrity processes to read and execute higher integrity files but not write them.
- It lets high integrity processes write lower integrity files, and it has the integrity of the process lowered on read and execution of lower integrity files.

The basic idea is that when for example a lower integrity process like network daemon is compromised by a software exploit, it will still be unable to affect any critical system file or process, even if root level access is obtained. This can be an effective protection against e.g. downloaded Trojans, viruses and other malicious programs. The system is more involved than described here but documentation is unclear and skimpy but this sketches the main principle. See [21] for more information.

5.4.4.2.3 Integrity Measurement Architecture Module

IMA is a very useful module. It allows a remote party to check what processes or files have been executed on a system, combined with their measured integrity in the form of hashes. This is basically remote attestation of the software stack. It takes integrity management into the application level or system runtime. It does this by maintaining a record of all accessed executables and system level integrity objects. It stores the file's path and hash. For every file stored it also writes the hash to a specified PCR. This way the measurement list can be checked for tampering remotely when comparing the calculated composite hash of the list with the PCR value of the list obtained through the quote command. If these values do not match, we know that something very fishy is going on and that the measurement list and consequently the entire system is not to be trusted.

5.4.4.3 Caching

As opposed to Enforcer, TCFL uses a caching technique to minimize the added overhead of integrity checking to a minimum. When a file is opened for the first time the hash is calculated and the signature of the stored hash is checked as well. While the file remains unchanged the result of this operation is cached. This takes down the overhead to a minimum.

5.4.5 Comparison Enforcer / IBM TCFL

Enforcer

- Offers a less flexible solution because it rigidly enforces a specified configuration and can only respond in predefined invasive ways. It is best used to safeguard relatively static systems like servers or embedded devices as opposed to dynamic desktop systems.
- It is harder to upgrade and maintain a system because it needs to be taken down and put into a special maintenance mode. This increases downtime.

- It does not use any caching techniques for the integrity checking. So, on each file access the file is rehashed. This can be a significant overhead for a busy system.
- Enforcer will enforce our predefined policy and not allow any altered applications or files to be accessed. It does not, however, provide a method for remotely attesting what applications have been executed on a system.

IBM TCFL

- Was designed to transparently protect a normal user Linux desktop against software attacks via e.g. downloaded Trojans and infected email attachments. It operates less invasively and correspondingly allows the execution of untrusted applications but will prevent the alteration of more critical system files.
- Does support built-in caching for the integrity measurement. So, offers better performance on busy systems.
- Is a much easier system to upgrade. The system does not have to be taken down to be upgraded. A special, trusted utility can be used for the installation of new trusted applications.
- Offers remote attestation of the application space through the keeping of a remotely verifiable measurement list.

Conclusion

It seems that TCFL is a more developed prototype than Enforcer. It is however geared towards securing Linux desktops and not our mostly static environment of the RFID reader. It seems that TCFL is not able to completely lock down a system because files can be added to the system and executed without detection. Enforcer on the other hand can. This is a functionality we need. We want to make sure that our platform is locked down completely and want to be able to detect this remotely. If Enforcer is active we know that our configured platform policy will be enforced.

We also need to make sure that applications cannot steal our secrets if the platform itself is in order. We already briefly touched upon this subject in the Enforcer section. The problem is that the password for access to the asymmetric key is hard coded into a compiled library, which could be retrieved relatively easily. Enforcer can help us out here. Because of the complete lockdown of the file system it should not be possible to mount this kind of attack.

In practice Enforcer is simpler to deploy. TCFL requires an impressive list of installation procedures and will only run on a particular Linux distribution (Fedora Core 4). Enforcer on the other hand can be installed on any distribution. Slackware Linux has been used on our test bed.

The measurement list offered by TCFL is a useful feature. It shows what files and applications have been accessed and what are the corresponding hashes, however, the measurement list is very dynamic in nature, which means that the value stored in the PCR register where the hashes are stored changes all the time as well. Hence, we cannot really tie our keys to this PCR value as it constantly changes. Therefore the sealed storage capability will not work up to the application level. Considering all these points we decided to use Enforcer in favor of TCFL.

5.4.6 Trousers software stack

Trousers is an open source Linux based implementation of The Trusted computing group Software Stack (TSS). The TSS is defined in the TCG Software Stack Specification [47]. Trousers currently implements the majority of version 1.10 golden of this specification. Version 1.2 [48] has been released recently but unfortunately is not backwards compatible.

Strictly speaking, the TSS is not part of the chain of trust as it does not perform integrity measurements of any kind. It seemed logical, however, to discuss it at the end of this chapter as it is an important part of the trusted reader platform.

The TSS allows normal applications to access and use TPM capabilities. Trousers is implemented as a regular application library and a permanently running daemon. The daemon is responsible for all communications with the TPM through the TPM driver present in the Linux kernel. Applications are not allowed to interact with the TPM driver directly. Only through the TSS should applications make use of TPM services. This is required as the TPM does not support concurrent sessions. The TSS will serialize concurrent requests to the TPM to overcome this design choice to keep the TPM a low cost device. The architecture and the API of the TSS are not easily understood. The TSS architecture consists of four distinct layers and the API description is a hefty 250 pages long.

Trousers is currently the only open source TSS implementation available. We have experimented extensively with Trousers during the learning period of getting to know the TSS API and it has proven to be a stable and very usable piece of software.

5.5 Integration issues

During the installation and integration of the different software modules, mentioned in this chapter, we have encountered some problems of which we will here report. We give a short overview of these issues as they can provide some insight into the problems that exist with using trusted computing in practice.

5.5.1 Enforcer and Trousers incompatibility

Enforcer is a kernel patch that is applied to a stock Linux kernel. The latest version of Enforcer is meant to run on Linux 2.6.5. This Linux version does not have a TPM driver built in yet. This is not a problem as Enforcer has a rudimentary driver included in the patch. Trousers, on the other hand, is designed to run on Linux 2.6.11 or higher which has a new TPM driver included that is able to interoperate with different TPM vendors. Atmel, National and Infineon TPMs are supported. The TPM driver in 2.6.11 or higher is a more developed version of the driver first released by IBM. The Enforcer driver is also based on the earlier driver but it has not been developed further.

So what we have attempted to do is to use the driver that came with Enforcer and try to get Trousers to work using the older driver. This seemed to work flawlessly until we began to execute more involved TPM functions through Trousers. For some reason these commands were not executed on the TPM and the kernel showed some vague errors about bad checksums. After looking at the source code it seems that the Enforcer people added a checksum function that inspects the incoming TPM requests. This checksum, however, was designed to only check the kind of TPM commands that Enforcer uses. When we tried to execute other TPM commands, e.g. the quote

command, we found that these commands were not accepted by this checksum function. We therefore took out the checksum function. After doing this the commands coming from the TSS worked flawlessly. As the TSS makes sure that the commands sent to the TPM are properly formatted we believe that the removal of the checksum will not cause any problems using the TPM through the TSS or by commands coming directly from Enforcer.

This problem poses an interesting question. What happens if Enforcer tries to communicate with the TPM concurrently with the TSS? In our setup Enforcer will only communicate with the TPM at boot time but it could be configured otherwise. Probably it could result in corrupted sessions with the TPM. Looking at the specifications, it is not exactly clear if the operating system can interact with the TPM after boot time.

5.5.2 Enforcer in practice

The mechanics of the Enforcer kernel have already been discussed. In practice, however, there are some issues with an evasive system like Enforcer.

Enforcer is configured to prevent the adding of new files to the file system. Of course a Linux desktop is not going to be happy about that. All kinds of processes want to write temporary files during execution but Enforcer just will not allow it. This simply breaks a lot of applications. For example, SSH will not allow remote logins because it tries to write temporary files when a connection is being established. The connections just hang. Similar problems arise during the start-up of X-windows because more popular window managers like KDE and GNOME try to open lots of temporary files. We circumvent this by using a really small window manager called windowmaker which does not need any temporary files. We partly solved this problem by making a special /tmp directory. We discuss this further on.

Enforcer also monitors hashes of files, user owner, group owner, size, permissions. If any of these properties are changed, these files become inaccessible. We found out which files simply need to be updatable like log files, configuration files, etc. We configured Enforcer to not enforce the hash, size and modification date of these files but still enforce the checking of the rest of the properties. By doing this, these files can be updated but they can never be executed because we keep enforcing the permissions of the files. We make sure that none of these files have the execute bit set. This configuration ensures that our system stays protected from executions of unchecked applications.

This statement only holds if there are no interpreters on the system. If for example a Perl interpreter is available on the system, then we can execute a Perl script using the interpreter even if the file has no execute bit set. To solve this problem we deleted all interpreters and even compilers from the system like Perl, Python, GCC and Java. But what about our Java based trusted reader software? We used a native compiler to transfer our Java program to a native executable which can be properly checked by Enforcer.

There are some other loose ends we needed to tie up. What about the /tmp directory? We configured Enforcer that this is the only place where temporary files are allowed to be written but we made some other changes as well. We made the tmp into a separate partition and mount this partition at boot time with the noexec parameter.

This means that files in the /tmp directory can never have the executable bit set, again for protecting the system from rogue executables. It is clear the Enforcer is not really suitable for enforcing a dynamic desktop environment. But for embedded systems like RFID readers, it is certainly an option.

5.5.3 Attestation Identity Keys

The TPM allows different kind of keys to be generated. E.g. signing keys for signing operations, encryption keys for encryption purposes and the mentioned attestation identity keys. These AIKs are special keys that can only be used for certain purposes. They are for example used with the quote command. Only AIKs can be used for this purpose. In practice however the output of the quote command is a signing operation. If a regular signing key could be used for this purpose it could be very easy to forge the quote command output. Simply create your own data blob that represents the composite hash of the PCR values and use the same signing key to create the encrypted blob. The decrypting party has no way to know if the decrypted data is in fact a composite hash from the output of the quote command or if the data is a random byte array. For this reason these activities are performed by different keys. This separation prevents this kind of attestation faking. To be able to perform this attack you do need to have access to the normal signing key.

The Atmel TPM used in our platform in fact does allow execution of the quote command using a regular signing key and we make use of that fact in our prototype. We will explain the reason for that shortly. The creation of an Attestation Identity Key is not as straightforward as creating a simple signing key. It requires interaction with a third party CA. The third party CA is used to vouch for this newly created identity. It ties the Attestation Identity Key to the Endorsement key of the TPM. The endorsement key is a unique key stored in the TPM and can uniquely identify a platform. The exact operation of this protocol is defined in the specifications but very unclear. Also there is no software that performs this function of the third party CA. In other words there is no way to create an Attestation Identity Key at this time. This is the reason we are working with a signing key.

We agree that this is not a very nice solution and it should be corrected as soon as this attestation identity key creation process is better understood. Because our signing key is tied tightly to our Enforcer module we are certain that the possible attack described earlier cannot take place on our trusted reader as the Enforcer module will not allow any signing key hijacking program to even execute.

Now that we have described in detail the software modules that form the chain of trust and their properties we can move to the implementation of the trusted reader application that will run on top of this chain of trust software stack.

5.6 Implementation details

5.6.1 High level overview

We will now look in detail at the actual implementation of the different modules of the trusted reader application. We will describe how the most important classes are constructed both graphically and descriptively and also have a look at the interactions between the classes of the individual modules.

5.6.2 Implementation of the reader core

5.6.2.1 Overview

The reader core module that is described in section 4.5.1 is implemented by the classes RfidInterface.java and ReaderCore.java. The RfidInterface.java class implements the RFID library interface. The ReaderCore.java class implements the RFID reader configuration and the reader core policy enforcement as depicted in figure 4-4. The ReaderCore.java class also provides the interfaces to the RFID application and the Policy Engine. Both the ReaderCore.java class and the RfidInterface.java class are shown in the UML class diagram in figure 5-3.

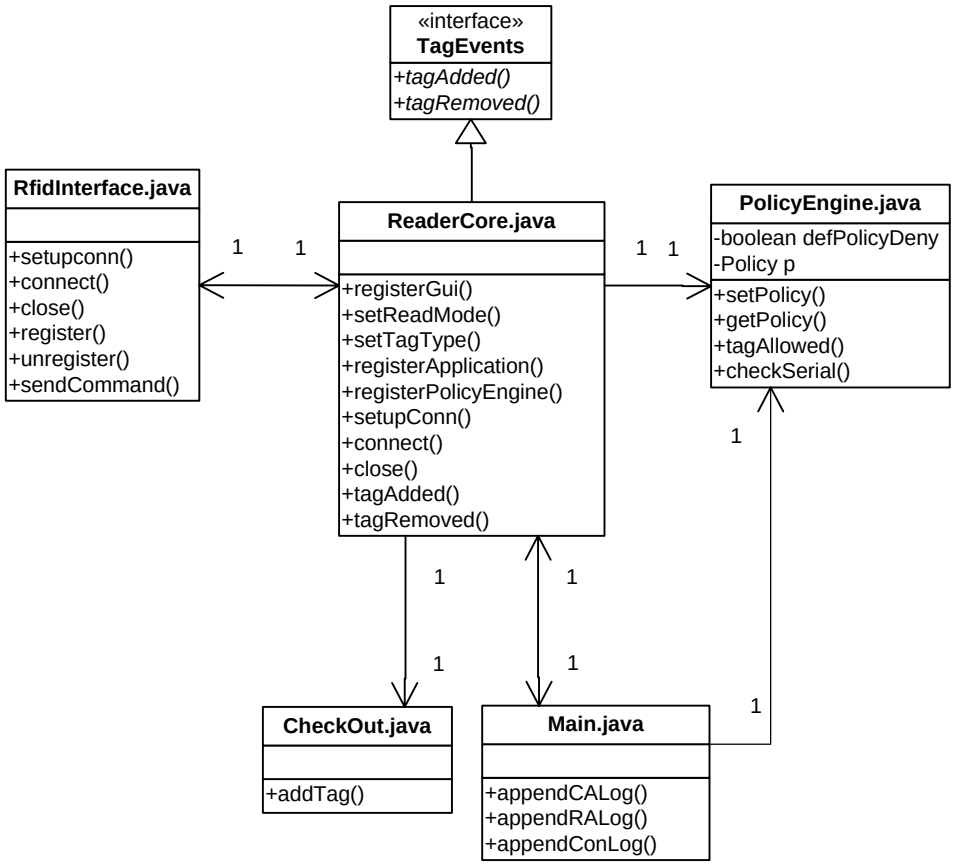


Figure 5-3: Reader core class diagram

The class diagram shows three more classes that the ReaderCore.java class depends on: PolicyEngine.java, Main.java and CheckOut.java. The PolicyEngine.java class

implements part of the policy engine module of section 4.5.2. This class file will be handled in more detail in section 5.6.3. It is useful to mention the PolicyEngine.java class here to understand how it interacts with the ReaderCore.java class.

The Main.java class file implements the graphical user interface of the trusted reader application. The ReaderCore.java class will call the functions appendCALog(), appendRALog() and appendConLog() to update the GUI with operational information. The GUI also allows policy updates and modification.

The Checkout.java is a simple RFID application to which tag information is sent if allowed by the policy engine. This is performed by executing the addTag() function which takes care that the GUI of the checkout application is updated accordingly. In our case it is a simple application simulating a shop register. Each tag identifier that is received is looked up in a small database. If the tag is found in the database, the description, the price and a small picture are displayed. The application runs as part of the trusted reader application. It could be imagined that this application is in fact a distributed logistics management system instead of an application running locally on the trusted reader platform. The checkout application is used in the workshop demo that is described in chapter six.

5.6.2.2 RfidInterface.java class

The main purpose of the RfidInterface.java class is to set up a network connection with the RFID reader. This is performed through the functions setupconn(), connect() and close() listed in table 5-1.

Table 5-1 Method Summary	
void	setupconn(String conntype, String ip, String port) Prepares a connection to the reader with the specified network settings. This can be a TCP network connection or a serial port connection. Only TCP connections are used in the trusted reader application.
void	connect() Sets up the actual network connection.
void	close() Ends the network connection.

Once the connection to the reader is set up the reader is ready to read RFID tags. When RFID tags are detected, the tag identifiers are sent back to any objects that have been registered with the register() function. All objects that are registered with this method need to implement the tagEvents Interface. This interface defines the functions tagAdded() and tagRemoved(). The tagAdded method is called when a tag enters the RFID reader's field and the tagRemoved method is called when a tag leaves the field. These methods are listed in table 5-2.

Table 5-2 Method Summary	
void	tagAdded (tagEntry tag) Method is called when tag is read when entering the reader's field. TagEntry is an object that provides a lot of information about the tag like the tag type and the tag identifier.
void	tagRemoved (tagEntry tag) Method is called when the tag was in the reader's field but is no longer detected.

In this implementation we register the readerCore.java class instantiation as a listener so that it can handle all incoming tags. As we can see in the UML class diagram in

Figure 5-3, the ReaderCore.java class indeed implements the TagEvents interface and implements the tagAdded() and tagRemoved() functions. The tagAdded() function will be covered in more detail when the ReaderCore.java class is discussed. Note that the TagEvents interface is offered by the Samsys library. More information can be found in [49].

Registered objects can be unregistered using the unregister() function. The register() and unregister() functions are described in table 5-3.

Table 5-3 Method Summary	
void	register(Object listener) Adds the object listener as a TagEventListener.
void	unregister(Object listener) Removes the object listener as a TagEventListener.

RfidInterface.java offers one more function, the sendCommand() function. This function allows us to send commands directly to the RFID reader for configuration and management functions.

Table 5-4 Method Summary	
void	sendCommand(BaseCommand command) SendCommand offers the capability to send commands to the RFID reader for management and configuration purposes. All possible commands are modeled using an extensive number of command classes provided by the Samsys library. The BaseCommand class is the parent class of these command classes.

5.6.2.3 ReaderCore.java class

As can be seen from the UML class diagram in figure 5-3, the ReaderCore.java class interacts with a lot of different classes of the trusted reader application. The most important function of this class is the reception of read tag identifiers and querying the policy engine whether or not to pass the tag identifier to our checkout application.

As has been mentioned in the previous section, the tagAdded() function and the tagRemoved() functions are called by the RfidInterface.java class when tags respectively enter or leave the RFID reader field. Only the tagAdded() function has actual code added to the function. In fact it is the tagAdded() function that implements the reader core policy enforcement state machine described in figure 4-5.

To clarify the operation of the tagAdded() function a sequence diagram has been provided in figure 5-4. This diagram shows the function calls that can take place after a tag has been read by the reader. Two different cases are shown. Case 1 shows a scenario where the tag is filtered by the policy engine. Case 2 shows a scenario where the tag is approved by the policy engine for further use by the RFID application.

Case 1: The first tag is read by the reader and the RfidInterface calls the tagAdded() function of the ReaderCore. The Readercore in turn executes the tagAllowed() function of the PolicyEngine. This functions determines if the tag is allowed to be read according to the set privacy policy. We will discuss this function call in more detail later on. For now it's sufficient to know that the tagAllowed() function returns a Boolean value to signify if the tag is to be used further or that the tag is to be dropped. In this case a false value has been returned and the tag is not to be passed on to our checkout application. The ReaderCore now executes the appendCALog() function of

the Main.java class to show the tag and policy outcome via the GUI of the application. No more actions will be undertaken. The tag identifier is not delivered to our checkout application.

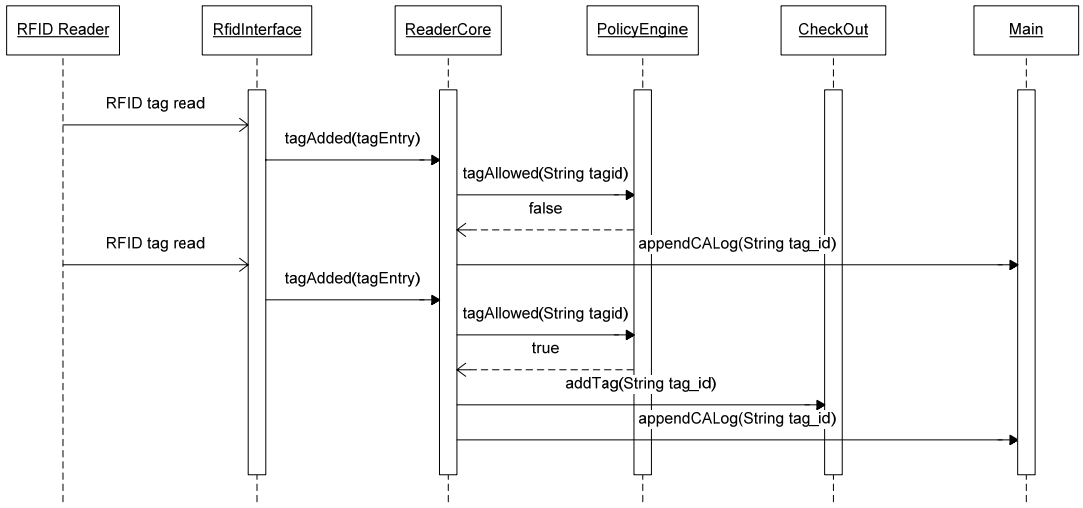


Figure 5-4: Sequence diagram reader core policy enforcement

Case 2: The second tag is read by the reader. Again the tagAdded() function is called by the RfidInterface. The ReaderCore executes the tagAllowed() function of the PolicyEngine. This time the policy allows the tag to be read and a true Boolean value is returned. Because the tag is cleared for use, the addTag() function of the CheckOut.java class is executed to provide the tag identifier to our simple checkout application. Also the appendCALog() function is executed to show the results via the graphical interface.

The ReaderCore.java class also provides two functions for the configuration of the RFID reader. The setReadMode() function can be used for setting the reading mode of the reader. The setTagType() function is used to configure the type of tags that are to be detected. The functions are listed in table 5-5.

Table 5-5 Method Summary	
void	setReadMode(int readMode) This method can adjust the reading mode of the reader. It can be set to auto read mode or polling mode. Polling mode requires explicit reading commands to detect tags while the auto read mode will continuously try to read tags. 1 = polled mode, 2 = auto read mode.
void	setTagType(tagType type) This method configures the tag type of the tags that the reader should read. Tagtype is an enumeration of {GEN2, ISO6B}. GEN2 stands for generation 2 tags while ISO6B is for reading Philips UCODE tags.

The ReaderCore.java class offers the functions setupconn(), connect() and close(), identical to the RfidInterface.java class. They are a one-to-one mapping onto these functions except for the fact that ReaderCore.java also takes care of the RfidInterface object creation and the registration of the ReaderCore object as a TagEventListener. This approach is chosen so that the ReaderCore class is the only class required to interface with from the standpoint of other classes when concerning RFID reader functionality.

The ReaderCore.java class also contains three distinct register functions: registerGUI(), registerApplication() and registerPolicyEngine(). These functions serve the purpose of acquiring the object references or handles to the class instantiations the ReaderCore.java class needs to interact with. The RFID application interface and the PolicyEngine interface modeled in the design of the ReaderCore in figure 4-4 in section 4.5.1 are in fact these object references to respectively the CheckOut.java class instantiation and the PolicyEngine.java class instantiation. The registerGUI() function takes care of the handle to the GUI class instantiation. The GUI interface (handle) is not taken into account in the design of the trusted reader application.

Table 5-6 Method Summary	
void	registerGUI(Main gui) This method supplies the handle of the Main.java class instantiation to the reader core. The handle has to be of type Main, the GUI class.
void	registerApplication(CheckOut co) This method supplies the handle of the CheckOut.java class instantiation to the reader core. The handle has to be of type CheckOut, our checkout application.
void	registerPolicyEngine(PolicyEngine pe) This method supplies the handle of the PolicyEngine.java class instantiation to the reader core. The handle has to be of type PolicyEngine, the implementation class of the policy engine.

The register functions can be made a bit more modular by defining the called methods in interfaces and then implementing these interfaces in the Main.java, CheckOut.java and the PolicyEngine.java classes. We can then use the interface types for the handles to the different class instantiations, allowing easier extension of the application in the future.

5.6.3 Implementation of the Policy Engine

5.6.3.1 Overview

The policy engine module described in section 4.5.2 is implemented by the classes PolicyEngine.java, Policy.java and PolicyRule.java. The functional building blocks depicted in figure 4-6: the policy management, the policy algorithm and the policy engine state machine modules are implemented by the PolicyEngine.java class. The privacy policy is defined using the Policy.java class. A Policy.java class instantiation is composed of zero or more PolicyRule.java class instantiations which specify the different policy rules the policy is composed of. The different classes are depicted in the class diagram shown in figure 5-5.

The class diagram also shows the ReaderCore.java class and the ConsumerAgent.java class. The ReaderCore.java class has already been discussed in the previous section. The ConsumerAgent.java class will be presented in section 5.6.4. Both classes are included in figure 5-5 to clearly show how the PolicyEngine.java class relates to these two classes.

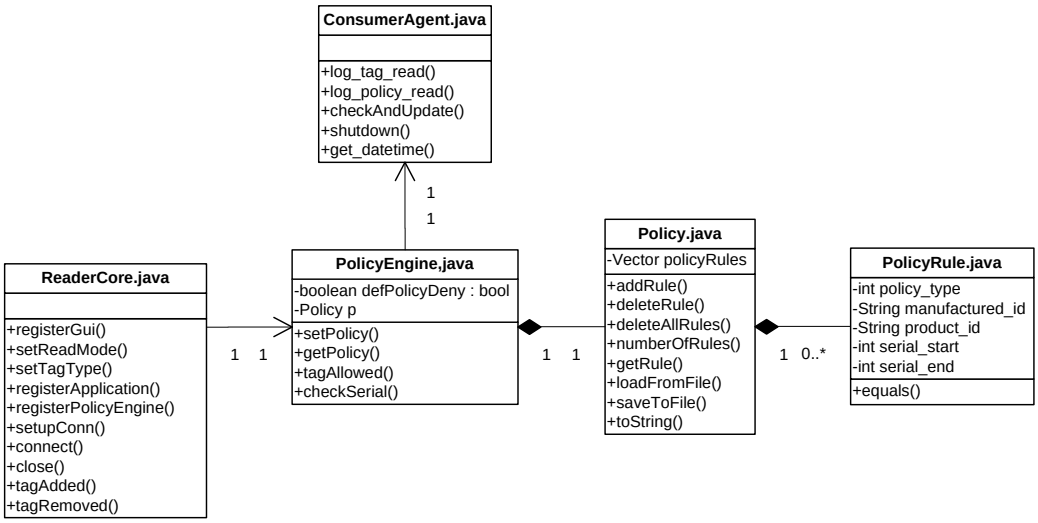


Figure 5-5: Policy engine class diagram

5.6.3.2 PolicyRule.java

A PolicyRule.java class instantiation allows us to define a policy rule according to the policy definition given in section 4.5.2.2. The integer policy_type specifies the policyrule type. The other variables are used according to the specified policyrule type. If variables remain unused, zero values or empty strings are set accordingly. All the variables in the PolicyRule.java class are set during object creation by the constructor method. The class has one equals() method, and is described in table 5-7.

Table 5-7 Method Summary	
boolean	equals(Object o) Compares the current object with the passed object o. If object o is a PolicyRule object and all the variables in both objects are equal a true Boolean value is returned. Otherwise a false Boolean value is returned.

5.6.3.3 Policy.java

The Policy.java class is used to specify a privacy policy. As mentioned in section 4.5.2.2, a policy is a composition of multiple policy rules. This is implemented in the Policy.java class by defining a vector which stores the defined PolicyRule.java objects. Furthermore the Policy.java class offers a number of methods to manage the PolicyRule.java objects stored in the vector. These methods are described in table 5-8.

Table 5-8 Method Summary	
void	addRule(PolicyRule rule) Stores the passed PolicyRule object in the vector.
void	deleteRule(PolicyRule rule) Deletes the first occurrence of the passed object rule from the vector if present.
void	deleteAllRules() Clears all PolicyRule objects from the vector.
int	numberOfRules() Returns the integer number of rules present in the vector.
PolicyRule	getRule(int index) Returns the PolicyRule with the given index from the vector.
void	loadFromFile(String filename) Loads a policy from a file with the given filename and stores the PolicyRule objects in a newly created vector. The Policy is active immediately.

void	saveToFile(String filename) Saves the active privacy policy to a file with the given filename.
String	toString() Returns a textual representation of the active privacy policy.

The file format of the files used in the loadFromFile() and saveToFile() methods also need to be defined. The format is text based. Each policy rule is defined on a separate line, terminated by a newline character. Each line starts with the integer value identifying the policy rule type. Depending on the policy rule type, additional variables are appended, using semicolons for separation of the different variables. Unused variables are not stored.

5.6.3.4 PolicyEngine.java

The main function of the PolicyEngine.java class is to determine whether or not a supplied tag identifier is permitted to be processed further or not, according to the specified policy. This determination is performed by the policy engine state machine and the policy algorithm as described in section 4.5.2. Both the state machine and the algorithm are implemented by the tagAllowed() function and the checkSerial() helper function. A description of these functions is given in table 5-9.

Table 5-9 Method Summary	
Boolean	tagAllowed(String tag) This method is passed the tag identifier of the detected tags. The implemented algorithms check if the tag is allowed to be read according to the active privacy policy. If this is the case a true Boolean value is returned, if not a false value is returned.
Boolean	checkSerial(int start, int end, String str_value) This method is a helper function for the tagAllowed() method. It checks if the serial number of the tag, given as a hexstring, falls within the range of the defined policy rule given by integers start and end.

First the tagAllowed() function checks whether or not a policy has been defined. If there is no defined policy, a default policy will be applied. This default policy is stored in the defPolicyDeny Boolean variable. The default policy, unless updated, is to deny all tags. The function terminates and returns the appropriate Boolean value.

If the policy is defined, the policy algorithm is executed. The algorithm iterates through the different policy rules contained in the Policy.java class instantiation to check if the given tag identifier matches any of the privacy rules. If any of the rules match, a true Boolean value is returned. If none of the rules match, a false Boolean value is returned.

Figure 5-6 shows a sequence diagram where the tagAllowed() function is called by the ReaderCore.java class. The PolicyEngine.java class then proceeds with the execution of the policy algorithm. The checking for a set policy is not taken into account in this diagram. We assume that a policy is defined and the tagAllowed() function proceeds directly with the execution of the policy algorithm. Furthermore the sequence diagram shows that the PolicyEngine first queries the Policy for the number of PolicyRules that have been defined. The PolicyEngine then goes into a while loop that retrieves the PolicyRule objects one by one. The while loop is depicted in the sequence diagram by means of a square box. In this while loop the PolicyRule objects are matched with the identifier that is passed along with the tagAllowed() function. When a match is found a Boolean variable is set to true. When all PolicyRules have

been checked, the Boolean result is returned. Note that the sequence diagram does not show the entire algorithm. A sequence diagram is not very suited for the accurate description of algorithms. Also the description of this algorithm is already given in section 4.5.2. The sequence diagram is partly given here to see more clearly how the PolicyEngine interacts with the Policy and PolicyRule objects.

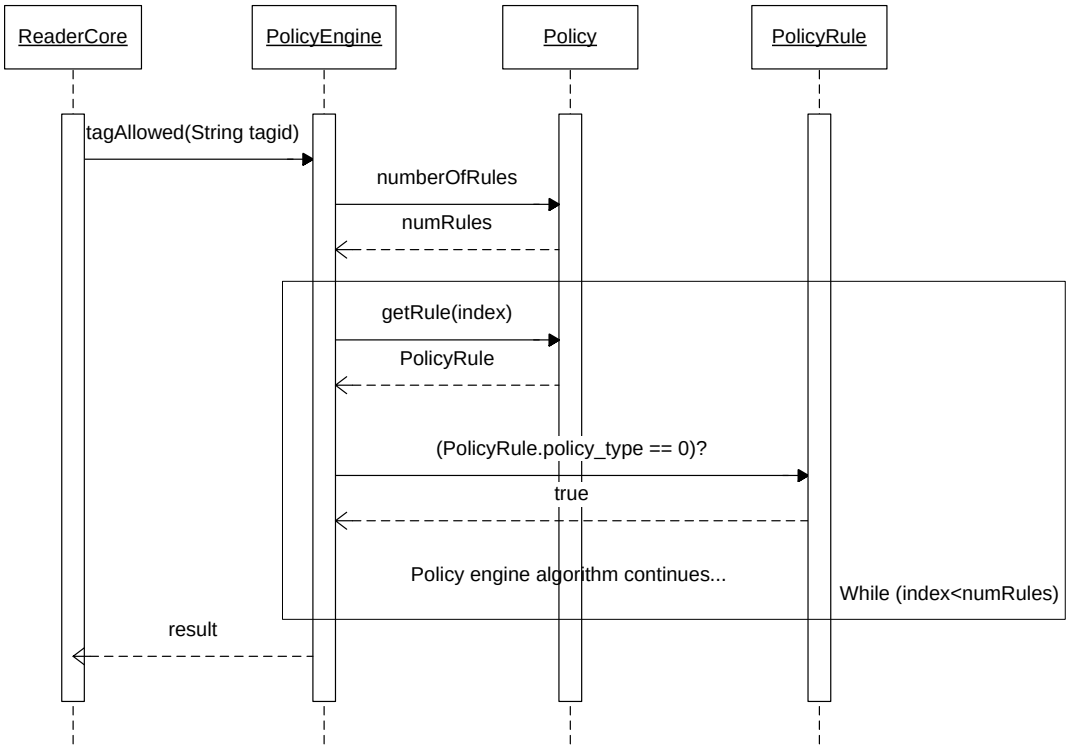


Figure 5-6: Sequence diagram policy engine policy enforcement

To manage the active policy, the PolicyEngine class offers the setPolicy() and getPolicy() functions. They are listed in table 5.10.

Table 5-10 Method Summary	
Policy	getPolicy() This method returns the Policy.java object active in the policy engine.
void	setPolicy(Policy p) This method allows the setting of a new privacy policy. The policy is also saved to a file on disk (policy.dat) so that the privacy policy can be restored on the next execution of the trusted reader application.

Until now the ConsumerAgent.java class shown in the policy engine class diagram of figure 5-5 has not been discussed. The ConsumerAgent.java class is responsible for logging all the policies that are enforced by policy engine. We will look at the implementation of the consumer agent in section 5.6.4 We currently only need to know that the ConsumerAgent.java class offers a function called log_policy_read(). This function takes a Policy object as argument and makes sure that the passed Policy object is written to the log file. The interactions between the policy engine and the consumer agent are pretty straightforward. Figure 5-7 provides us with a sequence diagram of these interactions.

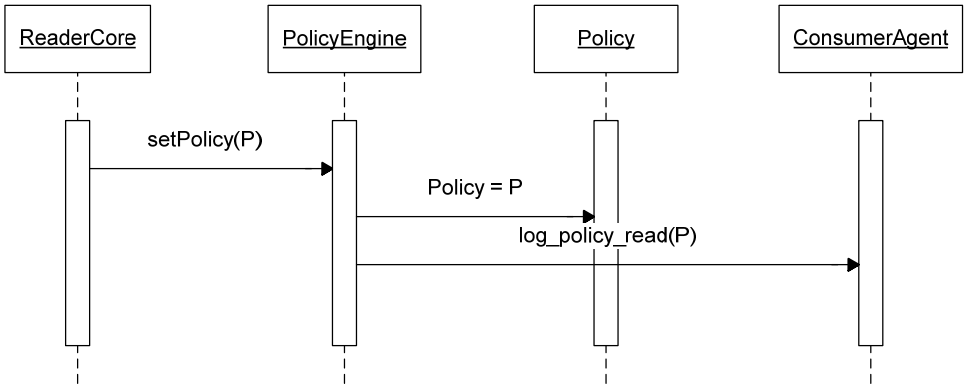


Figure 5-7: Sequence diagram of policy logging function

The policy of the policy engine can only be modified by using the setPolicy() function. This function immediately replaces the current policy with the newly supplied policy P. It is at this moment that the log_policy_read() function of the consumer agent is executed which takes care of updating the logs. More details on the logging functionality in section 5.6.4.

5.6.4 Implementation of the consumer agent

5.6.4.1 Overview

The consumer agent module described in section 4.5.3 is implemented by the ConsumerAgent.java class. The functional block design in figure 4-10 shows four distinct building blocks: the consumer agent state machine, the policy engine interface, the trusted computing software stack interface and the policy log.

The consumer agent state machine is implemented by the ConsumerAgent.java class. The policy engine interface has no actual implementation. It shows that the PolicyEngine.java class uses functionality offered by the ConsumerAgent.java class but not the other way around. The ConsumerAgent.java class is unaware of the existence of the PolicyEngine.java class. The ConsumerAgent.java class will be covered in more detail in section 5.6.4.2.

The trusted computing software stack interface is implemented by functions of the Utils.java class. These functions will be covered in section 5.6.4.3. The policy log building block models two concrete files on the harddisk: audit.log and audit.log.sig. The audit.log file is an ascii text file describing the various policies that have been enforced by the policy engine. The audit.log.sig file contains an encrypted hash of the audit.log file. The encrypted hash file allows us to detect any kind of tampering of the audit.log file as the key that is used to encrypt the hash is safely stored by the TPM.

Figure 5-8 provides a class diagram to show how the different classes relate to each other. The PolicyEngine.java class has already been discussed in section 5.6.3.4. The Utils.java class is a helper class that offers a number of static methods that are used throughout the application. These helper methods are explained in section 5.6.5.4. The functions we are interested in here are writeSignature() and checkSignature() as they provide the trusted reader application with the required TPM interoperability. We will look more closely that the implementation of these functions in section 5.6.4.3.

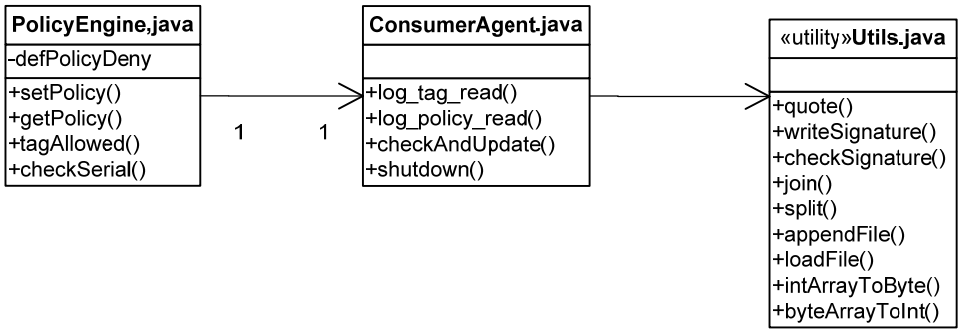


Figure 5-8: Consumer agent class diagram

5.6.4.2 ConsumerAgent.java

The consumer agent has only one purpose, the logging of privacy policies. The logging process is performed by the log_policy_read() function. The consumer agent also offers a log_tag_read() functions. This function was used to log the individual tags as well. We concluded later on that logging only the policies would suffice. The log_policy_read() and log_tag_read() functions are described in table 5-11.

Table 5-11 Method Summary	
void	log_tag_read (String tag, Boolean allowRead) This method appends the logfile with a textual representation of the tag reading event. The String tag contains the tag identifier. The Boolean allowRead specifies if the tag was allowed or denied by the policy engine. These two pieces of information are also saved in the log file.
void	log_policy_read (Policy p) This method appends the logfile with a textual description of the policy that was activated. The policy information is obtained from the parameter Policy p.

The ConsumerAgent.java class offers two more functions that haven't been discussed: checkAndUpdate() and shutdown(). CheckAndUpdate() takes care that the log file is updated by a given text only when the signature of the log file still checks out. After the log file is updated, it makes sure the signature is recalculated. Shutdown() is responsible for shutting down the trusted reader application when the application is terminated by the user or when the signature check of log fails. It also adds the shutdown event to the audit.log log file. The functions are listed in table 5-12.

Table 5-12 Method Summary	
boolean	checkAndUpdate (String update) Appends the String update to the audit.log file if the signature of the log file still matches the actual log. The signature is also recalculated when required.
void	shutdown () This method terminates the trusted reader application. If the shutdown is performed by the user, a shutdown timestamp is also added to the log file. If the trusted reader application is terminated because of detection of tampering with the log file, the log file remains unchanged.

The log_policy_read() function is called by the policy engine every time the policy is updated. Lets have a closer look at what the log_policy_read() function carries out. Figure 5-9 provides a sequence diagram of a log_policy_read() execution.

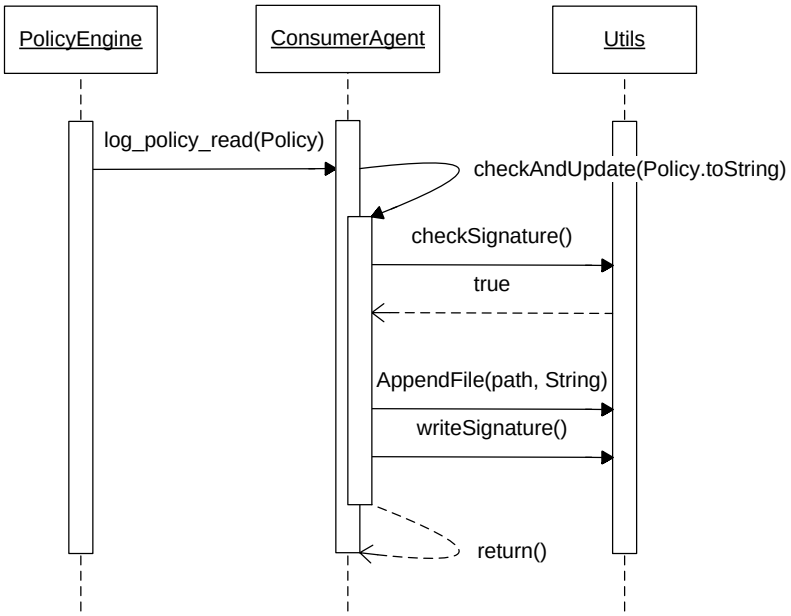


Figure 5-9: Sequence diagram consumer agent

When log_policy_read() is called, the checkAndUpdate() function is executed by the ConsumerAgent.java object instance. A String is passed as a parameter and contains a textual representation of the activated policy.

Next the checkSignature() function of the Utils.java class is executed to check if the current hash of the logfile is still correct. In this case a true Boolean value is returned, which means the signature still matches. If the signature no longer matches the log file, the application is terminated using the shutdown() function.

Now the AppendFile() function is called to update the policy log by appending the policy text. See section 5.6.5.4 for more details on this function. Because the log file has now been changed, the signature needs to be recalculated. This is performed by executing the writeSignature() function. The function recalculates the hash and re-encrypts the hash with the proper encryption key. The implementation of the checkSignature() and writeSignature() functions is discussed in section 5.6.4.3.

5.6.4.3 Utils.java

The checkSignature() and writeSignature() functions are not implemented in Java. Java offers the possibility to implement functions in other programming languages as long as the used programming language can be compiled into native platform libraries. Java calls these natively compiled libraries directly when a procedure needs to be executed. This gateway between programming languages is called the Java Native Interface. We use the JNI to implement the mentioned function using the C programming language. The implementation makes extensive use of the API offered by the Trousers software stack. The API conforms to [47]. We will refer to this API specification for the most critical function calls. Let's have a look at the implementation of these two functions using pseudo code.

start of writeSignature() pseudo code

```
void writeSignature(void)
{
```

The first step is to load the required encryption keys. Keys are stored in the TPM in a tree-like structure. The keys need to be loaded following a top-down path in the tree. Since in our case only one key has been created, we first load the storage root key (SRK) followed by our own generated signing key. This is performed on rule number 01 and 02 respectively.

```
01:   hsrk = loadSRK(srk_UUID, srkpass);
```

hsrk is a handle to a key object that is created. The handle will refer to a key object that represents the SRK. The SRK is referenced to by a unique identifier called the **srk_UUID**. The value of srk_UUID is predefined. The **srkpass** parameter is a string representing the required password required to load the key. This password is set during key creation.

```
02:   hkey = loadKEY(key_UUID, hsrk);
```

hkey is again a handle to a key object. The key object represents a signing key we have created. The UUID for our key was obtained during key creation and we keep it in a binary file called uuid.key. We use this file in the actual code but not in this pseudo-code. We assume that the **key_UUID** variable is set properly. We also pass the reference **hsrk** to the SRK key object as it is the parent of our generated key.

```
03:   inputdata = loadfile(audit.log);
```

After execution **inputdata** will point to an array of bytes that is filled by the bytes loaded from the **audit.log** file.

```
04:   hash = SHA1hash(inputdata);
```

hash is initialized to point to an array of 20 bytes that is filled with the SHA1 hash of the **inputdata** array. Now it is time to encrypt the newly calculated hash. This is performed by the hashSign() function. The official name of this function is tspi_Hash_Sign and can be found in section 3.3.3.6.2 on page 162 in [47].

```
05:   sigdata = hashSign(hash, hkey)
```

HashSign performs the encryption of **hash** array with the private key part of our generated key, referenced by **hkey**. The encrypted output is stored in the byte array **sigdata**.

```
06:   writefile(sigdata, audit.log.sig)
```

The encrypted signature containted in the sigdata byte array is written to the audit.log.sig file on the harddisk.

```
}
```

end of writeSignature() pseudo code

start of checkSignature() pseudo code

```
boolean checkSignature(void)
{
```

Similar to the writeSignature() function, we start off with the loading of the storage root key (SRK) followed by our own generated signing key. This is performed on rule number 01 and 02 respectively.

```
01:   hsrk = loadSRK(srk_UUID, srkpass);
```

hsrk is a handle to a key object that is created. The handle will refer to a key object that represents the SRK. The SRK is referenced to by a unique identifier called the **srk_UUID**. The value of srk_UUID is predefined. The **srkpass** parameter is a string representing the required password required to load the key. This password is set during key creation.

```
02:   hkey = loadKEY(key_UUID, hsrk);
```

hkey is again a handle to a key object. The key object represents a signing key we have created. The UUID for our key was obtained during key creation and we keep it in a binary file called uuid.key. We use this file in the actual code but not in this pseudo-code. We assume that the **key_UUID** variable is set properly. We also pass the reference **hsrk** to the SRK key object as it is the parent of our generated key.

```
03:   inputdata = loadfile(audit.log);
```

After execution **inputdata** will point to an array of bytes that is filled by the bytes loaded from the **audit.log** file.

```
04:   hash = SHA1hash(inputdata);
```

hash is initialized to point to an array of 20 bytes that is filled with the SHA1 hash of the **inputdata** byte array.

```
05:   sigdata = loadfile(audit.log.sig);
```

The encrypted signature stored in the file **audit.log.sig** is loaded into the sigdata byte array. Now that both the logfile and the signature are ready for use, it is time to verify if the encrypted signature will still match the actual log file. This is performed by the verifySig() function. The official name of this function is tspi_Hash_VerifySignature and can be found in section 3.3.3.6.3 on page 164 in [47].

```
06:   return(verifySig(hash, hkey, sigdata));
```

VerifySig will decrypt the encrypted hash located in **sigdata** with the public key part of the key referenced by **hkey**. The resulting hash is then compared to the newly computed hash of the audit.log file, which is stored in **hash**. If the hashes match a true Boolean value is returned, otherwise a false Boolean value is returned.

```
}
```

end of checkSignature() pseudo code

5.6.5 Implementation of the remote attestation module

5.6.5.1 Overview

The remote attestation module described in section 4.5.4 is implemented by the classes Daemon.java and ReceiveThread.java. The Daemon.java class implements the network connection interface building block depicted in figure 4-12. The remote attestation state machine is implemented by the ReceiveThread.java class. The implementation of these classes will be discussed in more detail in sections 5.6.5.2 and 5.6.5.3. The functional block design of the remote attestation module also shows a trusted computing software stack interface. This interface provides the trusted reader application with access to the TPM for retrieval of the platform integrity measurements. The interface is implemented by methods in the Utils.java class and is described in more detail in section 5.6.5.4. The policy log building block models the policy log file and signature maintained by the consumer agent. These files are accessed directly by the remote attestation module when required. The mentioned java classes are listed in the class diagram of figure 5-10.

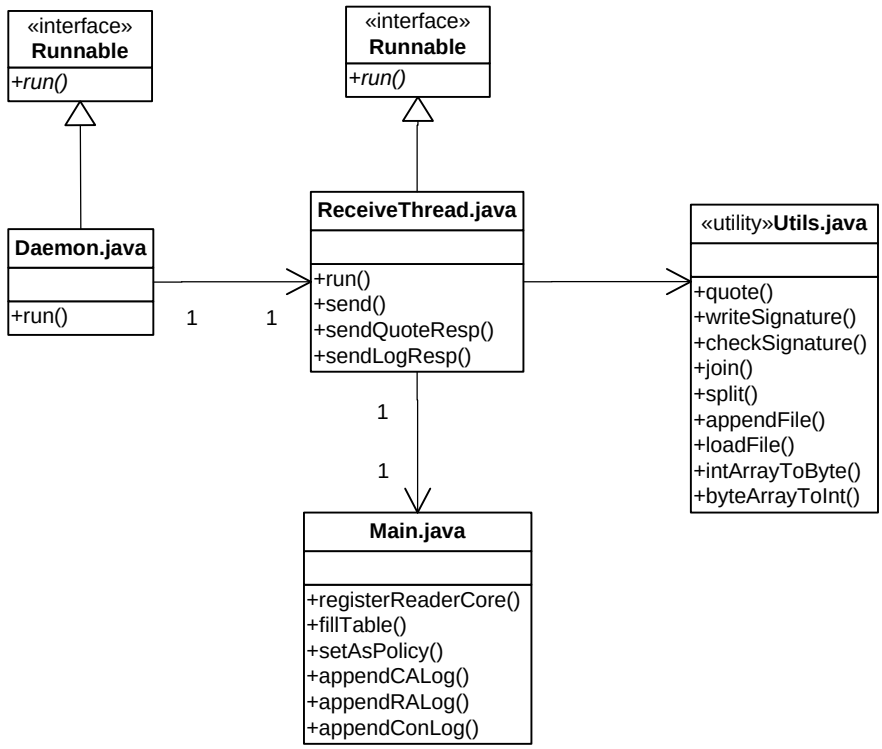


Figure 5-10: Remote attestation module class diagram

Figure 5-10 also shows the Main.java class. The Main.java GUI class has already been discussed earlier. It is listed here to show that the ReceiveThread.java class interacts with the GUI class to output connection and trusted computing generated information to the graphical interface. The GUI will show what requests are received through the established network connection and which responses are sent back to the third party auditor.

5.6.5.2 Daemon.java

The Daemon.java class is responsible for setting up a TCP listening socket on port 5000. It will monitor this port for incoming connections from third party auditors. Whenever a connection is established on this port, the Daemon.java class instantiation will launch a new thread that executes a new instantiation of the ReceiveThread.java class. This ReceiveThread.java class instantiation is designated to handle the incoming request for that particular connection. The corresponding socket is passed to the ReceiveThread.java object through its constructor.

The Daemon.java class object is created on startup of the trusted reader application and runs in its own separate thread. The Daemon.java class consists of only the run() method. The run() method is defined by the Runnable interface and is required to be implemented when passed through the constructor of a new Thread object (java.lang.Thread). The chain of events discussed so far is depicted by the first six procedure calls of the sequence diagram of figures 5-11 and 5-12. The other interactions in the diagrams will be covered further on.

5.6.5.3 ReceiveThread.java

The ReceiveThread.java class also implements the Runnable interface as the ReceiveThread.java class instantiations are executed in newly created dedicated threads. The run() method implements the finite state machine described in section 4.5.4.2. The finite state machine operates with the PDUs as they have been defined in section 4.5.4.1. Lets look at what interactions take place when a getLogRequest PDU is received. Figure 5-11 shows us the sequence of interactions that follow this event.

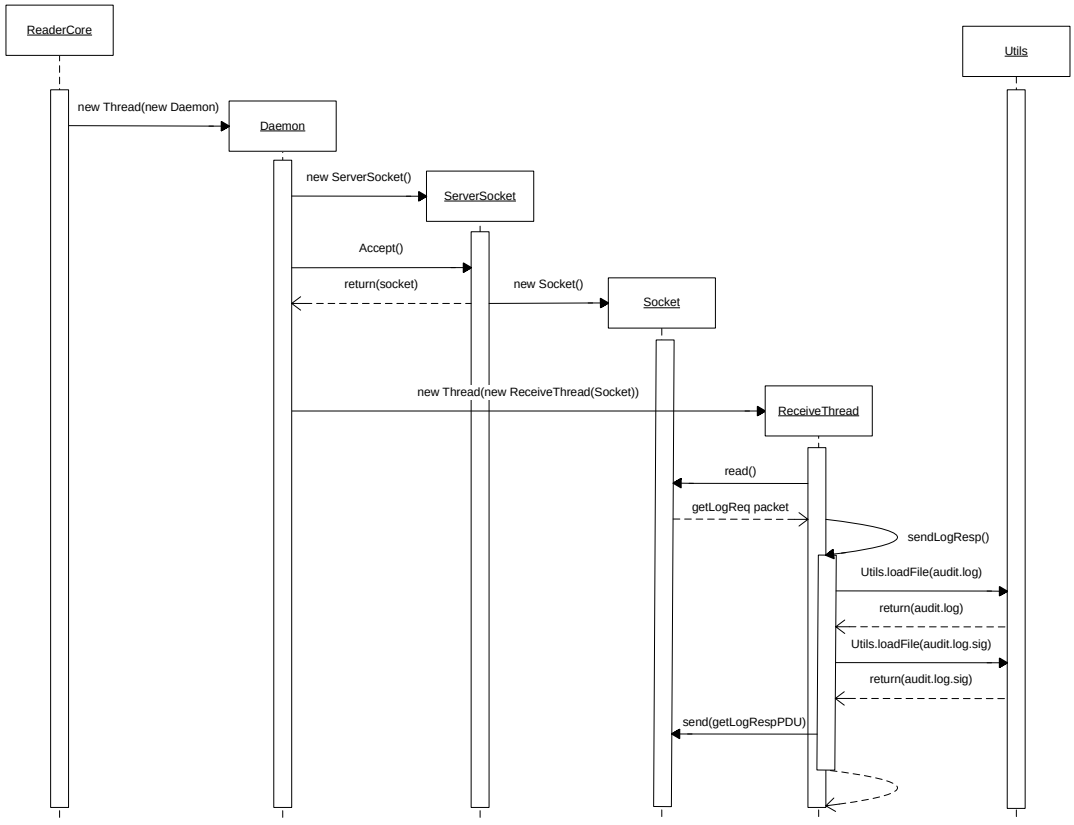


Figure 5-11: Sequence diagram remote attestation module (getLogRequest case)

When the finite state machine receives a getLogRequest PDU, a getLogResponse PDU is constructed. The PDU construction is performed by the sendLogResp() method. This method creates the appropriate PDU header and concatenates the contents of the consumer agent policy file and the encrypted signature. The retrieval of the policy file and signature is performed by the loadFile() method of the Utils.java class. The resulting PDU is then sent back to the third party auditor using the send() method. The functions send() and sendLogResponse() are described in table 5-11. Now lets have a look at the case when a getQuoteRequest PDU is received. Figure 5-12 provides us with the corresponding sequence diagram.

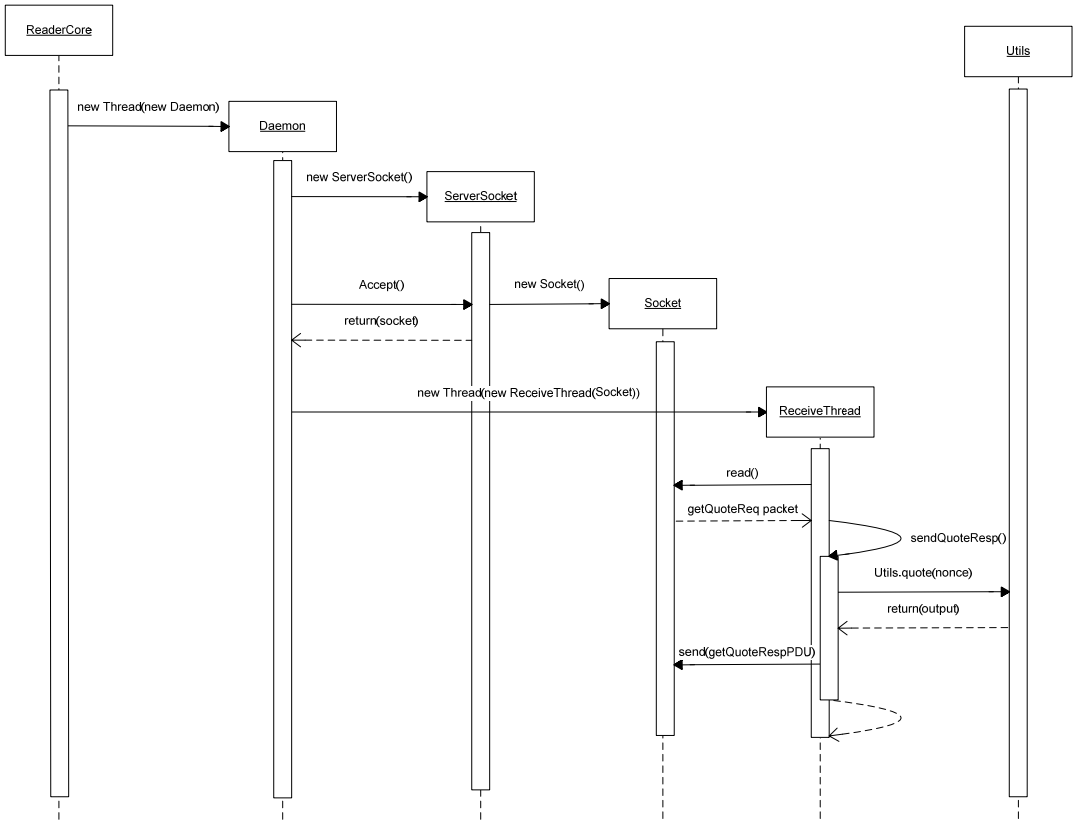


Figure 5-12: Sequence diagram remote attestation module (getQuoteRequest case)

When the finite state machine receives a getQuoteRequest PDU, a getQuoteResponse PDU is constructed by the sendQuoteResp() function. This function creates the appropriate PDU header and concatenates the header with platform integrity information obtained from the TPM. The platform integrity information is retrieved using the quote() function of the Utils.java class. The quote() function is passed the nonce that was sent along with the getQuoteRequest PDU. How the quote() function is implemented will be discussed in section 5.6.5.4. The resulting PDU is then sent back to the third party auditor using the send() method. The functions send() and sendQuoteResponse() are listed in table 5-13.

Table 5-13 Method Summary	
void	send(byte[] string) Sends the passed byte array string to the socket of the TCP connection and flushes the buffer of the socket to make sure the data it is sent immediately across.

void	sendLogResp() Creates a getLogResponse PDU carrying the consumer agent policy log and signature in its payload. The resulting PDU is supplied as a byte array to the send() function to be sent back to the third party auditor.
void	sendQuoteResp(byte[] nonce) Creates a getQuoteResponse PDU carrying the platform integrity measurements in its payload. The resulting PDU is supplied as a byte array to the send() function to be sent back to the third party auditor.

5.6.5.4 Utils.java

The Utils.java class is a class that contains a number of static helper functions that are used throughout the trusted reader application. These helper functions are listed in table 5-14. Utils.java also implements a number of methods that interact with Trousers and the TPM. These functions will be dealt with in the next paragraph.

Table 5-14 Method Summary	
static byte[]	join(byte[] b1, byte[] b2) Returns a byte array that is a concatenation of byte array b1 and b2.
static byte[]	split(byte[] b1, int offset, int length) Returns a byte array that is a subsection of byte array b1. Offset defines the starting position in b1 and length defines the number of items that are returned.
static void	appendFile(String filename, String line) Appends the line String to the file specified by the filename String.
static byte[]	loadFile(string filename) Reads the file specified by the filename String as an array of bytes and returns this byte array.
static byte[]	intToByteArray(final int integer) Converts an integer to a corresponding big-endian oriented byte array and returns this byte array.
static int	byteArrayToInt(byte[] b) Converts a big-endian oriented byte array back into an integer value and returns this integer value.
static String	printHex(byte[] array) Returns a string of the hex values of the elements of the passed byte array. The bytes are interpreted as unsigned bytes (like in C); java itself uses only signed bytes.

The remote attestation module also uses the quote() function of the Utils.java class. This function is used to retrieve the platform integrity measurements of the platform from the TPM. The quote() function is also implemented in C, making use of the JNI.

We will use pseudo-code to provide some insight into the mechanics of this function. The nonce that is passed to the function is obtained from the getQuoteRequest PDU, and is nothing more than an array of 20 random bytes, generated by the third party auditor. The quote() function will return a byte array that contains the encrypted blob containing the platform integrity measurements.

start of quote() pseudo code

```
byte[] quote(byte[] nonce)
{
```

The first step is to load the required encryption keys. Keys are stored in the TPM in a tree-like structure. The keys need to be loaded following a top-down path in the tree. Since in our case only one key has been created, we first load the storage root key (SRK) followed by our own generated signing key. This is performed on rule number 01 and 02 respectively.

```
01:   hsrk = loadSRK(srk_UUID, srkpass);
```

hsrk is a handle to a key object that is created. The handle will refer to a key object that represents the SRK. The SRK is referenced to by a unique identifier called the **srk_UUID**. The value of srk_UUID is predefined. The **srkpass** parameter is a string representing the required password required to load the key. This password is set during key creation.

```
02:   hkey = loadKEY(key_UUID, hsrk);
```

hkey is again a handle to a key object. The key object represents the signing key we have created earlier. The UUID for our key was obtained during key creation and we keep it in a binary file called uuid.key. We use this file in the actual code but not in this pseudo-code. We assume that the **key_UUID** variable is set properly. We also pass the reference **hsrk** to the SRK key object as it is the parent of our generated key.

Now that the keys are loaded and ready for use, we need to specify which PCR registers are to be taken into account with the generation of the "quote" or the composite hash of the integrity measurements. This is performed on line 03.

```
03:   hpcrs = setPcrComposite(PCRvalue);
```

hpcrs is a handle to a newly created PcrComposite object. The object is set to take into account the pcr values #0 till the integer value of **PCRvalue**. We chose to incorporate all 16 PCR values and initialize PCRvalue with the integer value 15. Now we are ready to execute the actual quote command. The specification of the quote command (official function name tspi_TPM_quote) can be found in section 3.3.3.4.28 on page 137 in [47]. The quote command is executed on line 04 of the pseudo-code.

```
04:   output = quote(hkey, hpcrs, nonce);
05:   return(output);
```

The quote command generates a composite hash based on the values of the PCRs that are selected through the PcrComposite object with handle **hpcrs**. This composite hash is concatenated by the **nonce**. The resulting data is encrypted with the private key part of our generated key, referenced by **hkey**. The ciphertext is stored in byte array output. The output is the return value of the quote() function.

```
}
```

end of quote() pseudo code

Chapter 6: Evaluation of the trusted reader implementation

6.1 Introduction

The purpose of this chapter is to show that the implementation of several important elements of the trusted reader operate according to the set requirements. We aim to prove the working of the reader by means of results obtained from several functional experiments, written testing code and the detailed description of a workshop demonstration performed with the trusted reader prototype. The chapter consists of four distinct parts:

- In section 3.5 and section 5.4 the design and the implementation of the chain of trust has been presented. We will now show that the integrity measurement functions of the different elements that form the chain of trust indeed measure the integrity of the higher layer and that these results, in the form of hashes, are stored correctly in the PCRs of the TPM.

- The design of the trusted reader application has been specified in section 4.3 and 4.4. The implementation details of the trusted reader application are presented in section 5.6. We will now show that the critical components of the trusted reader application operate correctly by showing the results from executed testing code and performed experiments.

- Lastly we will give a detailed description of the trusted reader used in a real world scenario. We use the trusted reader as a basis for a trusted RFID checkout counter in a retail environment.

6.2 Evaluation of the chain of trust integrity measurement

Each element in the chain of trust has the responsibility to correctly measure the next link in the chain of trust. If one of the components in the chain does not perform this function reliably then the integrity of the chain is broken and the integrity of the trusted platform cannot be guaranteed. Through the manipulation of the different components we will show indeed that the corresponding hashes are changed accordingly. Note that we do not prove with mathematical rigor that the implementations are correct but we do want to show that the implementation works reliably enough for this proof of concept prototype.

The Atmel TPM used in our prototype has 16 PCRs available for integrity measurement. The first eight PCRs are reserved for pre- and post-booting and operating system use. The last eight PCRs are available for use by trusted aware applications. To understand exactly what the different PCRs represent, we give an overview of the PCRs and their defined usage in table 6-1. More detailed information about the PCRs and their definition can be found in the TCG PC specific implementation specification [50] and to a lesser degree in the TCG PC client specific TPM interface specification [31].

PCR Index	PCR Assignments
00	Integrity of the BIOS Boot Block, the regular BIOS and any option ROMs or firmware of integrated components on the motherboard.

01	Integrity of the configuration and data of the motherboard and other included hardware components.
02	Integrity of option ROMs of non-platform adapters.
03	Integrity of the configuration and data of non-platform adapters.
04	Integrity of initial program loader (IPL) code. The initial program loader code is located in the master boot record.
05	Integrity of the configuration and data of the initial program loader code. For example the harddisk geometry stored in the master boot record.
06	The description of this PCR is very unclear in the specification. It is specified as: “Events recorded to this PCR are events related to State Transitions and Wake Events”. The value of this PCR remains constant in the prototype platform and thus plays no role for the integrity of the platform.
07	Reserved for future use.
08	Integrity of the first sector of stage two of the trustedGRUB bootloader.
09	Integrity of the rest of stage two of trustedGRUB.
10	Integrity of the selected kernel image and boot parameters
11	Integrity of all the files listed in the trustedGRUB checkfile and the checkfile itself.
12	Hash of the public key used for sealing the Enforcer databases.
13	unused
14	unused
15	unused

Table 6-1: PCR assignments

6.2.1 Scenario 1 – normal boot

We start the experiment with a clean and functioning trusted reader. We boot up the system and make sure that the Enforcer kernel is loaded successfully. We then make a print-out of the values of the platform configuration registers. The values of the PCRs are listed in figure 6-1. As can be seen in the output the value of PCR #11 is all zeros. This is because during the experiment the checkfile feature of trustedGRUB was not used. Not much can be deduced from the actual values of the hashes. We will refer back to these values in the next scenarios to point out what values have changed and what actions precipitated these changes.

```
00: FF C6 2C AF 32 E0 F4 5B 61 AC 36 91 BB 09 B8 65 87 B0 C7 AD
01: 08 9D 1D 91 DF 73 EA F4 9D 5A 7E 08 B3 69 AF 2B 44 00 97 5D
02: EB B3 BA AE E7 57 4B B6 37 AA AB 67 0F 9A C1 BC EB 6F 80 F3
03: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
```

```
04: C4 80 2B 17 C1 47 1E 74 DB 0E 91 B1 70 56 B7 53 0B 73 66 CE
05: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
06: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
07: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
08: 3F F2 A6 30 63 1B C0 A4 51 34 5C 2A A3 ED 5C F9 7D E6 05 79
09: 43 F8 2B 54 B1 D8 D6 B0 9A 77 55 EA AF 16 37 82 6E 0B 9B 6D
10: B3 AA 0A 3D 4F 64 04 20 BE 0E 59 39 67 02 4B 7B 77 BB 28 60
11: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12: 45 11 59 7C 6B 63 59 BB 74 95 2E 15 8A 27 00 3F B6 17 FE 7C
13: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 6-1: PCR values – normal boot

6.2.2 Scenario 2 – changing the BIOS configuration

In this scenario we change two configuration settings in the BIOS. We enable the serial port and turn “numlock on boot” on. After rebooting the system we compare the newly obtained PCR values (see figure 6-2) to the values obtained in scenario 1. We can see that the value of PCR #01 has changed because of the bios configuration changes. This is an expected result as PCR #01 is defined to hash the configuration and data of the motherboard.

The fact that PCR #01 has changed has major consequences for our trusted reader platform. Enforcer checks the PCRs during boot time of the kernel. If not all of the PCRs match exactly, Enforcer will simply refuse to load up in enforcement mode. Enforcer will still store a hash of the public key in PCR #12. The same assessment can be made remotely. If any of the PCR values #00 to #15 are off, then we know that Enforcer will not be running. The value of PCR #12 can play a special role for determining the status of the Enforcer module. This will be covered in section 6.2.6.

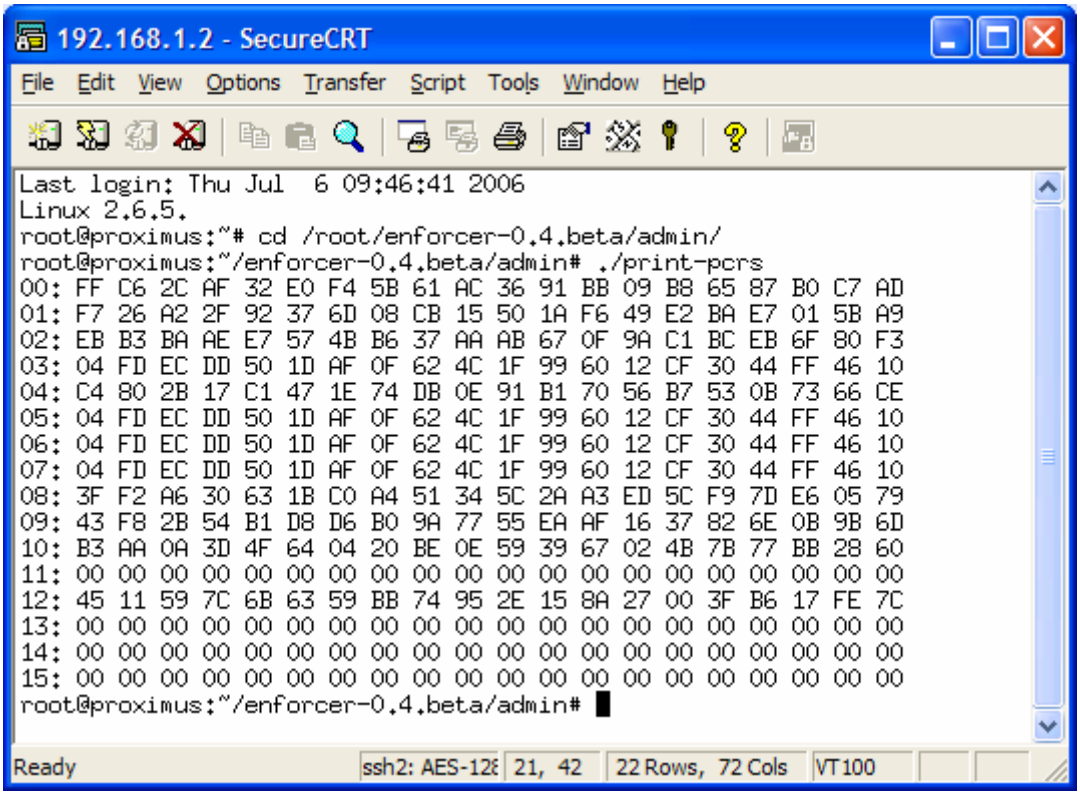


Figure 6-2: PCR values – changing the bios configuration

6.2.3 Scenario 3 – trustedGRUB adaptation

In this scenario we restore the settings that were changed in the BIOS in scenario 2. We also make a trivial modification to the trustedGRUB configuration file. We change the timeout setting of the boot loader. The timeout defines the time the boot loader waits for a selection in the actual boot menu. After the duration of the timeout, a default choice is executed.

If we examine the PCR values (see figure 6-3) after booting, it is surprising to see that PCR #01 does not have the expected value of scenario 1. PCR #01 has obtained yet another value. Somehow the settings in the BIOS are different from scenario 1. We can only guess what happened. This is illustrative for how picky the TPM really is. Even the smallest changes affect the PCR values and it is not always clear what caused the change. Sometimes reversing changes does not result in obtaining previous results, as can be clearly seen here.

We also see that the value of PCR #10 has changed. This change is due to the alterations we made to the trustedGRUB configuration file. This is expected as PCR #10 is responsible for measuring the boot parameters. Note that because of these two changes Enforcer again fails to load in enforcement mode.

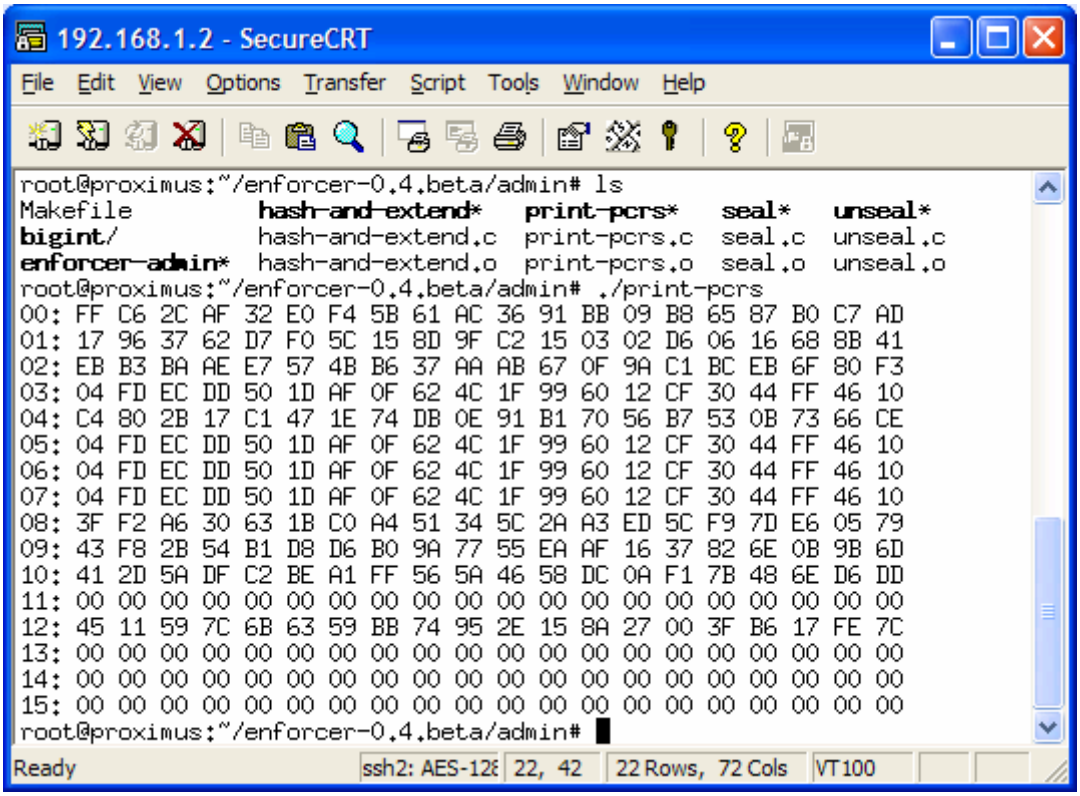


Figure 6-3: PCR values – trustedGRUB adaptation

6.2.4 Scenario 4 – changing the kernel

In this scenario we swap the kernel used in scenario 3. We use the same kernel source of Linux 2.6.11 with the version 0.4 enforcer patch as the old kernel, but we configure it to include NTFS read support. We do not change the trustedGRUB configuration file compared to scenario 3.

After replacing the kernel and rebooting the system we can again inspect the PCR values. The values are listed in figure 6-4. As we can see the value of PCR #10 has changed again compared to scenario 3 to reflect the change in kernels as expected. As a consequence Enforcer does not go into enforcement mode here either.

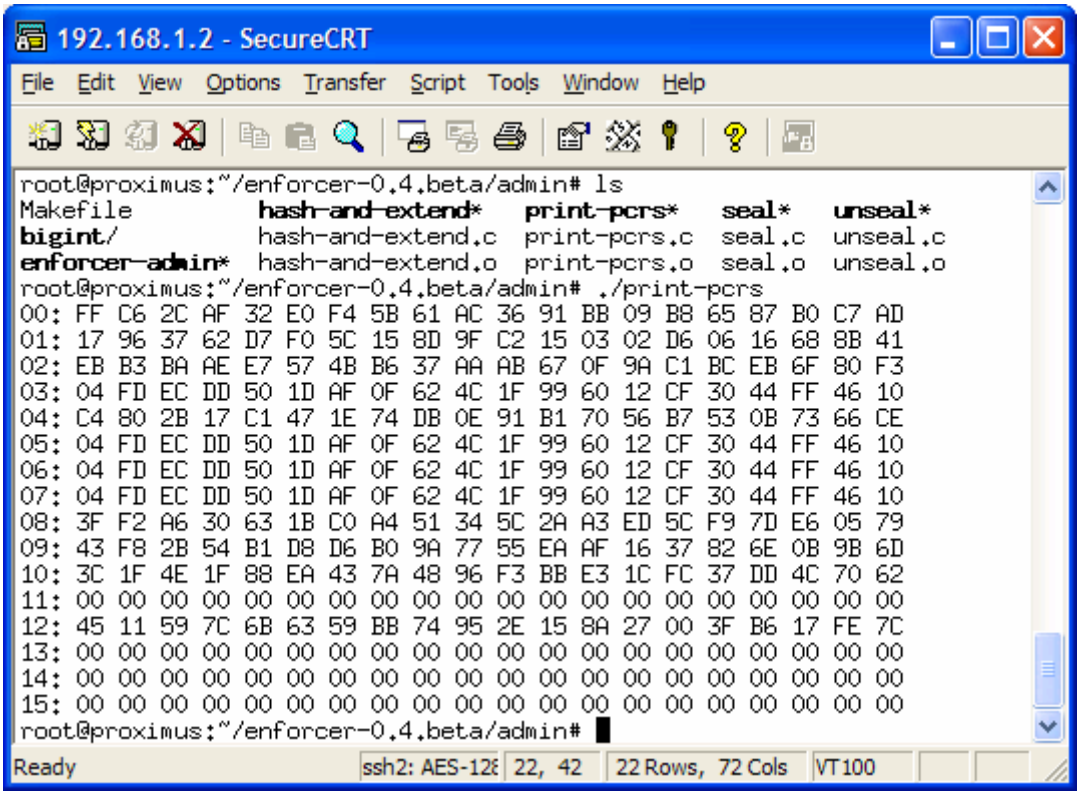


Figure 6-4: PCR values – changing the kernel

6.2.5 Scenario 5 – reset or cold boot?

In this scenario we want to show some unexpected behavior of the BIOS and TPM that we have encountered while working with the integrity measurement functionality. It seems that the TPM makes a distinction between the case where the laptop is turned on (as in being turned off for 10 seconds and then turned on) and a normal reset.

In the scenarios we have done up until now, all results have been measured using cold boots. We will now perform a warm boot by using the standard reboot command line application to reboot the Linux system. Note that the values we obtain here need to be compared to the values of scenario 1, as we have performed this test without making any other changes compared to scenario 1. If we look at the new PCR values listed in figure 6-5, we can see that the value of PCR #04 is indeed different than in scenario 1. It is not clear whether this is intentional and what causes it.

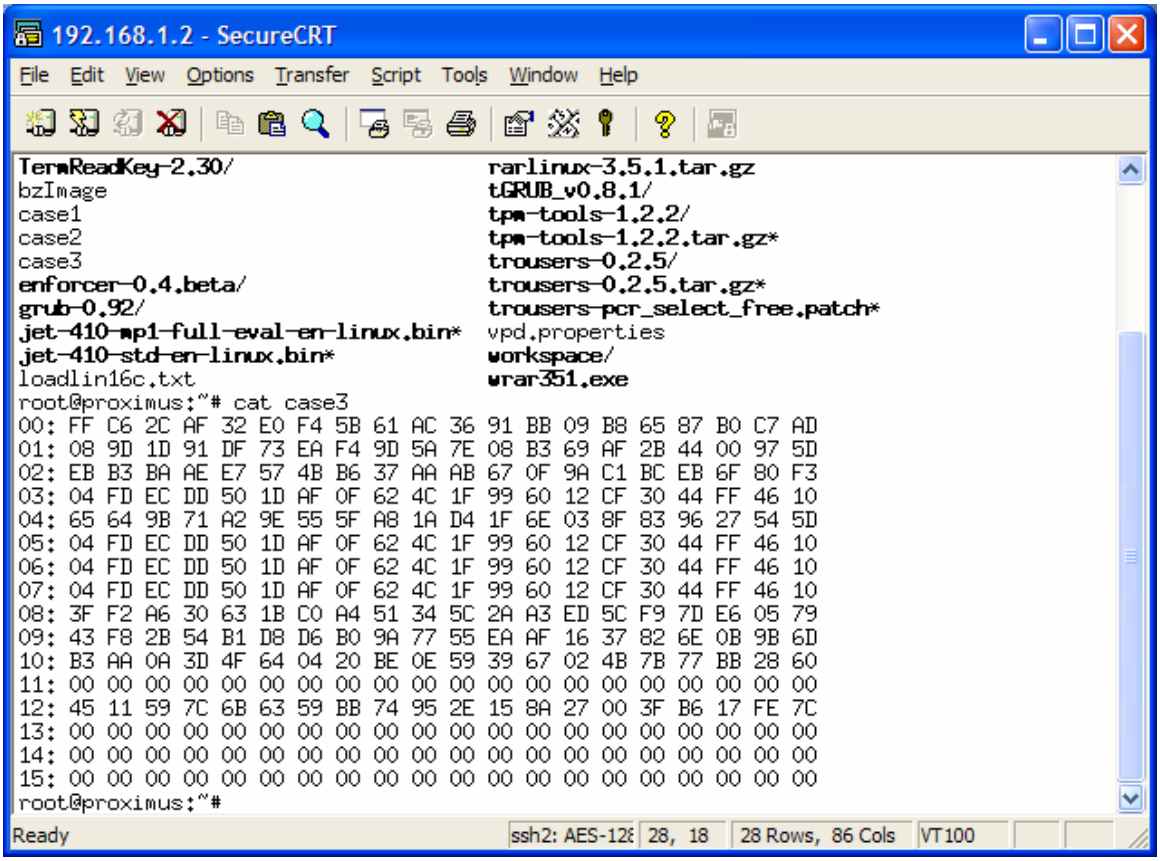


Figure 6-5: PCR values – reset or cold reboot?

6.2.6 Scenario 6 – Enforcer behavior

As we have seen in the previous scenarios, Enforcer will check the PCRs during boot of the kernel before going into enforcement mode. What hasn't been discussed yet is that Enforcer also checks its own configuration files for tampering detection through encrypted hash files. What will happen if the PCRs check out, but some of the configuration files are tampered with?

To show the behavior we intentionally adapt one of the hashes of an Enforcer configuration file so that the check will fail. Enforcer detects the discrepancy and will respond by not going into enforcement mode and by sending a randomized hash to PCR #12. This way we can remotely detect from the PCR values that Enforcer failed to load while all the actual PCR values prior to loading Enforcer were correct.

In figure 6-6 we show the encrypted hash in enforcer-binding.db.sig, the signature file for the binding database (enforcer-binding.db). The binding database allows us to define which applications are allowed to access which files. We added an "i" in front of the hash to corrupt it.

Figure 6-7 shows the PCR values obtained in this scenario. The value of PCR #12 indeed shows a new PCR value. The values should be compared to the PCR values in scenario 1. It can be seen that indeed all the PCR values are identical, except for the randomized PCR #12.

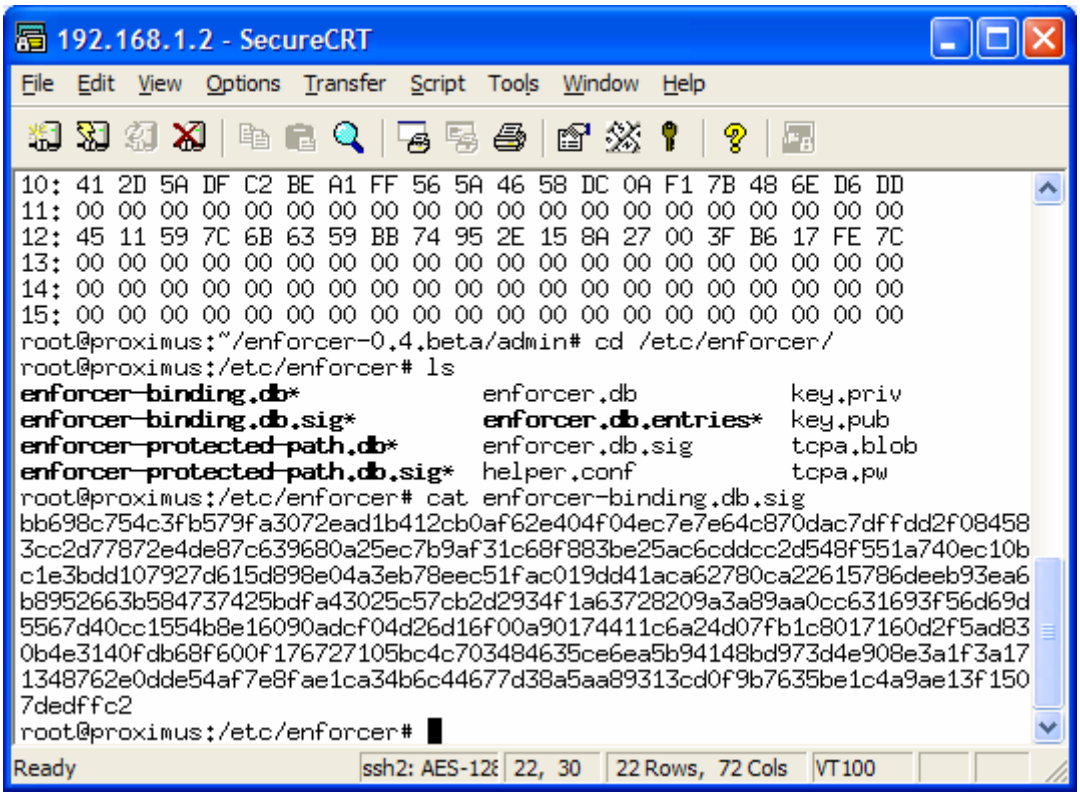


Figure 6-6: Enforcer configuration file signature

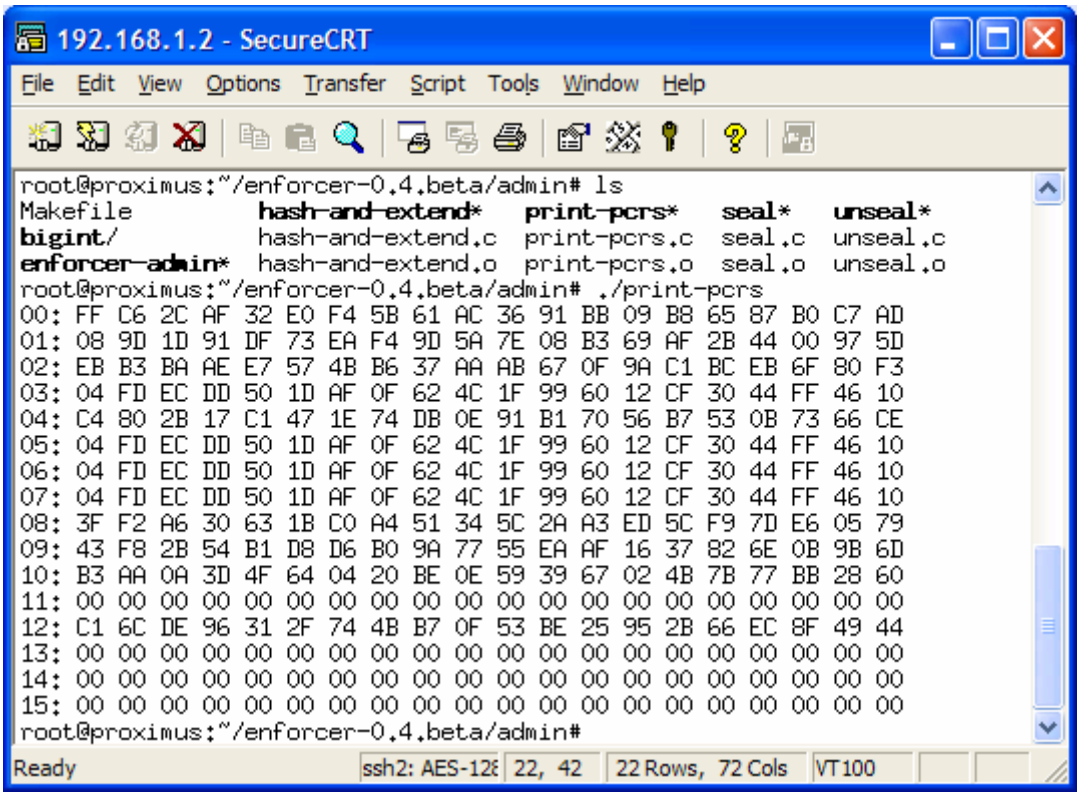


Figure 6-7: PCR values – Enforcer behavior

6.2.7 Conclusion

Summarized, we have shown the following:

- changes to the BIOS are reflected in the PCR values.
- changes to the configuration of trustedGRUB are also detected
- changes to the kernel are registered.
- it is detectable whether Enforcer is active (enforcement mode) or not, through the PCR values, and therefore also remotely through remote attestation.
- the TPM is very thorough in measuring a system's integrity. Even the smallest changes to the system are registered. Sometimes it may be too exhaustive as has been shown by the cold boot / warm boot situation.

It has been our experience that updating the partition table, master boot record and kernel parameters also has effect on the PCR values, as they should. There was only limited time available to perform these tests. Adjusting other items like the boot sector and the second stage of trustedGRUB would have been a good addition. We are, however, confident that we have shown that indeed the principle of the chain of trust is working in the trusted reader platform.

6.3 Testing code and experiment results

6.3.1 Overview

The purpose of this section is to show that the critical parts of the trusted reader application are implemented correctly. This is performed by constructing and executing testing code. Testing code is source code that consists of testing procedures which verify that certain parts of project source code is working properly. Section 6.3.2 will cover the policy algorithm used in the policy engine. Section 6.3.3 deals with the logging function of the consumer agent. Section 6.3.4 finishes off with verification of the remote attestation module.

6.3.2 Policy engine testing code

The most important element of the policy engine is the policy algorithm described in section 4.5.2.3 and section 5.6.3.4. The algorithm determines if a given tag identifier matches with the defined privacy policy. To determine that the algorithm works correctly we will specify a number of policies. For each of these policies we will expose a number of tag identifiers to the policy algorithm. For each tag the algorithm will decide whether it matches the policy or not. We can then verify these outcomes to see if the algorithm performed its duty in a correct fashion.

To set up this test we slightly modified the source code of the policy engine so that it can be run separately from the reader core, consumer agent and GUI classes. We have not made any changes to the logic. The algorithm still performs exactly the same operations as the original policy engine class.

6.3.2.1 Scenario 1 – no policy defined

In this scenario we set up the policy engine with an empty policy. The policy algorithm should then fall back to the default setting, which is to deny all tags. We execute the tagAllowed() function with seven tag identifiers. As can be seen from the testing code output, all seven tags are denied, as expected.

```
Policy Engine creation
Active policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY APPLIED
Executing tagAllowed, TAG: A1A050804A131031AA020101 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131031AA020102 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131052AA021001 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131049AA020901 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131034AA020801 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011002 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011003 POLICY ALGORITHM: DENIED
```

6.3.2.2 Scenario 2 – read all policy defined

In this scenario we set up the policy engine with a read all policy. The policy contains a single policy rule – type 0. The policy rule definitions can be found in section 4.5.2.2 for more details. We perform the tagAllowed() function with the same tag identifiers. As expected, all the tags are now allowed by the policy algorithm.

```
Policy Engine creation
Active policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
Executing tagAllowed, TAG: A1A050804A131031AA020101 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131031AA020102 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131052AA021001 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131049AA020901 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131034AA020801 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050803A155703AA011002 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050803A155703AA011003 POLICY ALGORITHM:
ALLOWED
```

6.3.2.3 Scenario 3 – manufacturer identifier defined

In this scenario we configure the policy of the policy engine to only read tags of a particular manufacturer. This is done using a single policy rule – type 1. The manufacturer identifier is set to “050803A”. Again we execute the tagAllowed() function with the same tag identifiers. The output shows that only the last two tags are allowed. This is correct behavior as these two tags are the only tags with the right manufacturer code. The manufacturer identifier in the tag identifiers has been highlighted for clarity.

```
Policy Engine creation
Active policy:
POLICY RUlES START
MANUFACTURER ID: 050803A
POLICY RUlES END
Executing tagAllowed, TAG: A1A050804A131031AA020101 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131031AA020102 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131052AA021001 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131049AA020901 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131034AA020801 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011002 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050803A155703AA011003 POLICY ALGORITHM:
ALLOWED
```

6.3.2.4 Scenario 4 – manufacturer and product identifier defined

In this scenario the policy is configured to allow tags which have both the correct manufacturer identifier and product identifier. The policy is defined by a policy rule – type 2. The manufacturer identifier is set to "050804A" and the product identifier is set to "131031". Executing the tagAllowed() function with identical tag identifiers yields no surprising results. Only the first two tags are allowed. Their manufacturer and product identifiers have been highlighted. All other tags have a different manufacturer code or a different product code or both.

```
Policy Engine creation
Active policy:
POLICY RUlES START
MANUFACTURER ID: 050804A PRODUCT_ID: 131031
POLICY RUlES END
Executing tagAllowed, TAG: A1A050804A131031AA020101 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131031AA020102 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131052AA021001 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131049AA020901 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131034AA020801 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011002 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011003 POLICY ALGORITHM: DENIED
```

6.3.2.5 Scenario 5 – manufacturer, product and serial identifier defined

In this last scenario we define a policy with a policy rule - type 3. This policy rule defines not only the manufacturer and product identifier but also a range of possible serial numbers. In this case we narrow this range to a particular serial number "131329". Note that this value is a decimal value and the value of the tag identifier is a hexadecimal value. The hexadecimal value "20101" matches the decimal value given. The manufacturer identifier is set to "050804A" and the product identifier is set to "131031". Executing the tagAllowed() function shows that only the first tag matches all the requirements. The second tag also has the correct manufacturer and product identifier but is not inside the serial number range. Again the algorithm behaves as expected.

```
Policy Engine creation
Active policy:
POLICY RUlES START
MANUFACTURER_ID: 050804A PRODUCT_ID: 131031 SNR_START: 131329 SNR_END:
131329
POLICY RUlES END
Executing tagAllowed, TAG: A1A050804A131031AA020101 POLICY ALGORITHM:
ALLOWED
Executing tagAllowed, TAG: A1A050804A131031AA020102 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131052AA021001 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131049AA020901 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050804A131034AA020801 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011002 POLICY ALGORITHM: DENIED
Executing tagAllowed, TAG: A1A050803A155703AA011003 POLICY ALGORITHM: DENIED
```

6.3.2.6 Conclusion

The testing code used in section 6.3.2 does not prove that the implementation of the policy algorithm is completely error free. It does however allow us to have more confidence in the implementation of the algorithm as the policy algorithm passed the testing of the most common scenarios.

6.3.3 Consumer agent testing code

The consumer agent defined in section 4.5.3 and 5.6.4 provides a tamper detecting policy logging mechanism. This mechanism is critical as it protects the policy auditing process of the trusted reader application. In this section we will discuss the

testing code that was used to verify that this critical function is indeed carried out correctly. We should bear in mind that the code of the policy engine and consumer agent has been adapted to allow them to be instantiated stand alone from the rest of the trusted reader application for inclusion in the testing code.

The testing code creates instances of the policy engine and the consumer agent. We then update the policy through the setPolicy() function (see section 5.6.3.4) of the policy engine class. The consumer agent should then append the audit.log file with the newly activated policy and update the signature of the log file in audit.log.sig.

After each update of the policy we will check that the log file has indeed been updated and that the signature has been recalculated. We will also verify that the signature actually matches the log file. We remark that we use the exactly the same policies as those used in the previous section.

6.3.3.1 Log update 1 – no policy defined

After the creation of the consumer agent and policy engine objects, we set up the policy engine with an empty policy. If we look at the contents of the audit.log file, then we can see that it has indeed been updated with the right policy information. The new policy is highlighted in the audit.log logfile output. Also the audit.log.sig file has been created, containing the signature listed. We check whether the signature matches the log file by invoking the Utils.checkSignature() function (see section 5.6.4.3). The output of the checkSignature() function shows that the signature does indeed match the logfile.

output testing code:

```
Consumer Agent creation
Policy Engine creation
Setting new policy:
NO POLICY SPECIFIED, DEFAULT DENY POLICY APPLIED
```

start contents audit.log:

```
09/25/06 16:11:10 read policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY
APPLIED
```

end contents audit.log:

start contents audit.log.sig:

```
73 81 AA 9E 74 0D 55 19 27 A3 EB 00 92 B6 36 5D 67 7B 8D 71 9F CA D7 89 39
8E FB F4 7A 22 38 74 AB 96 F1 AA 0A 43 07 02 63 3F 76 33 23 F0 14 FA 49 1C
DE 3B 28 B4 3A 1C 2F 06 73 B4 F3 97 14 8A B9 DC BF D1 C6 C6 7E E0 57 03 68
C5 53 46 EC CD 93 92 58 98 54 D8 D7 F7 6B 00 F9 55 04 5C 30 73 25 8D EE 9E
4F 33 52 7F 0E F8 54 B6 16 05 B4 95 BC B1 14 6A EC 14 63 D7 F0 BE A9 83 53
40 F2 AD DE 4D 7D 49 60 19 CC BA D6 59 2A 2A 17 5A E7 83 35 51 4E 4C 10 5A
8C F2 F7 41 E9 58 DE BD 5A 54 F4 8D 9F 54 9E D7 D7 A6 9B CD 7F BF 46 71 0A
B0 D3 FB 84 34 1A D7 77 53 60 F2 D5 CA 01 58 C2 35 CF AA BF 62 DF C3 10 8C
0E B8 90 02 EA 6A 9B 9C A7 6F AF 06 B6 90 64 EA AC F4 3E 0A 5F 80 60 6F 3F
08 4C DB 25 07 5A 28 3B 73 74 A9 A9 C8 93 86 CA C2 01 E1 1F 82 47 5A 3D 84
E0 46 C5 52 6C 89
```

end contents audit.log.sig:

checkSignature() function outcome:

```
Performing signature check: MATCH
```

6.3.3.2 Log update 2 - read all policy defined

Now we update the policy of the policy engine with a read all policy through the setPolicy() function. Again we can see that the log file has been updated with the new policy information. The new policy is highlighted in grey. The content of the log

signature has also been changed. Performing the checkSignature() function shows that the signature again matches the log file.

output testing code:

```
Setting new policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
```

start contents audit.log:

```
09/25/06 16:11:10 read policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY
APPLIED

09/25/06 16:11:13 read policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
```

end contents audit.log:

start contents audit.log.sig:

```
17 D3 A0 C7 D2 98 1B 89 AA 4E 57 FA 42 93 93 EE AE CF F3 44 8A 92 AA B0 BF
A6 62 B6 42 03 08 04 FC F4 1A B2 0A 06 7F 20 75 CA D0 51 97 C4 40 DD 3D 25
87 0F CC 33 CF 64 3F E6 30 19 4D 17 EE C1 C0 5F BF 4F 95 F4 F3 FB B8 C4 1D
0E 8E A3 22 0A 61 31 20 EA 11 A6 06 15 01 D2 6D B7 9E BF 0C FF 06 28 FA D7
9A 92 8A 30 12 41 7E 24 CB 94 82 13 DA EC 63 5A E9 90 2D 37 65 84 1D D2 10
1A 00 49 1F EC 57 22 0E D5 82 03 5E F8 FC 1D 9E EF 80 F1 0D 89 77 86 7E D5
A9 05 6C AA CE B7 81 7A 25 D8 6D 8E FB 25 3C 6E 4D 35 1D 88 2C 34 C8 5E 5B
B0 03 74 3C 49 1C 3E B8 CE 02 52 D4 03 93 CE 21 74 B9 E7 78 4C 56 E0 53 F6
44 29 79 71 6B 46 7F 1A 6F C4 7F E5 5E 91 D0 49 8C F5 36 8E FF 45 48 64 0C
37 3B 7B 79 FF 0C B4 34 80 6E B8 6B 1E 04 EA 72 51 A6 A5 82 E8 E0 CB 19 C5
65 93 2B 27 7A 5C
```

end contents audit.log.sig:

checkSignature() function outcome:

```
Performing signature check: MATCH
```

6.3.3.3 Log update 3 - manufacturer identifier defined

We now set a new policy that allows the reading of tags with a particular manufacturer number. The manufacturer identifier is set to “050803A”. Again the content of audit.log shows that the correct policy information is appended to the log file. Again the output of the signature is updated and proved correct by the checkSignature() function.

output testing code:
```
Setting new policy:
POLICY RUlES START
MANUFACTURER_ID: 050803A
POLICY RUlES END
```

start contents audit.log:
```
09/25/06 16:11:10 read policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY
APPLIED

09/25/06 16:11:13 read policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
09/25/06 16:11:17 read policy:
POLICY RUlES START
MANUFACTURER ID: 050803A
POLICY RUlES END
```
end contents audit.log:

start contents audit.log.sig:
```
8A AF 1A 1C F2 D0 57 BE 7E B3 EF 4D 6E 32 F4 DE 6D 99 2E F4 49 CA 40 65 AF
93 54 C6 46 9C FD E1 2B 34 F4 FF 0D BA 71 A8 1F 47 8E BB 15 29 FD A4 24 8F
DE E8 50 6D 7D 40 2B E3 BB EC 29 07 20 C0 7F 86 23 A7 EF CA 40 4C 1A B3 A1
0A 6B EC 93 E4 2E D6 41 D5 8B 6B 0B 44 98 3F 31 92 65 70 19 D4 58 5D F0 36
C1 85 94 2B B5 B5 1D 92 59 85 9F BD 98 27 8D D3 F9 D1 09 E0 97 10 DB 27 63
C4 11 15 17 A9 3F 0B 16 D4 F8 83 2C 61 A2 F7 11 55 99 03 1A BF D1 8B 4D B9
99 01 BC 98 7F C6 16 D3 FE 64 78 E5 04 79 25 CB 01 7F AC E4 C1 E9 29 74 4A
2C 36 09 97 16 F0 B1 24 78 23 70 9A 06 64 A9 35 7D 3E BC 71 FF 43 DD 91 68
25 65 C9 B0 43 3A 5C 72 9E 48 52 0F 2E E4 60 C2 00 69 3B 0C 17 C2 A7 85 33
97 C9 45 B8 17 E4 C9 86 FB A9 E8 D2 AB 66 D5 8A E9 06 E5 E4 BD C4 53 D3 8F
B2 ED 5B 58 9E C8
```
end contents audit.log.sig:

checkSignature() function outcome:
```
Performing signature check: MATCH
```

6.3.3.4 Log update 4 - manufacturer and product identifier defined

We will now test a policy defining the manufacturer identifier as “050803A” and the product identifier as “131031”. The audit.log output again shows that the correct policy information is appended to the log file. The signature is updated and matches the updated log file.

output testing code:
```
Setting new policy:
POLICY RUlES START
MANUFACTURER_ID: 050804A PRODUCT_ID: 131031
POLICY RUlES END
```

start contents audit.log:
```
09/25/06 16:11:10 read policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY
APPLIED

09/25/06 16:11:13 read policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
09/25/06 16:11:17 read policy:
POLICY RUlES START
MANUFACTURER_ID: 050803A
POLICY RUlES END
09/25/06 16:11:21 read policy:
POLICY RUlES START
MANUFACTURER ID: 050804A PRODUCT_ID: 131031
POLICY RUlES END
```
end contents audit.log:

start contents audit.log.sig:
```
13 2C 1C 9F 5D 7A 8D 08 00 6F 4E 06 4D B8 70 F0 48 C6 90 64 82 C8 2B CF 0C
EE BC FC 00 03 AE B4 16 CD 32 93 C3 97 E2 95 D4 47 4C 7A A5 83 4E BC B2 23
```

```
1B 2A 27 C5 2E 48 15 A4 7E A0 84 39 55 48 98 B3 CE FF CC 92 0A D7 95 A9 9B
99 C3 E6 81 A8 95 FF 80 93 AC FE 4F EB B9 19 3B 58 97 90 09 EB 52 99 54 7A
CB E4 09 72 6B B7 28 79 FA A6 08 31 DD 08 89 02 91 EC 30 9B D9 2C DB 05 AE
55 69 E7 D3 18 5F 71 26 8A 36 E0 94 12 6B EE 84 19 B7 CA 4C B5 13 92 24 0A
61 87 D5 64 56 8C 2A 6C D6 65 29 E0 93 85 7C 69 53 F0 3A 36 96 A2 44 AD 84
7E 54 08 7F DC 5E 76 8D B8 18 87 02 48 4C 59 05 30 54 FF F7 24 03 64 D4 9C
48 A1 91 6E E7 63 45 AE D2 FF 83 31 DD F5 5F 34 BC 11 12 76 D2 D1 C7 28 86
C7 66 EE B1 01 5F 79 0A 42 D7 AD 84 61 AB D6 ED 80 D8 37 4A B2 72 80 0D 63
63 3A A7 11 D8 84
```

end contents audit.log.sig:

checkSignature() function outcome:

```
Performing signature check: MATCH
```

6.3.3.5 Log update 5 – manufacturer, product and serial identifier defined

We will now test with a policy defining the manufacturer identifier as “050803A”, the product identifier as “131031” and the serial number identifier “020101”. The new policy is added to the audit.log output, highlighted in grey. The signature has been updated and matches the updated log file according to the checkSignature() function.

output testing code:

```
Setting new policy:
POLICY RUlES START
MANUFACTURER_ID: 050804A PRODUCT_ID: 131031 SNR_START: 131329 SNR_END:
131329
POLICY RUlES END
```

start contents audit.log:

```
09/25/06 16:11:10 read policy: NO POLICY SPECIFIED, DEFAULT DENY POLICY
APPLIED

09/25/06 16:11:13 read policy:
POLICY RUlES START
READ ALL TAGS POLICY APPLIED
POLICY RUlES END
09/25/06 16:11:17 read policy:
POLICY RUlES START
MANUFACTURER_ID: 050803A
POLICY RUlES END
09/25/06 16:11:21 read policy:
POLICY RUlES START
MANUFACTURER_ID: 050804A PRODUCT_ID: 131031
POLICY RUlES END
09/25/06 16:11:25 read policy:
POLICY RUlES START
MANUFACTURER_ID: 050804A PRODUCT_ID: 131031 SNR_START: 131329 SNR_END:
131329
POLICY RUlES END
```

end contents audit.log:

start contents audit.log.sig:

```
4B F3 43 2D DA F8 A8 86 AC FF 5C 58 D2 B2 13 C4 AE BC EC 73 69 45 5E F5 BB
69 5A 24 DF 77 F3 46 6A 3A E3 07 CF 68 CA D6 25 75 CC 7C 1A 70 1F FE 93 2F
83 7F 5F 02 21 EF 61 79 0A 7C 2A 51 9E 81 06 D6 83 B8 10 74 18 50 07 4D DE
C1 53 54 21 52 D7 C5 D2 4D DA F8 FC 11 9D 4F 3B 87 7A AB D8 78 B1 AB 4D C9
2A 9F 65 C4 2A 47 BE D9 7B B4 32 BF FD 32 2E 77 CE F8 EA 41 BD 35 66 5C FA
24 1D C9 07 FB D2 49 26 33 25 D9 44 FC 63 F2 F9 DC AA F4 F2 81 C5 CB 4C 3C
DC 49 12 58 1A 7C AE 20 60 D5 48 5B 03 D6 2A 40 6E 33 BB FE 65 8E 13 E9 0D
83 31 AF 43 7C DD 09 82 6C DE 8C F9 BE AC 92 63 E2 D4 77 FC 46 F3 8F B7 E0
4E C5 D3 29 D1 52 52 11 F5 14 E4 7D 5F C5 95 BF 13 D3 B0 06 55 A1 4D 8C D1
92 DE 84 02 F2 21 E8 90 CA 46 FC F6 D1 8B 79 68 3B 3C EB 71 13 66 E9 ED CE
54 5C FA C4 74 0B
```

end contents audit.log.sig:

checkSignature() function outcome:

```
Performing signature check: MATCH
```

6.3.3.6 Conclusion

The test cases we have performed together with the consumer agent, we show that it can handle all forms of policies using the different policy rules. It correctly appends the correct policy information to the log file each time the policy is updated. The accompanying signature is also updated promptly and correctly.

6.3.4 Experiment with the remote attestation module

In this section we want to look in more detail at the remote attestation module and the remote attestation process. The most crucial element of this module is the dynamic generation of the integrity measurements of the platform. We want to show through this experiment that the remote attestation module functions as it should. We will show the process with actual values like the used key, nonce, composite hash and encrypted blob. We also show some screenshots of the application in action.

First we provide a brief overview of all the steps that are performed during a remote attestation network query. C denotes a client-side action and S a server-side one. A theoretical treatment of remote attestation process can be found in chapter 3.

The remote attestation process step by step

1. C: generate a random nonce at the client side
2. C: send a quote request packet to trusted reader together with the random nonce.
3. S: generate the composite hash of the PCR values and concatenate with the received nonce.
4. S: encrypt this output from step three with the private key part of the asymmetric key. Note that the key can only be accessed if the PCR values are all valid.
5. S: send the encrypted blob back to the client
6. C: decrypt the blob as the client possesses the public key part of the used key.
7. C: the client has a composite hash of a known stable configuration of the trusted reader platform. The content of the decrypted blob is compared with the stored hash and the nonce that were generated. If these values match it is established that the platform integrity has not changed since the composite hash was stored on the client side.

1. Nonce generation at the client side

A random sequence of 20 bytes is generated. This so called nonce is incorporated in the encrypted reply to prevent replay with future requests. The value generated by our attestation client is shown here.

D0 24 96 EF E4 3C C5 1C BF 34 71 C4 FF 9F 66 64 C8 94 0E 65

2. List of PCRs at the moment of performing the Quote command on the trusted reader

The PCR values are shown as they were at the moment the remote attestation request was received. They represent the current state of the system. All the PCR values are taken into account by the calculation of the composite hash. The composite hash is basically a hash of the PCR values, which are hashes themselves.

```
00: FF C6 2C AF 32 E0 F4 5B 61 AC 36 91 BB 09 B8 65 87 B0 C7 AD
01: 17 96 37 62 D7 F0 5C 15 8D 9F C2 15 03 02 D6 06 16 68 8B 41
02: EB B3 BA AE E7 57 4B B6 37 AA AB 67 0F 9A C1 BC EB 6F 80 F3
03: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
04: C4 80 2B 17 C1 47 1E 74 DB 0E 91 B1 70 56 B7 53 0B 73 66 CE
05: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
06: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
07: 04 FD EC DD 50 1D AF 0F 62 4C 1F 99 60 12 CF 30 44 FF 46 10
08: 3F F2 A6 30 63 1B C0 A4 51 34 5C 2A A3 ED 5C F9 7D E6 05 79
09: 43 F8 2B 54 B1 D8 D6 B0 9A 77 55 EA AF 16 37 82 6E 0B 9B 6D
10: B3 AA 0A 3D 4F 64 04 20 BE 0E 59 39 67 02 4B 7B 77 BB 28 60
11: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12: 45 11 59 7C 6B 63 59 BB 74 95 2E 15 8A 27 00 3F B6 17 FE 7C
13: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
15: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

3. The computed composite hash of the PCR values

The composite hash represents the state of an entire platform. How the composite hash is calculated is quite involved. More details can be found in [18] concerning the TCPA_QUOTE_INFO structure. The actual value calculated in this case is displayed here.

```
01 01 00 06 51 55 4F 54 92 04 12 5B D6 F1 72 07 5C F4 77 A2
3D A8 42 CE 0E AA BA C7
```

4. Remote attestation module screenshot

Figure 6-7 provides us with a screenshot of the remote attestation part of the trusted reader application. The screenshot shows the arrival of the getQuoteRequest packet through the established network connection with the attestation client. It also shows the nonce that was the payload of the getQuoteRequest packet. The nonce is the same nonce provided in step 1.

The computed composite hash of step 3 is encrypted by a signing key. The resulting encrypted blob is again shown in the screenshot of figure 6-7. The encrypted blob is sent back in the payload of a getQuoteResponse packet to the attestation client.

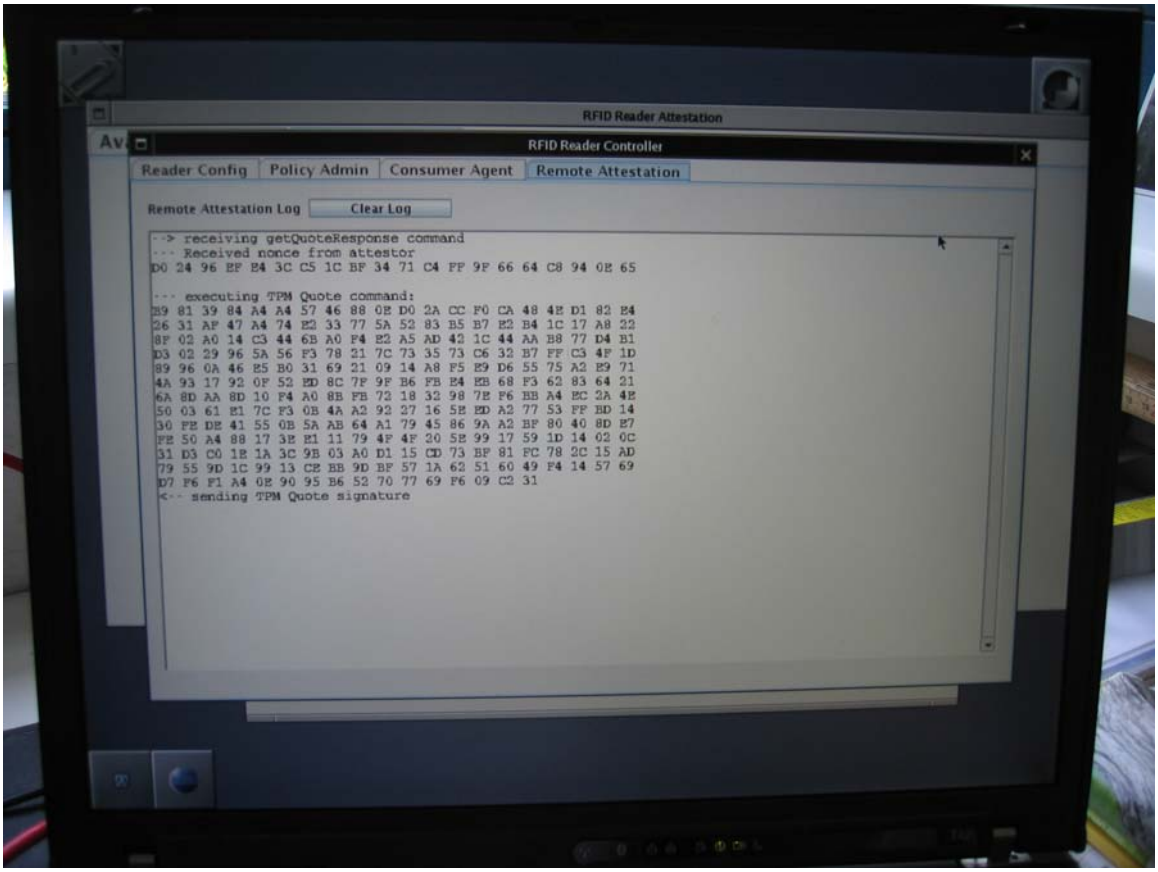


Figure 6-7: Screenshot remote attestation module

5. Attestation client screenshot

The attestation client receives the getQuoteResponse packet with the encrypted composite hash in the payload. The attestation client saves this blog to disc and decrypts it using the public key part of the encryption key. Evidently this public key needs to be supplied to the attestation client beforehand to make verification possible. We produced a utility that allows easy extraction of the public key part of an asymmetric key loaded in the TPM.

Once the decryption is performed by the openSSL crypto library, the attestation client has the composite hash and the nonce ready for verification. The hash and nonce are compared to the values that the client has stored. It has the nonce as it was the attestation client itself which has generated it. The attestation client also needs to have a composite hash value of which it knows that it represents a stable system state of the trusted reader. This composite hash value needs to be known beforehand, similar to the public key part of the used encryption key.

Figure 6-8 provides us with a screenshot of the attestation client. The attestation client has received the getQuoteResponse packet and the verification of the composite hash and nonce is correct. Based on these results we can conclude that the integrity of the trusted reader platform operates satisfactorily.

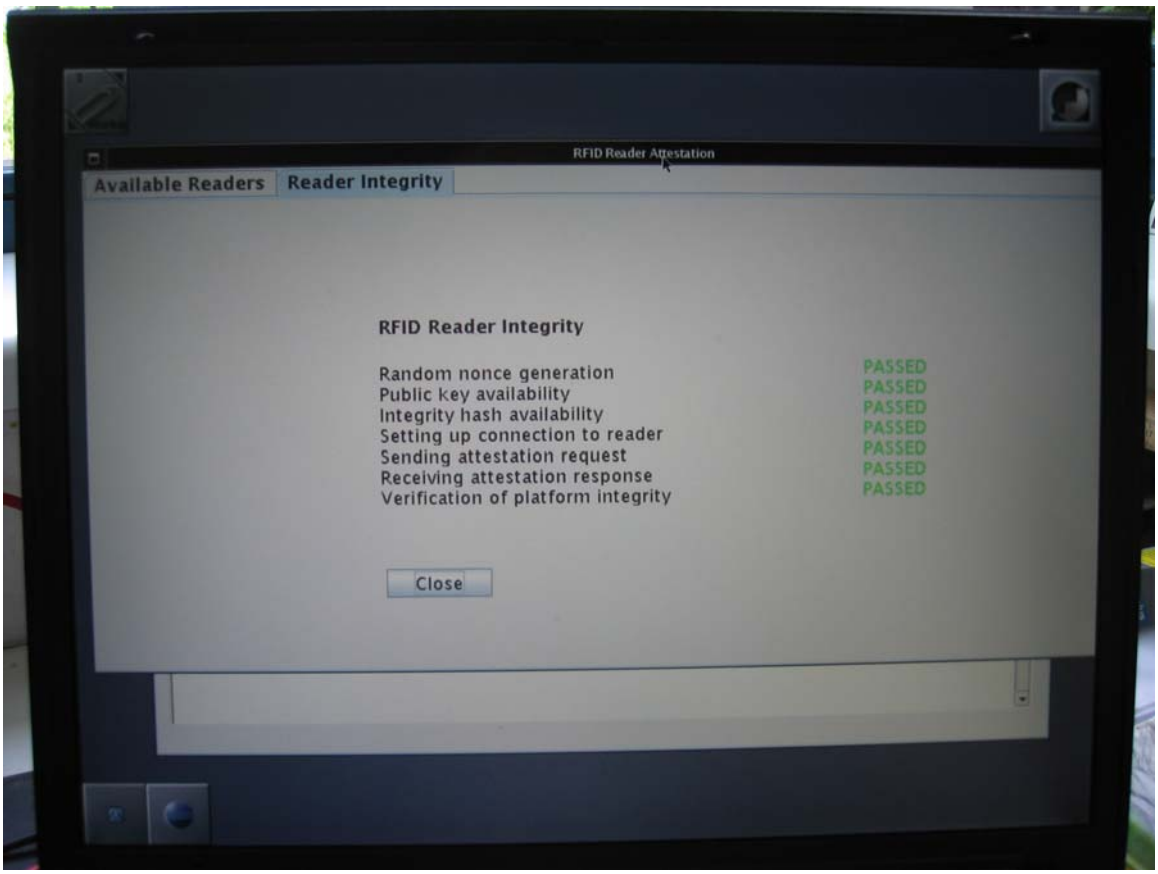


Figure 6-8: Screenshot attestation client

6.4 Description of the workshop demonstration

6.4.1 Introduction

In this section we will give an overview of the demonstration we performed with the trusted reader prototype at the ICT Innovation Open Day held on the 9th of March 2006. We refer to appendix A for the Open Day brochure. We will also mention some of the practical issues we faced during the preparations for the demo.

Because of the technical nature of trusted computing we decided to make the demonstration practical and visual. We decided to build a simple checkout register application that runs as part of the trusted reader software. This checkout register has a small database that links RFID tags with their corresponding description, price and a small picture. When registered tags pass by the reader, the products are listed on the checkout application. The user can then press checkout to purchase these items. The tag identifiers are marked as sold. If the items come into the reader field the next time, the reader will not show the items on the checkout application as an item can only be sold once.

In this case we used the database of the application itself, to store whether or not a particular product has been sold. Another option could be to store this information on the tag itself, protected by a form of encryption or password. We tried to implement this by using rewritable tags, however, the rewriting of the tags was very unreliable even at extremely close range. We decided to use the read-only tags that have the best range and use the database for keeping track of sold and unsold items.

Figure 6-9: Trusted RFID reader demonstration set-up.

In figure 6-9 we show a picture of the set-up during the BT Open Days. This is our Samsys reader and IBM laptop with four RFID reader antennas for providing a very reliable reading range of up to five meters. It seemed that the magnetic field of our reader interfered with another demo involving a radar buoy. By lowering the power of the reader and aiming the antennas further away from the other demonstration the interference was acceptable and allowed both demos to work properly.

As can be seen from the picture we used an actual plastic shopping basket in the demo. The shopping basked was filled with five shopping products. We also tried with a metal basket, but it completely shielded off the tags and was unusable. When we inserted the tags into the boxes of the products, problems occurred when the boxes contained metallic print or if the packaging itself contained metallic pieces, e.g. the aspirins were encased in plastic with a metallic strip on the bottom. The tags would not work reliably when placed next to the metallic strip. In that case we taped the tag to the outside of product. This worked reasonably well.

Another issue turned up when we stacked the product boxes. The stacking had a negative impact on the reading. In general the tags located at the bottom of the stack were not detected. We had to make sure that the boxes are well separated from each other inside the shopping basket. The angle of the RFID tag compared to the reader is also of big importance. If the tag antenna is placed in parallel with the reader antenna then the range is optimal as the antenna can absorb the most energy. If the tag is positioned on a right angle with the reader the antenna is not able to absorb any energy from the reader and the tag will not be able to reply to the reader. For this

reason it is advised to place multiple antennas at different angles to make sure that all tags are read independent of their angle. Using the four antennas at different angles, as depicted in figure 6-9, worked well.

For the demo we used and defined the following tags in the checkout database. Note that we applied spaces in the tag identifier to more clearly show the manufacturer identifier, product identifier and serial number. Also, the condoms are actually not sold in our virtual store but we do show the manufacturer and product description for this specific product (condoms in this case) to make detection more visible.

Tag Identifier	Description	Price
A1A 050804A 131052 AA021001	toothpaste	5.00
A1A 050804A 131049 AA020901	aspirin	7.50
A1A 050804A 131034 AA020801	band aids	12.00
A1A 050804A 131031 AA020101	towel	3.00
A1A 050803A 155703 AA011002	condoms	-

Table 6-2: List of registered tag identifiers in our database

We also specified a simple privacy policy in the policy engine.

Manufacturer ID	Product ID	Serial number start	Serial number end
050804A	ANY	ANY	ANY

Table 6-3: Defined privacy policy

6.4.2 The Demonstration

6.4.2.1 Scenario 1

The demonstration consists of running two scenarios. The first scenario is set up with a read all tags policy defined in the policy engine. When we pass our reader with the shopping basket, the checkout application will show the five products that are in our basket. We imagine that the fifth tag, corresponding to the condoms, represents a product that was purchased earlier in another store. This is exactly the kind of situation the privacy policy needs to prevent. It could be embarrassing for the customer if the register would show exactly what the customer is e.g. wearing, what medication they have in their purse or what products they bought before coming into the store. When we press the checkout button, the products are marked as sold in the database and the customer will not be charged again if they are within range of the checkout counter in the future.

6.4.2.2 Scenario 2

Scenario two starts with defining the privacy policy of the policy engine. We define the policy as it has been defined in table 6-1. The policy will only allow tags to be read if the manufacturer code equals 050804A. The product identifier and serial number are not important (see table 6-3). We imagine here that our store only sells products from manufacturer 050804A. Now we reset the checkout application and again pass by with our shopping basket. This time the fifth tag with manufacturer code 050803A is not shown on the register. The policy engine prevents this tag identifier from being received by the checkout application. The privacy of the customer is protected. Nobody needs to know that condoms were purchased earlier nor should any record be kept of this fact. While these scenarios are very basic, it does show the added value a privacy protecting RFID reader or architecture can offer.

Chapter 7: Conclusions and future work

In section 1.3 we discussed the research questions that need to be answered by this thesis. The first question is what the requirements are for the trusted RFID reader. The second question is what the design of the trusted RFID reader needs to look like, using the requirements as a starting point. The third question is how the reader design is mapped to the actual implementation. The final question that needs to be answered is whether or not the implementation of the reader design in fact conforms to the set requirements.

7.1 Requirements

We want to design an RFID reader that operates in a privacy-friendly manner. The design should provide a component that allows the definition of a certain privacy policy. This policy component defines which tags are allowed to be read and which tags are to be kept private. The privacy policy should be updatable at run-time. How a privacy policy can be defined will also need to be thought up.

The RFID reader should be auditable. This means that the reader requires a logging component that keeps track of current and past enforced privacy policies. These logs can then be queried for an audit of the reader. As the validity of these logs is crucial for a meaningful audit system, it should be possible to verify that the logs are not tampered with.

The validity of the policy logs can only be assured if the integrity of the trusted RFID reader platform itself can also be audited. We therefore require a system to measure the hardware and software configuration of the reader. These platform integrity measurements should be unforgeable and allow detection of tampering with any part of the trusted RFID reader platform.

It is required that audits of the trusted RFID reader can be performed remotely in a reliable way. The design should therefore supply a component which allows retrieval of the policy logs and the platform integrity measurements through the network interface.

7.2 Design

Based on the requirements, the design of the trusted reader application is divided in four distinct modules: the reader core, the policy engine, the consumer agent and the remote attestation module.

The reader core is merely responsible for interfacing with the actual RFID reader. The reader core fulfills the role of a pass-through window for tag identifiers that are read by the RFID reader to the RFID reader application. For every tag the reader core checks with the policy engine if the tag is allowed to be passed on the RFID reader application or not.

The policy engine module provides the privacy protection capabilities of the RFID reader. The policy engine contains a defined privacy policy. When queried by the reader core, the policy engine will match the read RFID tag identifier to the current privacy policy. If the tag matches the policy, a positive response is returned to the reader core. Alternatively, a negative response is returned. The reader core will take

the appropriate actions of passing through or discarding the tag identifier, based on the returned response by the policy engine.

The consumer agent is the logging module of the trusted RFID reader application. It keeps a log of all the privacy policies that are activated in the policy engine. The consumer agent interacts with the trusted computing module to provide tampering detection. An up to date encrypted hash of the log file is maintained for this purpose.

The remote attestation module provides remote auditing of the trusted RFID reader. The remote attestation module uses the trusted reader network protocol for transfer of the policy log file and the platform integrity measurements via the network interface to the auditor. The remote attestation module interacts with the trusted computing module to retrieve the current platform integrity measurements.

7.3 Implementation

The trusted RFID reader makes use of a trusted computing module for the monitoring of the reader platform and for providing tampering detection for the audit data. For correct operation of the trusted computing module, a chain of trust software stack needs to be in place. This chain of trust software stack is composed of a number of open source trusted aware components that are loaded one by one up to the kernel level. Each part of the software stack performs an integrity measurement of the higher software layer and stores these measurements inside the trusted computing module. These measurements can be provided to the auditor for platform integrity verification.

The first software element of the chain of trust is the trustedGRUB boot loader. It is responsible for booting and measuring the second software element, the Enforcer kernel. The Enforcer kernel is an adaptation of the regular Linux kernel. The enforcer module provides us with attestation of the trusted reader all the way up to application level by tightly locking down the entire operating system. This ensures that only a valid, untampered version of our trusted RFID reader application is executed on the system when Enforcer is active and the corresponding platform integrity measurements are as expected.

The second part of the implementation is the implementation of the actual trusted RFID reader application. The trusted reader application implementation has been performed using the Java programming language. The design of the trusted reader application, consisting of the reader core, policy engine, consumer agent and remote attestation module, has for the largest part been mapped one-to-one onto java classes. The interactions of the trusted reader application with the trusted platform module library have been programmed in C and are executed through the Java Native Interface.

7.4 Functionality Evaluation

We performed several evaluations to show that the implementation of the design operates according to the requirements. We performed an evaluation of the different layers of the chain of trust software stack to show that indeed each layer correctly measures the higher layer. This is important as the correctness of the integrity measurements is crucial for correct operation of the trusted platform services. Note that we have not performed an extensive evaluation of the functioning of the Enforcer kernel. We have had contact with one of the lead developers of the Enforcer project

and he has assured us that they have done extensive testing on their software prior to releasing three technical papers about the subject.

We also produced testing code to show that the implementation of the policy engine and the consumer agent is correct. We end the evaluation with an experiment that shows the correct execution of remote attestation of the trusted reader platform by a third party attestation client that is capable of verifying the validity of the platform integrity and the policy logs.

Having performed these evaluations, experiments and the demo at the BT Open Days, we can conclude that the implementation indeed conforms to the requirements as they have been defined at the beginning of this thesis.

7.5 Future work

The trusted computing module measures the integrity of software and hardware components through hashing. We have found that this is a very coarse way of measuring integrity as a single byte change will result in a completely different measurement. This method of measuring might be suited to very static environments, like embedded systems, but is not very well suited for more dynamic environments like a regular PC where patches and applications are constantly updated, installed and modified. It seems that some functionality is lacking in this regard. We think that more research should take place in this area. It will be interesting to see how future releases of trusted computing aware operating systems will handle this problem.

In the implementation of the trusted RFID reader we make use of the Enforcer kernel. This kernel is used to support remote attestation of our platform up to application level. For our purposes this solution works but Enforcer does cause a lot of problems. We feel that the trusted reader prototype can be improved by finding a solution which allows more freedom at the application level but still allows the platform to be remotely attestable up to the application level, much like the dynamic measurement list feature of IBM's TCFL.

Another issue we faced with regards to the trusted computing chip is the extra steps required for updating a system that has data sealed to its integrity measurements. Each time an update is performed, the sealed data needs to be unsealed and resealed to the new integrity measurements. This is not a procedure one wants to be performed with crucial data.

During the implementation of the trusted RFID reader we also encountered problems with the generation of identity keys. Identity keys are used to sign integrity measurement information. The process for creating these keys is not well understood and there is no supporting software available for the actual creation. This functionality needs to be better defined and understood to be of use. We used normal signing keys instead of identity keys, which according to the TCG standard should not be possible. This is definitely an aspect of the implementation that can be improved.

References

[1] S. Garfinkel and B. Rosenberg, “RFID Applications, Security and Privacy”, Addison-Wesley, July 2005.

[2] J. Landt and B. Catlin, “Shrouds of Time, the History of RFID”, The Association for Automatic Identification and Data Capture Technologies, October 1, 2001, located at: http://www.aimglobal.org/technologies/rfid/resources/shrouds_of_time.pdf

[3] K, Finkenzeller, “RFID-Handbook, Second Edition”,Wiley & Sons, April 2003.

[4] Main webpage of EPCglobal Inc, located at: http://www.epcglobalinc.org/home

[5] EPCglobal Inc., “900 MHz Class 0 Radio Frequency (RF) Identification Tag Specification”, 23 Feb 2003, located at: http://www.epcglobalinc.org/standards/specs/900_MHz_Class_0_RFIDTag_Specification.pdf

[6] EPCglobal Inc., “Class 1 Generation 2 UHF Air Interface Protocol Standard Version 1.0.9: Gen 2", January 2005, located at: http://www.epcglobalus.org/dnn_epcus/KnowledgeBase/Browse/tabid/277/DMXModule/706/Command/Core_Download/Default.aspx?EntryId=292

[7] A. Juels, “A minimalist Cryptography for low-cost RFID tags”, in Security in Communication Networks (SCN ’04), 2004.

[8] A.Juels, R.L. Rivest, M. Szydlo, “The Blocker Tag: Selective Blocking of RFID tags for Consumer Privacy”, in Eighth ACM Conference on Computer and Communications Security, pp. 103-111, ACM Press, 2003.

[9] D. Molnar, A. Soppera, and D. Wagner, “Privacy for RFID through trusted computing”, in Proceedings of the 2005 ACM workshop on Privacy in the electronic society, ACM Press. 2005. pp. 31-34.

[10] FIPS 180-2, Secure Hash Standard (SHS), August 2002, located at: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[11] Trusted Computing Group, “Architecture overview”, 2006, located at: https://www.trustedcomputinggroup.org/specs/IWG/TCG_1_0_Architecture_Overview.pdf

[12] E. Chien, “Techniques of Adware and Spyware”, Symantec, from the proceedings of the VB2005 Conference, located at: http://www.symantec.com/avcenter/reference/techniques.of.adware.and.spyware.pdf

[13] B. Balacheff, L. Chen, S. Pearson, D. Plaquin and G. Proudler, “Trusted Computing Platforms, TCPA technology in context”, Prentice-Hall, 2003.

[14] ITU-T, “Z.100: Specification and Description Language (SDL)”, August 2002, located at: http://www.itu.int/ITU-/studygroups/com17/languages/Z100.pdf

[15] Y. Byun, B.A. Sanders and C. Keum. “Design Patterns of Communicating Extended Finite State Machines in SDL”, in Proceedings of the 8th Conference on Pattern Languages of Programs, Monticello, Illinois, 2001.

[16] EPCglobal Inc., “EPCglobal Tag Data Standards Version 1.3 Ratified Specification”, April 2006, located at: http://www.epcglobalinc.org/standards_technology/Ratified%20Spec%20March%208%202006.pdf

[17] R.C. Martin, “UML for Java Programmers”, Prentice Hall, August 2003.

[18] Trusted Computing Group, “Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b”, February 2002, located at: https://www.trustedcomputinggroup.org/specs/TPM/TCPA_Main_TCG_Architecture_v1_1b.pdf

[19] T. Mcfadden. “Overview of laptops and TPMs”, 2006, located at: http://www.tonymcfadden.net/tpmvendors.html

[20] Main webpage Slackware Linux distribution, Patrick Volkerding, located at: http://www.slackware.com/info/

[21] D. Safford and M. Zohar, “A Trusted Linux Client (TLC)”, IBM, 2005, located at: http://domino.research.ibm.com/comm/research_projects.nsf/pages/gsal.TCG.html/$FILE/tlc.pdf

[22] Main webpage Fedora Core Linux distribution, Red Hat, located at: http://fedora.redhat.com/About/

[23] J. Marchesini, S.W. Smith, O. Wild, and R. MacDonald, “Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear”, Technical Report TR2003-476, Department of Computer Science, Dartmouth College, 2003.

[24] R. MacDonald, S. Smith, J. Marchesini, and O. Wild, “Bear: An open-source virtual secure coprocessor based on TCPA”, Technical Report TR2003-471, Department of Computer Science, Dartmouth College, 2003

[25] Main webpage of TrouSerS, the open-source TCG Software Stack, located at: http://trousers.sourceforge.net

[26] Main webpage of the Eclipse development platform and application framework, located at: http://www.eclipse.org/org/

[27] Main webpage of the Vim (Vi IMproved) editor, located at: http://www.vim.org/about.php

[28] Main webpage of the Emacs editor, located at: http://www.gnu.org/software/emacs/

[29] Main webpage of GCC, the GNU Compiler Collection, located at: http://gcc.gnu.org/

[30] Main webpage of the Mono project, located at: http://www.mono-project.com/Main_Page

[31] Trusted Computing Group, “TCG PC Specific Implementation Specification Version 1.1”, August 2003, located at: https://www.trustedcomputinggroup.org/groups/pc_client/TCG_PCSpecificSpecification_v1_1.pdf

[32] Main webpage of GRUB, the Grand Unified Bootloader, located at: http://www.gnu.org/software/grub/grub-legacy.en.html

[33] Applied Data Security Group, “TrustedGRUB website”, University of Bochum, located at: http://www.prosecco.rub.de/trusted_grub.html

[34] C. Stüble, “The PERSEUS Security Framework”, 2006, located at: http://www.perseus-os.org/

[35] IBM, “CryptoCards - IBM eServer Cryptographic Hardware Products”, located at: http://www-03.ibm.com/security/cryptocards/

[36] S. Jiang, S. Smith, and K. Minami, “Securing Web Servers against Insider Attack”, Appeared at the 17th Annual Computer Security Applications Conference, Dec 2001.

[37] Wikipedia, “IBM 4758“, April 2006, located at: http://en.wikipedia.org/wiki/IBM_4758

[38] IBM, “IBM PCI Cryptographic Coprocessor”, located at: http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml

[39] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, and S. Weingart, “Building the IBM 4758 Secure Coprocessor”, in IEEE Computer, vol. 34, pp. 57-66, 2001.

[40] Dartmouth College Office of Public Affairs, “Dartmouth collaborates with Sun Microsystems to develop secure technology”, November 2005, located at: http://www.dartmouth.edu/~news/releases/2005/11/18.html

[41] R. Merritt, “Group forms to plug holes in PC security”, April 2002, located at: http://www.eetimes.com/story/OEG20020426S0124

[42] T. Mcfadden, “TPM Matrix”, March 2006, located at: http://www.tonymcfadden.net/tpmvendors.html#software

[43] Microsoft, “Next-Generation Secure Computing Base (NGSCB)”, 2006, located at: http://www.microsoft.com/resources/ngscb/default.mspx

[44] D. Safford, “Take control of TCPA”, April 2006, located at: http://www.linuxjournal.com/article/6633

[45] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura, “Trusted platform on demand (TPOD)”, Research Report RT0564, IBM Corporation, 2004.

[46] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, “Design and implementation of a TCG-based integrity measurement architecture”, in USENIX Security Symposium, USENIX, 2004, pp. 223-238.

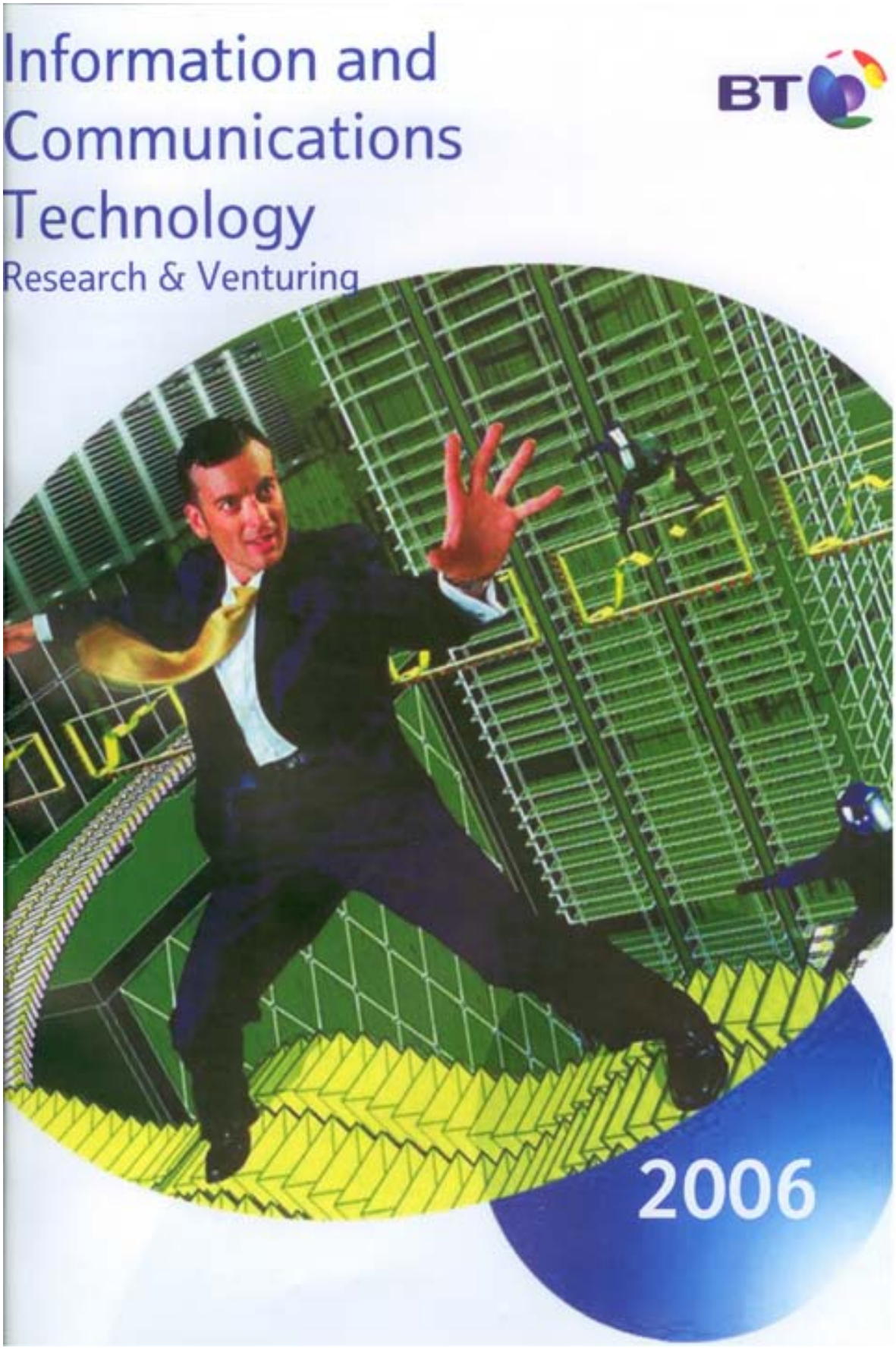
[47] Trusted Computing Group, “TCG Software Stack specifications version 1.10 golden”, August 2003, located at: http://www.trustedcomputinggroup.org/groups/software/TSS_Version__1.1.pdf

[48] Trusted Computing Group, “TCG Software Stack specifications version 1.2 Level 1”, January 2006, located at: http://www.trustedcomputinggroup.org/specs/TSS/TSS_Version_1.2_Level_1_FINAL.pdf

[49] Samsys, “Samsys RAPID Developer’s Guide”, located at: http://www.samsys.com

[50] Trusted Computing Group, “TCG PC Client Specific TPM Interface Specification (TIS) Version 1.2 FINAL”, July 2005, located at: http://www.trustedcomputinggroup.org/groups/pc_client/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf

Appendix A: ICT Open Days brochure

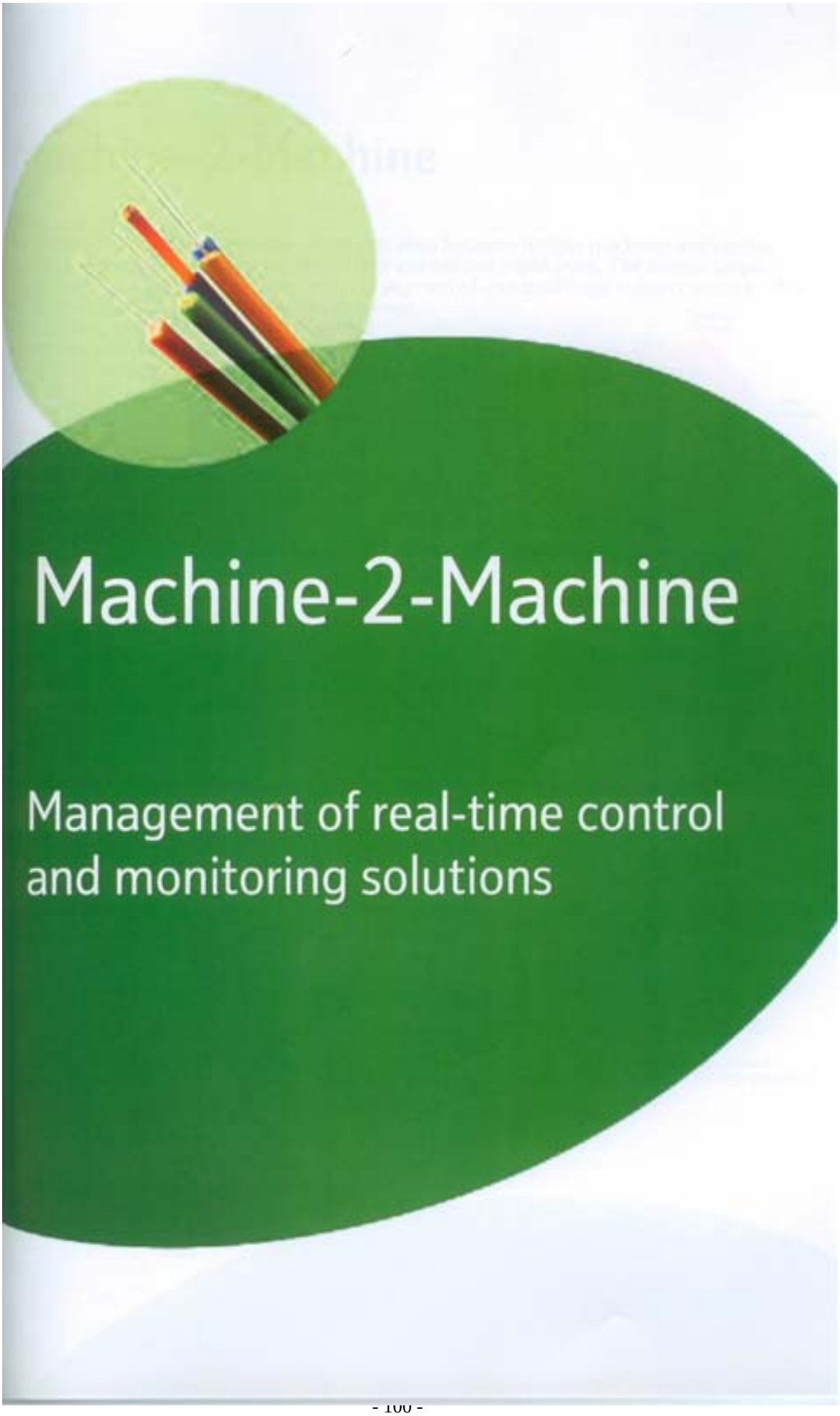


Welcome

Welcome to the ICT Innovation Open Day presented by Group CTO Research & Venturing. BT is now over two years into its transformation into a world-leading ICT company and GCTO's R&V ICT venture is itself approaching its 2nd anniversary. On show today we have a selection of the ICT research themes that we have been pursuing over this time, including Infrastructure for the Digital Networked Economy, ICT Security, the Real Time Enterprise and Machine2Machine. We are also pleased to showcase our sister Venture, Understanding Business Customers, which is led by John Seton.

Thank you for taking the time to attend this event and I hope you find the experience rewarding and insightful.

Jonathan Legh-Smith
ICT Venture Leader
jonathan.legh-smith@bt.com


Machine-2-Machine
Management of real-time control
and monitoring solutions

M2M

Machine-2-Machine

What is M2M?

BT defines M2M as the automated communication between remote machines and central management applications to enable real-time control and monitoring. The diverse scope of M2M does not allow the market to be easily segmented into traditional industry sectors – it is better defined by the object, structure or person that is being monitored, resulting in market categories such as Goods/Assets, Vehicles, People, Environment, Commercial Buildings and the Home.

Why M2M?

Very soon all human artifacts including consumer products, industrial plant and buildings will have the ability to sense and report on their status, usage, performance and the environment they operate in. M2M is about harvesting and leveraging this wealth of information to offer extraordinary business advantage to the companies that manufacture and provide services. Harbor Research describes M2M as a sweeping paradigm-shift from human-centric computing to device-centric computing and BT calls it "Living Intelligence".

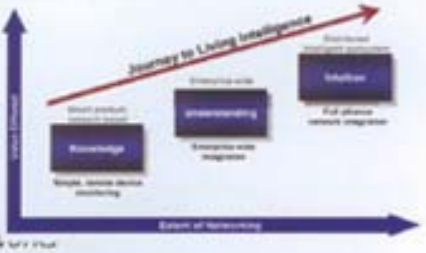
M2M Opportunities for BT

The opportunities lie in the creation of (information) value from M2M data (such as the weather and traffic conditions) and the ability to reliably manage a geographically dispersed ICT infrastructure. At the crossroads between a carrier and a new wave ICT company BT is extremely well positioned to offer information management solutions such as M2M data fusion, aggregation and analysis, QoS, assurance, data integrity, consistency and security, and efficiently managing the pervasive devices (e.g. remote sensors, hubs, etc.) and the infrastructure that connects them.

Key challenges in M2M

There are many key research challenges in M2M in the areas of information management, device and infrastructure management, security, interoperability and optimal architectures. The 2006 ICT Open day focuses on the following key challenges:

- M2M hub and sensor network management
- Security and Privacy in RFID systems
- Interoperability
- Deployment & operational efficiency of sensor netw
- Process assurance using RFID



Source: Harbor Research

Results of addressing the above key challenges are demonstrated in the areas of environmental monitoring, transport telematics and tagging and tracking.

Contact:
George Bilchev: george.bilchev@bt.com

Concept | Prototype | Trialled | Deployed

Machine to Machine

Trusted M2M Infrastructure

Objective

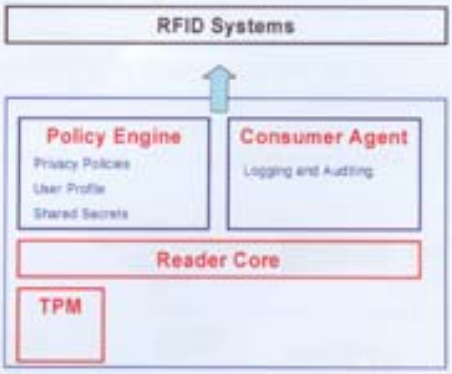
Trusted computing modules are becoming available on many M2M and other computing devices. The exploitation of this capability can assure the shared operation of devices and network services in a multi-service environment. The objective is simple - if an M2M device is trusted, we can separate the operation of the device from the control for services and content. Role-based access control mechanisms will provide policy driven control that can be remotely configured and audited.

We foresee a future with trusted computing embedded in our devices and network services. This enables the building of trust between applications and businesses operating over a shared M2M infrastructure for service provisioning.

Case Study "Trusted RFID Readers"

Radio Frequency Identification (RFID) technology raises significant privacy issues because it allows the tracking of items and people without their knowledge or consent. One of the biggest challenges for RFID technology is to provide privacy protection without raising tag production and management cost. We introduce a new architecture that uses a trusted computing platform embedded in an RFID reader. Readers use remote attestation to prove they are operating a known capability. The tags that the reader is allowed to read and pass on to applications is controlled by policies. The current and historical policies are logged and made available to 3rd party auditors.

The combination of Trusted Computing devices and remote attestation functions allows the verification that M2M devices comply with privacy and security policies and regulations. Operators can verify that the machine has not bee compromised, while auditors can check the compliance of operating policies



Progress

The Intel XScale 2 currently used in PDAs, RFID readers and embedded devices includes a Trusted Platform Module chip, and will allow the development of this work to control many M2M devices and services. The current RFID Trusted reader prototype has been implemented on an IBM TPM using the Enforcer TPM-aware Linux kernel.

Contact:
Andrea Soppera: andrea.2.soppera@bt.com