Design and Realization of Privacy Guaranteeing Means for Context-sensitive Systems

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Dipl.-Informatiker

Michael Maaser

geboren am 11.12.1975 in Jena

Gutachter: Prof. Dr. rer. nat. Peter Langendörfer

Gutachter: Prof. Dr.-Ing. Jörg Nolte

Gutachter: Prof. Dr. rer. oec. habil Günter Müller

Tag der mündlichen Prüfung: 06. Juli 2010

To my lovely daughter

Abstract

Privacy issues are becoming more and more important, especially since the cyber and the real world are converging up to certain extent when using mobile devices. Means that really protect privacy are still missing. The problem is, as soon as a user provides data to a service provider the user looses control over her/his data. The simple solution is not to provide any data but then many useful services, e.g., navigation applications, cannot be used.

The dissertation addresses two aspects of privacy protection. The first aspect regards not producing private information if possible. Such unnecessary information are traces of access controlled service uses. Hence, one approach in this dissertation enables *k*-anonymous authorization for services uses. It equips the users of the system with trusted pseudonymous certificates reflecting their respective authorizations. Analogous to anonymous e-cash, the certificates are issued by a trusted authority with knowledge of the actual authorizations of an identified user. The certificates can be verified by any service supported by the trusted authority but without knowledge of the user's identity. Not even the issuing authority is able to reveal the users identity from the pseudonym of a certificate. Hence, service usage cannot be tracked, neither by the service nor by the authority. This protects the privacy of service usage behavior of users.

The second aspect of privacy protection is to remain in control over private data released to others. Temporary release of private data is essential to context-sensitive services, which rely on these context data to provide or improve added value. Therefore, the dissertation designs a Privacy Guaranteeing Execution Container (PGEC), which enables applications to access private user data and guarantees that the user data is deleted as soon as the service or application is finished. Basically, the concept is that the application obtains access to the user data in a specially protected and certified environment, the PGEC. The PGEC also restricts the communication between the application and the service provider to what is explicitly allowed by the service user. In addition to those means, the PGEC also implements countermeasures against malicious attacks such as modified host systems and covert channel attacks, which might be misusing CPU load to signal data out of the PGEC. Thus, the PGEC guarantees a "one time use" of the provided private data.

Zusammenfassung

Privatsphären- und Datenschutzfragen gewinnen immer mehr an Bedeutung. Vor allem seit die Cyber- und die reale Welt durch die Verwendung mobiler Geräte zu einem gewissen Grad konvergieren. Mittel, die tatsächlich die Privatsphäre schützen, fehlen noch immer. Das Problem ist: Sobald ein Benutzer seine Daten an einen Diensteanbieter übermittelt, verliert er die Kontrolle über seine Daten. Die einfachste Lösung wäre es, überhaupt keine Daten zu übermitteln. Dann können jedoch viele nützliche Dienste, z. B. Navigations-Anwendungen, nicht verwendet werden.

Diese Dissertation adressiert zwei Aspekte des Privatsphärenschutzes. Der erste Aspekt betrifft die Nicht-Erzeugung privater Informationen wenn möglich. Solche unnötigen Informationen sind Nutzungsspuren an zugriffsbeschränkten Diensten. Deshalb ermöglicht ein Ansatz dieser Dissertation eine k-anonyme Zugangsberechtigung. Er stattet die Benutzer des Systems mit vertrauenswürdigen pseudonymisierten Zertifikaten aus, welche die entsprechende Autorisierung reflektieren. Analog zu anonymen e-Cash-Ansätzen werden die Zertifikate von einer vertrauenswürdigen Autorität mit Kenntnis über die tatsächlichen Berechtigungen identifizierter Nutzer ausgestellt. Die Zertifikate können von jedem Dienst mit Unterstützung der Autorität verifiziert werden, ohne die Identität des Benutzers zu kennen. Nicht einmal die ausstellende Autorität ist in der Lage, die Identität des Benutzers aus dem Pseudonym des Zertifikates zu ermitteln. Daher kann die Servicenutzung nicht nachverfolgt werden, weder durch den Dienst noch durch die Autorität. Dieses schützt die Privatsphäre des Dienstnutzungsverhaltens der Nutzer.

Der zweite Aspekt des Privatsphärenschutzes ist, die Kontrolle über ausgegebene private Daten zu behalten. Die vorübergehende Herausgabe privater Daten ist unerlässlich für kontextsensitive Dienste, die auf solche Kontextdaten angewiesen sind, um den Mehrwert des Dienstes anzubieten oder zu erhöhen. Dazu wird in dieser Dissertation ein Privatsphären garantierender Ausführungscontainer (PGEC) entworfen, der Anwendungen den Zugriff auf private Nutzerdaten ermöglicht, dabei aber garantiert, dass die Nutzerdaten gelöscht werden, sobald der Dienst oder die Anwendung beendet ist. Im Grunde ist das Konzept, dass die Anwendung den Zugriff nur innerhalb einer speziell geschützten und zertifizierten Umgebung, dem PGEC, erhält. Der PGEC beschränkt außerdem die Kommunikation zwischen der Anwendung und dem Dienstanbieter auf das Maß, welches explizit durch den Dienstnutzer festgelegt wird. Zusätzlich zu diesen Mitteln wendet der PGEC Gegenmanahmen an gegen böswillige Angriffe, wie modifizierte Hostsysteme und Angriffe über verdeckte Kanäle, die zum Beispiel die CPU-Last missbrauchen könnten, um Daten aus dem PGEC herauszuschleusen. Damit garantiert der PGEC eine “Einmalnutzung” der herausgegebenen privaten Daten.

Contents

Acknowledgements

With having this dissertation eventually finished, I would like to thank my family and friends for constantly nagging me about the state of affairs regarding my dissertation, which made me not give up.
I would like to thank my wife, as she put pressure on me by having finished her dissertation prior to me. She also provided the cover art of this book.
Finally, I would like to thank my colleague and friend Steffen Ortmann for his peer reviewing and especially for being a good discussion partner on many issues that occurred in this work.

Chapter 1

Introduction

In the world we used to know a few decades ago, privacy was simple. You closed the door and there you go. What happens behind closed doors stays behind closed doors. While privacy was your right to self-determine when, how and to what extent information about yourself is communicated to others[1], you used to be in almost full control of your information. Even if you told private information to someone, this information was not shared widely. Additionally, the details or the degree of truth decreased naturally when telling from one person to the next. But things have changed. Large parts of human life and social interactions migrated into the Internet. Hence, a vast amount of data is put into and stored in the net. Since machines do not forget and are able to make exact copies of what they know, the natural degradation and limitation in travel distance of information are gone. Besides, the machines are able to gather information that is not explicitly given but can be inferred from user behavior or context [2]. A context is defined as

> "the conditions and circumstances that are relevant to an event, fact, etc."[3]

These conditions and circumstances can be used to improve a service or to provide a service at all, e.g., a navigation service directing based on current position. On the other hand, private information could be derived that is not even known to the owner. In such case, it is very hard to remain in control over the own private information.
The remainder of this chapter motivates the demand for privacy and introduces a number of scenarios with different requirements for privacy.

[1] Derived from definition[1] by Prof. em. Alan Westin, Public Law and Government, Columbia University

1.1 Motivation

From a personal point of view, the lack of privacy protection by annoying advertisements may be noticed. Here e-mail spam, cold calls from call centers or mailings with credit offers from banks could be considered. Sometimes the information is gained from a publicly available source, e.g., the phone number from the telephone book or yellow pages. Even though this is not actually a privacy breach, the publication of the phone number was not supposed for those unsolicited calls. Hence, persons or even companies releasing personal information require the purpose being adhered. Technologies that bind data to a purpose or other restrictions, e.g., P3P[4] and GeoPriv[5], exist (see section 2.1.1). On the other hand, some advertisement or information that exactly fits a persons current needs may not be considered annoying but well placed. A large number of personal information including the context of the aimed person has to be known to figure out the exactly fitting unsolicited information. One may consider various recommendation systems. Those can be either systems that recommend on the basis of a community as in online shops like Amazon[2] or music playlists like last.fm[3]. These are based on previously purchased products and shopping behavior of other customers or other listeners music liking. It can further be music and television show recommenders based on key words, artists, genre or any other meta-data. Such systems live on the release of some private information such as behavior tracking, interests and contexts. Privacy Enhancing Technologies (PET) that aim to act in this field must not hinder the release of those data but ensure to keep control over it, to prevent it from being misused for any unintended purpose.
Governmental institutions like the Art.29 Data Protection Working Party[4] demand for PET as well [6]. This is especially provoked by an amount of publicly known privacy breaches. Those already occurred some years ago, like JetBlue's Privacy Policy Violations [7] or the publication of AOL's pseudonymized search log that enabled data miners to identify at least one person [8]. Privacy breaches are recently going on. There are not only accidental privacy breaches as in the British HM Revenue & Customs[5], which lost CDs with personal information of about 7.25 Million British Families receiving child benefits [9]. The Italian ministry of finances intentionally published the annual tax declarations of all citizens [10]. That is, even institutions that gather private information by law do not always obey privacy concerns. Furthermore, especially since 9/11 there exist covetousness of the government and security organs of the United States of America to get many information about people traveling into the U.S.A. [11] as well as information on financial transactions. The latter is known as the SWIFT affair [12].
The given examples show that private data is not safe, not even at parties that should be trusted as they are organs of the government you elected. These are supposed not to be

[2] http://www.amazon.com
[3] http://www.last.fm
[4] http://ec.europa.eu/justice_home/fsj/privacy/workinggroup/index_en.htm
[5] http://www.hmrc.gov.uk

interested in gaining profit from the collected data. It may not be feasible to counteract the growing interest in private data of governmental institutions by technical means such as presented in this work, but the past and current leaking issues shall demonstrate that there cannot be trust into any party with regard to private data. Effective privacy protection means should not rely on trustworthy parties but rather giving or regaining full control to the owner of the data. As data and especially personal information is useful only when being processed, means that allow private information to be processed by services provided by other parties are required. These services as well as the additional information brought with the service are somewhat private to the other party and hence, worth to be protected as well. The privacy protection technology presented in Chapter 4 aims to achieve that omni directional protection. It protects data users and services and the service providers' code as well.

1.2 Scenarios

There are various scenarios, where personal/private data arise or are being used. Hence, multiple aspects of privacy protection must be considered. Two of those aspects are addressed in this dissertation. The first aspect regards the prevention of generation of private information, such as behavioral traces. Protection of and control over data actually released to others is the second privacy protection aspect addressed here. A third aspect concerns only about the deduction of information from released or sensed private data [2]. It is considered, that fulfillment of the first two aspects obsoletes the third.
Those mentioned scenarios differ in the way private data are gathered and processed. This comprises the source of the data, which data or type of information is actually used, what is the data used for and whether or how does the data positively affect the purpose. The following sources may be considered:

- Content of own data bases from previous data gathering,
- Information explicitly provided by the owner,
- Information gathered from user's behavior (e.g., by tracking) or
- Aggregation and inference of information from original information.

Such data could be rather static information like

- Birth date,
- Credit card or social security number,
- Phone number or e-mail address,
- Postal address,
- Income,

- Health records,
- ...

or more dynamic information, e.g.,

- Proclivities,
- Interests or
- Behavior like shopping history, movement patterns or daily schedule.

Some of those data are obviously necessary to accomplish certain service, e.g., a postal address is needed to send a package. Information may also affect the purpose, such as the insurance premium for a health or accident insurance is lower when being healthy and young. Even without releasing health record and birth date such an insurance may procured, but probably the highest possible rate will have to be paid. Another example is an automobile liability insurance. The rate depends on static information like birth date, gender, date of receiving the drivers license, affiliation with certain groups (public service, farmers etc.) and even the type of car as well as dynamic information like a drivers record or years without claims. Some information entitles for certain rebates. If one cannot or do not want to present a clean drivers record, she/he is likely to start with the highest possible rate.
The following sections describe scenarios, in which a person uses services provided through computers that involve personal data. The affection on privacy and hence, the requirements on protection from misuse of that data is investigated.

1.2.1 Anonymous Authentication/Access Control

The first addressed scenario cares about private information that is inferred from a user's behavior. This scenario considers services, which do not actually need context information. The user tracking and behavior inference becomes possible only if users utilize services with revealing their identity. A prime example of such scenario can be a hotel complex. Let there be a number of guests that paid different rates. The rates dictate which facilities may be used. Such facilities can be a pool, a gym, a sauna, a squash or tennis court, a billiard room or even varied class restaurants. Depending on the age, the access to the bars might also be restricted. Drinks may be included or not, which is currently distinguished by colored wrist bands. Access to the rooms is restricted to the current occupants at any time as well as for security staff. The French maid should access the room in a restricted time frame and when no one else is in the room only. On the other hand, security staff and French maids may access more than one room. One may also consider a teacher of a school class or the trainer of a sports team that may have granted access to all rooms assigned to the class or team. There may further be restrictions on the use of the land-line phone or the TV and video offer in the room.
This hotel and its guests may be concerned about their privacy. But the hotel still has to

assert the adherence to the restrictions that apply, either due to paid rates or just by legal issues. The guests do not want to be tracked in their behavior within the hotel. Nobody should know, when a certain guest goes for breakfast, in the pool or the gym. How long did someone spend in the bar? Which movies did someone watch before going asleep?
To accomplish this an access control system is needed that does not associate the execution of an access right with an identity. In other words, such control system must enable persons to legally access restricted activities or areas without the need to reveal their identity. Some of the requirements could be fulfilled by offline solutions like the mentioned colored wrist bands or handing out mechanical keys. Even those wrist bands violate the privacy of the wearer, since everyone can see it and hence, can gain knowledge about the rate/package she/he has paid. Proving age requires a drivers license, passport or ID card. This is the initial point where a mapping to an identity occurs. When using electronic key cards, guests have to trust that these are not assigned with an identifier and the doors being opened do not track this identity. Hence, a suitable approach shall make tracking technically impossible. When tracking of movement or service use (consuming movies or TV) is impossible, the inference of profiles including personal proclivities becomes infeasible. The drawback of such system is, that in case of maybe criminal attempts the deniability or provability of presence or absence becomes very difficult. This problem of deniability or provability is also addressed in private payment methods and e-cash schemes like in [13].
Such mechanism for anonymous and untraceable access control is presented in Chapter 3. It leverages digitally signed certificates to prove granted rights by possession. Since digital certificates may be copied or eavesdropped, the legal possession of a certificate can be proven by knowledge. The revocation of rights is addressed by limited validity duration of the certificates. Presence can be proven only if both parties agree. The access control unit, e.g., the door, presents the used certificate and the guest presents her/his prove of ownership of the certificate. To further prove the time of access, the guest should also digitally sign the current time stamp upon accessing or using a certificate. Absence can only be proven indirectly with the prove of presence at some other place on mutual agreement between the user and the other access control unit.

1.2.2 Stationary/mobile use

With opening the Internet for commercial activities in 1992 and growing services like access to news, communication means and shopping facilities, lives of people and the society started to change. On the one hand, Internet users benefit from being well informed and up to date with ”What is going on in the world?”, staying in contact with interesting people independent of the distance or being able to shop 24 hours a day, 7 days a week. On the other hand, they lost some privacy. Online shops became able to track your shopping habits and even the stuff that you were not buying but just looking at. This allows to create a user-related profile. The publication of wish lists helps in creating of profiles as well. This includes lists of books that have been read, as it is available from public

libraries, that have been bought or wished and even certain Internet news that have been subscribed to or have been read. These are used by parties to interpret the interests of a person and deduce actions or restrictions. Several cases are known, where people with certain reading habits were debarred from entering the U.S.A.[14].
Modern communication means, such as ICQ[6], Windows Live Messenger[7] or Skype[8], announce your availability status to others. This is an intrusion into privacy and hence, the respective protocols introduced privacy mechanisms that allow to distinguish between users that are allowed to see the state and those that are not or to hide the online state at all. In recent few years community portals came up that allow to express yourself and to get in contact with users having similar interests. Some prominent representatives are MySpace[9], Facebook[10], StudiVZ[11] and its offshoots. All these platforms allow for some kind of exhibitionism. For the purpose of finding the right people with shared interests this just seems to be appropriate. The backside of the medal is the possible loss of privacy, since everyone is allowed to access the information provided. Increasingly often such platforms are seeked by personnel managers to get private background information for job applicants. More often than not, the information found is not conducive for getting the job.
With third generation mobile devices and the upcoming mobile Internet people are enabled to access all the services anytime and anywhere now. Various means to determine the position of a wirelessly connected mobile device are known. The position of mobile phones can be estimated from the GSM base station and its sectoral antenna it is connected to and the received signal strength or the packet travel time[15]. The same holds true for devices connected via 3G and 4G communication channels, e.g., UMTS or WiMax. This capability is widely used for cell phone features like the home-zone introduced in Germany by former VIAG Intercom and continued by its successor O_2[12] or competitors Vodafone[13] and T-Mobile[14]. For Wireless LAN (WLAN) enabled devices such as laptop computers or Personal Digital Assistants (PDAs) the used WLAN hotspot can be determined. Last but not least, a device may actively determine its position using Global Positioning System (GPS) or comparable systems. When incorporating such position information, not only cyber world habits but also real world motion habits can be tracked and linked together. Of course, the position information can bring in lots of benefits such as being exploited by numerous Location Based Services (LBSs) with navigation systems being the most prominent representative. On the other hand, it holds serious risk for users' privacies. Users are most likely aware that their current position is

[6] http://www.icq.com
[7] http://www.live.com
[8] http://www.skype.com
[9] http://www.myspace.com
[10] http://www.facebook.com
[11] http://www.studivz.net
[12] http://www.o2online.de
[13] http://www.vodafone.de
[14] http.//www.t-mobile.de

needed for the service currently used or brings some additional benefit into it. But they are as likely not aware that the same service could also track position data over time and deduce information for a different purpose. Keeping a position at a shopping window over a certain time could allow to deduce an interest in the products displayed.

1.2.3 Exchange of Private Data for Purpose

While a number of services could be provided without any information from the user, there are at least the same amount of services that definitely require information from or about the user. Otherwise, those services would render useless. When the service can convince the user that such information is required, the user is most likely willing to release the information as requested. At the same time, the user wants to ensure that the released information is not used for any other purpose than providing the requested service. For instance, a user may hand out her/his mail address to receive an ordered product. She/he is not expecting the address being told or sold to some insurance agent that comes over to talk her/him into a new insurance, for example. Similarly this can hold true in the opposite direction. Consider a musician or his agent that want to sell some MP3 track to a potential listener. While the payment could be considered as the service provided by the listener the MP3 track is the personal information of the musician. The musician is definitely interested in getting the money and agrees to hand out his music to the listener. Besides that, the musician wants to assert that the listener does not tell or give the bought music track to someone else without receiving further payment. Therefore the musician is using some Digital Rights Management (DRM) on the music track to remain in control over his personal information (the music track).
As one can see from the given examples, binding some data to a certain purpose is actually some kind of copy protection. When considering existing DRM solutions from the content providers point of view, it is also some retention protection, since DRM allows to specify license durations such as play only once or play only within 24 hours etc. The purpose may be specified in this license as play only, no copy, no output to analog device or any other restriction. When the mechanisms from the DRM technologies could be applied to any kind of data and for more flexible processing than just rendering using a given audio or video codec, it was just appropriate for privacy protection, too. Any person ought to be enabled to encapsulate her/his private data with a DRM harness that hinders a service provider from using these data for any other than the specified purpose.

1.2.4 Scenario of Accessing Medical Data of Patients

Medical data of patients bear critical information that does not only influence the personal lifestyle but may even affect the financial or professional situation of the patient or her/his relatives or acquaintances. Thus, medical data require special protection from unauthorized access and from information inference. This high demand for privacy

makes them a good vehicle to derive privacy requirements and to demonstrate the capabilities of privacy protection means.
During medical treatment it occurs very often that findings have to be shared between various physicians. This is the case in almost any referral. There it is required to transfer findings or even a medical history to the next treating physician. In case of involved X-ray photography, such transfer is even required by law §2c RöV[15]. An X-ray photo appears to be the patient's private data and requires appropriate protection when transferred to the treating surgeon. Using digital radiography, the X-ray photo exists as a digital image stored in a Picture Archiving and Communication System (PACS). The picture is available at every Personal Computer (PC) connected to the PACS introducing a privacy risk, without control of the patient. Similar is true for any other medical data stored in Electronic Patients Records (EPRs). A patient may further want to be in control over what data is communicated to her/his health insurance and how this data may be passed to other insurance companies. In all these cases, i.e., creation, transfer, storage and usage the patient shall remain in control over her/his data.
To show the primary functions of the Privacy Guaranteeing Execution Container (PGEC) introduced in Chapter 4 the following scenario shall be used. This scenario has been chosen to illustrate interactions between several services within the PGEC in a simplified way. Of course, other medical services that make use of patients context, like emergency handling, diagnosis support or long term monitoring, are feasible as well. Please note, the PGEC is a distributed system and each participant of the scenario runs her/his own instance. The instances form a logical unit with one inside and one outside. The logical unit is depicted by the dashed lines forming a closed ring in Figure 1.1. A patient sees a radiologist to have an X-ray photo taken. The radiologist creates the digital image and puts it in the container. Thereby the radiologist states the patient as the owner of the image. This enables to demonstrate how data items like the X-ray image are identified in the system and how the ownership of an item is set. Similarly any data with privacy relevance will go into the container. Further, the patient allows an EPR to store the image, requiring that it can be stored but not seen by the EPR service. Obviously, this needs an encryption and key management scheme, which are to be demonstrated by this. The respective EPR storage service is triggered by the radiologist after creation of the image. Next, the treating surgeon shall also be enabled to access the X-ray photo of the patient. This can either be during her/his visit or even without presence of the patient. This demonstrates how the patient, as the owner of the data, remains in control over her/his data. This will also demonstrate the complex relationship between owners, data items, service providers, service users and access right management. Again, the key management scheme required for decryption of the stored medical data is stressed. While all involved services run inside the container, the patient and the physicians can trust in appropriate privacy protection of the data. Even the EPR can concentrate on providing its services without worrying to much about privacy. By never seeing the content

[15] Röntgenverordnung in der Fassung der Bekanntmachung vom 30. April 2003 (BGBl. I S. 604)

of the stored data and not knowing the required keys, the data is securely protected.

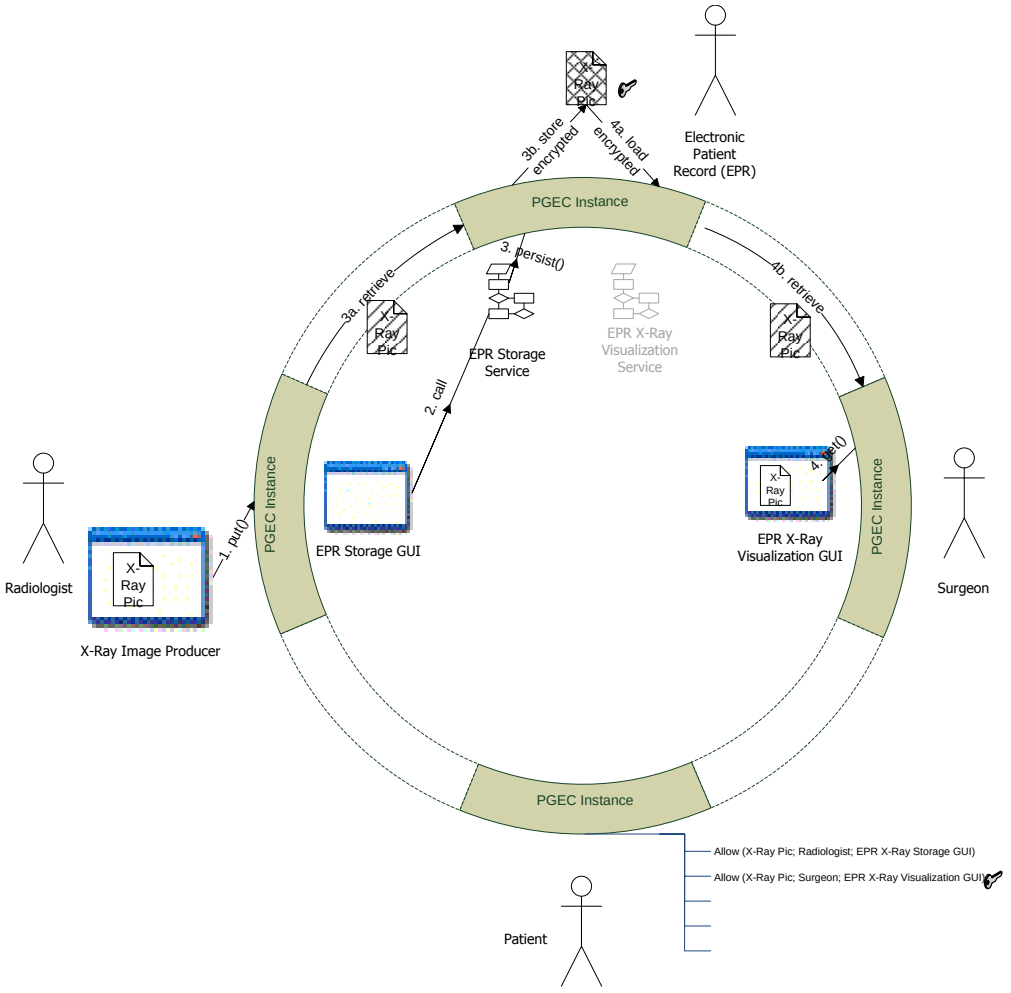


Figure 1.1: Transfer of patients data using an EPR inside the PGEC. The ring displays the logical unit of the distributed container, whereas the darker shaded sectors are the instances running on the machines of the respective scenario participants. The inside of the ring is the environment, in which service can be executed and access private data depending on the access rights granted by the data owner. Private data can be filled into the container but never leave it un-encrypted. (The faded EPR Visualization Service has no actual function but to provide the Visualization GUI, which in turn implements the required functionality. This is reasonable due to the transparent distribution of the container instances.) The numbering reflects the sequence of the calls. Calls numbered with a and b are triggered by the according API calls to the PGEC.

Structure of the Dissertation Since the requirements for privacy vary throughout the introduced scenarios, various solutions for privacy protection exist addressing these requirements. Such privacy protection solutions and technologies are described in the following chapter. Chapter 3 presents a solution for scenarios where actual information is rather irrelevant, but merely some access authorization by possession or knowledge is important. One could compare that to the possession of a mechanical key. An approach that aims to regain control over all released personal information, even such that is inferred secretly, without loosing the benefits from processing such information as a context is presented in Chapter 4. The privacy gains from the presented solutions as well as remaining lacks in absolute privacy protection are summarized in the last chapter.
Despite all technological possibilities and capabilities, people should not loose a certain amount of sanity and reason and develop a healthy privacy awareness.

Chapter 2

Related Technologies

Privacy is an aspect relevant for a number of fields of application and research. Thus, it is addressed either as the main focus of research projects, e.g., PRivacy and Identity Management for Europe (PRIME)[1], or as an addition to research in e.g., LBSs or Wireless Sensor Networks (WSNs). It can be distinguished between pure declarative technologies (see Section 2.1.1) and technologies that technically support privacy enforcement (see Section 2.1.2). The latter are more relevant to this work.
This work presents two approaches for privacy protection. One approach prevents the user from inadvertently leaving marks while accessing restricted services or areas. This preserves the users anonymity. This chapter presents various technologies for access control and authentication and assesses its contribution to protect a user's privacy. The second approach developed in this dissertation aims at allowing to benefit from releasing private data for a particular purpose. The technology presented therein asserts that this private data cannot be misused for some other purpose, either inadvertently or with intent. Similar behavior is known from DRM systems, which are presented in Section 2.4.

2.1 Existing Privacy Protection Technologies

2.1.1 Declarative Technologies

Technically, the Platform for Privacy Preferences Project (P3P) [4] defines an XML dialect for the description of privacy policies. So service providers can state which data they are gathering for which purpose. A P3P Preference Exchange Language (APPEL) [16] can be used to express what a user expects to find in a privacy policy. P3P and APPEL merely provide a mechanism to describe the intentions of both sides rather than means to protect user data after agreeing to use the service.
IETF's GeoPriv working group is developing an architecture for handling location infor-

[1] https://www.prime-project.eu/

mation in a privacy aware manner [5]. One of the benefits of this architecture is that the privacy rules, are stored as part of the location object [5]. Thus, nobody can claim that she did not know, that access to the location information was restricted. But misuse is still possible and it is still not hindered somehow by technical means.

2.1.2 Enforcing Technologies

There are several approaches that try to protect privacy in location aware middleware platforms [17, 18, 19, 20, 21]. In [17, 18, 19] means are discussed that enable the user to declare how much information she is willing to reveal. [20] discusses a middleware that applies user defined rules, which describe who may access the user's position information and under which circumstances. The approach investigated in [21] intentionally reduces the accuracy of the position information in order to protect privacy. This helps to protect privacy to certain extent, but it cannot be used in systems that need an accurate position to work properly, e.g., navigation services. In all these approaches means to enforce privacy are missing.
There is a lot of work done in the area of digital rights management to protect content [22, 23] as well as code from misuse [24]. Those approaches rely on specialized hardware such as Smartcards, or are vulnerable to data extraction [25]. Further, these systems do not provide means to execute any code to be freely defined as it was needed for services. They merely protect media content, which could be considered as the service provider's data. But, despite the protection of user data is in principle an equivalent problem these approaches do not provide a solution for protecting service users' data.

Agent Platforms

To the best of the knowledge, there are only two approaches [26, 27] that try to make sensitive data available to a third party while ensuring secrecy of that data. [27] proposes an architecture that ensures secure data processing by exploiting the java sandbox model as execution environment for data processing code and by limiting the feedback from the data processing code to the out side world. In order to allow correct interpretation of data processing results as well as development of appropriate algorithms, a part of the data has to be publicly accessible. In addition, sensitive data is always kept at its owner's site. The prerequisites of this concept render it impractical to implement location based or context sensitive services, although it is well suited for privacy preserving data mining.
The approach presented in [26, 28] tries to avoid that user data is accessible outside a specially secured execution environment. User data is enclosed in an agent and securely transferred into an isolated closed-door one-way platform provided by a trusted third party. The service agents proceed analogous with their own data. Those entire agents interoperate within that trusted environment and agree on a certain result. The result is forwarded by all involved agents independently to the closed-door platform, which posts the result to the agents' origins if the forwarded results are equal. This ensures that no

private data is transmitted to the opposite party if the agent did not agree to. In order to ensure the privacy of the user and to protect the services data, all agents including their enclosed data are deleted after service completion. In contrast to this approach, PGEC does not rely on a trusted third party that provides processing capabilities such as a server plus a specific agent platform. Encapsulation of sensitive data and its deletion after service completion are provided by the PGEC by design. It allows for bilateral cooperation between service users and service providers. User and service provider data do not need to be transferred a priori but only when really needed. Their data may even be used without being transferred. This is especially helpful if location based services are realized inside the container, because they may need a huge amount of data, such as a catalog of all restaurants in New York City. This feature is ensured by the concept of a distributed PGEC, which consists of at least two instances at either participant side but represents virtually one single PGEC.

Statistical Data Protection

In scenarios where individual information is not of interest but rather the information that can be derived from a larger number of individuals, such as a sum, average, minimum/maximum or range, variance or standard deviation, the privacy of the individual can be sustained. Statistics literature knows two methods to modify the value of a field to protect the privacy of the individual value but to retain the statistical properties of the set of values [29]. Those are:

- Value-Class Membership
- Value Distortion

Besides, there exists *Value Dissociation*. When requesting a field's value of a particular record in a database, this method returns the value of the respective field in another record of the database. Similar to value distortion, this is not very useful when requesting individual values. It can protect privacy when selecting subsets from the database or requesting general values like average. An advantage is, that it is also applicable to alphanumeric values. In [29] the similarity to value distortion is shown as well as a suitable attack is presented. That reduces the number reasonable applications for this method. The following paragraphs explain the more useful methods of statistical data protection.

Value-Class Membership This method of statistical data protection partitions the domain of values. Each partition is a set of values and mutually disjoint and every possible value is contained in exactly one set x_i. A request for a particular value is then answered with the set x_i instead of the actual value. This method is commonly known and widely used to hide individual values, e.g., in questionnaires. The sizes of the respective sets do not need to be equal. Sometimes the partitions are created by discretization into intervals. Annual salaries, for example, could be partitioned into intervals of 10.000 € for

values lower than 100.000 €, and intervals of 50.000 € for higher values.
This is especially useful for location data. For many applications a coarse grained location resolution might just be sufficient. The intervals can also be multidimensional, such as an area on earth's surface or combined with a temporal interval. Such intervals were used by Gruteser and Grunwald [21] for spacial and temporal cloaking in LBSs. The position of an individual at a certain time is mapped into a three-dimensional interval in a way that the positions of a number of other individuals fall into the same interval. The maximum size of the interval depends on the service. By the pure knowledge of the interval passed to the LBS, the actual individual cannot be distinguished, which provides a certain amount of anonymity. To assure that the interval contains more than a given minimum of individuals the authors take advantage of a central server as a location anonymizer, which in turn knows the exact positions of each individual over time. Such central anonymizer is the weakest link in privacy protection of that approach and a potential target of privacy attacks. Privacy was protected far more when no central instance was needed. To accomplish that, the size of the interval has to be determined by each individual with some confidence that she/he is not the only element within that interval.
Another approach to maintain location privacy within spatial intervals is described in [30]. It uses dynamically created intervals observed at the individual's devices during the anticipation of two individuals coming into close vicinity. Upon event registration the intervals are chosen as circles around the current positions of the involved individuals. The radiuses r_i and r_j of any pair of individuals are limited to $r_i + r_j + v <= d_{i,j}$ with v being the requested vicinity distance and $d_{i,j}$ the current distance between this pair of individuals. As long as neither of the involved individuals leaves her/his interval (area) there is no need to communicate the actual position. Only if an individual leaves her/his interval, the current positions of all individuals involved in the affected event description have to be retrieved by the event server. Accordingly, the event might be raised or new intervals are calculated and passed to the individuals. While the "observed" individuals stay rather distant from each other, their respective latitude is large enough to rarely require position updates, which keeps the level of location privacy high.

Value Distortion A common approach of value distortion is random data perturbation. Each individual value is adjusted with an offset or scalar. The offset or scalar has to be chosen in a way that it renders into the neutral element regarding the applied operation of interest. For adding or averaging, the sum or average of the offsets respectively must be zero. For multiplication the product of all scalars must be one. To ensure that the individual value cannot be deferred by subtracting the offset or dividing the scalar, these offsets and scalars have to be different and randomly assigned to the individual values. The random value is drawn from a well known distribution such as the uniform distribution within interval $[-\alpha, +\alpha]$ or the Gaussian distribution with mean $\mu = 0$ and a known standard deviation σ [31].

It is clear that with random values it is almost impossible to reach the gain of getting n random numbers that actually sum up $\sum_{i=1}^{n} r_i = 0; r_i \in \mathbb{Z}$ or multiply to $\prod_{i=1}^{n} r_i = 1; r_i \in \mathbb{R}$. Methods that try to reconstruct the original distribution from the distorted values have been introduced and discussed by [32]. They approximate the original distribution of values $F_U(u)$ as

$$F'_U(u) = \frac{\int_{-\infty}^{u} f_V(w-z) f_U(z)\, dz}{\int_{-\infty}^{\infty} f_V(w-z) f_U(z)\, dz}. \tag{2.1}$$

With differentiation and averaging over all data samples w_i, they obtain the posterior distribution from

$$f'_U(u) = \frac{1}{n} \sum_{i=1}^{n} \frac{f_V(w_i-u) f_U(u)}{\int_{-\infty}^{\infty} f_V(w_i-z) f_U(z)\, dz}. \tag{2.2}$$

As $f_U(u)$ remains still unknown, the authors iterate by using the left hand side of Equation (2.2) on the right hand side of the next iteration by starting with $f^0_U(u)$ being the uniform distribution. The iteration stops when the difference between two consecutive approximations becomes sufficiently small (1% of the threshold of the χ^2 test [32]).
Even though this most likely does not produce exact values, it appears to be sufficient for many statistical evaluation of measurements. On the downside it has been shown that preservation of privacy in certain types of data perturbation is not necessarily good [33]. For processing exact values either individual or aggregated ones other approaches have to be taken into account. To gain an exact average or sum, means are needed to extract the randomness in the distortion without determination of the individual random values. A simple approach was to distribute the random offsets on the participants that they exactly sum up or multiply to the neutral element. Let there be data sources $v_i; i = 0, 1, 2, ..., n$ mutually sharing cryptographic keys, which can be obtained by key distribution mechanism described in [34]. The first v_0 may randomly choose a preliminary u'_0 and send it to v_1. Each $v_i; i < n-1$ adds an own randomly chosen u_i and forwards it to v_{i+1}. v_{n-1} adds a random offset as well and forwards the sum u_Σ to v_0. Finally v_0 determines $u_0 = u'_0 - u_\Sigma$, which it uses as its secret offset. Each data source may use its secret offset to distort its data $d_i + u_i$ and send them to the sink. Without knowing the offsets u_i it can calculate the sum $\sum d_i = \sum d_i + u_i$ because $\sum u_i = 0$. Alternatively, the first data source v_0 might add a random offset to its actual value d_0 and send it to the next data source. Each data source is adding its actual value and transmits it to the next. The last data source sends the sum to v_0, which can then subtract its random offset, resulting in the actual sum and forwarding it to the sink. Due to the mutual link encryption between the data sources, it can be ensured that no entity knows both the incoming and the outgoing sum of any other entity but itself. Hence, no entity may defer the actual value of another entity. This distortion of adding a random number could also be considered a very simple privacy homomorphism regarding the sum function of the actual values. More sophisticated privacy homomorphisms are described below in Section 2.1.2.
With Cluster-based Private Data Aggregation (CPDA) and Slice-Mix-AggRegaTe (SMART), the authors of [35] introduce two schemes that allow

the exact additive aggregation of values in a distributed sensor network while preserving the privacy of the individual data. In both schemes they use mutual encryption keys obtained by the mechanism in [34].

CPDA uses additive properties of polynomials. They build clusters of n nodes that share n seeds $s_i; i = 0, 1, ..., n-1$. Each of them calculates n polynomials of degree $n-1$. In this polynomials

$$v_{j,i} = d_i + r_{1,i}\, s_j + r_{2,i}\, s_j^2 + ... + r_{n-1,i}\, s_j^{n-1}; i, j = 0, 1, ..., n-1 \tag{2.3}$$

the coefficients are $n-1$ random numbers and the actual value. Every node i sends the $v_{j,i}$ to respective node j. Then these nodes j add the received values

$$F_j = \sum_{i=0}^{n-1} v_{j,i} = \sum_{i=0}^{n-1} d_i + \sum_{i=0}^{n-1} r_{1,i}\, s_j + \sum_{i=0}^{n-1} r_{2,i}\, s_j^2 + ... + \sum_{i=0}^{n-1} r_{n-1,i}\, s_j^{n-1} \tag{2.4}$$

and share their respective results. This gives a system of n equations with n unknown coefficients. Which can be solved like this.

$$\begin{bmatrix} \sum d_i \\ \sum r_{1,i} \\ ... \\ \sum r_{n-1,i} \end{bmatrix} = \begin{bmatrix} 1 & s_0 & ... & s_0^{n-1} \\ 1 & s_1 & ... & s_1^{n-1} \\ & & ... & \\ 1 & s_{n-1} & ... & s_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} F_0 \\ F_1 \\ ... \\ F_{n-1} \end{bmatrix} \tag{2.5}$$

This can also be used to additively aggregate multiple values (up to n) at a time if these are used in replacement of the random values.

SMART The second approach, which the authors of [35] pursue, is Slice-Mix-AggRegaTe (SMART). It uses the associativity of the addition. In the first step each node chooses k nodes and slices its value into $k+1$ summands.

$$v_i = \sum_{j=0}^{k} v_{i,j} \tag{2.6}$$

Second these slices are transmitted to the chosen nodes and one is kept for itself. Each node adds all its received slices and the slice kept for itself

$$\sum_{i=0}^{k} v_{i,j} \tag{2.7}$$

and sends it in the third step to the aggregator or data sink. There the sums of slices are added resulting in the sum of all original values without any node knowing the actual value of any other node.

$$\sum_{i=0}^{k} v_i = \sum_{i=0}^{k} \sum_{j=0}^{k} v_{i,j} = \sum_{j=0}^{k} \sum_{i=0}^{k} v_{i,j} \tag{2.8}$$

CPDA and SMART turn out to preserve privacy of the individual values very well especially with growing cluster sizes or slice counts. One of their drawbacks is the introduction of computation and communication overhead. Besides that, these mechanisms can only be applied if an additive aggregation of data is sufficient for the envisioned purpose. Note, other statistical requests, such as average or minimum/maximum, can be deferred from additive aggregation as well. Moreover, it is possible to also map the multiplication into addition by using logarithmic identities.

$$a \cdot b \cdot c = e^{\ln(a \cdot b \cdot c)} = e^{\ln(a)+\ln(b)+\ln(c)} \tag{2.9}$$

The SMART approach might further be used to retrieve the set of actual values without knowledge of their actual mapping to nodes.
The presented technologies are applicable in distributed sensor networks for a priori known services of rather low complexity. Those services are pure data aggregation or simple statistical in-network evaluations. Complex processing of data or inference of contexts as required by the scenarios described in Section 1.2 is not feasible.

Privacy Homomorphisms There are more complex operations that may even not be executed on the source of the data. Reasons for this constraint may be limited resources or more prominent parameters to be kept secret from the data source, as the private key to be used in digital signatures. To preserve the privacy of the data there must be a way to obtain the result of the complex operation without knowing the operation or its secret parameters and without passing the actual data. This can be reached through the use of homomorphisms or homomorphic encryptions.
Simple types of homomorphisms are group and ring homomorphisms. These map either a group $(A, +)$ into $(B, \oplus)$ or a ring $(C, +, \cdot)$ into $(D, \oplus, \otimes)$ respectively. With $f(a)$ being the respective homomorphism the following expressions are true.

$$f(a_1 + a_2) = f(a_1) \oplus f(a_2); a_i \in A; f(a_i) \in B \tag{2.10}$$

$$f(c_1 + c_2) = f(c_1) \oplus f(c_2); c_i \in C; f(c_i) \in D \tag{2.11}$$

$$f(c_1 \cdot c_2) = f(c_1) \otimes f(c_2); c_i \in C; f(c_i) \in D \tag{2.12}$$

This definition can also be generalized to an arbitrary number of functions or operation with even arbitrary arity.

$$f(g(a_1, a_2, ..., a_n)) = h(f(a_1), f(a_2), ..., f(a_n)) \tag{2.13}$$

If further $f(a)$ is an encryption function ε_k and a corresponding decryption function δ_k with key k exists, this can be used as a privacy homomorphism. Some literature [36] denotes the quadruple $(\delta_k, \varepsilon_k, G, H)$ as the privacy homomorphism. $G = g_1, g_2, ..., g_m$ and $H = h_1, h_2, ..., h_m$ are sets of functions over which the privacy homomorphism is defined.

$$\delta_k(h(\varepsilon_k(a_1), \varepsilon_k(a_2), ..., \varepsilon_k(a_n))) = g(a_1, a_2, ..., a_n) \tag{2.14}$$

Well known examples are the multiplicative homomorphism of Rivest-Shamir-Adleman (RSA) and the new privacy homomorphism of Domingo-Ferrer [37]. Using the RSA homomorphism the encryption of the product of two messages can be determined as the product of the encryptions of its factors.

$$c_i \equiv \varepsilon_e(m_i) \equiv m_i^e \ mod\ N \tag{2.15}$$

$$c_1 \equiv m_1^e \ mod\ N \tag{2.16}$$

$$c_2 \equiv m_2^e \ mod\ N \tag{2.17}$$

$$c_1 \cdot c_2 \equiv m_1^e \cdot m_2^e \ mod\ N \equiv (m_1 \cdot m_2)^e \ mod\ N \tag{2.18}$$

Analogous is true for the decryption, where only the exponent is changed. An inversion of this homomorphism is widely used in RSA blind signatures. The blind signatures utilize that the signer of $m_1 \cdot m_2$ cannot determine the factors m_1 and m_2, whereas the requester of the signature can choose $m_2 = k^d \ mod\ N$ by encrypting a random k with the public key of the signer and further determine $k^{-1} \ mod\ N$.

$$(m_1 \cdot k^d)^e \ mod\ N \equiv m_1^e \cdot (k^d)^e \ mod\ N \equiv m_1^e \cdot k \ mod\ N \tag{2.19}$$

With k^{-1} the requester can then determine $m_1^e \ mod\ N$ without letting the signer know m_1.

$$m_1^e \cdot k \ mod\ N \cdot k^{-1} \ mod\ N \equiv m_1^e \cdot k \cdot k^{-1} \ mod\ N \equiv m_1^e \ mod\ N \tag{2.20}$$

The new privacy homomorphism of Domingo-Ferrer is an enhancement of Rivest's et al. privacy homomorphism [38] that supports modular addition, subtraction and multiplication. The enhancement was necessary since it has been shown by Brickell and Yacobi [39] that it is vulnerable to a known-plaintext attack. The enhancement leverages the indeterminism of $k \geq 2$ summands similarly to the slicing in SMART [35] to withstand known-plaintext attacks. Therefore the cleartext a is split into k secret summands a_i, whereas the likelihood of guessing the summands decreases with increasing k. Hence, it is named a security parameter. The sum of these summands is modular equal to the secret cleartext $a \in \mathbb{Z}_n$ to a module of $n = pq$. p and q are secretly chosen prime numbers by the owner of the private data. With that, even small values less than n are not trivially encrypted. The owner of the private data further chooses two prime constants $r_p \in \mathbb{Z}_p$ and $r_q \in \mathbb{Z}_q$. The summands are multiplied to the summands. For big numbers it is known to be extremely difficult to determine the actual prime factors if these are also big enough. n can be public since it is equally difficult to factorize. The encryption of cleartext a is

$$\begin{aligned}\varepsilon_k(a) = ([a_1 r_p \ mod\ p, a_1 r_q \ mod\ q], [a_2 r_p^2 \ mod\ p, a_2 r_q^2 \ mod\ q], \ldots, \\ [a_k r_p^k \ mod\ p, a_k r_q^k \ mod\ q]).\end{aligned} \tag{2.21}$$

While the cleartext is $a \in \mathbb{Z}_n$, the ciphertext is $\varepsilon(a) \in (\mathbb{Z}_p \times \mathbb{Z}_q)^k$. The operation on the ciphertexts (a vector addition/subtraction) that is homomorphic to the plain addition/-subtraction would also map into $(\mathbb{Z}_p \times \mathbb{Z}_q)^k$. Without the knowledge of p and q it can

only be narrowed to $(\mathbb{Z}_n \times \mathbb{Z}_n)^k$. That is for $k = 2$,

$$\varepsilon_k(a) = ([a_1 r_p \ mod \ p, a_1 r_q \ mod \ q], [a_2 r_p^2 \ mod \ p, a_2 r_q^2 \ mod \ q]) \tag{2.22}$$

$$\varepsilon_k(b) = ([b_1 r_p \ mod \ p, b_1 r_q \ mod \ q], [b_2 r_p^2 \ mod \ p, b_2 r_q^2 \ mod \ q]) \tag{2.23}$$

$$\begin{aligned}\varepsilon_k(a+b) = ([&a_1 r_p \ mod \ p + b_1 r_p \ mod \ p, a_1 r_q \ mod \ q + b_1 r_q \ mod \ q], \\ [&a_2 r_p^2 \ mod \ p + b_2 r_p^2 \ mod \ p, a_2 r_q^2 \ mod \ q + b_2 r_q^2 \ mod \ q])\end{aligned} \tag{2.24}$$

The addition/subtraction of the vector elements may be executed modular to n, if n is public. The multiplication with an un-encrypted integer can be derived from this addition/subtraction, too. Each vector element is multiplied with that un-encrypted integer. The operation, which is homomorphic to the multiplication, pairwise multiplies all the elements containing r_p as well as those containing r_q. The multiplication may be modular to n, too. For simplicity the *mod n* is omitted in the given equations. The exponents of r_p or r_q respectively are the degrees, which add up during multiplication. Similar to Equation (2.24), terms with the same resulting degree are added up (see Equation (2.25)).

$$\begin{aligned}\varepsilon_k(a \cdot b) = ([0,0], [&a_1 r_p \ mod \ p \cdot b_1 r_p \ mod \ p, a_1 r_q \ mod \ q \cdot b_1 r_q \ mod \ q], \\ [&a_1 r_p \ mod \ p \cdot b_2 r_p^2 \ mod \ p + a_2 r_p^2 \ mod \ p \cdot b_1 r_p \ mod \ p, \\ &a_1 r_q \ mod \ q \cdot b_2 r_q^2 \ mod \ q + a_2 r_q^2 \ mod \ q \cdot b_1 r_q \ mod \ q], \\ [&a_2 r_p^2 \ mod \ p \cdot b_2 r_p^2 \ mod \ p, a_2 r_q^2 \ mod \ q \cdot b_2 r_q^2 \ mod \ q])\end{aligned} \tag{2.25}$$

The decryption is done by multiplication of each element with the respective inverses r_p^{-1} and r_q^{-1} powered to the necessary degree.

$$r_p^{-1} r_p \equiv 1 \ mod \ p \tag{2.26}$$

$$r_q^{-1} r_q \equiv 1 \ mod \ q \tag{2.27}$$

Then the elements are added and solved by using the Chinese Remainder Theorem.

$$c = ([c_{1,p} r_p, c_{1,q} r_q], [c_{2,p} r_p^2, c_{2,q} r_q^2], \ldots, [c_{m,p} r_p^m, c_{m,q} r_q^m]) \ with \ m = lk; l \in \mathbb{N} \tag{2.28}$$

$$\begin{aligned}D''(c) = ([&c_{1,p} r_p r_p^{-1} \ mod \ p, c_{1,q} r_q r_q^{-1} \ mod \ q], \\ [&c_{2,p} r_p^2 r_p^{-2} \ mod \ p, c_{2,q} r_q^2 r_q^{-2} \ mod \ q], \ldots, \\ [&c_{m,p} r_p^m r_p^{-m} \ mod \ p, c_{m,q} r_q^m r_q^{-m} \ mod \ q])\end{aligned} \tag{2.29}$$

$$D'(c) = (\sum_{i=1}^{m} c_{i,p} r_p^i r_p^{-i} \ mod \ p, \sum_{i=1}^{m} c_{i,q} r_q^i r_q^{-i} \ mod \ q) \tag{2.30}$$

$$D'(c) = (c_p \ mod \ p, c_q \ mod \ q) \tag{2.31}$$

$$D(c) = c_p q q^{-1} + c_q p p^{-1} \ mod \ n \ with \ q q^{-1} \equiv 1 \ mod \ p; p p^{-1} \equiv 1 \ mod \ q \tag{2.32}$$

Even though this privacy homomorphism provides good privacy protection within the operations addition, subtraction and multiplication (encrypted and mixed) on integers, it

does not explicitly address division or floating point values. When considering floating point values in fractional representation (numerator and denominator) as a pair of two integers also the division could be derived. In contrast to the original privacy homomorphism by Rivest et al., it is resilient to known-plaintext attacks. Also numbers less than p and q are non trivially encrypted. On the backside the multiplication is computationally expensive and memory-intensive. Each multiplication of two encrypted values of degrees k_1 and k_2 result in an encrypted value of degree $k_1 + k_2$. This increases the effort for any following operation on this result. To reduce the degree of an intermediate result this would have to be transfered to the owner of the original values for decryption and new encryption, which produces values of degree k only. While this would reduce the memory and computation effort on the encrypted values in the privacy processing environment, it would increase the communication and the computation effort on the data owners side. The optimal distribution of further homomorphic operation and re-encryption of intermediate results at the owners side can be determined, but is dependent on the particular expression that has to be calculated. Further, not every task can be accomplished using the four basic arithmetic operations. Hence, the set of possible applications relying on this homomorphism is limited. Those applications must purely rely on calculations using the basic arithmetic operations. More complex math, text processing or even any generic data processing using loops, conditions and other programming constructs are not feasible. The aforementioned scenarios cannot be addressed with this approach.

Vanish The authors of [40] aim at self-destruction of data. Their approach named Vanish gives the data owner control over unlimited copies of that data even without knowing the existence of such copy. Similarly to DRM mechanisms (see Section 2.4), the actual payload is encrypted with a key randomly assigned to that data. That key is stored at a different place than the encrypted code and is the part that actually self-destructs. The only ways for an attacker to get hold of the data, is to gather the complete key while it exists or to obtain a decrypted copy of the data. Retroactive attacks are not possible. To accomplish the destruction of the encryption key this is split into shares using threshold secret sharing [41] and distributed in a Distributed Hash Table (DHT). The indexes to which the shares are distributed is derived from a random access key that is stored with the encrypted private data. The underlying DHT of Vuze[2] provides automatic deletion of aged entries after eight hours and OpenDHT [42] lets even specify a maximum duration of the stored entries. That way it is ensured that key shares disappear over time. If DHT nodes disconnect, shares may be removed before the key share expiration. When reconnecting, DHT nodes are usually assigned to a different index scope responsibility that effectively makes the key share unavailable. To cope with that, the threshold ratio of the required key shares for reconstruction should be less that 100%. The key shares disappear at least after the expiration given by/to the DHT. When less than the required

[2] http://www.vuze.com

number of key shares can be retrieved from the DHT, the data cannot be decrypted anymore. A collaboration attack with nodes that do not delete their key shares requires as many attackers as the threshold, which never change their index scope responsibility, and the unlikely event that these attackers where chosen by the random access key. With the millions of nodes in the DHT of Vuze, such an attacking approach is negligible. Despite the Vanish system can rely on the oblivion of the DHT, it has to trust the receiver of certain private information that she/he does not keep an un-encrypted copy nor any party keeps the complete key. To ensure this, the key share aggregation and reconstruction as well as the decryption should be executed within a closed software environment like the Protected Media Path (PMP).
Since Vanish imposes no requirements on the type of the data to be protected, it appears to be applicable for generic and complex services and scenarios as the described in Section 1.2. On the downside, the data is not protected while being processed. That is, the services have to be trusted not to keep un-encrypted copies of the data or the key. Vanish protects only from retroactive attacks. Its data and key distribution scheme may be applied in the PGEC to distribute data, keys and permission rules among the PGEC instances. This can enable privacy protected services even without current online presence or connectivity of users.

2.2 Access Control Technologies

Independent of whether data to be accessed and processed is encrypted or not, a broad spectrum of technologies can be used to control this access. The access can be controlled on various levels of abstraction. Commonly known are access modifiers in object oriented programming languages such as Java, C++ or C#. They use keywords, e.g., `public`, `protected`, `private` or `internal`, to control access of objects to methods and fields in other objects or classes. The access of users or services to a system at whole or entities in the system is controlled on a higher level of abstraction. This includes also applications acting on a user's behalf or initiative, i.e., processes in the user's context. Some of the technologies used here are rather declarative, e.g., eXtensible Access Control Markup Language (XACML) where others actively implement protection mechanisms. Other technologies address even privacy aspects, e.g., Kerberos (see Section 2.3.1) and various Role Based Access Control (RBAC) systems. The following section discusses a few access control techniques with regard to their applicability in privacy protection systems as the PGEC or for anonymous access control.

2.2.1 XACML

Access control and security policies have to be enforced in various areas of information systems. Those could be e-mail and remote-access systems as well as companies business systems restricting access to internal business data. The number of elements and points of enforcement of security policies increases with the size of the enterprise. The

policy elements may even be managed by multiple departments, e.g., the Information Systems department, Human Resources, the Legal department or the Finance department. The independent configuration of each enforcement point is current practice. Due to the lack of a consolidated view of the safeguards, a modification of such access control and security policy is expensive and unreliable. Even without the need of modification, such implementation of security policy is hard to communicate to consumers, shareholders and regulators.
In an effort of addressing consolidated view, modification and comprehensible communication, a common language for expressing security policies was developed. The eXtensible Markup Language (XML) provides easy extensibility to the requirements of access control, a widespread platform and tool support. Based on that, XACML has the potential to express any access control and security policy independent of application and corporate internal characteristics. With an enterprise-wide implementation of XACML, a unified management including writing, reviewing, testing, approving, issuing, combining, analyzing, modifying, withdrawing, retrieving and enforcing of the security policy becomes feasible for all components of the information systems of the enterprise.
Listing 2.1 displays a very simple policy as an example for XACML. It defines a rule that permits (line 12) access to any element (line 10) for each subject that has an e-mail (RFC822) address in the domain med.example.com (line 21). The determination whether the subject fulfills the given requirement is specified via the match function `rfc822Name-match` (line 19) and its two parameters (lines 20-23). The essential part of the `rfc822Name-match` function is shown in Listing 2.2.

The XACML language is very generic with regard to the description of target elements and accessing subjects. Even the matching functions can be extended using math expressions described in MathML [44]. Besides the actual policies including the access rules, XACML also formally describes the requests for data. The requests share some elements with the policies, which simplifies the comparison and matching on evaluation of an incoming request against a given policy. The elements and their structure are defined in an XML Schema Definition (XSD) provided by Organization for the Advancement of Structured Information Standards (OASIS)[3]. It allows static syntax and validity checks. With including other name spaces, it can further embed structures from other XML documents to restrict and enforce access control on data items in XML-based databases.
For applications and frameworks that control access to any data or entity the utilization of this language is an appropriate means to maintain a somewhat standardized way of access and compatibility to other systems. Due to the flexible description of rule subjects including according match functions, it is even applicable to privacy aware and anonymized access control systems.

[3] http://www.oasis-open.org/home/index.php

Listing 2.1: Example Policy in XACML[43]

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os http://docs.oasis
             -open.org/xacml/access_control-xacml-2.0-policy-schema-os.xsd"
        PolicyId="urn:oasis:names:tc:example:SimplePolicy1"
        RuleCombiningAlgId="identifier:rule-combining-algorithm:deny-overrides">
  <Description>
    Medi Corp access control policy
  </Description>
  <Target/>
  <Rule RuleId="urn:oasis:names:tc:xacml:2.0:example:SimpleRule1"
        Effect="Permit">
    <Description>
      Any subject with an e-mail name in the med.example.com domain can perform any action
           on any resource.
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              med.example.com
            </AttributeValue>
            <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0
                 :subject:subject-id" DataType="urn:oasis:names:tc:xacml:1.0:data-
                 type:rfc822Name"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
    </Target>
  </Rule>
</Policy>
```

Listing 2.2: Code snippet from `rfc822Name-match` function in Sun's XACML implementation

```
String arg0 = ((StringAttribute)(argValues[0])).getValue();
String arg1 = ((RFC822NameAttribute)(argValues[1])).getValue();

if (arg0.indexOf('@') != -1) {
    // this is case #1 : a whole address
    String normalized = (new RFC822NameAttribute(arg0)).getValue();
    boolResult = normalized.equals(arg1);
} else if (arg0.charAt(0) == '.') {
    // this is case #3 : a sub-domain
    boolResult = arg1.endsWith(arg0.toLowerCase());
} else {
    // this is case #2 : any mailbox at a specific domain
    String mailDomain = arg1.substring(arg1.indexOf('@') + 1);
    boolResult = arg0.toLowerCase().equals(mailDomain);
}
```

2.2.2 Discretionary Access Control (DAC)

It is commonly known to provide credentials to gain access to a system, i.e., a name or login and a respective password. This holds true for logins to operating systems, web-sites, e-mail accounts and so on. Actually it is only an identification (user name) and authentication (proof of knowledge). An authenticated identity provides the basis for the actual access control. If there is more granularity required than an all or nothing access to a system, this identity has to be mapped to respective permits or access

rights. On community web-sites such as Facebook[4], it maps to full access to the profile belonging to that given identity. Further, it maps to some access to profiles that state this identity being a 'friend'. The identity to access right mapping can also be cascaded. An identity may belong to a group to which certain rights are granted. Well known examples are Linux' and UNIX' file systems. Access rights are reflected as attributes to the file or directory (or other entity) using a bit mask representing *read*, *write* and *execution* rights to the owner (an identity), to a group or all others (the group to which all known system identities belong). These access permissions attributed to an entity are sufficient to decide whether the access requested by a particular identity is to be granted. Such mechanism is a DAC.
Systems that evaluate the access right in the moment of access using a mapping with this identity are obviously not suited for high privacy demands. It can still be used as a widely accepted paradigm for identification and authentication in applications where the time and sequence of access of an identity to certain services, systems and data entities are not in the focus of protection.

2.2.3 Mandatory Access Control (MAC)

In opposition to the DAC, the MAC examines the access on attributes of the accessing subjects and the to-be-accessed objects using a set of authorization rules (policy). MAC is commonly associated with multilevel security systems. In such systems, the required attribute for the objects is the security level it belongs to, e.g., public, confidential, secret, top secret. A subject also possesses a security level attribute. The simple rule is that the security level of the subject must be equal or higher than the security level attributed to the object to be accessed. Rules might be more complex in other scenarios.
Since the actual access examination requires the attribute only instead of the identity, it supports somewhat privacy. If the attribute can be provided without revealing the identity of the subject such mechanism can be incorporated in an anonymous access control system. The system presented in Chapter 3 is actually capable of providing these attributes without an identity. The attributes, which directly map into rights, are organized in a hierarchy. This makes it equivalent to an RBAC system [45].

2.2.4 RBAC

RBAC is a more flexible approach to restrict system access to authorized users than MAC and DAC. In organizations with lots of activities and functions as well as lots of users/subjects, the maintenance of discretionary mapping of subject to objects cause high effort. Subjects may change their functions or position within or drop out from an organization. Further, new subjects mights also get in the organization more frequently

[4] http://www.facebook.com

than the organizational structure is updated. Stability in this mapping can be gained by modeling the activities and functions as a role [46]. Assigning subjects to roles, to move them from one role to another or to un-assign a subject from a role, significantly reduces the administration effort. Flexibility is achieved through the following facts.

- A subject can have multiple roles.
- A role can have multiple subjects.
- A role can have many permissions.
- A permission can be assigned to many roles.

Roles can even be composed from other roles, inheriting the assigned permissions. In Figure 2.1 exist nine users and six objects with a permission on each object. The users are assigned to three roles that include each other. For instance, User 9 is a doctor and has by that the permissions of an intern and a healer, too. User 5 is an intern inheriting the permissions from the healer role, but excluding permissions granted to doctors only. A role could also inherit from multiple roles, which is not depicted.
The indirection between a permission and a subject make roles a suitable vehicle to protect the privacy of a user/subject when executing an action requiring certain permission. For full protection, the presentation and authorization of one or more roles should be possible without identification of the subject taking on these roles. This requirement is comparable to the requirement stated in Section 2.2.3 and is solved in Section 3.1. It is obvious that the degree of privacy protection raises with the number of subjects taking the same role.

2.3 Authentication Technologies

This section describes two authentication technologies, which are applied in information systems besides straightforward user name and password checks. Other technologies claiming authentication, such as Remote Authentication Dial-In User Service (RADIUS) [48], usually embed those basic authentication technologies.

2.3.1 Kerberos

Kerberos (RFC1510 [49]) describes an architecture to authenticate users and services in open and untrusted networks. It provides authentication without sending a password, neither as plain text nor encrypted. The Kerberos architecture exchanges encrypted tickets for authentication between the involved parties, which provide protection against eavesdropping in the network. Besides the user and the service, there exist a Kerberos Authentication Service (KAS) and a Ticket Granting Service (TGS). The KAS provides

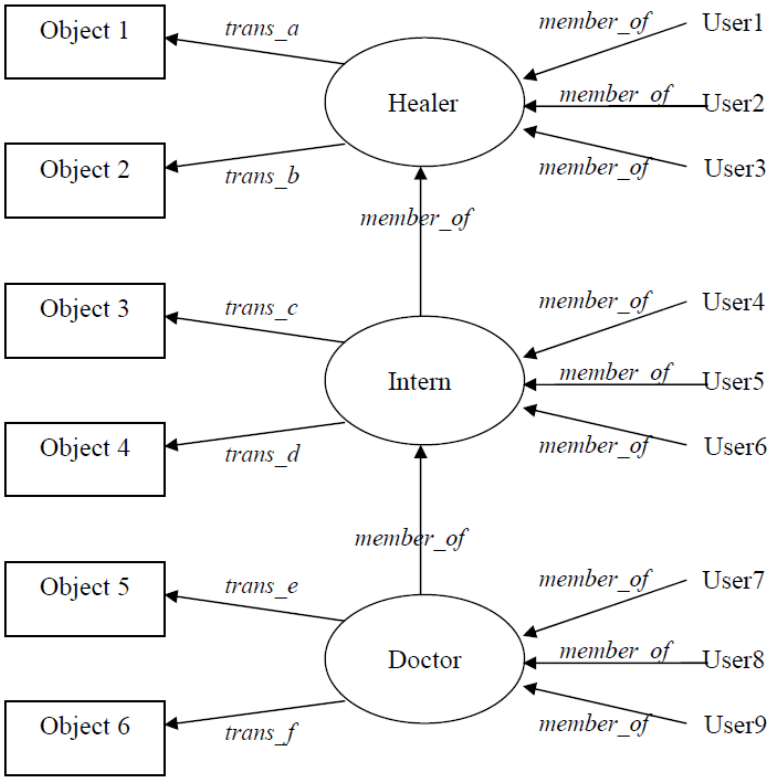


Figure 2.1: Multi-Role relationships [47] in an RBAC system. Access rights to *Objects* are assigned to respective roles *Healer*, *Intern* or *Doctor*. The roles include each other by a *member_of* relationship. That is, a *Doctors* transitively inherits the access rights from *Interns* and *Healers*. *Users* assigned to a particular role are explicitly granted with the rights assigned to that role and implicitly with the rights assigned to the more general roles.

a Single-Sign-On, which means that users have to login at only one entity but are enabled to use all services using the Kerberos system. By that, the KAS is the only party that learns the user identity, whereas the TGS learns the service identity to be used by certain client. The mapping between the client and the user is known to the KAS only. That is, as long as the KAS can be fully trusted, the privacy of the user is protected, with regard to her/his identity. Since, the ticket granting ticket, issued by the KAS, is encrypted with a key, shared only between KAS andTGS, it cannot be assured that the user identity remains secret from the TGS. Also other channels for collusion between KAS and TGS are not excluded. Similarly, the service may learn a users identity by collusion with KAS andTGS.

In [50] the Kerberos protocol has been adapted to protect users' privacies by changing the roles in the architecture. It assumes a platform containing the KAS and a service providing private information about users. The latter service provides the information only on presentation of a ticket issued and signed by the owner, which has the role of the

TGS. That is, every user of the platform runs her/his own TGS. The service requesting private information on the users behalf appears as the Kerberos client. Here, privacy is not protected for the services identity but rather for the centrally available information of the platform users, such as positions.

2.3.2 Pre-shared key (PSK) Authentication

Pre-shared keys (PSKs) are a priori known secrets between communicating parties. Usually those are symmetric keys. PSK is used for implicit authentication, like it is used in Kerberos. It is further commonly used in WLAN encryption like, Wired Equivalent Privacy (WEP) and Wi-Fi Protected Access (WPA). This symmetric key can be used directly for encryption of the communication between the parties sharing that key. In Internet Protocol Security (IPsec) [51], the symmetric key is generated at runtime and exchanged using Diffie-Hellman key exchange algorithm [52]. Mutual authentication is accomplished using the PSK. Application of PSKs is feasible for small networks. In larger networks a Public Key Infrastructure (PKI) or RADIUS (RFC2865 - RFC2869)[48] provide better maintainability. A PSK authentication is easy to implement and provides good privacy protection, since the key cannot be mapped to an identity. On the other hand, it provides no means for revocation. If a user's granted access shall be revoked, all others have to pre-share a new key.

2.4 Digital Rights Management (DRM)

DRM summarizes technologies that limit or control the access to digital content such as video, audio, documents or software. Usually it is applied by copyright holders to prevent their content from being used in ways other than supposed and to protect their business model on exploiting their creative content. This includes prevention from copying, transforming or consumption over the time permitted. Those technologies effectively prevent the protection means of their contents from being circumvented by manipulating the file, such as stripping of some header or skipping an initial serial number check, which has been used on numbers of software applications. Most of them encrypt the actual content and use various techniques to hide or control the keys. Apple's[5] FairPlay is quite simple to understand. The media content is encrypted with an AES master key. This master key is encrypted with an individual user key, which is generated during the purchase. Since only the master key has to be individually encrypted and overwritten in the file, this can be accomplished in no time making it a reasonable vehicle even for larger media files. At the time of media consumption, the QuickTime[6] software contacts the license server to check whether the particular user key can be retrieved. This enables the license server to revoke user keys, e.g., due to elapsed rental time. Only with this user key the encrypted master key in the media file can be decrypted, which

[5] http://www.apple.com
[6] http://www.apple.com/de/quicktime/

consequently enables to decrypt the media.
The most commonly known and probably the oldest widely used DRM system is the Content Scrambling System (CSS)[7], which was introduced in 1996 and is used by almost every commercially produced DVD. The required keys are hidden in the lead-in area of the protected disc and in licensed hardware or software players.
Since the CSS uses relatively short encryption keys of only 40 bit length, it has been cracked relatively easy [53]. Following DRM systems have been designed using stronger encryption schemes. The Advanced Access Content System (AACS)[8] that has been introduced for HD-DVD and Blu-Ray discs, applies the Advanced Encryption Standard (AES) with key lengths of 128 bits and a more complex key organization. The first approaches to break this system extracted the keys from software players using debuggers and memory space inspection [54][9,10]. Compromised keys have been revoked, which prevented the affected players from decrypting newer digital content.
As we learn from all these successful breaches, there are only two ways to prevent such attacks. That are, either using hardware based protection mechanisms on the PC platform, such as Trusted Computing [55], or not providing the keys to software players. It is expected, that a software-only based PGEC is vulnerable to the same attacks. For a productive environment and a widely distributed usage it should rely on a Trusted Computing Platform. Besides an effective key protection, which can eventually not be accomplished in a software only approach, there are other measures for runtime data protection to be taken. Those can be learned from software based DRM systems that are deeply integrated in the Operating System (OS). The most prominent, if not even the only one, is the Protected Media Path (PMP) by Microsoft[11]. The PMP is executed in a protected environment, which supports to protect any data flow within, especially protected content from and through Windows Vista. Protection from inside attacks is attained by permitting only trusted code in the Protected Environment (PE). Trusted code is digitally signed and the used key is certified with a PMP-PE certificate issued by Microsoft. Participating user-mode display driver component, user-mode audio driver components or audio processing objects and Media Foundation[12] pipeline plug-ins (codecs, mf transforms) use this certificate for signing. Other modules such as kernel-mode display device drivers and kernel-mode audio driver components require other types of certificates (Protected Video Path - Output Protection Management (PVP-OPM) , Protected Video Path - User-Accessible Bus (PVP-UAB) and Protected User Mode Audio (PUMA)) to be embedded and used for signing. The PVP-OPM for instance ensures that a PC's integrated graphics adapter outputs provide reliable control of output protection schemes, such as High-bandwidth Digital Content Protection (HDCP),

[7] http://www.dvdcca.org/css/
[8] http://www.aacsla.com
[9] http://forum.doom9.org/showthread.php?t=122664
[10] http://forum.doom9.org/showthread.php?t=122969
[11] http://msdn.microsoft.com/en-us/library/aa376846(VS.85).aspx
[12] http://msdn.microsoft.com/en-us/library/ms694197(VS.85).aspx

Macrovision, and Copy Generation Management System-Analog (CGMS-A) as it is required under license agreement with content owners [56]. This allows to extend the protected environment with a Protected Video Path (PVP) and PUMA. Hence, the protected content never leaves the PMP as un-encrypted data between the media source and the physical display or audio device (see Figure 2.2).

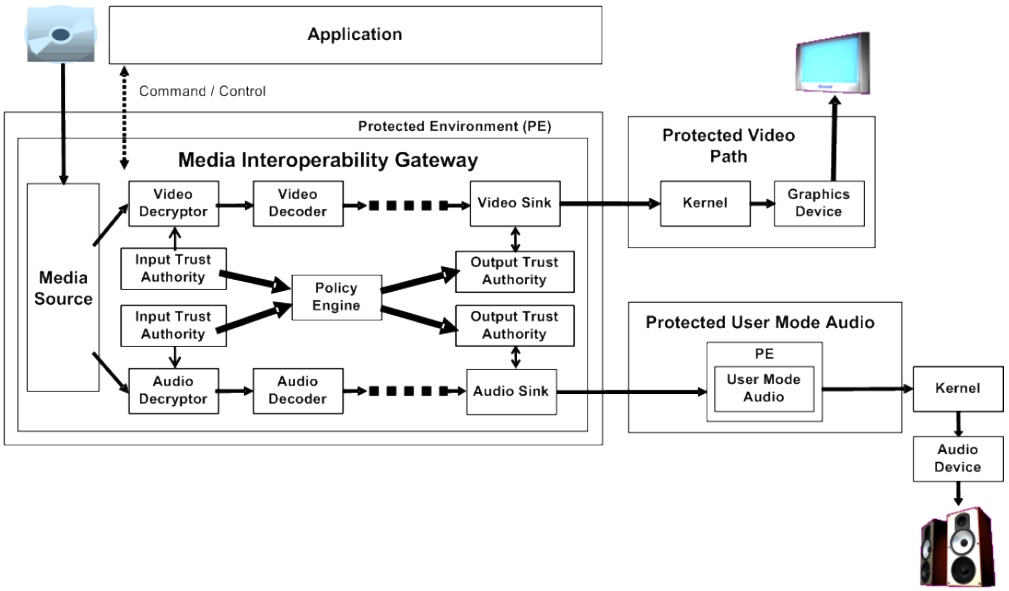


Figure 2.2: Protected Media Path (PMP) Overview [56]. It displays data and control flow from the protected digital media to the rendering devices. Decryption of data happens only within a specially Protected Environment (PE) by trusted components. Decrypted data may further leave the PE only via Protected Video Path (PVP) and Protected User Mode Audio (PUMA). This ensures that no copies of the decrypted streams can be obtained.

Protection from outside tampering with the PE and the components running within is approached by a number of measures.[13] These apply to the operations that a typical process can or cannot perform on a protected process or its threads:

- Inject a thread into a protected process
- Access the virtual memory of a protected process
- Debug an active protected process
- Duplicate a handle from a protected process
- Change the quota or working set of a protected process
- Set or retrieve context information of a thread

[13] http://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/process_Vista.doc

- Impersonate the thread

The OS kernel monitors continuously which components and modules are loaded in the environment by periodically performing a cryptographically-protected handshake with the PE components. If an untrusted kernel mode component is present, this handshake fails and the PMP stops processing.
For further protection of the PE and content in the PMP, revocation of trusted components is introduced. Compromised trusted components can be revoked to prevent further execution and replaced with a newer trusted version of the component when it becomes available.
Still the PE is completely implemented in software, which makes it vulnerable to software-based attacks. Patching the Windows kernel remains possible.[14] It is not known up to here, whether the content protection can be successfully circumvented using this approach. Even though it might appeal to execute the PGEC inside this PE, which might increase the required effort to break in, there are various obstacles. Currently only media applications get certified to run in the PMP. The bigger problem is that every service that is going to run inside the PGEC has to be signed and certified by Microsoft then, which is likely not to happen. For the development of a PGEC, certain security measures from the PE and PMP may be inherited but still have to be adapted to the needs of a multi-purpose protected environment that shall even run untrusted and uncertified code as well.

[14] http://www.alex-ionescu.com/?p=35

Chapter 3

Anonymous Access Control

Usually access control goes along with identification and authentication. This process often requires to give a proof of identity or to verify released personal data. In real life this proof can be given by confidential documents like passports, whereas electronic media prefer the usage of credentials or digital certificates for these issues. Digital certificates are trustworthy documents that are issued by a trusted Certificate Authority (CA), where the user has been identified through official documents. Certificates issued by a CA can be used as digital identity cards for electronic services. Given the user data and her/his certificate, a service can request the CA to check the user data. For example, an age-restricted pay-TV service could verify the age of a user by her/his certificate and thereby authorize the service use. Besides the age of the certificate's owner, the verification unnecessarily releases additional information contained in the certificate. Thereby the service gains all certified user data and the CA acquires knowledge about the usage of the certificate. That way an anonymous usage of certificates becomes infeasible. In consideration of privacy, a service should receive essential information only and the CA does not need to know where, when and for what purpose an issued certificate is used for. The presented certification method, which was also published in [57], allows evidencing granted rights by the use of anonymous individual-related certificates. The content and the regularity of any issued certificate are preserved while the legal ownership can be proved anonymously. In the pay-TV example, neither the CA can find out who consumed which content nor is the service able to map certificates to real identities, not even in mutual collaboration.

k-Anonymity Full anonymity and individual-relation of certificates are actually contradictory. To solve this conflict pseudonyms are used, which can be mapped to an individual only by that individual. The other information in the certificate, i.e., the granted right, is chosen such, that it cannot be matched into an individual. This introduces *k*-anonymity, which is defined as follows:

"Each release of data must be such that every combination of values of quasi-identifiers can be indistinctly matched to at least k individuals [58]."

That is, the presenter of a certificate can be identified only as one of the k individuals, which are granted the particular right denoted in the certificate. When presenting multiple certificates at the same time, the presenter must be aware that her/his k-anonymity decreases to the intersection of the respective k_i possible individuals quasi-identified from the presented certificates i. In the worst case this may even result in $k = 1$, i.e., no anonymity. Hence, the presented system especially benefits from large user groups and relatively low number of required certificates to be presented for access authorization.

3.1 Technology

There exist many approaches that consider privacy and anonymity for trustable electronic media. The major goal is to establish trustable user data and authentication methods for service usage. Chaum developed the concepts of credential system [59] and group signature schemes [60] to establish verifiability on user data. Credential systems are made to give a proof of identity when interacting with electronic services. Signatures are used to prevent electronic documents from being changed by unauthorized entities. Nowadays, certificates are the preferred means for authorization and identification issues. Increasingly providing privacy and anonymity for users comes into consideration when interacting with services.

Most approaches focus on enhancing the user's privacy towards services, e.g., by using pseudonyms in certificates and e-cash schemes as provided in [61, 13]. Releasing data implies a trusted third party, for example a CA or a bank, where the pseudonym can be mapped to the real identity of the user. Services may request the trusted third party to verify the correctness of certified data for a given pseudonym. Hence these approaches support anonymity of users against services but enable the trusted third parties to track issued certificates. Even privacy enhancing certification frameworks as presented in [62] and [63] still use a trusted third party and do not provide a solution for fully anonymous granting of certificates. Preventing trusted third parties from tracking issued certificates either needs using untraceable pseudonyms or blinding the content of the certificate before signing. Chaum determined the idea of blinding certificates [64] that allows concealing the content from the CA. Therein the CA is requested to issue a certificate with unknown content. Consequently, this method is inefficient particularly with regard to verifiability of certified data. The CA might issue invalid certificates or could grant certificates for unauthorized identities. In an approach to avoid this, the CA demands n blinded versions of the content to be transmitted and then requests to reveal the blind factors of $n - 1$ versions. Thereby, it can discover attempts of fraud. While the remaining subject is randomly chosen, the attacker has limited chances ($p = \frac{1}{n}$) to foist a fraudulent certificate.

Even a combination of known approaches would not bear up against a collaborated at-

tack of services and trusted third parties. To the best of knowledge, there exists no known approach that provides fully anonymous use of certificates, where neither the service nor the CA is able to map the certificate to a real identity. Nevertheless, the approach presented herein can guarantee that only valid certificates are issued by the CA.

3.2 Function Principle

This section sets up the means for a privacy aware certification system. In such system a user retrieves pseudonymous certificates for rights legally granted to her/him. Thereby the CA does not get hold of the pseudonym. Services gain the pseudonym from verification of the presented certificates by the CA. For authentication the user proves ownership of the pseudonym to the service.

3.2.1 Requirements

It is supposed that users have certain rights, either by explicit grant or due to physical properties, e.g., age. If a service requires certain rights those must be presented by a potential user. Usually this is done by authenticating the user, whereby the service knows which rights are granted to her/him. For an anonymous access control it is not required and even undesired that a user is known to the service she/he uses. Thus, the user has to present the granted rights explicitly. To ensure that a user presents valid rights only, those rights must be issued as certificates by a trusted authority, called CA, which knows all users and their respective granted rights. Certificates shall further be revocable or valid for a limited time. Prevention from double-spending of a certificate within the time of validity, as in digital payment schemes, is not required. In addition, certificates shall be prevented from eavesdropping, so that they cannot be used by other users. In an effort to prevent eavesdrops, the certificates are issued to particular users. To maintain privacy, it must not be possible to track the issued right certificate. As for privacy reasons, the CA shall not get to know which user presented a particular right certificate to a service. Hence, a right certificate must contain:

- a verifiable right
- a verifiable expiration date
- a means to prove ownership, which cannot be mapped to the user identity by the service, the CA or both

As proof of ownership, a pseudonym in form of a public-private key pair is used. Hence, the certificate contains the public key whereas the user possesses the private key of that pair. Since the certificate shall not be mapped to the user's identity, the authority must not know the pseudonym when issuing the certificate. The way to certify something without knowing it is blind signing [64], as mentioned. The downside of blind signing is that the signer has no means to verify what is to be actually signed. It is not reasonable

to sign a right or an expiration date in blind. Otherwise, the authority might sign a right that is not granted or for an unrealistic expiration date, which breaks the possibility of right revocation. A means to interdigitate the openly signed right and expiration date with the blindly signed pseudonym has been found. This enables to verify the right and the expiration date during signing and still not knowing the pseudonym within. The next section explains the math details of this interdigitated signature algorithm. In the subsequent section potential attacks from malicious users, CAs or eavesdroppers are addressed. Means to guard against those attacks are introduced.

3.2.2 Mathematical Background

The following shows the math of the algorithm. When applying blind signing, the only value that is under control of the signer is the key. So, the idea for interdigitating blinded information with verifiable information is to encode the verifiable information into the key itself. That can be accomplished be using unique signature keys for each right or expiration date. Since at least the set of expiration dates is unbounded, a static assignment of keys is not feasible. Therefore, it has been chosen to algorithmically build these keys. That is, signing some message with the $n + 1^{st}$ key is exactly the same as signing it with the n^{th} and once more with the first key. Thus, interdigitating an information i can be achieved by signing the blinded message i times. Obviously, the information that is openly interdigitated has be $\in \mathbb{N}$. Please note, despite the inductive description of the derivation of the $n + 1^{st}$ key, it is possible to directly calculate that key, which dramatically reduces the calculation effort of the signing process (see Section 3.3).
To map the verifiable information into $\mathbb{N}$ the rights and the possible expiration dates are numbered. For the expiration dates, this is as days from 2007-01-01. The blinded pseudonym is encrypted r times with a key dedicated for rights and t times with a different key for expiration dates, being r the number of the right to be certified and t the expiration date of the certificate. Obviously, a malicious user could try to get a pseudonym signed multiple times to gain rights with higher numbers that are not actually granted. Prevention of this illegal right acquisition is presented in Section 3.2.3. Encrypting a certain message r times and afterwards decrypting it r times with the respective keys of one RSA key pair, will result in the same original message (see Equations (3.1) to (3.4)). Therein, the public key is $(e; N)$, the private key is $(d; N)$ and the message to be encrypted is m.

$$ed \equiv 1 \ mod \ \varphi(N) \tag{3.1}$$

$$\Rightarrow m^{ed} \equiv m \ mod \ N \tag{3.2}$$

$$e^r d^r \equiv 1^r \ mod \ \varphi(N) \tag{3.3}$$

$$\Rightarrow (m^{e^r})^{d^r} \equiv m^{(ed)^r} \equiv m \ mod \ N \tag{3.4}$$

Let the user choose a pseudonym p. Further she/he chooses randomly and secretly a blind factor k. With k and the publicly known modulus of the signature key of the CA,

the user can calculate the modulus inverse k^{-1} with

$$kk^{-1} \equiv 1 \; mod \; N. \tag{3.5}$$

According to Chaum, the pseudonym or a hash value of it can be blinded by modular multiplying it with the encrypted blind factor. The public key of the rights authority is used to encrypt k. The result is given to the authority to be encrypted with the respective private key. Without actually knowing the factors in the product, this results in an encryption of the pseudonym p and a decryption of the blind factor k, which can be neutralized by multiplication with the inverse k^{-1}. In Equation (3.6) one can find the result of an openly signed pseudonym and compare it with the result of the blind encryption (see Equation (3.9)). Equations (3.7) and (3.9) are processed at the user's side and (3.8) at the authority's side. It can easily be seen, that the authority does not get to know the actual pseudonym p nor the encrypted pseudonym s. At the same time, the user gets a signed pseudonym, without knowing the authorities private key $(d; N)$.

$$p^d \equiv s \; mod \; N \tag{3.6}$$

User: $$p(k^e) \equiv p_{blinded} \; mod \; N \tag{3.7}$$

CA: $$p_{blinded}{}^d \equiv p^d(k^{ed}) \equiv p^d k \equiv s_{blinded} \; mod \; N \tag{3.8}$$

User: $$s_{blinded}k^{-1} \equiv p^d k k^{-1} \equiv p^d \equiv s \; mod \; N \tag{3.9}$$

In Chaum's blind signature the information is just: "Yes, this has been encrypted with the signer's private key." The intention here is to let it say: "This has been encrypted with the signer's private keys for right r or other stated purpose." Therefore the blind encryption procedure was extended in two ways. The first is to encrypt the blinded pseudonym multiple times. The number of encryptions encodes separate information, i.e., the right. This way the authority gains some control of what it signs. Still there is no knowledge about the pseudonym that is signed. Only, decrypting it exactly the number of times that it was encrypted results in the pseudonym, which can then be verified. If it cannot be verified, it is either not the pseudonym of the presenting user or was not decrypted the right number of times and hence, is not certifying the right it claims to.

User: $$p(k^{e^r}) \equiv p_{blinded} \; mod \; N \tag{3.10}$$

CA: $$p_{blinded}{}^{d^r} \equiv p^{d^r} k^{(ed)^r} \equiv p^{d^r} k \equiv s_{blinded} \; mod \; N \tag{3.11}$$

User: $$s_{blinded}k^{-1} \equiv p^{d^r} k k^{-1} \equiv p^{d^r} \equiv s \; mod \; N \tag{3.12}$$

The operations in Equation (3.11) are processed in the CA with knowledge of neither the pseudonym p nor the blind factor k. The operations in Equations (3.10) and (3.12) are executed on the user's side to generate a signed pseudonym without knowing the private key $(d; N)$ of the authority.
The second extension of the procedure is to use multiple key pairs to encode orthogonal information, i.e., the right and the expiration date. There is only one constraint on the keys to be used. All have to use the same modulus. Equations (3.13) to (3.15) show the

process of creating a blind signature with two key pairs [$(e;N)$,$(d;N)$ and $(\hat{e};N)$,$(\hat{d};N)$]. This can easily be extended to more than two key pairs as long as these use the same modulus.

$$\text{User:} \quad p(k^{(e^r\hat{e}^t)}) \equiv p_{blinded} \; mod \; N \tag{3.13}$$

$$\text{CA:} \quad {p_{blinded}}^{d^r\hat{d}^t} \equiv p^{d^r\hat{d}^t}k^{(ed)^r(\hat{e}\hat{d})^t} \equiv p^{d^r\hat{d}^t}k \equiv s_{blinded} \; mod \; N \tag{3.14}$$

$$\text{User:} \quad s_{blinded}k^{-1} \equiv p^{d^r\hat{d}^t}kk^{-1} \equiv p^{d^r\hat{d}^t} \equiv s \; mod \; N \tag{3.15}$$

Using both extensions of the blinding algorithm, the user gets an encrypted pseudonym as if being encrypted using the private keys of the CA but without letting the authority know the pseudonym. On the other hand, the authority can ensure that the signature is used for the intended purpose only, because the signature turns into a valid pseudonym only if decrypted using the public keys $(e;N)$ and $(\hat{e};N)$ the correct numbers of times r and t.
There is a chance that the pseudonym could be decrypted correctly even when applying a wrong right or expiration date. This is called a collision. A collision may occur when

$$e^{r_1}\hat{e}^{t_1} \equiv e^{r_2}\hat{e}^{t_2} \; mod \; \varphi(N). \tag{3.16}$$

With e and $\hat{e}$ being relatively prime to $\varphi(N)$ they and their powers belong to the multiplicative group of integers modulo $\varphi(N)$. The likelihood for finding two congruent pairs of $e^{r_1}\hat{e}^{t_1}$ and $e^{r_2}\hat{e}^{t_2}$ is the reciprocal to the order of that group, which is $\varphi(\varphi(N))$. According to [65] $\varphi(N) \geq \sqrt{N} \; \forall N > 6$. Hence, the likelihood of colliding rights and expiration dates is equal or less than $\frac{1}{\sqrt[4]{N}}$. For bit lengths of N at 1024 or more, this becomes negligible.

3.2.3 Attacks by Malicious Users

Up to here, the basic approach is still vulnerable to attacks like fraudulent falsification and eavesdropping. So, a malicious user is able to gain a right, which was not actually granted by requesting blind signatures for certain granted rights multiple times. That is, a user might be granted the rights r_1 and r_2, whereas a right $r_1 + r_2$ is not granted. Since the CA has no means to check the content of the pseudonym to be encrypted, it does not notice when encrypting an already encrypted pseudonym (see Equations (3.17) to (3.19)).

$$\text{User:} \quad p^{d^{r_1}}k^{e^{r_2}} \equiv p_{blinded} \; mod \; N \tag{3.17}$$

$$\text{CA:} \quad {p_{blinded}}^{d^{r_2}} \equiv p^{d^{r_1+r_2}}k^{(ed)^{r_2}} \equiv p^{d^{r_1+r_2}}k \equiv s_{blinded} \; mod \; N \tag{3.18}$$

$$\text{User:} \quad s_{blinded}k^{-1} \equiv p^{d^{r_1+r_2}}kk^{-1} \equiv p^{d^{r_1+r_2}} \equiv s \; mod \; N \tag{3.19}$$

A similar attack is gaining a not granted right with a number lower than the number of a granted right. Assume a right r_1 is granted, whereas the right $r_2 = r_1 - 1$ is not granted.

Once the user possesses a pseudonym signed for r_1 she/he can create a valid one for r_2 by decrypting it once with the public key of the CA, see Equation (3.20).

$$p^{d^{r_1}e} \equiv p^{d^{r_1-1}} \; mod \; N \tag{3.20}$$

To tackle these, both keys of a pair are used privately to the CA and only the modulus is shared with users to prevent from reduction of the certified right. An additional private-private key pair $(\bar{e}; N)$,$(\bar{d}; N)$ is used to count the number of signature steps. Independent of the right to be certified, the blinded pseudonym gets signed exactly once with $(\bar{d}; N)$. A valid signature must not be signed more than once with this key $(\bar{d}; N)$. In case, the attacking user tries to get a signature for an already signed pseudonym in order to increase the ordinal number of the signed right and to extend the expiration date, this key $(\bar{d}; N)$ is applied more than once. Hence, an attack is detected while verifying the pseudonym's rights.
In the blinding procedure the blind factor needs to be encrypted with the public keys of the CA before multiplying it to the pseudonym. Since all keys are private here, the blind factor has to be encrypted by the CA as in Equation (3.21). On the other hand, the CA must not know the applied blind factor. It could determine the pseudonym otherwise. It is feasible to use the CA-encrypted blind factor to the power of n modulo N with n being randomly chosen and unknown to the CA (see Equation (3.22)). After the blind encryption in Equation (3.23) only k^n remains, for which the modular inverse k^{-n} can easily be determined by the user via Equation (3.24).

$$\text{CA:} \qquad k^{\bar{e}e^r} \equiv k_{CA-enc} \; mod \; N \tag{3.21}$$

$$\text{User:} \qquad p({k_{CA-enc}}^n) \equiv p_{blinded} \; mod \; N \tag{3.22}$$

$$\text{CA:} \qquad {p_{blinded}}^{\bar{d}d^r} \equiv p^{\bar{d}d^r} k^{\bar{e}\bar{d}(ed)^r n} \equiv p^{\bar{d}d^r} k^n \equiv s_{blinded} \; mod \; N \tag{3.23}$$

$$\text{User:} \qquad s_{blinded} k^{-n} \equiv p^{\bar{d}d^r} k^n k^{-n} \equiv p^{\bar{d}d^r} \equiv s \; mod \; N \tag{3.24}$$

In fact, CA-encrypting the blind factor and encrypting the blinded pseudonym are the corresponding en- and decryption operations. This could be exploited to remove the flag, that reflects the number of signature steps, when signing a pseudonym twice. To prevent this, a pre-shared generator is used as k. Note, CA-encrypted blind factors may even be exchanged between users, which prevents the CA from encoding user identity related information into the blind factor. Since, now the service does not know the authority's keys, the decryption and verification has to be forwarded to the CA. If a user presents the signed pseudonym to a service claiming that it was issued for right r and expiration date t, the CA to decrypts the signed pseudonym r times with $(e; N)$, t times with $(\hat{e}; N)$ and once with $(\bar{e}; N)$ as shown in Equation (3.26).

$$s \equiv p^{\bar{d}d^r\hat{d}^t} \; mod \; N \tag{3.25}$$

$$s^{\bar{e}e^r\hat{e}^t} \equiv p^{\bar{d}d^r\hat{d}^t\bar{e}e^r\hat{e}^t} \equiv p \; mod \; N \tag{3.26}$$

3.2.4 Attacks by Malicious CA

While the CA knows the identity of a user, when blindly signing her/his pseudonym, it is possible for a malicious CA to append identity information to the blind signature by multiplication. To accomplish this, the CA chooses a large prime number f and indexes the known identities. Equation (3.23) is then extended to Equation (3.27) with i being the index of the identity. The un-blinding at the user's side does not notice this identity foisting, see Equation (3.28). During the verification of the signature, which is done at the CA, the signed pseudonym including the foisted identity is decrypted resulting in a product of pseudonym and the power of the chosen prime number f and the identity index (Equation (3.29)). As the format of the pseudonym is known, this product can iteratively be multiplied with the inverse of f until such well-known format is reached. The count of the required iterations is the identity of the user.

$$p_{blinded}{}^{\bar{d}d^r} f^{i\bar{d}d^r} \equiv p^{\bar{d}d^r} k^{\bar{e}\bar{d}(ed)^r n} f^{i\bar{d}d^r} \equiv p^{\bar{d}d^r} k^n f^{i\bar{d}d^r} \equiv s_{blinded} f^{i\bar{d}d^r} \ mod\ N \tag{3.27}$$

$$s_{blinded} f^{i\bar{d}d^r} k^{-n} \equiv p^{\bar{d}d^r} f^{i\bar{d}d^r} k^n k^{-n} \equiv p^{\bar{d}d^r} f^{i\bar{d}d^r} \equiv s f^{i\bar{d}d^r} \ mod\ N \tag{3.28}$$

$$(s f^{i\bar{d}d^r})^{\bar{e}e^r} \equiv p f^i \ mod\ N \tag{3.29}$$

This foisting may not be prevented but it can be detected by the user. Therefore, the user splits her/his chosen pseudonym into at least two factors p_j, see Equation (3.30). She/he lets the CA blindly sign each factor and the actual pseudonym as well in arbitrary sequence. That way, the CA cannot distinguish, which one is the pseudonym and which are factors, and hence, it has to treat all equally. If a malicious CA foists the signatures with identities the results are as in Equation (3.31) and (3.32). Multiplication of the signed factors must result in the signed pseudonym. If it does not, as in Equation (3.33), a manipulation by the CA is discovered.

$$\prod_{j=1}^{n} p_j = p \tag{3.30}$$

$$p_j^{\bar{d}d^r} f^{i\bar{d}d^r} \ mod\ N \tag{3.31}$$

$$p^{\bar{d}d^r} f^{i\bar{d}d^r} \ mod\ N \tag{3.32}$$

$$\prod_{j=1}^{n} (p_j^{\bar{d}d^r} f^{i\bar{d}d^r}) \equiv (\prod_{j=1}^{n} p_j)^{\bar{d}d^r} f^{i\bar{d}d^r n} \equiv p^{\bar{d}d^r} f^{i\bar{d}d^r n} \not\equiv p^{\bar{d}d^r} f^{i\bar{d}d^r} \ mod\ N \tag{3.33}$$

The service has no information on the identity of the user and hence, the CA does not get to know the identity either. Since the CA has never known the encrypted pseudonym, un-blinded during the signature process, it cannot map it to an identity in the verification process. The CA returns the pseudonym to the service if and only if the decrypted result for the claimed right, expiration time and exactly one flag is in fact a valid pseudonym. This can be achieved by checking the format of the pseudonym. While using the public key of an arbitrary key pair $((\tilde{e}; U),(\tilde{d}; U))$ concatenated with a secure hash of itself as pseudonym, the CA is enabled to verify the format of the decrypted pseudonym. If

the format cannot be verified, the certificate has probably been claimed with wrong parameters (fraudulent use of certificate). Further the public key $(\tilde{e}; U)$ can be used to encrypt a challenge (Equation (3.35)) that only the owner of the private key $(\tilde{d}; U)$ belonging to the pseudonym can decrypt (Equation (3.36)) and hence, prove her/his ownership of the pseudonym. If the user cannot decrypt the challenge, she/he is not the owner of the pseudonym (eavesdropped certificate).

$$p = (\tilde{e}; U) \tag{3.34}$$

$$c_{challenge}{}^{\tilde{e}} \equiv c_{enc} \ mod \ U \tag{3.35}$$

$$c_{enc}{}^{\tilde{d}} \equiv c_{dec} \equiv c_{challenge} \ mod \ U \tag{3.36}$$

3.3 Performance

According to the algorithm described, the exponents in the used RSA keys have to be raised to the power of the right or the expiration date respectively. As this is not a modular operation this power increases very quickly. Assuming an exponent of about 512 bit and a right 13 to be specified, the respective power has a length of 6656 bit. For expiration dates this length can be even much higher. Such huge numbers are hard to handle, but raising the encryption exponent to a power of n means exactly the same as encrypting n times. Thus, one could sequentially execute n modular operations using only the encryption exponent (see Equation (3.37)).

$$m^{(e^3)} \equiv ((m^e)^e)^e \ mod \ N \tag{3.37}$$

The sequential execution of n modular operations definitely decreases the amount of used memory but rather increases the calculation effort. In the presented approach all encryption operations, which use an encryption exponent raised to a power, are executed in the CA, which also knows $\varphi(N)$. Using the dependency in Equation (3.38) the calculation of the encryption exponent can be reduced from a memory and time consuming calculation of e^n to $e^{n \ mod \ \varphi(N)}$. Using the reduced exponent for encryption results in the same value as if using the very large power or encrypting n times using the original exponent e (see Equation (3.39)).

$$x \equiv y \ mod \ \varphi(N) \Rightarrow m^x \equiv m^y \ mod \ N \tag{3.38}$$

$$m^{(e^n)} \equiv m^{(e^n \ mod \ \varphi(N))} \ mod \ N \tag{3.39}$$

The calculation effort for this reduction is the same as one regular encryption with one key. So, the effective calculation effort for encrypting n times using the original exponent reduces from n to two operations. This reduction has to be accomplished for each key in use $e, \hat{e}, \bar{e}$. Hence, for the authority the encryption effort does not grow with the numbers r or t to be encoded, but linearly with the number of different keys to be used.

Network load and repeated verification effort Each certificate in the existing implementation that is transmitted to a service as proof of possession of a right has a size of about one to two kByte. While each certificate contains only one particular right, possibly in combination with a device or device group that this right applies to, various service require more than one right at a time. Hence, a user would have to transmit multiple certificates for each service call. Additionally, there is the forwarding to the CA and the verification of the certificates. To circumvent unnecessary network traffic and calculation effort at the CA, implicit sessions are proposed. By sending a set of certificates, a user retrieves a list of Transaction Authentication Numbers (TANs) from a service. These TANs are associated with the verified certificates or with the claimed rights respectively. The service assures that none of the generated TANs is already assigned to another set of verified certificates. By using sufficiently long numbers as the TANs, the probability for successful guessing a TANs becomes negligible. All these TANs belong to an implicit session. Each service call is accompanied with one of the TANs. With this TAN, the service can match the associated rights with required rights for that particular service call and deny service if necessary. So, the calculation effort for verification can be reduced to once per certificate and service. The network traffic per service call can be reduced from several kByte to a few bytes. The mentioned list of TANs serves an additional purpose while simultaneously protecting itself. After generation, the list is encrypted with the public keys from the pseudonyms in the associated certificates. In fact, it is encrypted with a random symmetric key (AES), which in turn is encrypted using the public keys. The encrypted AES key is transmitted along with the encrypted list of TANs. Only the legal owner of the presented certificates is able to decrypt the key and thereby the list. It is not required to return a 'response'to the 'challenge'. The usage of a valid TAN implies the correct decryption of the list of TANs proofing the legality of its recipient. Note, it is the nature of a TAN that it can be used just once. Hence, the service keeps track of used TANs. Possible interception of a TAN by a Man-in-the-Middle attacker is not addressed here, but can easily be circumvented if the connection between user and service is encrypted using an appropriately exchanged key, e.g., using Diffie-Hellman [52].
All blind factors are encrypted by the CAstarting from an a priori known generator k. As these are the basis for the mentioned power in Equations (3.21) to (3.24) and the same for every user, these can already be produced in advance of actual requests. This does not reduce, but rather increases, the overall effort of the CA but it increases the responsiveness of the CA upon request of such pre-encrypted blind factor. Analogous, the combined keys for verification according to Equations (3.25) and (3.39) are prepared and cached, which halves the response time for verification of a certificate.

3.4 Simplified Example

This section explains the presented anonymous access control method in an example. For better visualization of the involved parties and exchanged certificates the certified information is highly simplified.

3.4.1 Issuing of Certificates

In the example it is assumed, there is a user named Bob. A right with the reference number 2 has been granted to him. Bob wants to get his right certified for later use. The CA knows at least a user named Bob and the right number 2 granted to him. Please remember, the k-anonymity grows with the number of users granted with the same right. Further, the CA holds several key pairs sharing a common modulus for signing and verifying pseudonyms in certificates. For simplicity, only the key pairs for rights and expirations dates are displayed in Figure 3.1. Actually, further key pairs are used, e.g., for devices, priorities and especially for the signature counter.
Bob randomly chooses a key pair as a pseudonym. He creates an unsigned certificate, stating right 2 and a rather fictitious expiration date 3. In advance, Bob fetched a blind factor from the CA and raised it to a power with a random exponent, which is only known to him. He appends the hash value of the public key of his pseudonym with well defined markers and multiplies this with the power of the blind factor. That way, he gains a blinded pseudonym, i.e., it is not possible for anyone but Bob to regain the hash value from this nor even the public key itself.

Bob sends this unfinished certificate to the CA with releasing his identity to the CA. It can be assumed that appropriate means for identification and authorization of a user exist, e.g., RSA key pairs. Each user identity possesses such a pair, while its public key is known to the CA and associated with the identity. All responses to an identity are encrypted using this public key. Only the legal recipient can decrypt the response and hence, implicitly authenticate. This encryption is not depicted in the figures. The CA checks whether the right 2 claimed in the certificate is granted to Bob and the demanded expiration date lies within the next 24 hours. If this is positive, the blinded pseudonym gets signed with the signature keys for the respective number of times (see Figure 3.2). Afterwards it is returned using the above mentioned encryption, which implicitly authenticates Bob as the legal receiver of the signed certificate.

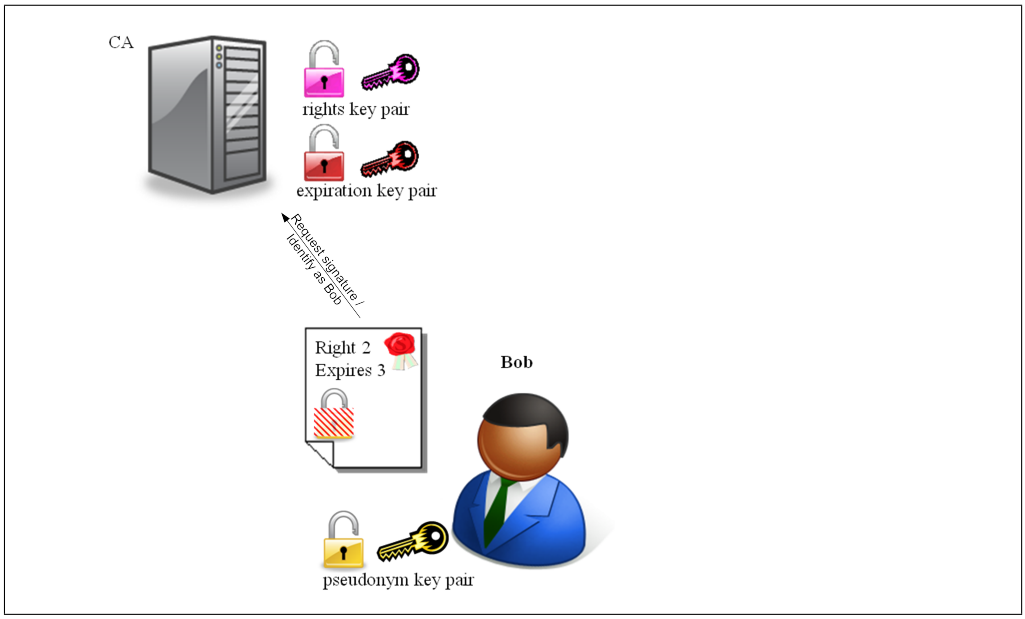


Figure 3.1: A user Bob known to the CA creates an unfinished certificate with right 2 and expiration date 3. He adds a blinded self chosen pseudonym, displayed as the hatched lock. The pseudonym is part of a key pair, which is like the blind factor only known to Bob.

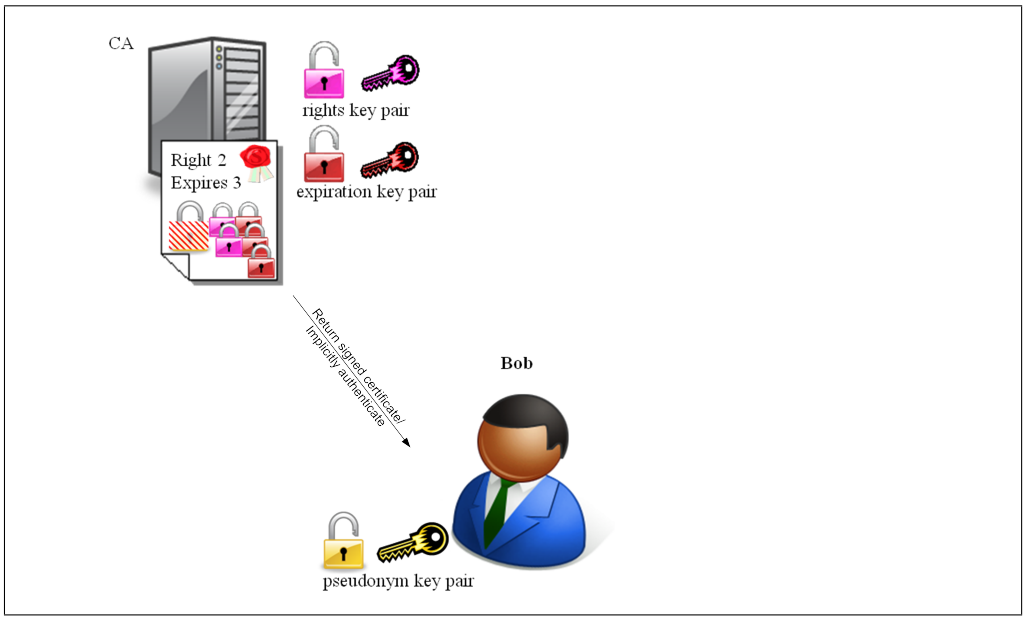


Figure 3.2: Bob sends the unfinished certificate to the CA. After identification and authentication of Bob, the CA verifies whether the demanded right and expiration date are granted to Bob. In the successful case, the CA signs the certificate twice with the key for rights and three times with the key for expirations dates.

Since Bob knows the effective blind factor, i.e., the raised power with his secretly known exponent, he can use its inverse to un-blind the pseudonym. Thereby, it remains validly signed (see Figure 3.3). Actually, only a secure hash of the pseudonym's public key is signed and Bob also adds the plain version of that public key to the certificate. With such completely signed certificate, Bob can disrobe from his identity and interact under his chosen pseudonym with services. The correlation between the identity Bob and the pseudonym is known to him only.

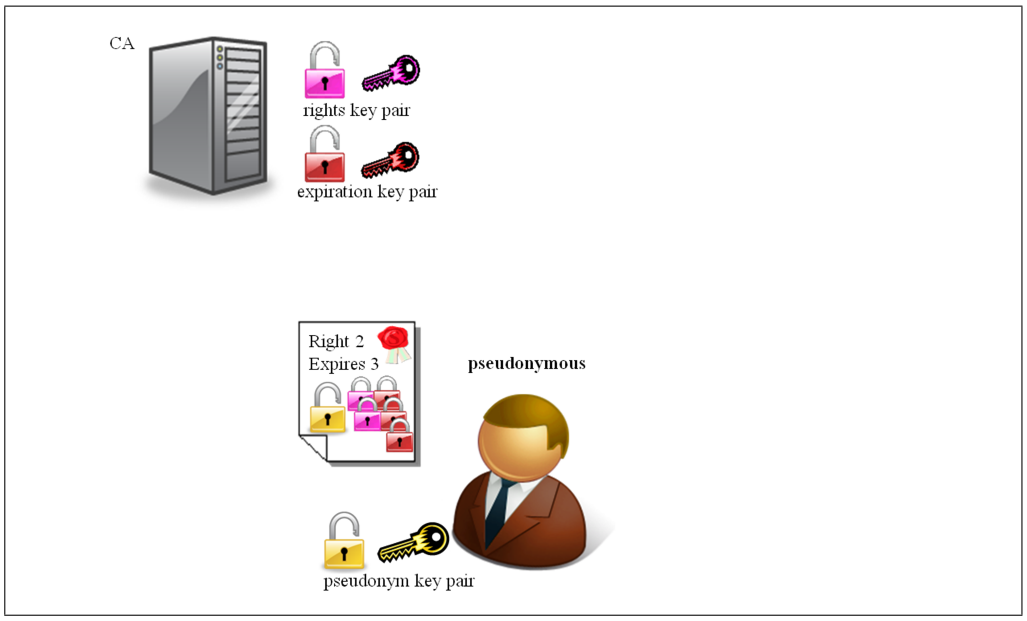


Figure 3.3: Bob uses the inverse of the blind factor, known only to him, to un-blind the pseudonym after the CA returned the signed certificate (removal of hatching)). He gains a validly signed certificate, which cannot be mapped to an identity. Using this, he can prove his right 2 until date 3 without revealing his identity.

3.4.2 Anonymous Authorization and Authentication

There is an unidentified user possessing a certificate, claiming the right 2 and expiration date 3. In the example, there is further a Video On Demand (VOD) service, which requires its potential users to possess right 2, which could represent certain age for example. The unidentified user sends her/his certificate to the VOD service (see Figure 3.4).

The service requests the CA to verify the certificate. As described in Section 3.2.3, the keys for verification must be kept as secret as the keys for signature. Therefore, the verification cannot be accomplished by the service itself. The CA decrypts the signed pseudonym with the respective number of verification keys, according to the claims of the certificate. In this example that is, twice with the key for rights verification and three times with the key for expiration date verification (Figure 3.5). Actually, the signature

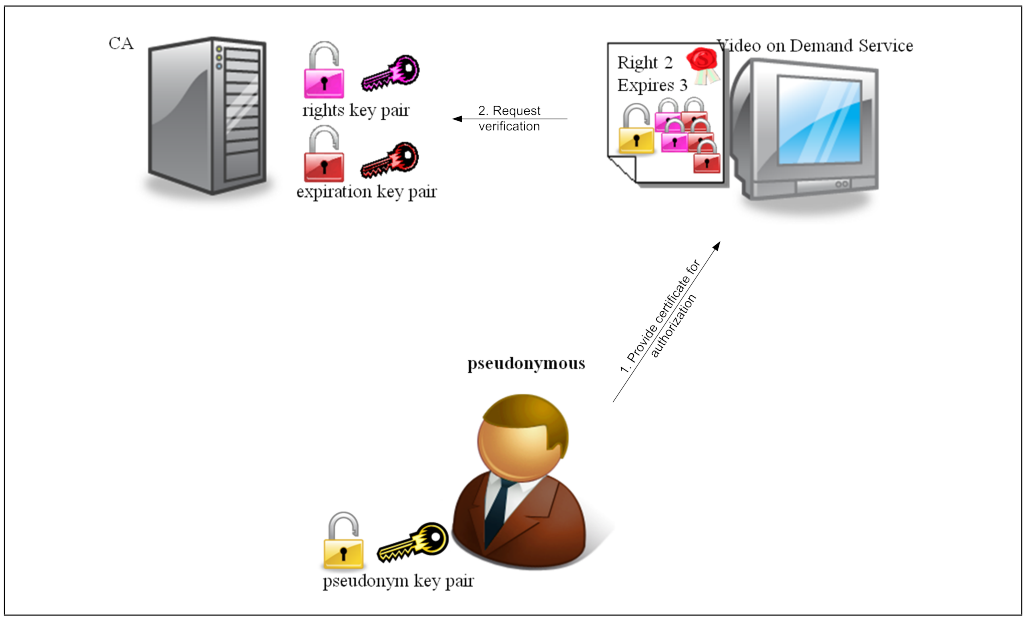


Figure 3.4: A user wants to use a VOD service without revealing her/his identity. The service requires the possession of right 2 at the time of usage. For the proof of ownership of right 2, the user sends her/his certificate to the service.

counter verification key and further verification keys for devices or priorities are applied here, too.

If the result of the decryption with the according verification keys is a well-formed pseudonym, that is, a hash value of the contained public key appended with the well-known markers, it is returned to the service (Figure 3.6). Please note, while the verified pseudonym is either a public RSA key or the hash value of that key, eavesdropping and replaying by other users pose a threat. But if the authenticating user could forge a CA and respond a her/his pseudonym or according hash value to the service, she/he can fraudulently obtain authorization. Hence, a trusted and encrypted connection between service and authority is required. The format check of the pseudonym under verification is depicted in Figure 3.5 that each of the colored locks is assigned with exactly one equally colored key. The number of keys is determined by the claims of the certificate. In case there was a key, which could not be assigned to one of the locks, a right with a higher ordinal number or a later expiration date than certified was claimed. The signed pseudonym would be determined as invalid in such case. Remains a lock without respective key, that means that a right with lower ordinal number or an earlier expiration date than certified was claimed. Also then the certificate is recognized as invalid. In neither case, the CA returns the decryption result to the service.

In the example here, the signed pseudonym was validly decrypted and returned to the service. The service may therefore assume that right 2 is granted to the legal owner of the certificate. Certainly, it has to be assured that the certified expiration date is not reached

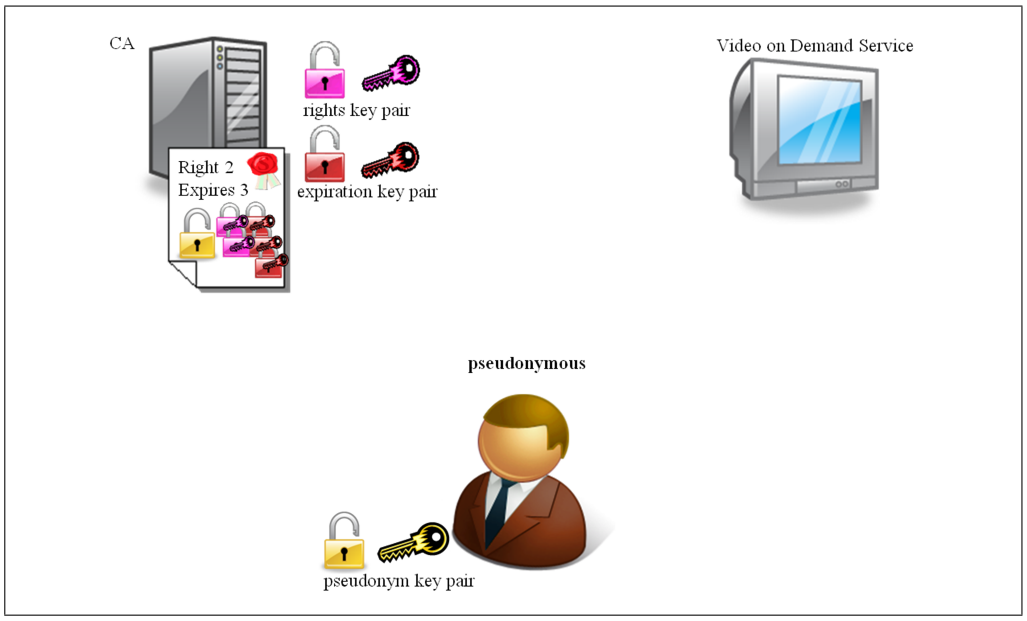


Figure 3.5: For verification of the certificate received from the user, the service forwards it to the CA. The CA applies, according to the claimed right and expiration date, twice the verification key for rights and three times the verification key for expiration dates.

yet, i.e., the certificate has not expired. Figure 3.7 depicts, how the service verifies the legal possession of the presented certificate by the presenting user. As already described in Section 3.3, the service creates a list of TANs (in Figure 3.7 denoted as 'Challenge') and encrypts that list with the public key of the pseudonym embedded in the presented certificate. The encrypted list is sent back to the user.

Only if this user has legal ownership of the certificate, she/he possesses the corresponding private key belonging to the pseudonym. With this she/he is able to decrypt the list of TANs (see Figure 3.8). Illegal presenters of the certificate, e.g., eavesdroppers, cannot decrypt this list. The later transmission of a valid TAN from the list implies legal possession of the rights associated with this list of TANs.

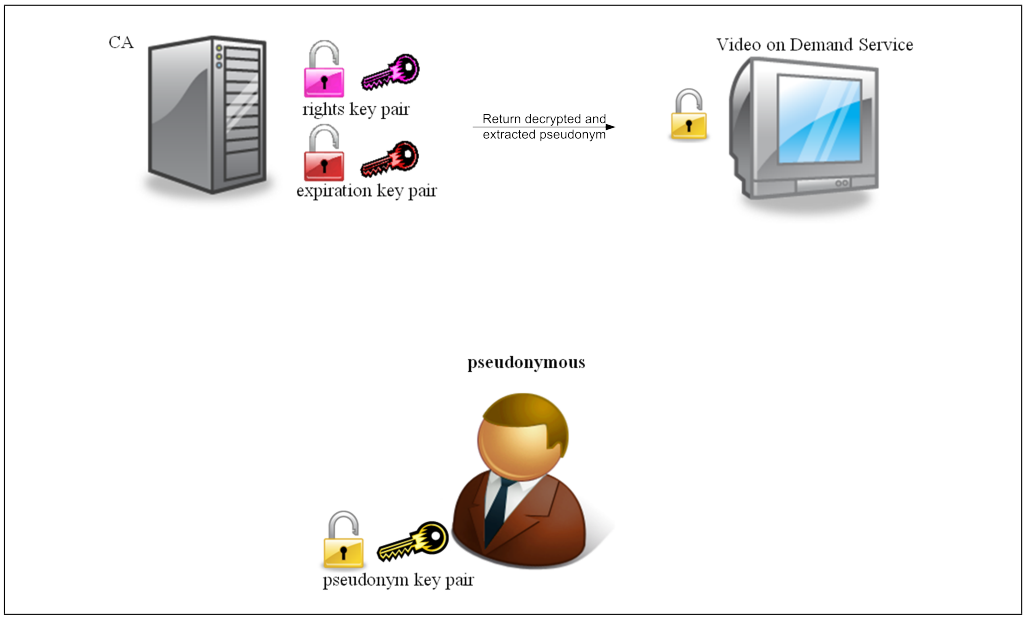


Figure 3.6: If the verification results in a well-formed pseudonym, the CA returns this to the requesting service. Otherwise nothing is returned and the service excludes the user from the service usage.

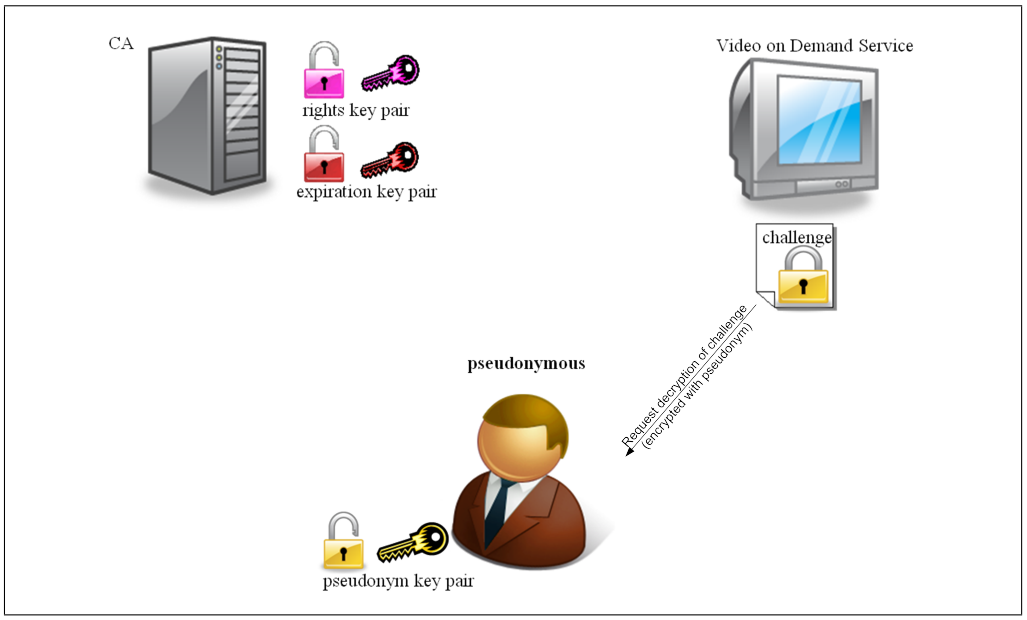


Figure 3.7: The well-formed pseudonym received from the CA is a public key, which can be used for verification of legal ownership of the presented certificate. Therefore, the service generates a random challenge and encrypts it with the embedded pseudonym. This encrypted challenge is returned to the presenting user.

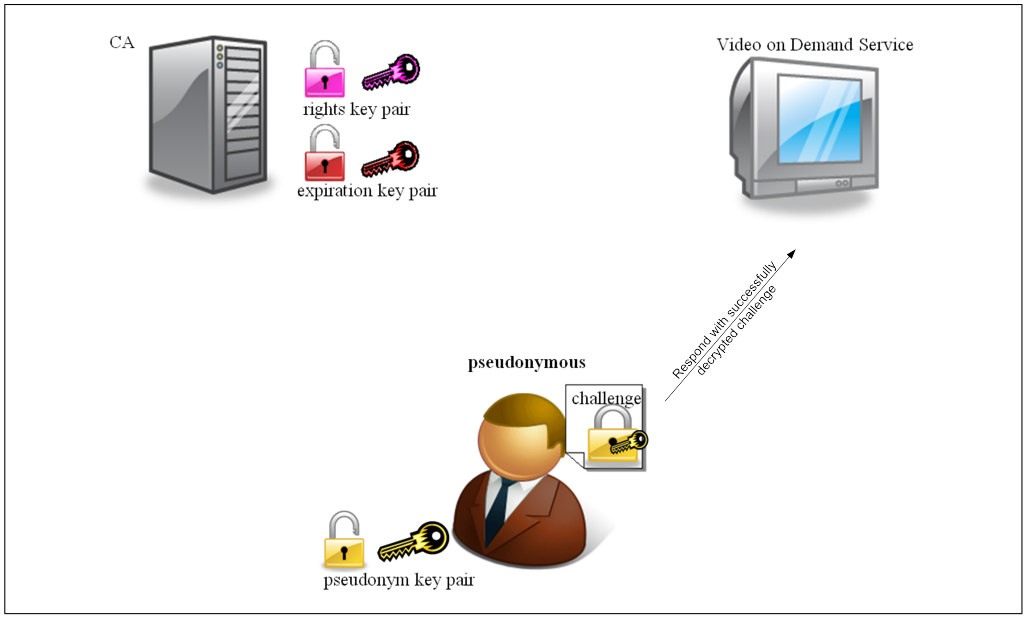


Figure 3.8: If the user is legal owner of the certificate, she/he possesses the private key belonging to the pseudonym. Only with this private key the challenge can be decrypted. With a correct decrypted challenge the legal possession of rights can be proven, which entitles for service usage.

Chapter 4

Protecting Private Data by Technical Means

Besides special issues with privately relevant data it is a fact that everything that can be read, can also be copied. A further generalization of that fact is:

"Everything that can be sensed, can be copied".

Once data are copied, the original owner has no actual control over the copy of these data. Hence, the data could be used at any time for any purpose by the party that possesses this copy. If one is concerned about her/his private data and wants to prevent such copies, the only 100% safe solution is not to give her/his personal data to someone else. Therein giving can be something like transmitting, showing or making it audible. On the other hand, a lot of services, especially in the Internet, do not make sense or are not usable without some personal data of the user. The dilemma is how to hand out personal data to the service without giving it the capability to create a copy. A copy is considered a manifestation of data that can be sensed by another entity. If there is a technical solution that prevents the service from manifesting the data at all or at least having that manifestation not being sensable by other entities, a user could faithfully pass her/his data to the service and trust that the data are deleted afterwards or at least not be used for any other purpose. Note, the user would not have to trust in the service and its privacy policy, but merely in the service independent privacy protection solution.
It is assumed that content of volatile memory, such as registers in CPUs and RAM, is not sensable by others. Modern OSs provide means to prevent access to certain memory segments by others than the owner. Encrypted data in persistent memory or transferred across a network is also supposed to be not sensable as long as the key for decrypting is unknown to the entity trying to sense.
A technical means to protect private data shall hence assert that these private data must not be sensable by other entities than the one they have been handed out [66]. To be on

the safe side, all possible senses and not just the obvious ones have to be addressed. Humans can obviously see and hear or read and listen respectively. Hence, data should not be written or spoken to protect information from humans. But humans can also smell, taste and feel. Consequentially, odors and pheromones or flavors and condiments must not be used. With 'feeling' describing the ability for tactile cognition, temperature cognition and cognition for pain or balance, rather not obvious communication means as surfaces, heat, coldness, needles or shock must also not be used.
For computers the following senses are considered:

- Reading a hard drive, floppy, optical drive or similar
- Listening to the network
- Scanning
- Recognizing key press or mouse motion
- Sampling audio
- Grabbing video or capturing screen content

Less obvious capabilities are to:

- Read from shared memory
- Scanning and parsing system streams (in, out, err)
- Catching exceptions
- Synchronizing on semaphores or object monitors
- Determine CPU and memory load
- Recognize file access

It can be easily deduced, which operations and actions must be prohibited to prevent a service that processes private data from copying. A technical privacy protection means should prohibit just these operations for service code running accompanied by that means. This is to be done even on the service providers premises. That is, the service provider imposes a restriction on her-/himself that she/he cannot circumvent. If the service provider can further prove that she/he is using this restriction enforced by the protection means, she/he can provide confidence to the user. In other words, private user data can be used by the service but not copied to be used for unexpected purpose. The technical means that is going to accomplish this is called a Privacy Guaranteeing Execution Container (PGEC) and presented in the Sections 4.2 to 5.1. Beforehand, some definitions are given to describe and understand the application context of the PGEC.

4.1 Terminology

This section explicates a number of terms that are used in the description of PGEC. These descriptions shall support the understanding of the PGEC and its protection means and prevent misconceptions. Herein, services are classified and economical security, information load and negotiation of privacy contracts are defined.

4.1.1 Service Classification

The communication between the code, which is executed inside the container, and the service provider outside, is restricted by a privacy contract. The privacy contract has to be negotiated between the service provider and the service user. Appropriate means to do so have been proposed in [67, 68, 69, 70]. Both parties have to sign that contract. It defines which kind of messages may be sent by the code providing the service and to which communication endpoints they may be sent. The rules that have to be defined inside the privacy contract depend on the kind of service. Up to now three classes of services are identified.

1. Logically delivering services
2. Logically controlling services
3. Physically delivering services

Logically delivering services do not need to communicate with the service provider side. A navigation service is a representative of this class. In order to provide its functionality, it only needs to read local service data such as a map and the current position of the mobile device. No communication with the service provider is needed, except in case these services are charging a small amount of money. This class especially addresses the presentation of the service results. Presentation is usually considered as a display with a GUI or similar. This can be misused to retrieve private data from inside the container via an allowed means. A virtual graphical device can actually write the information to disk as a straight forward attack. Services are most likely not presenting own secret data to the service users. The user has no actual control over what is displayed. Hence, differentiation is required between a container running at service provider's side, suspected to fraudulently obtain private data, and a container on the user's side, which demands the private data to be protected. Thus, services running in the container on the user's device may be allowed to display any result or information while prohibiting any audio or video generation on the service providers machine. The output device might be stated in the contract as well and could be authenticated by knowledge of the private key that was used to generate the contract signature.

Logically controlling services do not provide any benefit without a chance to send messages back to the service provider. Examples of logically controlling services are re-

mote control services for cameras in surveillance applications. These essentially need to send messages back home, such as move *right*, *left*, *up* and *down*. The privacy contract has to enumerate a message vocabulary for this class of services. on the one hand, this vocabulary has to be rich enough to provide a comfortable handling. On the other hand, it shall be sparse enough to ensure that malicious code cannot misuse this vocabulary to encode privacy relevant information and send it back using the allowed messages. The negotiation does not need any knowledge on the semantics of the data items to be negotiated about. This is especially true for the messages or literals to be defined. Neither the negotiation nor the PGEC actually needs to know the meaning of a certain communication endpoint for such messages. It only needs to be able to identify and address that endpoint. Similarly, the names of the literals are of low meaning to the PGEC. The count of different literals is what matters. That is, the service and its negotiation entity can propose any literal of any name. This proposal is done before knowledge of private user data and hence, cannot contain any of such information. The literals have to be string comparable. So the PGEC can check the content of messages. Besides the pure number of different literals, their expected frequency or acceptable delay shall be specified. Additionally, literals may be grouped. This grouping is also defined by the service. Literals of different groups are considered to be orthogonal. By orthogonal it is meant that changing the sequence of those literals will not affect the result in the external world. The knight's move in chess gives a good example. Moving *1-step-ahead* and then *2-steps-right* results in the same position as *2-steps-right* and *1-step-ahead*. Hence, movements *ahead* and *backward* form one group and *right* and *left* form another group. To prevent misuse of literals for trickling private information to the outside, the number of literals should be kept low. The number of literals and their groups is defined in the privacy contract. Keeping the size of groups as low as possible allows to randomly mix orthogonal literals if several of them are triggered within a close time span. For the given example, if the maximum delay is defined to be one second and in this interval the two literals *ahead* and *right* are triggered for sending, the PGEC may send them in any order. Mixing orthogonal literals reduces the chance of encoding information in the sequence of sent literals. The amount of information that could be "morsed" with the literals can be estimated (see Section 4.1.3). This estimation can be used in the negotiation strategy to limit the number of different literals, their frequency or the overall number throughout the service usage.

Physically delivering services are the last identified class of services. Online-shops or print services are members of that class. Those actually require to disseminate private data such as shipping addresses or content of pages to print. Hence, they require weakening of the privacy guarantee. Consider a shopping service that needs at least the shipping address given to the delivery service as well as information for clearing to be given to the bank. In case such information disclosure is permitted by the privacy contract, the

service requests the container to send the information to the appropriate party. Thus, it is made sure that only pristine information is given. Further it is ensured that it is given to those parties only that the information was supposed for. But note, from the moment this privacy relevant information leaves the container, there is no control over it anymore.

4.1.2 Economical Security

A 100% secure solution has no chance to ever been broken. But this is rather theoretic, that is, every encryption or security solution can be broken. It just depends on time and effort how and when it can be done. Hence, economical security is defined:

"When the effort to break the protection of an entity/data is larger than the value of the entity/data, its protection is economical secure."

This must be considered in the protection means of private data inside the PGEC and the self-defense means of the PGEC. For the protected user data inside the weakest point in its protection perimeter is decisive. While existing encryption technologies, e.g., AES, RSA or Elliptic Curve Cryptography (ECC), are known to be strong with sufficiently long keys, possible design or implementation flaws in OS, the Java Virtual Machine (JVM) or the PGEC may be easier to exploit. The latter are usually not tackled in a brute-force manner, which makes it hard to estimate the actual effort. For information leaks like the introduced literals in the PGEC, an estimation can be calculated (see Section 4.1.3). It has been shown that a 768-bit integer can be factorized within 2.5 years using a cluster of 80 PCs [71]. Based on the date of first factorization of a 512-bit integer it is extrapolated that 1024-bit integers could be factorized within reasonable time in ten years. Thus, the authors recommend to withdraw keys of 1024 bit length no later than 2014. Hence, key length of 2048 can still be considered secure. One of the authors estimated and compared the efforts, by time and money or hardware respectively for various symmetric and asymmetric encryption or secure hashing algorithms, regarding the used key lengths [72]. This estimation could be used to estimate an upper bound for the value of data being securely protect or to determine a minimal key length to protect data of certain value.
In consideration of the vulnerability of software-only solutions as described in Section 2.4, software may be protected additionally using hardware dongles. Hardware dongles evolved throughout the years from providing a read-only serial number to cryptographic devices. Obviously, the encryption may not be broken brute-force. Depending on the intelligence, the software makes use of its dongle, the effort to hack certain functions can be increased. Appropriate means, to hack and to obstruct hacking were described in [73]. This article refers to a hacker contest [74], in which one of two tasks regarding such dongle was successfully hacked. It was possible to run certain function without the dongle while possessing the dongle and this particular function was actually licensed by the dongle. The other task, running an unlicensed function, which would

require breaking the keys inside the dongle was not solved. From the duration of the contest, which was six weeks, an upper bound and hence costs can be estimated. Estimation of effort and costs to physically break into the hardware was not found.

4.1.3 Evaluate Information Load of Literals

For services that need to communicate with the outside world in a somewhat limited way, i.e., some remote control service, the admittance and use of fixed literals are proposed. Those literals have to be negotiated before service usage. Those literals are then some kind of predefined commands. Those commands should not have parameters. Otherwise, the domain of the parameter has to be that small that it could also be mapped into a small number of commands without parameter.
Note, that those literals could be misused to transfer private data to processes that are not under control of the PGEC. To cope with that, it is proposed to limit the overall number of literals, the frequency of literal sending or the duration of a service session using literals. Such limitation is calculated as follows.

D the amount of data in bits to be privacy protected

T the amount of data in bits that can be transferred through the use of literals

L the number of admitted literals while each parameterized literal counts as many times as the size of the co-domain of its parameter

n the number of transferred literals

f the frequency of literals

t the duration of a session sending literals

$$2^T = L^n \tag{4.1}$$

$$T = n \cdot \log_2 L \tag{4.2}$$

$$n = f \cdot t \tag{4.3}$$

$$T = f \cdot t \cdot \log_2 L \tag{4.4}$$

$$T < D \tag{4.5}$$

From Equation (4.1) can be learned that a sequence of T bits (two possible values) builds the same number of combinations as n literals (L possible values). By solving this to T (Equation (4.2)), it can be determined how many bits of information can be transferred by a number n of L possible literals. When considering an average frequency of literals and the duration of a service usage session the number, the number of literals that can be transmitted during that session is determined by Equation (4.3). Hence, with a given average frequency, session duration and amount of possible literals the information load,

which can leave the control of the PGEC, can be calculated using Equation (4.4). For privacy protection it is required, that this information must be less than the data size of the sensitive information to be protected.
Assume a service that is using four literals at about one literal per second. This service is getting credit card number, expiration date and name of a user and is suspected to transfer these data out. A credit card number has 16 digits, with the first being anything between three and six. The expiration date has twelve months and one of four years including the current year. Assume, the user has an average 17 characters name like 'Christian Schmidt'. That results in 4×10^{15} possibilities for the credit card number, 4×12 possibilities for the expiration date and for the name $26 \times 27^{15} \times 26$ possibilities. About 139 bits ($D = 139$) are required to express those. Thus, $f \cdot t \cdot \log_2 L$ must be less than 139 bits. That is, with four literals and an average frequency of one per second, it needs 70 seconds to transfer the given data via the literals. Thus, the user may want to limit the duration of the session to less than 70 seconds in order to protect the privacy of her/his credit card data.
More interesting are quickly changing private information like position data that could allow tracking and creation of a personal profile, i.e., certain dwell time at places. For example, the position might be given in the 14-digit Gauss-Krueger-Notation, which is around 53 bits and the service is going to use five literals. Further, it is assumed between two known positions should be five minutes to remain kind of incognito. To ensure that position information cannot be transfered using the literals more often than every five minutes, the frequency of the literals throughout the service usage session must be limited.

$$f \cdot t \cdot \log_2 L < D \tag{4.6}$$

$$f < \frac{D}{t \cdot (\log_2 L)} \tag{4.7}$$

$$f < \frac{53bit}{300sec \cdot (\log_2 5)\frac{bit}{literal}} \tag{4.8}$$

With application of Equations (4.6) to (4.8) it can be figured that the frequency must be less than one literal every 13 seconds to meet the privacy requirements regarding the position information.

4.1.4 Negotiation of Permitted Data

Privacy protection requires adequate declaration of which data may be used by whom and for what. Note, this may even be no data for nobody and no purpose. Approaches for such declaration have been presented in Section 2.1.1. There is even some tool support for these technologies. AT&T's Privacy Bird[1], for instance, is a free plug-in for Microsoft Internet Explorer. It allows users to specify privacy preferences regarding

[1] http://www.privacybird.org/

how a website stores and collects data about them. The user's preferences are set by enabling/disabling different rules out of a set of fixed options. The user is not able to define further rules or specific restrictions. If the user visits a website, the Privacy Bird analyzes the policy provided and indicates whether or not the policy matches the user's preferences.

Depending on the service class users are ready to permit the release of other information. This is even true for specific services. That is, personalized privacy policies are required. Such personalized privacy policy is called a privacy contract. The way to achieve such privacy contracts is negotiation. To preserve privacy in this negotiation process differs from usual negotiations used in network technologies. Choosing one option from the intersection of the respective lists of options from both parties, will affect the privacy of the negotiation partners. A step-by-step modification of an initial proposal can be used to achieve privacy aware negotiation. To the best knowledge there are only two approaches that allow step-by-step modification of the originally proposed policy by service user and service provider namely [75, 76]. But in both approaches only the service provider is enabled to present new versions of its policy, whereas the user can only accept or reject the proposal. The user side can only provide hints, why the proposed policy was not accepted. These hints may help the service provider to calculate a more suitable proposal.

For real privacy respecting negotiation, both negotiation opponents should define their secret preferences that specify whose acceptable room to negotiate. Both negotiation parties should be capable to offer counterproposals based on former offers. Finally, successful negotiations should close with a mutually signed privacy contract that contains all negotiated content. Such privacy contracts have been introduced together with a procedure to negotiate them between a service provider and a user in [67, 68].

If this negotiation scheme and according privacy contracts are used together with a privacy guaranteeing means like the PGEC presented herein, it must distinguish between data that is allowed to be used within the container, for which copy protection is guaranteed, and those data that may leave the container, e.g. in case of physically delivering services. Also, the literals and their constraints (grouping, frequency, maximum amount or session duration) have to be negotiated and stated with the privacy contract. Even though, a privacy contract appears to be the most appropriate way to specify permissions for data access, literal sending and purpose binding, the effort of integration of the existing implementation of a privacy negotiation unit [69, 70] has been avoided in favor for the implementation of the actual protection features presented herein.

Further, such automated privacy negotiation systems [68, 69, 70], which determine individual privacy contracts [67] can be used in conjunction with standardized access control policies like XACML. An appropriate translation between the XML privacy contract and the XACML policy supports the integration of privacy negotiation into existing systems implementing XACML.

4.2 Privacy Guaranteeing Execution Container (PGEC)

The Privacy Guaranteeing Execution Container (PGEC) is designed for stand-alone execution as well as for distributed execution. When executed as a stand-alone application on the service user's side, it makes sense only for protecting the code and embedded data of the used service. Executed at services providers premises, it protects the user's private data as long as it can be ensured that those data cannot be grabbed on their way into the container. The most flexible way of usage allows a distributed container. Multiple instances, running at user's and service provider's premises, provide a transparent but secure exchange of private data throughout the distributed container instances. The architectural design is partially analogue in a stand-alone and a distributed container. Those shared architectural designs are described in Section 4.2.1, whereas those parts that allow for distributed execution of the container are presented in Section 4.2.2.

4.2.1 Stand-alone Architecture

Figure 4.1 shows the components of the PGEC, i.e., the execution environments, the communication interface, the privacy contract and covert channel attack protection means. Privacy contracts are a declarative privacy protection means as described in Section 2.1.1. These contracts have been introduced together with a procedure to negotiate them between a service provider and a user in [67, 68]. Implementations of their negotiation have been presented in [70, 69]. For convenient usage of the PGEC presented herein, a negotiation unit derived from [70] should be implemented in the PGEC.

Communication Interface

The communication interface restricts access to data as well as the literal exchange according to what is agreed between the service provider and the service user. It is obvious that the service usually consists of external code. The user is likely to not fully trust the service code. Hence, to control the communication of the service and thereby protecting her/his private data, the user executes the service code within the PGEC.

Restricting Network Communications Besides other privacy relevant information, the privacy contract states which literals may be sent to which communication endpoints and which data is to be retrieved from the user through the container. All decisions taken by the communication interface are based on this privacy contract. The communication interface is in principle a kind of rule engine applying the rules defined in the privacy contracts. In case, the executed service in the container definitely needs to communicate with the outside of the container at its origin or any other party, this communication must be limited in a way that it prevents transmission of privacy relevant data. The communication must not be established by the service itself but merely by the container. An API supporting to send literals is provided by the communication interface. The service may and can initiate the sending of literals only by using the communication

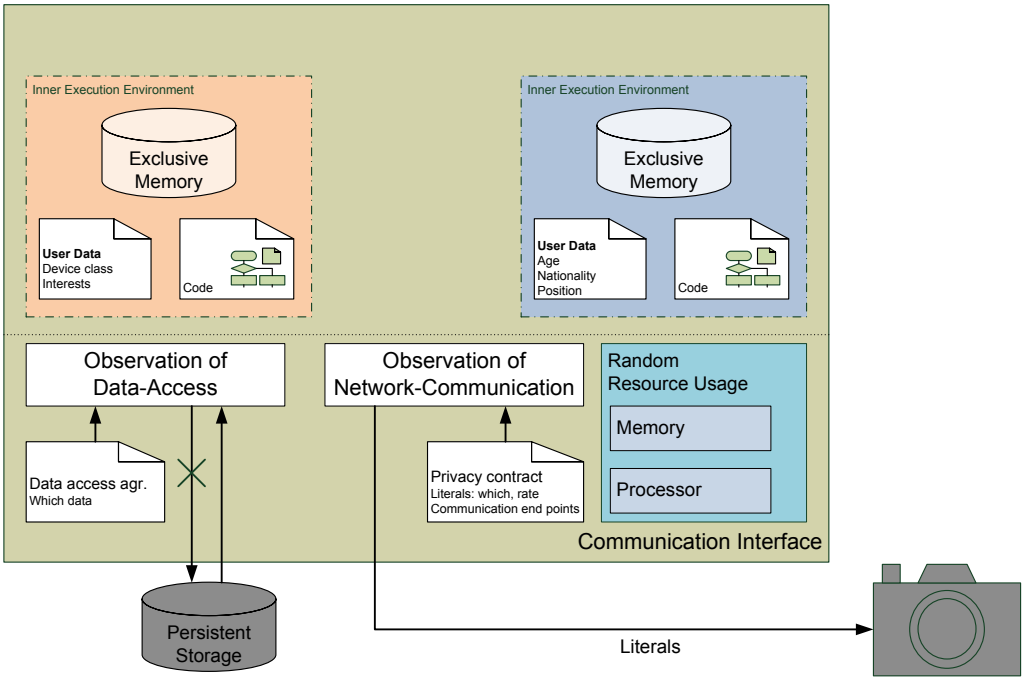


Figure 4.1: Architecture of the Privacy Guaranteeing Execution Container (PGEC). It features a communication interface observing and controlling communication with external processes or machines. In general external communication is prohibited. Exceptions can be specified in privacy contracts or data access agreements. Further, the PGEC provides an arbitrary number of execution environments. The separate execution environments effectively suppresses communication between their applications and services.

interface of the container. That way it can be ensured that the service does not send privacy relevant data to somewhere else. The destinations of the literals as well as the possible literals themselves are defined during the negotiation process of the privacy contract. The vocabulary has to be defined in a way that enables the communication interface of the container to check whether or not a certain literal is allowed. Therefore it is proposed that the vocabulary consists of literals. In this case, the communication interface can use a simple string compare operation to verify whether the literals are allowed or not. The container checks the literals given through the API on compliance with the privacy contract and sends the actual literal to the specified communication endpoint or dismisses it.

Even if the vocabulary of the literals is well defined, i.e., if it prohibits easy sending of sensitive data, it can be used for this purpose if an unrestricted number of messaged may be sent. Since the meaning of such literal cannot be generally determined, those could be used to encode sensitive data. An assessment on the amount of messages for encoding private data is made in Section 4.1.3. Hence, the container may also limit the frequency of such literals or it could deliver orthogonal literals in unspecified order. This decreases the chance of encoding information other than the control information to be transmitted

by the literal. Exemplary for such service, remote control services for electronic devices shall be named, e.g., an air conditioner. It does not need to know the privacy relevant information of someone's preferred room temperature, but only needs to be adjusted warmer or cooler until the measured temperature fits the personal preferences.
A service may need to receive additional information from its origin at the outside of the container. In this case, the communication interface provides another API to request such information by response of a function call or by providing predefined data streams. The container is enabled to suppress messages from the services origin to the executed service, to prevent from guessing information and acknowledging by literals.
If a service tries to send data not agreed to in the privacy contract or to communication endpoints not agreed to, a privacy exception is generated. Such privacy exceptions can also be logged. This log may help to prove fraudulent behavior during the negotiation process or to prove claims of reimbursement of unjustified service charges, if the service is not functional due to the lack of particular information.

Restricting Data Access The second task of the communication interface is to control data access. Here again, the privacy contract describes what is permitted, while anything else is be prohibited. In addition, write operations to persistent storage have to be blocked by the communication interface. Since the required data is application/service dependent, there has to be a specified way in which the container gains access to the data potentially passed to the services inside it. Two approaches have been identified.
The first approach is to specify an API to push information from the outside into the container. This enforces every application using the container to execute services to adapt to that API. The bigger problem comes with changing privacy relevant data that may be needed by services but not as often as these change. Thus, a data push approach would result in a never-ending push thread that takes up computational power, probably without any positive effect on the services in the container. If data is pushed into the container, the container itself has hold of these data. Unfortunately, this prevents the garbage collector from deletion of the demanded data upon service completion. Only data directly associated with services and their classes are deleted on service completion. In a distributed PGEC, it is likely that private data are pushed into the data owner's container instance at her/his device only. Hence, it is not critical if her/his data are not immediately deleted.
The second approach is to allow the container to access to the data at install-time or even at run-time. To accomplish that, a data access component as displayed in Figure 4.1, which grants read access only is proposed, and which uses the privacy contract to check which data may be read. Reading private data into the PGEC is not considered critical, since no data can leave the execution environment if it is not allowed in the privacy contract, i.e., the communication interface will filter all literals and other communication means are restricted, too. Private data may exist in various formats and be accessible by a number of means. Thus, unified access to all kinds of private data is not feasible.

Standardized access interfaces like Java Database Connectivity (JDBC) and query languages like SQL are not available for every data source. Therefore, a combination of both approaches is most promising for a productive application of the PGEC. For simplicity reasons, the first approach has been chosen for the implementation presented in this work.

Execution Environment

The execution environment is merely a logical construct ensuring that applications and services within the PGEC can access the container interfaces but have no access to data or code outside the container. In fact, the PGEC distinguishes between inner and outer execution environments, where outer execution environments are situated logically outside the PGEC, are less restricted but have no access to private data. The communication restrictions imposed on the inner execution environments allow the services within to access private data. Both types of environments provide the necessary infrastructure to services executed inside and outside the container, e.g., access to the processor, volatile memory etc. They are also responsible for the cleanup operations that have to be done when a service is no longer used, i.e., it has to make sure that all data is really deleted. While it is quite obvious that services running inside the PGEC are not supposed to communicate with the outside world, the inner execution environments have further to ensure that different services inside the PGEC shall not directly communicate with each other. It is an additional task of the execution environments to ensure that these services do not have any communication with each other. That means, even the use of a shared memory segment has to be avoided by the execution environment. The amount of execution environments in a PGEC instance is not fixed and usually for each service a new execution environment is built. Note, that the outer execution environments, which exist logically outside the container are not restricted by the communication interface and corresponding privacy contracts. On the other hand, those have no access to the private data accessible inside the container. This construction is not depicted in Figure 4.1 but was introduced to run additional processes besides the container. Those processes can for instance, receive literals or provide a management application for users to manage privacy preferences, insert private data into the container or even find and request services provided inside the distributed container. Detailed discrimination from inner execution environments can be found in Section 5.1.2.

4.2.2 Distributed Architecture

Services/applications that are using and processing private information in a stand-alone fashion are not threatening users' privacies very much. Usually, these are executed at the users' machine over which she/he has control. The only threat of those with regard to privacy was network communication. The user may protect her-/himself by applying a firewall. A number of firewall systems exist in the market. Those are either software

solutions so called personal firewalls such as the Microsoft® Windows Firewall or hardware solutions built into routers, i.e., using Linux' iptables. Mostly those are used to prevent unwanted access from the outside of the firewalled hosts. Firewalls can further be used to block certain protocols or several ports. As far as it is known, there is no way of a firewall to prevent certain data items to be sent to not blocked addresses. Even though there is some packet inspection, there is no chance to recognize the content of encrypted packets. Firewalls have no control over the data once it passed the firewall. Hence, a firewall may only assure that no data leaves the machine at all. This is similar to unplugging the network cable, which renders all Internet services unusable.
On the other hand, those Internet services are more interesting but also impose more privacy concerns. Such services will not bring any benefit or not even function, when not having an Internet connection. This is likely in cases where large databases, processing power or distributed community knowledge is required for providing the service. Using such services, requires to either transfer private data to the service provider with the database or the database to the user. The latter will obviously refused by the service provider. It is exhausting bandwidth and threatens its intellectual property. The first is not reasonable from the privacy perspective. This is addressed by the distributed architecture of the PGEC. The PGEC's distributed architecture stipulates to connect multiple instances of the PGEC with encrypted channels (see Figure 4.2). The keys used for encryption of the channels are only known to the PGEC. This can either be accomplished by using a dedicated symmetric key for all channels or appropriate key exchange between the instances. In the latter case, the PGEC instances need a mutual authentication mechanism. The channels are used to transparently transfer private data from one PGEC instance to another. This allows the services running in the execution environments of the PGECto access private data analogous to the stand-alone version. The communication interface forwards the data request to the PGEC instance, which holds the data. In opposite to firewalls, the PGEC transfers sensitive data securely over the network into other PGEC instances only. While regular firewalls do not prevent local applications to write onto local hard drives or similar persistent storage devices - even printers, the PGEC does. That way, all data transferred within PGEC instances remain within those. This is true for private data of service users as well as data and service code of the providers. Within the container instances both can be brought together and the service can usefully be completed.
The encrypted channels actually allow any communication between container instances. That is, forwarding of literals, control messages, data access authorization requests and even tunneling network connections or pipes between distributed components of services within the PGEC. Communication of networked services, which is not possible in a stand-alone PGEC, can be established when all service components run inside PGEC instances. Still, all regular network communication, which is not tunneled into another container instance, is suppressed.
Since data can only be used within PGEC instances, this principle may be compared

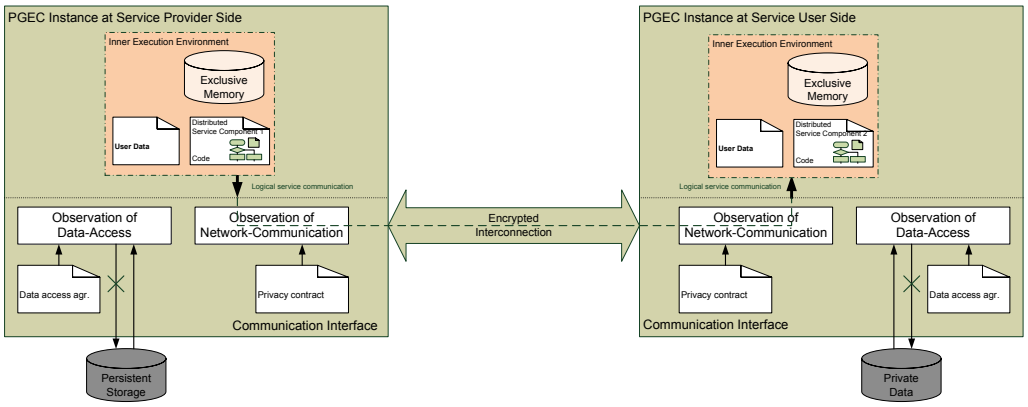


Figure 4.2: Architecture of the distributed container. A distributed PGEC consists of at least two instances of the stand-alone PGEC. These instances are connected through encrypted channels, which tunnel all communication of services distributed in these PGEC instances and messages for container management and control. Mutual authentication mechanisms ensure that only PGEC instances are the endpoints of those channels. These channels allow transparent access to private data available in other container instances.

to DRM systems where the protected content can only be decrypted and played within certified DRM enabled systems, which comply with the requirement not to persistently store the decrypted data. PGEC is a DRM system that allows everyone to protect her/his data from being copied, but even be used for a priori unknown purposes within the container. A key or license management similar to existing DRM systems (see Section 2.4) can ensure use limitation for particular data, users and services. The key management scheme introduced in the prototype presented herein is presented in Section 5.1.5. The constraints on the keys protecting particular private data shall be derived from the privacy contracts.

4.3 Protection Requirements

In order to provide useful context aware services, a lot of different information has to be taken into account. This ranges from a personal profile of the service user over related data from a third party to the algorithms used and developed by the service provider. The major problem to solve when it comes to privacy issues is how to guarantee that the service provider does not retrieve data of the service user. This task has to be tackled in such a way that the solution can adapt to different contexts, i.e., the data which may be exposed or protected can vary from application to application as well as from user to user even for the same application. In addition, service providers must be protected from malicious service users, i.e., it has to be ensured that the service user cannot get hold of the algorithms provided by the service provider. In order to provide mutual protection between service user and service provider PGECs are introduced. These are containers,

which are independent of the service provider as well as independent of the service user. They have to ensure the following properties:

1. All data may be stored in volatile memory only and will be deleted after completion of service use; this has to hold true for service provider as well as for service user data.

2. The communication between the code executed in the container as well as the communication between the container and any third party is to be restricted to what is agreed between the service provider and the service user. This agreement is denoted as the privacy contract, as mentioned before.

3. The local exchange of messages and implicit communication, e.g., via shared memory is prohibited.

If property (1) is fulfilled the container may be executed on any location (server or mobile) due to the fact that there is no way to get low level access to the data of the other side. The benefit is that load balancing becomes feasible. Computational expensive services do not have to be executed on the mobile device.
If property (2) is fulfilled, there is no chance to steal data during the service use. The problem here is to define a set of allowed messages. On one hand, it has to be sufficiently large to allow service fulfillment. On the other hand, it has to be as restrictive as possible in order to ensure that it cannot be misused to steal data, and to enable the container to verify the content of the allowed messages.
If property (3) is fulfilled, a service running in a container cannot share its knowledge about gained private data with other services that are concurrently executed within the PGEC. Hence, it is not possible to extract private data via an additional service and a faked user with a very loose privacy policy.
In order to make sure that the service user as well as the service provider will trust the PGEC, it has to be implemented by a trustworthy third party, and it has to be signed by that party using a PKI certificate, e.g., from VeriSign[2].
This section discusses the threats that are posed to the PGEC. Based on the identified classes of attacks, possible countermeasures are mentioned. Details on the implementation of these counter measures can be found in Section 5.1.

4.3.1 PGEC Threat Model

In order to understand the security means provided by the PGEC, the goal of a potential attacker needs to be defined clearly by formulating potential attacks and by defining required countermeasures. The prime goal of the PGEC is protecting service user's privacy. Thus, the most important aspect in the design of the PGEC is to prevent data extraction from the container. In this model, the PGEC itself is the only trustworthy

[2]http://www.verisign.com/

entity. It is implemented by an institution, which is considered to be trustworthy by all users of the PGEC, but will be executed at host systems that are owned by potential attackers. In other words, the PGEC needs to be capable of protecting itself against malicious execution environments, which are manipulated in order to steal data out of the container. In addition, it also needs means to prevent program code provided by the service user or service providers from transmitting data out of the PGEC. The following three classes of attacks can be used to retrieve data out of a PGEC:

1. Sending data via network, shared memory or file system to a collaborating entity,

2. Using covert channels [77] such as Central Processing Unit (CPU) load or similar to signal data to a collaborating entity,

3. Compromising the PGEC or its execution environment on the host system to undermine the protection means against threats of classes 1 and 2.

To emphasize the proof of concept, a list of possible attacks to gain private data protected in a PGEC is presented.

Regular Data Protection

Obviously, collections of private data can be gathered by sending/copying any private information, which is gained, to a collaborating entity. Such transmission over one of various applicable channels is considered an attack. In general these channels are network links, shared memory or a file system to which the collaborating entities have mutual access. To protect the private data in the container from those attacks, these channels and corresponding attacks are enumerated. It is considered that all these attacks are realizable by regular means using Java APIs. Table 4.1 enumerates the attacks together with the respective countermeasures to prevent those that are implemented in the container. The respective implementation issues are described in Section 5.1.

Protection from Covert Channels

Besides the channels applicable for communication, which are provided through a Java API, there are channels, which are not a priori identified as applicable for communication. Those are called covert channels. Table 4.2 lists a number of covert channels that might be exploited to communicate private data to the outside world. The presented countermeasures show awareness of a number of covert channels and the ability to minimize the chance of successful communication along those channels. Of course, reaction on unknown covert channels is hard or even impossible to achieve, which is not part of this work.

	Attack	**C**ountermeasure
1	Replace `SecurityManager` of the JVM	Override and hardcode `Permissions` in `checkPermission` Method of PGEC's own `SecurityManager`
2	Read a file	
3	Write a file	
4	Open a network connection (TCP/UDP)	
5	Wait for or accept network connection (Listen on a port)	
6	Redirect default system streams	
7	Print on printer	
8	Call native methods that are not managed by the JVM	
9	Access private methods of the `PrivacyManager` via reflection and override of Java method access control	
10	Read and write system properties or read environment variables	
11	Display a GUI window	
12	Access to databases via JDBC	JDBC access relies on socket or file access, which is solved by the `SecurityManager`.
13	Access private data via the `PrivacyManager` without it being instantiated and installed in the first place	The constructor of the `PrivacyManager` checks whether there is no other `SecurityManager` yet, and whether it is instantiated from the main thread and there was no other action before.
14	Exchange data via static fields (shared memory); includes trying to get references on objects in other execution environments.	Use various own `ClassLoaders` for each execution environment.
15	Render audio	Override security `Permissions` and replace audio system classes with own `ClassLoader`
16	Write to system streams	Redirect system streams to NUL device and prevent further redirection by hard-coded `Permissions` in own `SecurityManager`
17	Access information with a debugger	Prevent debugging; i.e., compile with option -g:none does not create debug information in the Java binary, prevent running the debug interface library (jdwp.dll) within the PGEC process. The latter is achieved a a side effect of the assertion of untampered host environment (see Section 5.1.4).

Table 4.1: Attacks launched via regular API means using Java and respective countermeasures provided by PGEC

	Covert Channel Attack	Countermeasure
18	Throw `Exception`s with messages containing private data	Catch all `Exception`s and filter messages in the container execution environment that started the service
19	Start threads with names containing private data and enumerate running threads from outer environments	Assign each execution environment with a separate thread group and prevent threads within that group from retrieving parental thread groups, which would contain threads or thread groups of other execution environments.
20	Synchronize on globally accessible object monitors or semaphores	PGEC holds locks on every globally accessible object's monitor This includes all box objects created during automatic boxing of primitive types, the `Class` objects of classes loaded by the system `ClassLoader` and all static properties of such classes. (see Listings 5.10 and A.4)
21	Encode information by the amount of allocated memory	PGEC allocates large amounts of memory itself in a random fashion; Drawback: the application in the container must be able to handle `OutOfMemoryError`s
22	Encode information in the CPU load	PGEC produces random CPU load or steady high CPU load by a background thread that consumes all remaining CPU cycles that are not assigned to any application outside or inside the container
23	Request loading of classes that are not actually class files but rather pipes open by external communication end points; similar "touching" of files can be detected outside and hence be used to trickle information out	Determine actually existing class files in the classpath at startup time and prohibit and suppress any useless class loading during runtime
24	"Morse out" information by alternating class loading of existing classes	Class loading is allowed only once per run
25	"Morse out" information with allowed literals	Random mixing of semantically independent literals; if possible all literals should be semantically independent
26	Encoding of information with literals over synchronized time	PGEC introduces a random latency in the transmission of literals
27	"Morse out" information by sending equal packets over the PGEC interconnection and sniff on the network	Use stream ciphers for PGEC intercommunication to make originally equal packets incomparable in the encrypted network communication
28	"Morse out" information by alternating sending of packets to collaborating communication partners in distributed PGEC; use network sniffing to evaluate the communication with the respective PGEC instances over time Packet from A to B encodes 0 Packet from A to C encodes 1	Send padding packets to all other associated instances of the distributed PGEC that are not involved in the actual communication.

Table 4.2: Covert Channels Attacks executed by program code inside the PGEC

Protection from Compromised PGEC or its Host System

Especially, the countermeasures enumerated in Table 4.1 rely on means provided by the implementation of the JVM and the Java security architecture. Hence, an attacker, e.g., a data collector or data miner, may attempt to circumvent the protection means of the PGEC by overriding the Java security measures or comparable measure in the OS. Such attacker could provide her/his own JVM including modified versions of the Java networking API. Such modification could access network endpoints, i.e., `Sockets`, without permission check by the `SecurityManager`. Thus, the PGEC has to self-defend against modifications of the JVM and its system classes or OS provided libraries. Since, the modifications are a priori unknown, the PGEC attempts to ensure that is runs in well-known environments only. That is, PGEC runs only in certified versions of OSs and JVMs. Means to assert this is implemented and described in Section 5.1.4.

Chapter 5

Prototype Implementation

PGECs have to fulfill mainly three tasks. They have to provide an execution environment that ensures proper clean up of data and code when the service is no longer used. It further restricts the information exchange via all possible channels to what is allowed in the privacy contract. Such functionality can be inherited by using runtime environments that ensure secure code execution such as Java runtime environment, .NET or even the Adobe[1] Shockwave interpreter. All of the mentioned approaches provide some kind of a sandbox model which limits the access of foreign code to local resources such as file system and network. Since the PGEC needs to protect itself against the runtime environment, additional conditions need to be fulfilled to guarantee correct behavior the PGEC. The prototypical implementation explained in this section is done in Java. This allows to take advantage of the sandbox and security management facilities of the Java runtime environment. The security manager, controlling the sandbox, can prevent file and network access, access to system properties, access to printers and even to methods implemented in native and hence unmanaged code.

5.1 Technical Aspects and Components of the PGEC

This section describes aspects of the implementation of a PGEC using the example of a prototypical implementation in Java. Since, all components of a PGEC instance in the distributed architecture also exist in the stand-alone version, almost all aspects apply to both architectures likewise. Differences exist in the API for data access (Section 5.1.5), which has to enable transparent access to private data throughout the distributed components and to provide logical unity. Finally, those distributed instances have to mutually authenticate to ensure that exchanged private data remain logically inside the PGEC (Section 5.1.6).

[1] http://www.adobe.com/

5.1.1 PGEC and its Host Runtime Environment

The code, that is the PGEC and the services within, is executed in a runtime environment and thus may only access system resources through that runtime environment. This circumstance allows limiting the access to resources by security managers. These security managers obey certain security policies. The code inside the sandbox may only access those resources that are explicitly granted.

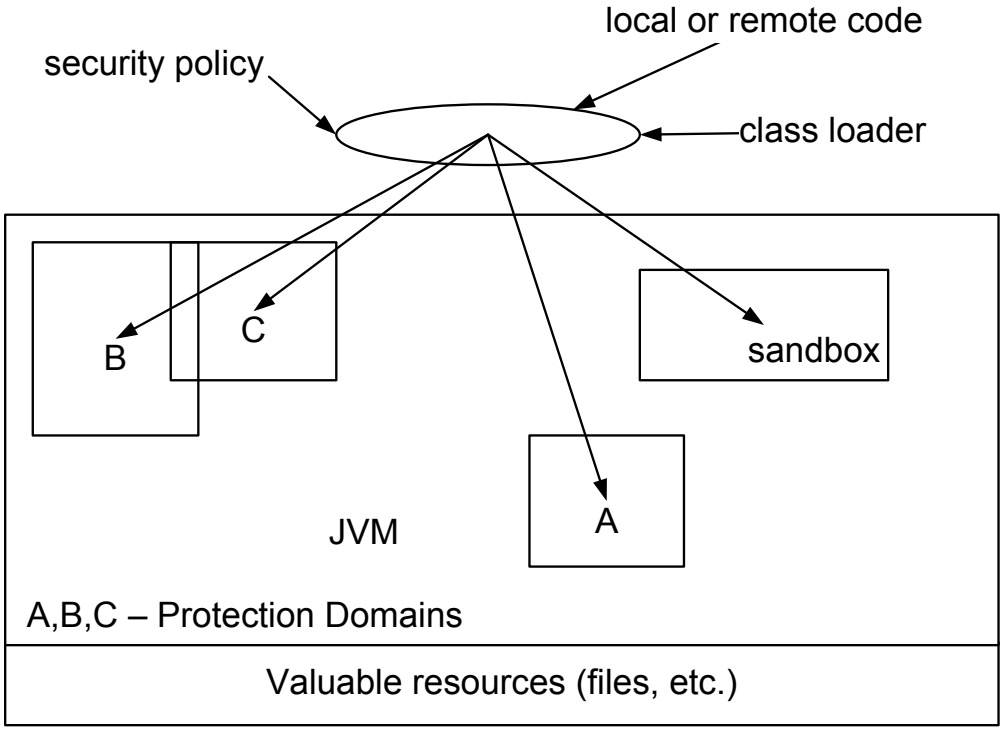


Figure 5.1: Java 2 Security Model [78]. The evolution of the Java security model even introduced protection domains, which resemble the execution environments of the PGEC's architecture. While the security policy and access permissions define the allowed actions, the protection domain and access control checking provide the enforcement. In the Java security model, the security policy and the access permissions are defined by the owner/executor of the JVM. In contrast to that, in the PGEC these are defined by (negotiated) privacy contracts and must be immutable by the executor of the JVM running the PGEC and the services within.

In earlier versions of Java, only remotely loaded code ran inside the sandbox and had limited permissions. That is, locally started code had no limitations in accessing system resources. The evolution of Java and its security model developed protection domains for any type of code, regardless of locally or remotely loaded and whether signed or not (see Figure 5.1). These protection domains provide a reasonable base for the execution environments of the PGEC. Unlike in the Java security model, the security policy and the access permissions must not be set or changed by the executor of the JVM hosting

the PGEC. This policy depends on the data access and usage restrictions imposed by the data owner. Additionally, the PGEC addresses and limits access to system resources, that usually have not much or no security relevance, e.g., GUI or audio system. The limitation of the service permissions can go that far that services are only able to communicate with their execution environment. Thus, the service is not able to send any information anywhere else but to the container. There is no need for a service to directly access any local resource. Every data or logical resource can be accessed through and only through the container. The container grants access to resources only if the service is authorized by the privacy contract with the data owner/service user. While the service providers may start a container component themselves, they are actually enabled to set the security policies on their own. In order to still guarantee proper functionality of the PGEC, the security policy that applies to the services in the container's execution environment must not be adjustable by the executor of the container. Thus, the according security policy must be fixed and an appropriate security manager must be running to obey this policy. To accomplish this, the container itself is designed and implemented incorporating a security manager including the constricted policy for the services inside. The PGEC's implementation inherits from `SecurityManager` and is named `PrivacyManager` to reflect the privacy protection purpose. To ensure that this embedded security manager is not bypassed, the container instantiates only if

(1) there is no other security manager already installed,

(2) the `PrivacyManager` is the very first component started in the JVM and

(3) the JVM has not been tampered with.

Condition (1) is needed to ensure that only the trusted and certified security manager of the PGEC is running. If there was another security manager installed beforehand, it may not be possible to install the containers own security manager and the other manager cannot be trusted by the container, which results in a security and privacy leak. This condition is checked by lines 203, 204, 208 and 209 of Listing 5.1.

Condition (2) is used to ensure that no components are started prior to the PGEC. Otherwise it was possible for those components to open and keep a network or file handle without control of the container. Such might be used to circumvent the container access restrictions. The critical point here is the constructor of the `PrivacyManager`, which is designed to check itself whether it was called from the right class and hence being installed as the very first action within the program flow. In the stipulated program flow, the `main` method of `PrivacyManager` shall be the entry point of any program using the PGEC. This is to ensure that there is no thread running or class loaded out of control of the PGEC. As can be seen in Listing 5.2 the singleton accessor method is called the very first, which will eventually call the constructor of `PrivacyManager` (see Listing 5.1).

Listing 5.1: This method provides convenient access to the `PrivacyManager` as a singleton implementation and in imitation of the static access to the `SecurityManager` including appropriate type casting.

```
202     public static PrivacyManager getPrivacyManager() throws PrivacyException {
203         SecurityManager secMan = System.getSecurityManager();
204         if (secMan == null) {
205             // no SecurityManager set yet set this PrivacyManager as new
206             // System wide SecurityManager
207             return new PrivacyManager();
208         } else if (secMan.getClass().equals(PrivacyManager.class)) {
209             return (PrivacyManager) secMan;
210         } else {
211             throw new PrivacyException("Privacy Manager could not be installed");
212         }
213     }
```

Listing 5.2: Entry point for the PGEC.

```
666     public static void main(String[] args) {
667         PrivacyManager privacymanager = getPrivacyManager();
```

Within the constructor, the call stack must look like this.

```
   com.endosoft.pgec.PrivacyManager.<init>(PrivacyManager.java:135)
at com.endosoft.pgec.PrivacyManager.getPrivacyManager(PrivacyManager.java:207)
at com.endosoft.pgec.PrivacyManager.main(PrivacyManager.java:667)
```

It appears to be sufficient to check whether the stack height equals three (line 137 of Listing 5.3) and that the bottom element points to the `main` method of `PrivacyManager` (lines 143 and 149). Since this check is done within the constructor, the topmost element is obviously this constructor itself. While this constructor is private and only called from the static `getPrivacyManager()` method, the next element in the stack trace is just this method. The next stack element has to be the `main()` method of the `PrivacyManager`. This is true only in two cases. The first is, the mentioned `main()` method is the applications entry point, which makes it the bottom element of the stack. In the second case, PGEC's `main()` method has been called from somewhere else, which increases the length of the stack trace to more than three.

Condition (3) ensures that the security features provided by the JVM are not circumvented by a malicious JVM implemented by an attacker. Such a JVM probably does not implement any of the security concepts built-in into Java. By that, private data may be released even if the security manager of the PGEC is enabled. Hence, the container has to check its hosting JVM. To prevent possible JVM manipulations only well-known JVM implementations are accepted. More details are found in Section 5.1.4.
These three conditions ensure that no private data can be disseminated without user consent. They ensure that the PGEC runs only in a proper and secure environment and

Listing 5.3: Stack trace evaluation to assure that the main method of `PrivacyManager` is the systems entry point and no other action is performed before instantiation and installation of the `PrivacyManager`.

```
129     private PrivacyManager() throws PrivacyException {
130         if (!checkSoftwareEnvironment()) {
131             throw new PrivacyException("there are modules loaded that are not "
132                     + "allowed, deprecated, manipulated");
133         }
134
135         StackTraceElement[] stackTrace = new Throwable().getStackTrace();
136         int stackLength = stackTrace.length;
137         if (stackLength != 3) {
138             throw new PrivacyException(
139                     "PrivacyManager must be instantiated from its own main "
140                             + "method (stacktrace length = 3)");
141         }
142         StackTraceElement bottomStackElement = stackTrace[stackLength - 1];
143         if (!bottomStackElement.getClassName().equals(
144                 "com.endosoft.pgec.PrivacyManager")) {
145             throw new PrivacyException(
146                     "only com.endosoft.pgec.PrivacyManager may instantiate "
147                             + "itself");
148         }
149         if (!bottomStackElement.getMethodName().equals("main")) {
150             throw new PrivacyException(
151                     "com.endosoft.pgec.PrivacyManager.main(String[] args) "
152                             + "must be the program entry point to instatiate "
153                             + "PrivacyManager");
154         }
```

thus, data is secured by PGEC means. Otherwise they prohibit container instantiation and by that data access, since services may access privacy relevant data only inside the container.

Class Design Practice

It is commonly not considered good practice to build classes with more than 2000 lines of code. Due to the special circumstances that require most of the code within and used by the `PrivacyManager` to be protected from external use, the most methods within have to be private. Those methods cannot be outsourced to external classes. As much as possible inner classes have been used to structure the code while maintaining the access protection achieved from private methods and fields. For clearer arrangement, those inner classes should be moved into separate files using technologies like *includes*, known from C, C++ and other programming languages or *partial classes*, known from C#. Unfortunately, the Java language provides neither of these means.

5.1.2 Other Protection Means

In the following, protection means, which are not addressed by the `SecurityManager`, are described. The Java sandbox model is able to restrict access to system resources. This was designed from the perspective to protect a user's machine from foreign code, that might be loaded over the Internet. These restrictions can be weakened for particular resources if the user trusts the foreign code. To increase trust code signing was intro-

duced. Unfortunately, the sandbox was not designed to extend the restrictions. There are some resources that were not considered problematic, i.e., memory access, audio devices. Untrusted access to the GUI is addressed by marking the respective window but complete window suppression is not generally considered. An access control on the programming level exists, but may be circumvented by reflection, if not appropriately prevented by the `SecurityManager`. Even the introduced error handling mechanisms can be misused for privacy attack.

Bypass Java Language Access

The Java language provides a number of access modifiers, such as `public`, `protected` and `private`. If the modifier is omitted, access is restricted to classes within the same package only. Usually these are checked when an object tries to access a field, method or constructor. In most cases this is already done at compile time. That makes it impossible to directly access a `private` method from outside the defining class. By using the reflection capabilities of the Java language fields, methods or constructors can be accessed dynamically. The access check is done at runtime. There is a flag that may be set to bypass the Java language access check. Using this flag would allow to access `private` fields, methods and constructors from places that are not supposed to. Fortunately the setting of this flag can be prevented by a `SecurityManager` implementation. The `PrivacyManager` representing the PGEC is designed to do so. Hence, there are two cases: the `PrivacyManager` is already installed as the system's `SecurityManager` or it is not. In the first case, an attempt to set the flag will be successfully avoided with throwing a `PrivacyException`. If there is no `SecurityManager` or an implementation of `SecurityManager` that might not care about `ReflectPermission`s, the flag can be successfully set. To actually gain access to the field or method under attack an instance of the containing class is needed. Such instance of `PrivacyManager` can be retrieved via the singleton accessor method or its constructor. The constructor is defined `private` but using the mentioned flag it could still be called. The singleton accessor method either returns the existing instance of the `PrivacyManager` or creates an instance by calling its constructor. This instance is returned in any subsequent calls. As described above, the `PrivacyManager` assures itself fulfilling the aforementioned conditions or does not instantiate otherwise. These measures ensure that the `PrivacyManager` is instantiated and installed under very tight restrictions only and thereby prevent access bypasses through reflection. Given this, it can be relied on the Java language access control by modifiers in all other places.

Isolating Execution Environments Inside the PGEC and Preventing Shared Memory Access

The Java runtime environment provides no concept of pointers or direct memory access. Objects in memory are accessed through references. Due to the lack of direct memory access there is no actual concept of shared memory either. On the other hand, any

object can be shared within the Java virtual machine by sharing its reference. Security permission checking is only done for external resources like files, sockets and native library linking but not for object access in memory. That is, if it was possible to share a reference between a service running within the container and some colluding application outside the container, this could be misused for uncontrolled transfer of data from inside to outside. Such data might be private data of the service user and hence, such transfer is a privacy breach. As there is no means to effectively divide privacy relevant data from non-privacy-relevant data, it is not an option to try to allow reference sharing for non-relevant data only. Effective protection can only be granted when no reference is shared at all. The ways of sharing a reference shall be evaluated. First is to share a reference by passing it as a parameter through a method call. Second is to fill it into an available property of another object. Third and forth are analogue but using statically available methods or properties. In the first two cases, a reference to the object must be already known. Creating the reference on its own by newly instantiating an object is not a problem, because the new object lies also either inside the container or outside just as the creating object does. When not creating the reference itself there exists a path along known references to that reference. Since each step in the path was created by instantiating an object - unless the step used a static reference - all objects along the path are actually on the same side of the container, which is no problem.
The open point here are static references. Static reference are accessible through the class itself rather than an actual instance. They are accessible even if there is no instance of that class at all. The only preconditions are that the class is loaded and its name is known to the object that tries to access it. It is not feasible to assert that an object does not know the name of a class, especially in the case that the service running inside and the colluding application outside the container are created by the same party. The obvious way to prevent the access to static references is to have the class not being loaded. But if the service and the colluding application are running, their respective classes are already loaded. There must be a way to have the class with the static reference being loaded but still not accessible or not even been found by the class that tries to access it. When a class is accessing another class, it requests its own `ClassLoader` for that other class. The `ClassLoader` knows all classes it has loaded itself or forwards the request to its parent `ClassLoader`. So when using different `ClassLoader`s inside and outside the container and making sure that each does not return a class loaded by the other one, a class with a certain name can be loaded multiple times. The multiple classes with the same names use different memory and hence, their respective static references point to different places. That way all changes to static references or the content behind are accessible to objects of classes loaded by the same `ClassLoader`.
PGEC uses different `ClassLoader`s for different execution spaces, which ensures that objects in different execution spaces do not have handles on instances from other spaces. Only the container or classes within just that other execution space know those handles. The container will not give those handles to the objects instantiated in another execution

space. To access static fields or methods of a class, the accessing object must have hold of that class with static fields. If it is loaded by a particular `ClassLoader` instance it will not get hold of the classes loaded by a different `ClassLoader` instance but load an own copy of a class of that name. Thus, no static changes in instances of that class are visible to objects instantiated by the other `ClassLoader`. Thereby there is no shared memory available. That allows to run various services in parallel while preventing access to each other.
In the implementation, a new `ClassLoader` was inherited that makes up an execution environment (see Listing A.3). Further, a `ClassLoader` for services within the container differs from `ClassLoader`s for other applications outside the container. This differentiation of logically being inside or outside the container has been introduced in Section 4.2.1. Hence, specializations for an `InnerExecutionEnvironment` and an `OuterExecutionEnvironment` have been inherited from that general `ExecutionEnvironment`. One of those differences is the capability of injecting replacement code for inner classes as described in Section 5.1.2 and displayed in Listing 5.6.

Audio System

One of the system resources that was not considered security relevant by the creators of Java and its security management is the audio system. The sound system distinguishes sampled and synthesized audio. The sampled audio system plays audio stream that are based on samples, e.g., WAV-files, MPEG-1 Audio Layer 3 (MP3)-files or Compact Discs (CDs). The synthesized audio supports the Musical Instrument Digital Interface (MIDI) protocol, which allows to generate sound by specifying tones rather than sound samples as well as to record those tones from adequately equipped music instruments connected to the MIDI interface at the sound card. As it is mostly used as an output device by synthesizing music with (multiple) instruments and it cannot be used to access or manipulate data of the user that is running a Java sandbox, it can be assumed not to do any harm. On the other hand, the specification of tones is pretty expressive and hence, could be used to emit information that can be sensed by some application that is not bound to the PGEC. Consequently, it might not be security relevant but may enable privacy breaches. Imaginable ways to do so are virtual sound devices or virtual MIDI implementations that actually just record to disc instead of emitting sound.
When accessing the MIDI system with the existing Java API, the `SecurityManager` is consulted only to access some system properties. Even suppressing those properties or emptying the respective values does not prevent the existing MIDI system from being used. In the standard Java implementation even without being able to enumerate the devices, a handle to the according `MidiDevice.Info` can be retrieved with explicit knowledge of the identifiers of the existing MIDI devices. Further, several MIDI devices may exist on a single PC and it is almost impossible to figure out the malicious ones. The easiest way to do so is to exchange the code for class `MidiSystem`, which is statically

used. The injected class code implements the same methods as the original class but never returns anything that can be used to produce MIDI audio. First of all, that is not to offer any `MidiDevice.Info` as to been seen in Listing 5.4. Analogous, the code for

Listing 5.4: Replacement of `MidiSystem` for inner execution environments.

```
24  public static MidiDevice.Info[] getMidiDeviceInfo() {
25      return new MidiDevice.Info[0];
26  }
```

class `AudioSystem` can be replaced with something that pretends not to have any audio device (mixer) at all (see Listing 5.5). When an application approaches to access one of

Listing 5.5: Replacement of `AudioSystem` for inner execution environments.

```
42  public static Mixer.Info[] getMixerInfo() {
43      return new Mixer.Info[0];
44  }
```

the audio systems it has to retrieve a handle to either a `MidiDevice` or `Mixer`. These handles can only be gotten from the classes `MidiSystem` and `AudioSystem`. PGEC provides a replacement code for both of them. When one of these classes is accessed the class loader of the accessing class is requested to find the class `MidiSystem` or `AudioSystem`. Since all classes inside the container or rather within an inner execution environment are loaded by an `InnerExecutionEnvironment`, the access requests are caught and the replacement code is injected, see code snippet in Listing 5.6.

Graphical User Interface (GUI)

The highest level of privacy protection can be gained for logically delivering services. Logical delivery requires some representation of the result, e.g., some changed or filled file or a display to the service user. This service class appears to be unreasonable without representation of the result. On the other hand, display of a GUI has been identified as Attack #11 in Table 4.1. Obviously, it is critical to allow a service to produce a GUI at devices under control of the service provider. As already mentioned in Section 4.1.1, GUIs may be presented at devices under the user's control. These devices can be identified as the origins of the user's private data. From the scenario introduced in Section 1.2.4, it is known that user's data may even accrue on other devices. Further, a user may allow, depending on her/his trust into another party, other devices to display a GUI as well. A chat service may be considered, which is provided by one service provider to two or more chatters. While the chatters should each see a GUI and hence, the communicated messages, the service provider should neither see nor log the conversation. PGEC's method in Listing 5.7 is consulted if and when any GUI operation, such as opening a window, is executed. Therein, callers from inner execution environments (line 1000) undergo further evaluation. It is checked, whether either for any container

Listing 5.6: Replace various class code in inner execution environments.

```
98      protected final byte[] loadClassData2(String name) {
99          if (classesToBeReplaced.contains(name)) {
100             String filename = name.replace('.', File.separatorChar).concat(
101                     ".class.replacement");
102             System.err.println("one should not load " + name + " within PGEC "
103                     + name);
104             return loadReplacementClassData(filename);
105         } else {
106             return null; // do not use super.loadclassData here
107             // if the class has not to be replaced
108             // it has to be checked whether it is a "global" PGEC class
109             // therefore use NULL here
110         }
111     }
112
113     private byte[] loadReplacementClassData(String name) {
114         // load the class data from the connection
115         log.debug("loading replacement class " + name);
116         // TODO get rid of hard coded pathnames
117         File classToLoad = new File("D:/development/dissertation/PGEC/classes",
118                 name);
119         if (classToLoad.exists()) {
120             int classLength = (int) classToLoad.length();
121             byte[] out = new byte[classLength];
122             try {
123                 FileInputStream fis = new FileInputStream(classToLoad);
124                 int ptr = 0;
125                 while (ptr < classLength) {
126                     ptr = fis.read(out, ptr, classLength);
127                 }
128             } catch (FileNotFoundException fnfe) {
129                 out = null;
130             } catch (IOException ioe) {
131                 out = null;
132             }
133             return out;
134         }
135         return null;
136     }
```

instance (line 1002) or explicitly for this particular local container instance (line 1003) GUI access is granted. The information, which container instances are granted with GUI access rights (line 1004), has to be derived from the negotiated privacy contract. In the existing implementation, this is explicitly set (see Listing 5.8).

Despite in Section 5.1.2 the use of the audio system by services in inner execution environments is generally prohibited and prevented, it shall be considered to grant access to the audio system in the same way as the to the GUI. To accomplish this, the necessity for class replacement in Listing 5.6 can be additionally checked analogous to the checks in lines 1001 to 1003 of Listing 5.7.

5.1.3 Covert Channels

In Table 4.2 a number of imaginable covert channel attacks have been enumerated. The table also provides brief descriptions of possible counter measures. The enumerated covert channel attacks start with those that are able to transfer blocks of informations, e.g., strings, at once and continue with channels that enable ‘Morseing’ of small pieces of information. These small information pieces can then be used to encode larger infor-

Listing 5.7: Actions regarding the GUI require `AWTPermission`s to be granted. Applications in inner execution environments may only create and display windows if explicitly permitted by the privacy contract.

```
996     private void checkPermission(AWTPermission perm) throws PrivacyException {
997         // TODO handle according to the name
998         // i.e. inner classes shall never read out pixels on the screen
999         // perm.getName();
1000        if (isInner()) {
1001            Set<ContainerID> grantedContainers = getEffectivelyGrantedVisualizationSites(
                    getEnvironmentID());
1002            if (grantedContainers.contains(null)
1003                    || grantedContainers.contains(getContainerID())) {
1004                // GUI is allowed from this environment at this container
1005            } else {
1006                throw new PrivacyException(
1007                        "inner classes in this execution environment may not open GUI
                                Frames in this container");
1008            }
1009        }
```

Listing 5.8: API to explicitly grant access to GUI.

```
2215    private PermissionStructure permissions = new PermissionStructure();
2216
2217    private HashMap<ExecutionEnvironmentID, HashMap<DataItemID, Object>>
            assignedPrivateData = new HashMap<ExecutionEnvironmentID, HashMap<DataItemID,
            Object>>();
2218
2219    @Override
2220    public void allowGUI(ExecutionEnvironmentID executionEnvironmentID)
2221            throws PrivacyException {
2222        allowGUI(executionEnvironmentID, null);
2223    }
2224
2225    @Override
2226    public void allowGUI(ContainerID containerID) throws PrivacyException {
2227        allowGUI(null, containerID);
2228    }
2229
2230    @Override
2231    public void allowGUI(ExecutionEnvironmentID executionEnvironmentID,
2232            ContainerID containerID) throws PrivacyException {
2233        if (isOuter()) {
2234            permissions.allowGUI(executionEnvironmentID, containerID);
2235        } else if (isInner()) {
2236            throw new PrivacyException(
2237                    "inner classes may not grant rights to display a GUI by themselves");
2238        } else {
2239            // actually it is pretty unlikely that the container or its classes
2240            // grant rights to inner execution environments
2241            throw new PrivacyException(
2242                    "only outer classes may grant rights to inner environments to display a
                            GUI");
2243        }
2244    }
```

mation blocks. This section details a few of them (covert channel attacks #18 to #22). To prove the counter measures to be effective, attacks using the respective covert channels have been implemented. It can be shown that the attacking code is capable of transmitting some information using its chosen covert channel. Further, respective counter measures have been developed that are embedded in the PGEC. With the counter measures enabled, it can be shown that the attacking code is not further capable of transmitting its information. In fact, the receiving part of the attacking code is not able to identify or

decode the information from the covert channel.

Throwing Exceptions

In Table 4.2 (Attack #18) the messages included in exceptions have been identified as a transportation means for private data from within the container. An attacker might throw a user defined `RuntimeException` with a message containing anything she/he likes. As this could even be encrypted, a pure filtering of the messages is not feasible for data protection. Further, it cannot be a priori known which exceptions may occur. Fortunately, Java provides a means to catch any uncaught exception. Uncaught `Exceptions` other than `RuntimeExceptions` or its heirs are already complained at compile time. The required handler, for catching `RuntimeExceptions` can be set by the `ClassLoader`. PGEC implements own `ClassLoaders` to create multiple inner and outer execution environments. These `ClassLoaders` are the barrier between the environments and can also be the barrier for uncaught exceptions inside their loaded classes. Therefore the class loader spanning inner execution environments only has to override `getUncaughtExceptionHandler()` and provide it with an exception handler that discards the leaking exception. In Listing 5.9 the exception occurrence is notified for debugging purpose. Note, exceptions that are caught by service code running inside the PGEC will not leave the inner execution environment anyway.

Listing 5.9: Discarding any uncaught exception raised within an inner execution environment

```
164     @Override
165     protected UncaughtExceptionHandler getUncaughtExceptionHandler() {
166         return new InnerUncaughtExceptionHandler();
167     }
168
169     private static class InnerUncaughtExceptionHandler implements
170             Thread.UncaughtExceptionHandler {
171
172         @Override
173         public void uncaughtException(Thread t, Throwable e) {
174             // this is an inner exception
175             // suppress all messages
176             System.err.println("Exception raised but don't tell anything");
177             // TODO remove this debugging output
178             System.err.println("DEBUG: " + t + " " + e.getMessage());
179         }
180
181     }
```

Named Threads

To give services the flexibility of concurrent execution, Java provides threads. Besides the program logic, these threads carry some meta data. These are, e.g., the priority, a name and the thread group it belongs to. An attacker inside an inner execution environment could try to start threads with names containing private data. Her/his collaborator

in an outer execution environment could enumerate all running threads and getting the names from them, containing the private information. The PGEC tackles this by assigning each execution environment with a separate thread group. Further it prevents threads within that group from retrieving parental thread groups. Only from these parental thread groups, sibling thread groups belonging to other execution environments or threads can be enumerated. That way the names of threads in other execution environments remain hidden and no information can flow through these names.

Encoding on unapparent information carriers

Thread Synchronization The first example of unapparent information carriers are monitors and semaphores (attack #20). When these are globally accessible to a software system, such as the services running in inner and outer execution environments, they can be used for thread synchronization across the borders of execution environments. A basic attack #20 requires three threads and three object monitors (X, A and B) to synchronize on. One thread is the sender of the information, while the other two are receivers. The monitor X is used to synchronize all three of them. That is, it must be ensured that the sender locks the critical section of X first and until both others are waiting directly before their critical sections of X. When the sender leaves X, the receivers pass their critical sections quickly and the sleep a bit to give the sender a head start. This head start allows the sender to enter critical sections of A and B. After the sleep of the receivers both try to enter these sections as well, one section after the other. One receiver tries to enter A first and the other enters B first. As long as the sender hold the locks both cannot enter their critical sections. Depending on the sequence, in which the sender leaves the critical sections of A and B, one or the other receiver is preferred in entering the sections. The receiver that wins this 'race' determines the received information. One is responsible for 0 and the other one for 1. An exemplary chart showing the sequences, in which the critical sections of X, A and B are entered and left is displayed in Figure 5.2. In this example the sender encodes a sequence of 0-1-1 using the mentioned object monitors.

To counteract this covert channel, PGEC holds locks on every globally accessible object's monitor. This includes all box objects created during automatic boxing of primitive types, the `Class` objects of classes loaded by the system `ClassLoader` and all static properties of such classes (see Listings 5.10 and A.4).

Listing 5.10: Gather lots of locks to prevent synchronization of threads in different execution environments as a countermeasure to covert channel attack.

```
669 //              new InfiniteLock(privacymanager);
670                 for (Class<?> clz : classesBelongingToPGEC) {
671 //                  new InfiniteLock(clz);
672                 }
673                 new InfiniteLock(); // locking the boxed primitive types
```

By holding all the locks during the complete runtime of the PGEC, every other thread attempting to gain a lock on one of these has to wait until the lock is released. The locks

are released only at the `System.exit()` call, which also shuts down the waiting threads. Thus, those have no chance to ever run, resulting in starvation of the attacker. Hence, there is no chance for synchronization on those globally accessible objects. Synchronization is possible only on objects loaded by the execution environments `ClassLoader`. From Section 5.1.2 it is known that these cannot be shared between inner and outer execution environments.
Unfortunately, it has been discovered that the class `Window` holds a lock on itself, respectively using a synchronized method, during the instantiation of `Window` or its descendants. This is required to create any GUI and hence, the `Window` class cannot be locked by the PGECand is exempt from locking (see Listing 5.11). Even though, there is only one class explicitly denoted, this list grows at runtime. Every class that contains static synchronized methods is added to that list. The more classes that are in that list, the more classes' monitors can be misused for exploiting covert channel attack #20 using inner/outer synchronization. For prevention of this, the system classes, such as `Window` and those in packages `java.*`, which cannot be replaced by own `ClassLoader`s, have to be redesigned to not use static synchronized methods.

Listing 5.11: Enumeration of objects that cannot be locked by the PGEC to prevent a deadlock, e.g., when opening a GUI.

```
290         private static final HashSet<Object> criticalObjects = new HashSet<Object>();
291
292         static {
293             criticalObjects.add(Window.class);
294             criticalObjects.add(SecurityManager.class);
295             //criticalObjects.add(System.err);
296             //criticalObjects.add(System.out);
297         }
```

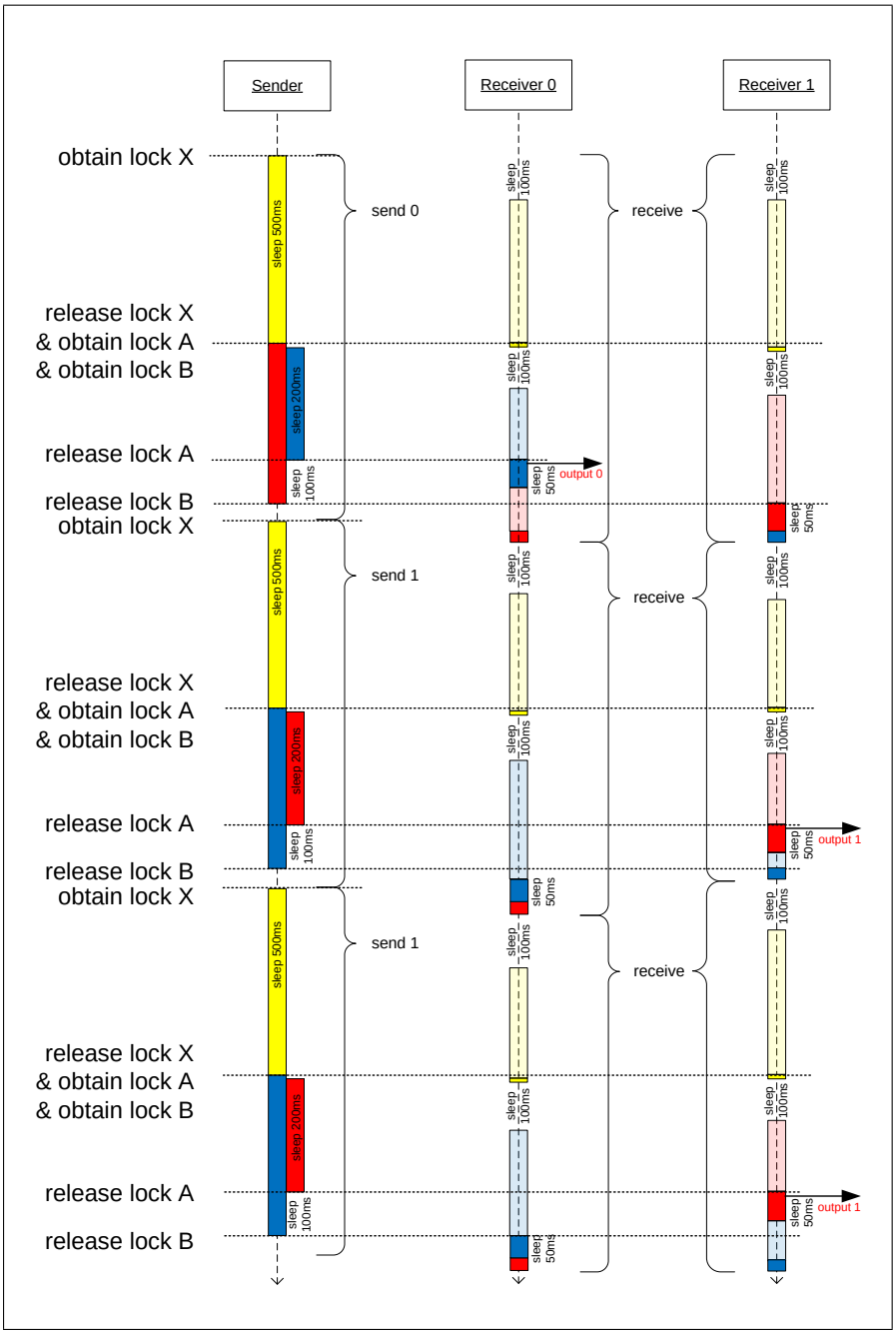


Figure 5.2: Sequence chart of concurrent threads transferring information by synchronization on globally accessible object monitors. The colors in the chart reflect the monitored objects. X - yellow, A - red and B - blue. A solid block of particular color means that the thread holds the lock on that objects monitor. A thread waiting for a particular lock is depicted as a faded block of the respective color. Within the critical sections there is not much logic but merely sleeping. The receiving threads use a shared object to determine which one is the first to enter the critical section of A or B respectively. The first outputs a 0 or 1 accordingly.

Memory Load The next example of attacks is using the memory load ((#21)) as the covert channel. The sending part of the attacker allocates or frees memory, while the receiver can easily determine the amount of free memory. Since Java applications have no actual control over the exact execution of the JVM's garbage collector, there remains some uncertainty on the moment of actual memory clearance. Further, other threads may also allocate or free memory blocks. Therefore, the attacking code takes advantage of the Manchester encoding [79] based on a not to small clock cycle. While the Manchester encoding is self-synchronizing, the jitter resulting from delayed garbage collection can be compensated with a sufficiently long clock cycle. The existing attack implementation uses as clock cycle length of 1200ms. To generate such code, the rising edge matches the allocation of a rather unusual amount of memory (up to 128 MByte), whereas the falling edge is achieved by freeing just this memory block. These are accomplished by the methods in lines 19-45 of Listing A.5. The receiver (Listing A.6) of the information samples the signal modulated on the memory allocation more often than the a priori chosen clock cycle. Due to the possibilities of other threads allocating or freeing memory in the JVM, it adjusts the median value between high and low signal every time it came to a decision on rising or falling edge. This allows to cope with memory allocation noise up to half the chosen block size (64 MByte). For demonstration purpose it prints an approximation of the signal graph and the received information on the console without further processing. A detected *high* signal, that is highly allocated memory, is displayed as -(dash). *Low* signals, i.e., larger amounts of free memory, are reflected by _(underscore). The console output reads like this:

```
_------------------_____________
____-----_________---------_____-----_________--------_____-----_____-----_H
____-----_________----_____--------_____-----_________---------_________e
---------_________-----_____---------_________-----_____---------_____-----l
_____-----_________-----_____--------_________----_____--------_____-----_l
_____----_________----_____--------_________-----_____-----_____-----_____-o
-------_____-----_________---------_____-----_____-----_____-----_____-----_
____-----_________---------_________---------_________-----_____-----_____-W
-------_________----_____---------_________-----_____-----_____----_____o
---------_________-----_____-----_____---------_____-----_________---------_r
_____----_________-----_____---------_________----_____---------_____-----_l
____-----_________----_____--------_____-----_________--------_____-----_d
_____________________________________________________________________________________
```

With disabled counter measures, the generator can run in an inner execution environment while the reader in an outer execution environment can receive the transmitted "Hello World". It is obvious that this could be any other information from inside the container, too.

The respective counter measure follows about the same approach as the attacker. But instead of modulating a signal on the memory allocation it produces noise. That is, it allocates and frees large amounts of memory randomly over time (see Listing A.7). With these blocks being large enough, the reader cannot distinguish between memory allocations done by its colluding generator and the one from the counter measure. The size of

the blocks can be determined in multiple ways. The counter measure can figure out the amount of available memory already before any service or attacker has been loaded and started by the PGEC. The noise may then use a block of about that size. If the JVM can adjust the available memory the size has to by dynamically adjusted. Another option is to observe the memory allocation and determine unusual large allocation operations. These may be assumed as an attack and the noise blocks of the counter measure should have that size.

CPU Load Finally, a covert channel attack using the CPU load as its covert channel (#22) has been implemented to prove the concept and to show effectiveness of the respective counter measure. Much like the implemented attack using the memory allocation, the attacking sender (Listing A.8) modulates its information onto the CPU load with Manchester encoding. High signal results in high CPU load and low signal results in low load. Figure 5.3 shows a snapshot of the CPU load graph captured by the Windows task manager.

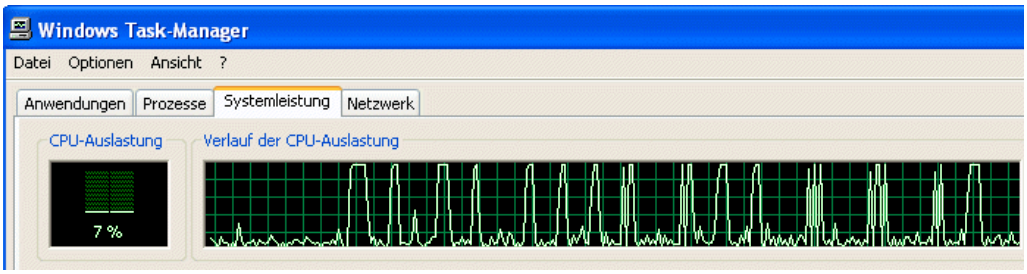


Figure 5.3: CPU load chart over time with modulated information.

The respective colluding receiver samples the CPU load and decodes the information from this. Java provides no API to determine the current CPU load. Therefore the Java implementation of the receiver uses a lowest priority thread using up all remaining CPU cycles to execute some dummy operation. From the number of dummy operations per time unit it can defer the CPU load (see Listing A.9). If the determination of the CPU load could be done with finer granularity the information could be encoded with even more symbols than just *high* and *low*. The counter measure against this covert channel also generates noise on the channel by randomly generating high CPU load (see Listing A.10).

5.1.4 Assertion of Untampered System

Besides harnessing against attacks from code running inside the container, the PGEC must also defend against external attacks. This form of self-guarding could be compared with the PE as described in Section 2.4. Obviously, a protection deeply embedded in the OS can be stronger than anything that could be implemented within a regular process or even a virtual machine. The efforts shown here shall merely act as a proof of concept. Assuming a clean demarcation between processes, the PGEC's process consists of the OS, the JVM, the PGEC itself and the applications within the container. Relying on the Java sandbox and security model, it can be assumed that these applications will not have access to the process and therefore not manipulate or access data around the `SecurityManager` implemented by the PGEC itself. Manipulation of the PGEC could be replacement of some of its classes.
By hash checking of the components of the currently running JVM, it can be assured that the privacy container and its native components are running besides an approved JVM. Hash checking and code signing of Java code already exists in the JVM and could be used. Attackers can of course self sign their manipulated version of PGEC or their very own implementation. Therefore the hash of the PGEC components must be compared to a hash code placed elsewhere than in a publicly readable certificate. Since a PGEC instance has to authenticate to other remote instances, it must prove knowledge of some cryptographic key. It is considered as difficult to get the key as to get the hash code. Further, it may be even more difficult to replace the hash code with the hash of the falsified implementation or to appropriately pad this implementation to have the same hash value. The native components may further hash check their callers from within the JVM to prevent Man-in-the-Middle approaches using an approved JVM. Code attestation approaches for similar purpose are also proposed by [80] and [55]. The secure hash checking also ensures that those classes accessing external resources are the ones that actually do the call for access checking at the `SecurityManager`. The code in Listings 5.12 and 5.13 shows how the task of checking this condition is forwarded to native code.

Listing 5.12: Passing the task of checking the integrity of the JVM to native code.

```
129     private PrivacyManager() throws PrivacyException {
130         if (!checkSoftwareEnvironment()) {
131             throw new PrivacyException("there are modules loaded that are not "
132                     + "allowed, deprecated, manipulated");
133         }
```

The existing implementation (Listing A.1) also hash checks the JVM's and OS's libraries and modules. Doing all this in software is not a 100% secure solution. A Trusted Platform Module (TPM) [55] appears to be a good choice to achieve trust not only in the OS but further in the JVM and the PGEC.
Since no clean demarcation between processes exists in regular processes, this approach remains vulnerable to thread injection, memory reading/dumping or other process ma-

Listing 5.13: Declaration and inclusion of the native code.

```
2205    public native boolean checkSoftwareEnvironment();
2206
2207    static {
2208        System.loadLibrary("PGECnative");
2209    }
```

nipulation. Prevention or protection from those attacks cannot be achieved by such approach. Means like the Protected Environment (PE) (see Section 2.4) or the self-guarding techniques used be anti-virus software can be considered.
Even though the present implementation of system assertion cannot provide perfect protection, it is still crucial. Therefore the assertion is started quite early in the program flow, that is, in the constructor of the `PrivacyManager` (see Listing 5.12).
The native code enumerates all modules running within the current process and determines the files each of them has been loaded from and calculates their hash values. Then it compares the hashes with those read from a configuration XML file. If a module found in the current process is not enumerated in the configuration file the check method returns false, which causes the PGEC to abort. The respective interesting code lines are 50, 59, 64, 68, 91, 106, 109, 132, 149 and 153 in Listing A.1.
The modules and libraries that are allowed within the JVM process are enumerated in an XML file. This contains names, sizes, hash values, base addresses, entry points and in memory sizes of the allowed components. This also allows to enumerate multiple versions of a library to reflect various versions of Java, OS or security patches. The Listings A.2 and 5.14 display the XML schema and a snippet of an exemplary environment descriptor file. The environment descriptor shown contains a number of libraries that are loaded in debugging mode only. During development it is useful to have those allowed. In a productive environment these must not be allowed since debugging is a privacy leak, which would allow to extract private data from a running (or suspended) PGEC. Further, a productive version of the PGEC must not allow to change this descriptor file. A simple solution was to add another hash check in the `PrivacyManager`, to assert the integrity of the environment descriptor. The more secure solution is to migrate the descriptor file into a Smartcard or similar dongle. It should be even considered to migrate the whole environment check into hardware dongles or TPMs.

Listing 5.14: Snippet of example environment descriptor.

```
<?xml version="1.0" encoding="utf-8" ?>
<PGEC xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="PGECEnvironmentDescriptor EnvironmentDescriptor.xsd"
    xmlns="PGECEnvironmentDescriptor">
    <Module>
        <Name>PGECnative.dll</Name>
        <Size>34304</Size>
        <Hash type="SHA1">0CE2A1C4991ECCF572764EBF425673F63809B625</Hash>
        <BaseAddress>268435456</BaseAddress>
        <EntryPoint>268446646</EntryPoint>
        <MemorySize>49152</MemorySize>
    </Module>
    <Module>
        <Name>privacymanager.netmodule</Name>
        <Size>16384</Size>
        <Hash type="SHA1">2A017F15A11743EFFDCAD197AD5DA707C12C7564</Hash>
        <BaseAddress>335675392</BaseAddress>
        <EntryPoint>0</EntryPoint>
        <MemorySize>24576</MemorySize>
    </Module>
    <Module canBeMain="true">
        <!--Java 1.6.0_10-rc2-b32-->
        <Name>javaw.exe</Name>
        <Size>144792</Size>
        <Hash type="SHA1">C62DB7635C120D87B790D73CE6ED40DB7FEE6BE3</Hash>
        <BaseAddress>4194304</BaseAddress>
        <EntryPoint>4228764</EntryPoint>
        <MemorySize>147456</MemorySize>
    </Module>
    <Module>
```

5.1.5 API for Data Access

Besides all the mentioned limitations and restrictions to services running inside the PGEC, these services shall gain access to private data in order to provide useful service. An API is provided for controlled access and limited manipulation, see Listing 5.15.

Listing 5.15: Interface implemented by the `PrivacyManager` to enable controlled access of service inside the PGEC to private data of users. The actual implementation asserts that this interface is accessible only from inner execution environments.

```
11 public interface PrivacyAwareAccess {
12
13     /**
14      * Requests private data from any of the known and connected containers.
15      * The first response from a container instance is taken as the actual value
16      *  of that private data.
17      * @param name
18      * @return
19      */
20     public Object getPrivateData(DataItemID name);
21
22     public void persistPrivateData(DataItemID name);
23
24     public void sendLiteral(ContainerID containerID, String literal);
```

The most important method `getPrivateData()` allows to retrieve certain personal data. The data itself is untyped to support maximum flexibility in the kinds of data. A data item is identified by an owner, an item name and an instance. The instance number allows to provide multiple versions or instances of a particular data, e.g., multiple private addresses or delivery addresses. That way it is even possible for users to act with fake identities, e.g., to provide different birth dates. After rejection of requests from outer execution environments (line 1771 of Listing 5.16), the implementation accesses locally available information (line 1772) or transparently retrieves the information from remote container instances (lines 1778-1780). This lets the distributed PGEC appear as one logical unit to services within.

Actual access to particular data from certain service or user of a particular service respectively is checked. Data is provided separately for each execution environment, which correlates to a particular service, or globally for all environments. If the data item is set, it is implicitly granted for that service.

Since services are not permitted to have access to system resources like file system, the data access API provides a method (`persistPrivateData()`) to persist private data under control of the PGEC (line 22 of Listing 5.15). The implementation of that method creates a random key and stores the denoted data item in the local file system of the current PGEC instance. Thereby the content and the data item identifier are encrypted. Hence, from the file system can neither the data itself be determined nor which data was persisted. The created key is stored and managed at the data owner's PGEC instance. The data item retrieval method (line 20 of Listing 5.15) allows also transparent access to persisted versions of data items if no volatile versions can be found.

Finally, this API provides a method (`sendLiteral()`) to send literals (line 24 of List-

Listing 5.16: Code section implementing one of the methods of the `PrivacyAwareAccess` API. This implementation hides the actual implementations of data access with regard to the source of the data, i.e., local or remote, and with regard to the storage, i.e., in volatile memory or persistent. Thereby, a transparency of data access is achieved, which is required for the logical unity of the distributed PGEC.

```
1758     @Override
1759     public Object getPrivateData(DataItemID name) throws PrivacyException {
1760         Object result = getPrivateData(name, true); // try to find an update
1761         // value in the outside
1762         // world
1763         if (result == null) {
1764             // use a persisted possibly outdated version
1765             result = getPrivateData(name, false);
1766         }
1767         return result;
1768     }
1769
1770     private Object getPrivateData(DataItemID name, boolean onlyFromOutside) {
1771         if (isInner()) {
1772             Object result = getPrivateDataLocally(name, onlyFromOutside);
1773             if (result != null) {
1774                 return result;
1775             } else {
1776                 // TODO search more intelligent
1777                 // use environmentID to determine the containerIDs
1778                 for (ContainerID containerID : knownRemoteContainers) {
1779                     result = getPrivateDataRemotely(containerID, name, null,
1780                             null, onlyFromOutside);
1781                     if (result != null) {
1782                         return result;
1783                     }
1784                 }
1785                 return null;
1786             }
1787         } else {
1788             throw new PrivacyException(name
1789                     + " may not be accessed outside the privacy container");
1790         }
1791     }
```

ing 5.15). The literal is given as a string, which can easily be compared to check, whether the particular literal is permitted. Only PGEC instances can be addressees of the literal. If the addressee is the local PGEC instance, the literal is piped to the outside of the container. Otherwise, the literal send request is forwarded to the actual addressee via the mentioned encrypted channel between the PGEC instances. This behaviour can be found in Listing 5.17.

Key Management

At various places in this work the negotiation and use of privacy contracts have been mentioned. In favor for the implementation of the actual protection features, a less complex structure to explicitly grant permissions to access data and to display GUI has been implemented in the existing prototype of the PGEC. It is sufficient to prove the concept of the built-in protection means, but lacks dynamic for productive use. The structure allows to specify the access of users, represented by their container instances, to data items depending on the service they use. A triple of container instance (user), execution

Listing 5.17: Code section implementing the method of the `PrivacyAwareAccess` API that enables to send negotiated literals, as may be required by logically delivering services. Also here the logical unity of the distributed PGECis created by hidden forwarding of literals to remote instances if necessary.

```
2071    @Override
2072    public void sendLiteral(ContainerID containerID, String literal)
2073            throws PrivacyException {
2074        if (isInner()) {
2075            checkLiteralPermission(containerID, literal);
2076            if (containerID == null) {
2077                sendLiteralLocally(literal);
2078            } else {
2079                sendLiteralRemotely(containerID, literal);
2080            }
2081        } else {
2082            throw new PrivacyException("instances running outside the "
2083                    + "container should cope with their literals on their own");
2084        }
2085    }
2086
2087    private void sendLiteralLocally(String literal) {
2088        try {
2089            this.literalOutletSnd.write(literal);
2090        } catch (IOException e) {
2091            // TODO Auto-generated catch block
2092            e.printStackTrace();
2093        }
2094    }
2095
2096    private void sendLiteralRemotely(ContainerID containerID, String literal)
2097            throws PrivacyException {
```

environment (service) and data item that is contained in the relation *ReadPermission* permits the particular data item being read by the service running in the stated execution environment on behalf of the user, which started the particular container instance. This relation and the related entities are represented in the Entity-Relation-Model depicted in Figure 5.4. The mentioned relation is additionally attributed with a flag, which represents whether the permission is valid also for access to persisted versions of the data item. Each persisted data item is encrypted by a key, which is also stored in that structure. Finally, there is a relation between container instances and execution environments (services) representing the permission to display a GUI. The permission relations are not maintained at single instance. The *ReadPermissions* are distributed with regard to the owner of the stated data items. *GUIPermissions* may even occur repeatedly in multiple container instances. That is, the display of GUIs is independently permitted by each user of services, locally at her/his respective container instance. The effective permissions are calculated as the intersection of the *GUIPermission* relations of all users. Requests for permission can be forwarded through the secured channels between the PGEC instances. On the same way, persistence keys, which are maintained at the container instance of the owner of the particular data, can be retrieved from a remote permission structure.
When it is considered to persist the granted permissions and the keys assigned to persisted data items, a standard relational database is not feasible. Keys could be read, or permission entries could be manipulated or injected. The complete permission structure must be stored in an encrypted way to ensure its security and integrity. The permission

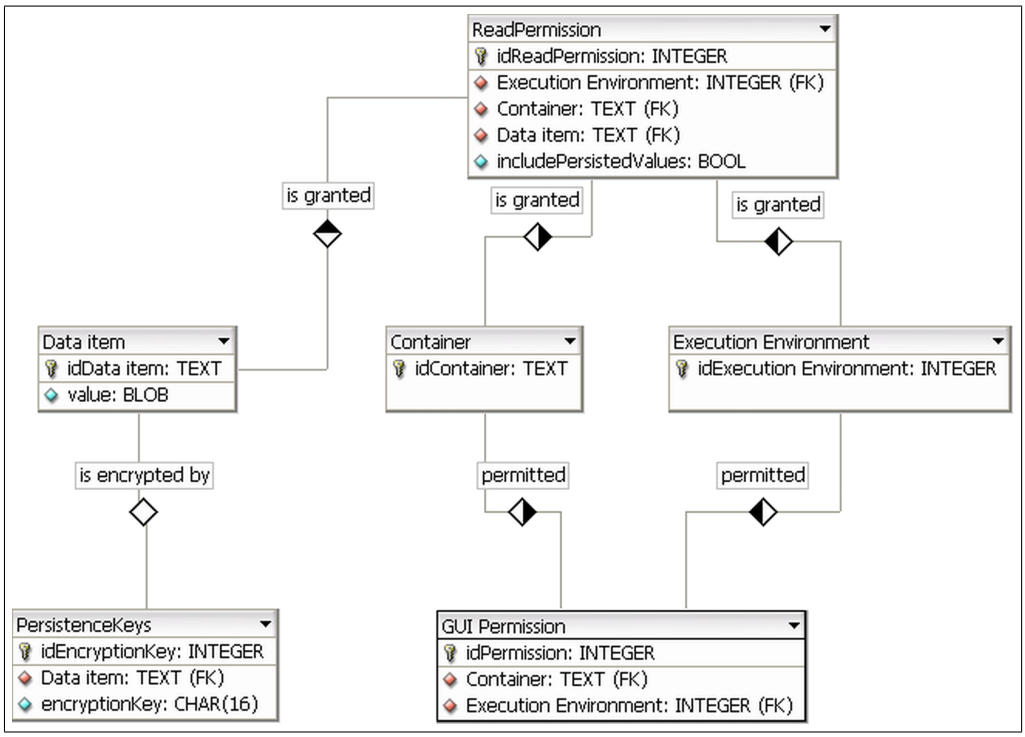


Figure 5.4: Entity-Relationship-Model of the permission structure that is implemented in the PGEC.

structure implemented in the prototype (Listing 5.18) uses mainly cascaded hash tables and does not support persistence of the permissions and keys.

Listing 5.18: The relations modeled in Figure 5.4 are implemented as nested `HashMaps` and `HashSets`. Appropriate methods to set and delete entries from the relations are implemented, but omitted in this code snippet.

```
18 public class PermissionStructure {
19
20     private HashMap<ContainerID, HashSet<ExecutionEnvironmentID>> whereGUIisAllowed = new
           HashMap<ContainerID, HashSet<ExecutionEnvironmentID>>();
21
22     private HashMap<DataItemID, byte[]> persistenceKeys = new HashMap<DataItemID, byte[]>()
           ;
23
24     private HashMap<ContainerID, HashMap<ExecutionEnvironmentID, HashMap<DataItemID,
           Boolean>>> permits = new HashMap<ContainerID, HashMap<ExecutionEnvironmentID,
           HashMap<DataItemID, Boolean>>>();
```

5.1.6 Mutual Authentication of Distributed PGEC Instances

The distributed architecture of the PGEC (refer to Section 4.2.2) requires encrypted interconnection between the container instances. Besides assertion of the integrity of the local PGEC instance and its host runtime environment, the instances have to mutually authenticate. Otherwise, a PGEC instance might send private information to an intruder masquerading as a PGEC instance. Even when the communication channel was encrypted, the intruder without the PGEC's protection means can misuse the private data.
The current prototype implementation uses a PSK authentication. The symmetric key is hard-coded in the `PrivacyManager`and also used for the encryption of the communication between the container instances. Distributed services inside the PGEC are provided with managed sockets to allow direct communication between their components. Though currently un-encrypted, they could also use this PSK. From IPsec it can be learned, that dynamically created key exchanged via the Diffie-Hellman algorithm [52] appears to be more reasonable.
PSKs provide no appropriate means for revocation if all instances share the same key. AACS uses a hierarchy of PSKs, which enables certain nodes or subtree to be revoked. Thereby, series of Blu-Ray players can be disabled from decrypting newer contents. While the PGEC is likely not to exist from multiple providers or in various series, this type of revocation appears not feasible. A promising approach is to provide each PGEC instance with an individually signed certificate. Such certificate can be revoked in revocation lists. Using these certificates the instances can use the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) protocol to mutually authenticate and at the same time exchange a symmetric key for encryption of the mentioned communication channels between container instances and service components within. While in Java the SSL sockets are specializations of sockets the PGEC can easily plant such SSL socket to the service components without notice. The underlying encryption cipher to be used can be specified upon creation of the SSL socket to ensure appropriate encryption strength.

5.2 Test Attacks and Effects of Counter Measures

In this section it is described how the implemented test attacks are executed. For each addressed attack (refer to the Tables 4.1 and 4.2) the involved attacking classes are introduced. Listings of those classes are kept at a minimum or even waived. Depending on the nature of the attack, the attacking classes may be started under control of the PGEC, in either an inner or outer execution environment or both. Some attacks are even executed out of control of the PGEC. That is, such classes are started directly as an application. The expected results for the case of a successful attack is described as well as the result of unsuccessful attacks. In the latter case, exceptions are mostly thrown. Usually exceptions in inner execution environments are suppressed. As mentioned they might be logged. The logging here includes printing of the exception with its message to

the error stream. According to the counter measure against attack #16, this error stream is redirected. For debugging purpose, it is redirected into a file instead of the NUL device. Since the exception and its message are of interest, the stack traces of the thrown exceptions are omitted for the most of the test attacks.
The existing prototype of the PGEC uses a startup configuration file. It contains descriptions for each execution environment to be opened in the container. An execution environment has a name, which correlates to the provided service therein. This identifier is also be used for granting access rights to private data. Hence, in a productive version of PGEC, this identifier must be trusted, e.g., being signed by a trusted entity. The environment description contains an arbitrary number of classes that shall be started within that environment. Each of those classes is started in an own thread. To enable starting of a class, this must be either an implementation of `java.lang.Runnable` or must contain a main method of signature `public static void main(String... args)`. For the latter, the arguments can be specified in the configuration file as well. Additionally, inner execution environments can be provided with the description of a client class. That class is not instantiated an started, but can be retrieved by remote PGEC instances to be started there in a newly opened inner execution environment. It is expected that this is a GUI implementation for the particular service. Listing 5.19 shows the top of the startup configuration file, that is used for the test attacks as well as for the introduced scenario from Section 1.2.4. To run each attack separately the respective blocks have to be uncommented.

Listing 5.19: Configuration file for startup of PGEC.

```
<?xml version="1.0" encoding="utf-8"?>
<PGECconfig xmlns="http://tempuri.org/PGECconfigSchema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="PGECconfigSchema.xsd">
  <OUTEREnvironment name="attack1">
    <CLASS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="MainClassType">
      <NAME>com.endosoft.privacyattack.ReplaceSecurityManager
      </NAME>
      <ARGS />
    </CLASS>
  </OUTEREnvironment>
  <INNEREnvironment name="attack1">
    <CLASS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="MainClassType">
      <NAME>com.endosoft.privacyattack.ReplaceSecurityManager
      </NAME>
      <ARGS />
    </CLASS>
  </INNEREnvironment>
  <!--OUTEREnvironment name="attack2and3">
    <CLASS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="MainClassType">
      <NAME>com.endosoft.privacyattack.FileAccess
      </NAME>
      <ARGS />
    </CLASS>
  </OUTEREnvironment>
  <INNEREnvironment name="attack2and3">
    <CLASS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="MainClassType">
      <NAME>com.endosoft.privacyattack.FileAccess
      </NAME>
      <ARGS />
    </CLASS>
  </INNEREnvironment-->
```

5.2.1 Test Regular Attacks

Attack #1 is implemented by class `com.endosoft.privacyattack.ReplaceSecurityManager`. It should be started in an outer or an inner execution environment. The attacking code is able to determine the currently installed `SecurityManager`, which should be the `PrivacyManager`. Afterwards it is replacing the current `SecurityManager` by `System.setSecurityManager(new RMISecurityManager());`.
If the attacking code is executed directly without the PGEC, an output similar to the following is expected.

```
currentSecMgr: null
replacedSecMgr: java.rmi.RMISecurityManager@42e816
```

Since replacement of the `SecurityManager` must not be allowed by any class, in both cases a `PrivacyException` is thrown. The exceptions are caught and printed to the error stream. This error stream can be seen on the console for outer execution environments or is redirected to the above mentioned log file.

```
currentSecMgr: com.endosoft.pgec.PrivacyManager@167d940
DEBUG: Thread[Outer Main Thread (com.endosoft.privacyattack.↩
↪ReplaceSecurityManager),5,Outer Execution Environment Threads (attack1)] com↩
↪.endosoft.pgec.PrivacyException: no class may create a securitymanager
java.lang.RuntimeException: com.endosoft.pgec.PrivacyException: no class may ↩
↪create a securitymanager
    at com.endosoft.pgec.ExecutionEnvironment$MainRunner.run(↩
    ↪ExecutionEnvironment.java:289)
    at java.lang.Thread.run(Unknown Source)
Caused by: com.endosoft.pgec.PrivacyException: no class may create a ↩
↪securitymanager
    at com.endosoft.pgec.PrivacyManager.checkPermission(PrivacyManager.↩
    ↪java:850)
    at java.lang.SecurityManager.<init>(Unknown Source)
    at java.rmi.RMISecurityManager.<init>(Unknown Source)
    at com.endosoft.privacyattack.ReplaceSecurityManager.main(↩
    ↪ReplaceSecurityManager.java:21)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at com.endosoft.pgec.ExecutionEnvironment$MainRunner.run(↩
    ↪ExecutionEnvironment.java:286)
    ... 1 more
```

Similarly, the content of the log file appears like this.

```
currentSecMgr: com.endosoft.pgec.PrivacyManager@167d940
Exception raised but don't tell anything
DEBUG: Thread[Inner Main Thread (com.endosoft.privacyattack.↩
↪ReplaceSecurityManager),5,Inner Execution Environment Threads (attack1)] com↩
↪.endosoft.pgec.PrivacyException: no class may create a securitymanager
```

Attacks #2 and #3 are implemented by class `com.endosoft.privacyattack.FileAccess`. It should be started in an outer or an

inner execution environment. Since services in outer execution environments are allowed to read and write files, this reflects the successful case of the attack. At first, it reads an obviously existing file, i.e., `C:\boot.ini` on Windows systems. The content of this file is printed to the system output stream. The next step is writing a file named `testfile.Outer.txt`. The content of the file becomes

```
greetings from com.endosoft.pgec.OuterExecutionEnvironment@dd20f6
```

When run in an inner execution environment, even reading of files is prohibited. Hence, the reading fails with a `PrivacyException`.

```
com.endosoft.pgec.PrivacyException: inner classes have no permission to read a↩
↪ file
    at com.endosoft.pgec.PrivacyManager.checkPermission(PrivacyManager.↩
    ↪java:788)
    at com.endosoft.pgec.PrivacyManager.checkPermission(PrivacyManager.↩
    ↪java:842)
    at java.lang.SecurityManager.checkRead(Unknown Source)
    at java.io.FileInputStream.<init>(Unknown Source)
    at java.io.FileReader.<init>(Unknown Source)
    at com.endosoft.privacyattack.FileAccess.<init>(FileAccess.java:24)
    at com.endosoft.privacyattack.FileAccess.main(FileAccess.java:59)
    ...
```

The following attempt to write file `testfile.Inner.txt` will also fail with a `PrivacyException`. Because the opening of a file for writing includes test on existence of the file, which requires a read permission to the file system, the message in the thrown `PrivacyException` complains about missing read permission instead of missing write permission.

```
com.endosoft.pgec.PrivacyException: inner classes have no permission to read a↩
↪ file
    at com.endosoft.pgec.PrivacyManager.checkPermission(PrivacyManager.↩
    ↪java:788)
    at com.endosoft.pgec.PrivacyManager.checkPermission(PrivacyManager.↩
    ↪java:842)
    at java.lang.SecurityManager.checkRead(Unknown Source)
    ...
    at java.io.FileWriter.<init>(Unknown Source)
    at com.endosoft.privacyattack.FileAccess.<init>(FileAccess.java:45)
    at com.endosoft.privacyattack.FileAccess.main(FileAccess.java:59)
    ...
```

If the PGEC would grant read permissions, e.g., selected files and folders, to services in inner execution environments, the attack would still fail, with a `PrivacyException`.

```
com.endosoft.pgec.PrivacyException: inner classes have no permission to write ↩
↪a file
    ...
```

Attacks #4 and #5 are implemented by class `com.endosoft.privacyattack.OpenNetworkConnection`. It should be started in an outer or an inner execution environment. Services in outer execution environments

shall have as little limitations as possible. Hence, they are allowed to open sockets to initiate or accept network connections. A successful attack would produce similar results as when run within an outer execution environment. That is, the attack implementation attempts to open a socket with initiating a Transmission Control Protocol (TCP) connection to a local listener on port 80. The next attempt is to open a User Datagram Protocol (UDP) socket, capable of sending and receiving packets. Finally, a TCP server socket listening on port 2048 and accepting network connections shall be created.

```
TCP Socket successfully created:Socket[addr=localhost/127.0.0.1,port=80,↩
↪localport=2291]
UDP Socket successfully created:java.net.DatagramSocket@8814e9
ServerSocket successfully created:ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,↩
↪localport=2048]
```

All three attempts from inside an inner execution environment will be rejected with a `PrivacyException`. In the special case of UDP sockets even the loading of the socket implementation will fail, due to the prohibition of reading the system property `impl.prefix`.

```
com.endosoft.pgec.PrivacyException: inner classes may not open sockets
...
Can't find class: java.net.nullDatagramSocketImpl: check impl.prefix property
com.endosoft.pgec.PrivacyException: inner classes may not open sockets
...
com.endosoft.pgec.PrivacyException: inner classes may not open sockets
...
```

Attacks #6 and #16 are implemented by class `com.endosoft.privacyattack.SystemStreams`. It should be started in an outer or an inner execution environment. Actually there is only one system stream for outputs and one for errors. The PGEC integrates a filter in these streams to support differentiation between inner and outer execution environments. That is, data put into the streams from outer environments may still pass as usual, while data from inner execution environments are redirected to the NUL device. In this prototype for display and for debugging, they are redirected into a file. In this file the suppressed output and printed stack traces of thrown exceptions can be read. To prevent circumvention of the installed filter, system streams may not be changed neither from inner nor from outer execution environments. The implemented attack prints to both system streams first. Then it creates two files `outputstream.txt` and `errorstream.txt` to which it redirects the respective system streams. A successful execution results in the following outputs. On the console:

```
If you can read this I have access to the OUTPUT stream
If you can read this I have access to the ERROR stream
```

In a file `outputstream.txt`:

```
Written to redirected OUTPUT stream
```

Finally, in a file `errorstream.txt`:

```
Written to redirected ERROR stream
```

When executed in an outer execution environment on the console can be read:

```
If you can read this I have access to the ERROR stream
If you can read this I have access to the OUTPUT stream
com.endosoft.pgec.PrivacyException: System streams may not be changed
...
Written to redirected OUTPUT stream
com.endosoft.pgec.PrivacyException: System streams may not be changed
...
Written to redirected ERROR stream
```

The mentioned files are created but remain empty. Inner execution environments redirect all system streams into a log file, prohibit file creation and stream redirection. Thus, the console remains empty, the mentioned redirection files are not created and the log file contains the following:

```
If you can read this I have access to the OUTPUT stream
If you can read this I have access to the ERROR stream
com.endosoft.pgec.PrivacyException: inner classes have no permission to write ↩
↪a file
...
com.endosoft.pgec.PrivacyException: System streams may not be changed
...
Written to redirected OUTPUT stream
com.endosoft.pgec.PrivacyException: System streams may not be changed
...
Written to redirected ERROR stream
```

Attack #7 is implemented by classes `com.endosoft.privacyattack.PrintSomething` and `com.endosoft.privacyattack.PrintSomething2`. Java provide two separate APIs for printing. The two attack implementations use these APIs respectively. Both should be started in an outer or an inner execution environment. While printing is permitted in outer execution environments, an execution in there represents the successful case. The result of `PrintSomething` will be a page with the content of Figure 5.5. The other implementation will print a picture from a given file or input stream.
Both approaches to print require a `RuntimePermission("queuePrintJob")`, which is not granted to applications and services in inner execution environments. Hence, the execution of the attack implementations will fail there with the following exception.

```
com.endosoft.pgec.PrivacyException: inner classes may not print
...
```

Attack #8 is implemented by class `com.endosoft.privacyattack.CallNative`. It should be started in an outer or an inner execution environment. The attacking code attempts to load a native library by `System.loadLibrary("dummy");`. Loading a native library is essential for any call to

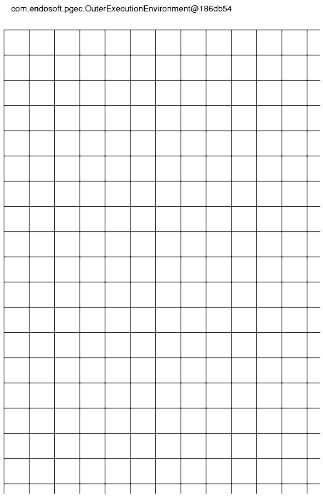


Figure 5.5: Successful print result of `PrintSomething`.

native code. To prevent calls to native code the loading of native libraries is prevented to all execution environments. Note, some libraries, i.e., `awt, fontmanager, dcpr, net, nio, jsound` and `jsoundds`, are explicitly permitted. These are required for displaying GUIs, printing and communication with additional permission. The PGEC has to ensure that the libraries of those names to be loaded are exactly the one originally provided with an approved JVM. Successful loading of the library does not produce any output. Unsuccessful library loading (failed attack) is responded with a thrown exception, regardless of the execution environment in which the attack is started.

```
com.endosoft.pgec.PrivacyException: no class may load native code
...
```

Attack #9 is implemented by class `com.endosoft.privacyattack.AccessPrivateFieldsMethodsConstructors`. It should be started in an outer or an inner execution environment. Four attack attempts are implemented. These are access to a static private field, to a private field, a private method and to the private constructor. A successful attack would present four (return) values.

```
permittedNativeLibraries=[Ljava.lang.String;@1270b73
localPort=12345
isPrivileged()=true
new PrivacyManager()=com.endosoft.pgec.PrivacyManager@e7b241
```

Even the direct execution of the attack implementation will not be fully successful. For the non-static fields and methods a reference to an instance of the accessed class is required. In case of the `PrivacyManager`, this must be instantiated first. In Section 5.1.1, conditions are described for instantiation of the `PrivacyManager`. These conditions are not fulfilled when running the attacking code directly. Hence, it will not gain a

PrivacyManager instance and thus cannot access the non-static entities. Similar is true for the access to the private constructor, which returns no instance due to the adherence to the same conditions.

```
permittedNativeLibraries=[Ljava.lang.String;@1270b73
...
com.endosoft.pgec.PrivacyException: only com.endosoft.pgec.PrivacyManager may ↩
↪instantiate itself
...
com.endosoft.pgec.PrivacyException: only com.endosoft.pgec.PrivacyManager may ↩
↪instantiate itself
...
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown ↩
    ↪Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at com.endosoft.privacyattack.AccessPrivateFieldsMethodsConstructors.main(↩
    ↪AccessPrivateFieldsMethodsConstructors.java:72)
Caused by: com.endosoft.pgec.PrivacyException: PrivacyManager must be ↩
↪instantiated from its own main method (stacktrace length = 3)
    at com.endosoft.pgec.PrivacyManager.<init>(PrivacyManager.java:138)
    ... 5 more
```

Inside PGEC's execution environments a ReflectPermission is checked. This permission is not granted to services inside the execution environments, but without this permission the Java language access control cannot be bypassed. Thus, private entities can be detected by reflection, but not be accessed. This results in four PrivacyExceptions.

```
com.endosoft.pgec.PrivacyException: reflection (suppressAccessChecks) is not ↩
↪allowed in connection with the PGEC
```

Attack #10 is implemented by class com.endosoft.privacyattack.ReadWriteSystemProperties. It should be started in an outer or an inner execution environment. In the first step it enumerates all available system properties and outputs them with their values. In the second step the system property user.name is replaced with "Bob". Finally, the available environment variables are enumerated and printed with their values. The following shows an excerpt of the printed results.

```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Programme\Java\jre6\bin
java.vm.version=11.0-b15
...
user.name=maaser
...
--- Changed system property ---
user.name=Bob
--- Environment variables ---
...
PROCESSOR_IDENTIFIER=x86 Family 6 Model 14 Stepping 8, GenuineIntel
SESSIONNAME=Console
...
```

While in inner execution environments, environment variables and system properties may neither be read nor written, the system properties may be read and even written in outer execution environments. That is, when the implemented test attack is run in an outer execution environment only the retrieval of environment variables is prohibited an throws an according `PrivacyException`.

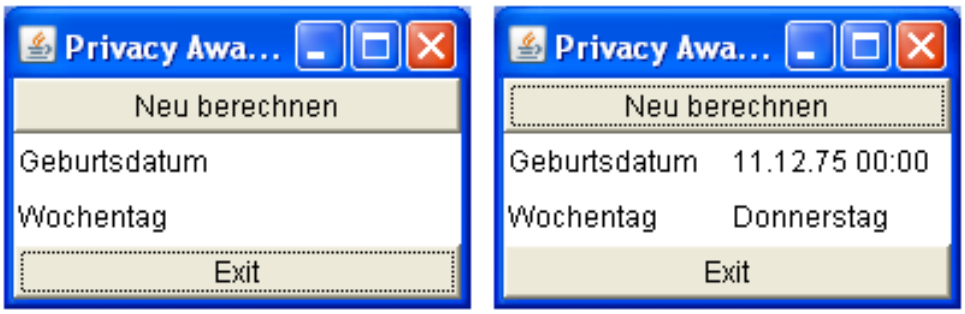
```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Programme\Java\jre6\bin
java.vm.version=11.0-b15
...
user.name=maaser
...
--- Changed system property ---
user.name=Bob
--- Environment variables ---
com.endosoft.pgec.PrivacyException: no class may read environment variables
...
```

In inner execution the test attack implementation has no success at all and for all three attempts a `PrivacyException` is thrown.

```
com.endosoft.pgec.PrivacyException: should inner classes be allowed to read ↩
↪system property * ?
...
--- Changed system property ---
com.endosoft.pgec.PrivacyException: inner classes may not write system ↩
↪properties
...
--- Environment variables ---
com.endosoft.pgec.PrivacyException: no class may read environment variables
...
```

Attack #11 is implemented by class `com.endosoft.privacyattack.ShowGUI`. It should be started in an outer or an inner execution environment. Since services in outer execution environment may arbitrary open GUIs, this will reflect the successful execution of the attack implementation. It opens a window containing a functional button (see Figure 5.6a). When this button is clicked a trivial program logic is executed and fills two text labels with some text (see Figure 5.6b). Services in inner execution environment may only open GUIs if explicitly permitted, which is not done for the `ShowGUI` attack code. Thus, the window cannot be opened but rather a `PrivacyException` is thrown, causing the respective inner execution environment not to start.

```
DEBUG: Thread[main,5,main] Environment could not be started, due to
com.endosoft.pgec.PrivacyException: Environment could not be started, due to
...
Caused by: com.endosoft.pgec.PrivacyException: inner classes in this execution↩
↪ environment may not open GUI Frames in this container
...
```



(a) Before button click (b) After button click

Figure 5.6: The GUI that is produced by the test attack implementation.

Attack #12 is implemented by class `com.endosoft.privacyattack.DatabaseConnect`. It should be started in an outer or an inner execution environment. The implementation waives actual reading from or writing to the database, but simply connects to it. Hence, a successful run as in an outer execution environment results in the following output.

```
JDBC Connect Example.
Connected to the database
Disconnected from database
```

The existing implementation uses SQLite[2], which relies on pure file access. Since file access is prohibited from inner execution environments, the database connection fails with a `PrivacyException` referring to file read prohibition.

```
JDBC Connect Example.
com.endosoft.pgec.PrivacyException: inner classes have no permission to read a↩
↪ file
...
```

When using other database connectors that rely on network connections, e.g., MySQL[3], the connection will fail similarly with a `PrivacyException` regarding network prohibition.

Attack #13 is implemented by class `com.endosoft.privacyattack.AccessUninstalledPrivacyManager`. It should be started directly without the `PrivacyManager`. If started within an execution environment, the reference to the `PrivacyManager` can easily be retrieved and the success of reading the private data “birthdate” depends on the execution environment and the granted access permissions. These control mechanisms are not of interest here. A direct start will show that currently no `SecurityManager` is installed, that may hinder

[2] http://www.zentus.com/sqlitejdbc/
[3] http://dev.mysql.com/downloads/connector/j/3.1.html

access to some private data. But to access the non-static method `getPrivateData()`, an instance of `PrivacyManager` has to be retrieved. According to the conditions known from Section 5.1.1, the `PrivacyManager` will not instantiate here. The attack results in a `PrivacyException`.

```
currentSecMgr: null
com.endosoft.pgec.PrivacyException: only com.endosoft.pgec.PrivacyManager may ↩
↪instantiate itself
    at com.endosoft.pgec.PrivacyManager.<init>(PrivacyManager.java:145)
    at com.endosoft.pgec.PrivacyManager.getPrivacyManager(PrivacyManager.↩
    ↪java:207)
    at com.endosoft.privacyattack.AccessUninstalledPrivacyManager.main(↩
    ↪AccessUninstalledPrivacyManager.java:17)
```

Attack #14 is implemented by classes `com.endosoft.privacyattack.SharedMemory` and `com.endosoft.privacyattack.FindOtherExecutionEnvironments`. `SharedMemory` should be started in an outer and an inner execution environment simultaneously. This class provides a static field, that is supposed to be used as a shared memory. The instance in the inner execution environment assigns it with some possibly private information, while the instance in the outer execution environment expects to read this information from the field. In case of a successful attack, the outer instance reads and outputs the content written by the inner instance.

```
I learned the following: This is a secret, please keep it privately.
```

In fact, due to the different `ClassLoader`s, the respectively loaded classes are not identical. Hence, also the static field differs between the execution environments. That is, the outer instance will read the initialized value:

```
I learned the following: something I already knew.
```

While the different `ClassLoader`s can ensure, that static fields and methods are not accessible across the borders of execution environments, the attack implemented in `FindOtherExecutionEnvironments` attempts to gain access to the other environments `ClassLoader`s. This might enable an attacker to get hold of references to classes or objects within other execution environments and to use these to share information. The references to sibling `ClassLoader`s are known only the `PrivacyManager`, which will not release this information (see attack #9). The second source of these references is the parent `ClassLoader`, which loaded the classes of the execution environments' `ClassLoader`s. Just this is attempted by `FindOtherExecutionEnvironments`. A successful execution, i.e., a direct start, would print the parent `ClassLoader`:

```
sun.misc.Launcher$ExtClassLoader@35ce36
```

When started in an outer or an inner execution environment an according `PrivacyException` is thrown instead.

```
com.endosoft.pgec.PrivacyException: classes in execution environments may not ↩
↪retrieve other classloaders than their own executionenvironment
...
```

Attack #15 is implemented by class `com.endosoft.privacyattack.ProduceAudio`. It should be started in an outer or an inner execution environment. Outer execution environments do not limit the access to audio device and hence, execution here will be successful. The first attempt to access the audio system is through sampled audio. A down-sampled (8kHz, 8bit mono) sequence of 0.5 seconds of the `notify.wav` that ships with windows is played. In the second attempt a MIDI device is used to play this gamut (Figure 5.7). Executed in an

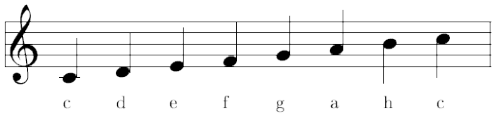


Figure 5.7: The gamut to be played on a MIDI device by the attacking code.

inner execution environment the access to audio devices will be suppressed. Besides the logged information of the `InnerExecutionEnvironment ClassLoader`

```
one should not load javax.sound.sampled.AudioSystem within PGEC javax.sound.↩
↪sampled.AudioSystem
one should not load javax.sound.midi.MidiSystem within PGEC javax.sound.midi.↩
↪MidiSystem
```

the attacking code will fail in retrieving the required audio devices. The redirected system streams of the inner execution environment read like this:

```
available audio mixer devices:0
javax.sound.sampled.UnsupportedAudioFileException: could not get audio input ↩
↪stream from input stream
...
available MIDI devices:0
javax.sound.midi.MidiUnavailableException: No MIDI available inside the PGEC
...
```

Attack #17 This attack does not require a special implementation. During the development process of the PGEC, debugging capabilities are indispensable. Therefore, the environment descriptor (Listing 5.14) contains descriptors for libraries, which are loaded during debugging only, to permit the PGEC to run in a debugger environment, too. When this environment descriptor is changed back to the version without debugging libraries, the use of a debugger will prevent the PGEC from instantiating. The embedded native code from Listing A.1 detects the running and not permitted library `jdwp.dll`, which cause a respective `PrivacyException`.

```
this process module [jdwp.dll] is not allowed
               its File name is: C:\Programme\Java\jre6\bin\jdwp.dll
               its File size is: 167936
               its base address is: 1832321024
               its Entry point address is: 1832439948
               its MemorySize is: 167936
Exception in thread "main" com.endosoft.pgec.PrivacyException: there are ↩
↪modules loaded that are not allowed, deprecated, manipulated
    at com.endosoft.pgec.PrivacyManager.<init>(PrivacyManager.java:131)
    at com.endosoft.pgec.PrivacyManager.getPrivacyManager(PrivacyManager.↩
    ↪java:207)
    at com.endosoft.pgec.PrivacyManager.main(PrivacyManager.java:668)
```

5.2.2 Test Covert Channel Attacks

Attack #18 is implemented by class `com.endosoft.privacyattack.ThrowException`. It should be started in an outer or an inner execution environment. The start in an outer execution environment matches the successful case. That is, an arbitrary `Exception` can be thrown and, if not caught, causes the application to stop and print the according stack trace. To ensure undisturbed execution in other execution environments, also the outer execution environments catch uncaught exceptions, to gracefully shut down this particular environment.

```
DEBUG: Thread[Outer Main Thread (com.endosoft.privacyattack.ThrowException),5,↩
↪Outer Execution Environment Threads (attack18)] java.lang.Exception: main():↩
↪ greetings from com.endosoft.pgec.OuterExecutionEnvironment@8965fb
java.lang.RuntimeException: java.lang.Exception: main(): greetings from com.↩
↪endosoft.pgec.OuterExecutionEnvironment@8965fb
    at com.endosoft.pgec.ExecutionEnvironment$MainRunner.run(↩
    ↪ExecutionEnvironment.java:289)
    at java.lang.Thread.run(Unknown Source)
Caused by: java.lang.Exception: main(): greetings from com.endosoft.pgec.↩
↪OuterExecutionEnvironment@8965fb
    at com.endosoft.privacyattack.ThrowException.main(ThrowException.java:18)
...
```

In inner execution environments, the `Exception` is caught as well and suppressed. For the aforementioned debugging reasons, the `Exception` is logged to a file. The content of that log file reads as follows:

```
Exception raised but don't tell anything
DEBUG: Thread[Thread-3,5,Inner Execution Environment Threads (attack18)] run()↩
↪: greetings from com.endosoft.pgec.InnerExecutionEnvironment@1f9dc36
```

Attack #19 is implemented by classes `com.endosoft.privacyattack.EnumerateThreadsTraitor` and `com.endosoft.privacyattack.EnumerateThreads`. They should be started in an inner and an outer execution environment accordingly. The `EnumerateThreadsTraitor` spawns a thread named `private data from sun.misc.Launcher$AppClassLoader@fabe9`.

`EnumerateThreads` enumerates all running threads in a thread group starting with the group of the current thread and traversing along the parents to the root thread group. If both classes run side by side in the same JVM without the protection means of the PGEC it would determine something like the following.

```
myThread = main
myThreadGroup = main
Thread [0]:main in Group: main
Thread [1]:Thread-0 in Group: main
Thread [2]:private data from sun.misc.Launcher$AppClassLoader@11b86e7 in ↩
↪Group: main
 parent thread group
Thread [0]:Reference Handler in Group: system
Thread [1]:Finalizer in Group: system
Thread [2]:Signal Dispatcher in Group: system
Thread [3]:Attach Listener in Group: system
Thread [4]:main in Group: main
Thread [5]:Thread-0 in Group: main
Thread [6]:private data from sun.misc.Launcher$AppClassLoader@11b86e7 in ↩
↪Group: main
```

Please note the respectively last enumerated threads in the groups. These represent the thread with the possibly private information. Inside the PGEC, the following threads are running.

```
main
locking boxes
AWT-Shutdown
AWT-Windows
AWT-EventQueue-0
Outer Main Thread (com.endosoft.privacyattack.EnumerateThreads)
Inner Main Thread (com.endosoft.privacyattack.EnumerateThreadsTraitor)
PrivacyManager
private data from com.endosoft.pgec.InnerExecutionEnvironment@91f005
```

The PGEC's protection means prevent services in execution environments from accessing the thread group that is the parent of {`Outer|Inner`} `Execution Environment Threads`. An attempt to access this parental thread group provokes a `PrivacyException`.

```
myThread = Outer Main Thread (com.endosoft.privacyattack.EnumerateThreads)
myThreadGroup = Outer Execution Environment Threads (attack19)
Thread [0]:Outer Main Thread (com.endosoft.privacyattack.EnumerateThreads) in ↩
↪Group: Outer Execution Environment Threads (attack19)
com.endosoft.pgec.PrivacyException: you may not break out of your thread group↩
↪ inside the execution environment
```

Attack #20 is implemented by class `com.endosoft.privacyattack.GlobalSynchronization`. It should be started concurrently in an outer and an inner execution environment. A parameter determines the behavior, i.e., whether it encodes or receives information. The algorithms for encoding and receiving are described in Section 5.1.3. To display the successful case, the

countermeasure of getting all locks on globally available objects is omitted for objects other than the boxed primitives. Using a second parameter, the attacking code can use either boxed primitives or some class objects, i.e., `PrivacyManager.class, String.class, Socket.class`, for the required synchronization. When run using the class objects, the sequence of 1-1-0-1-1-0-1-1-0 is correctly transmitted from the inner execution environment. The code in the outer execution environment outputs then:

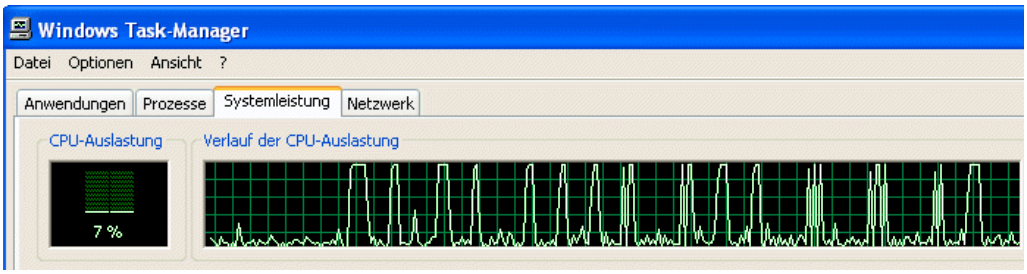
```
ONE
ONE
ZERO
ONE
ONE
ZERO
ONE
ONE
ZERO
```

When using the boxed primitives, which are locked by the PGEC, the attacking code deadlocks.

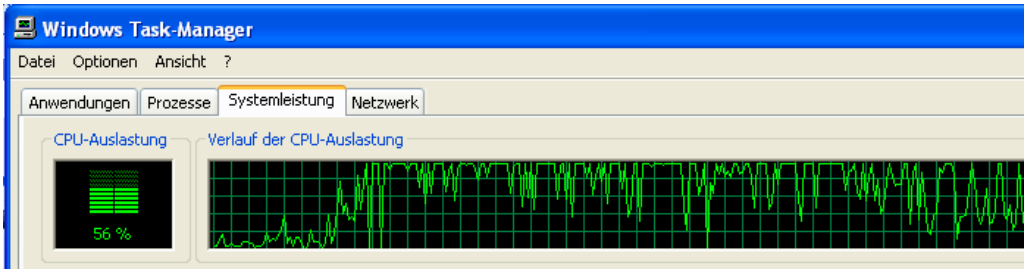
Attack #21 is implemented by class `com.endosoft.privacyattack.GeneratorMemoryLoadManchester` and `com.endosoft.privacyattack.ReaderMemoryLoadManchester`. They should be started in an inner and an outer execution environment accordingly. A successful attack outputs the detected detected high and low signals by printing - and _ and decodes the signals to “Hello World” as described in Section 5.1.3. With enabled countermeasures (uncommment lines 678 and 679 in `PrivacyManager.java`), the attack results in a randomized output like:

```
_---------------_____________________------_----------_----------_____↩
↪--------_______-----_____----_______----_______-0
-----------------_____----_____-----------------_________-_________------↩
↪_____--_____---_________-------____----------_____---_______----_________↩
↪-----_______-_________________-----_________________________-_____------↩
↪_________----_________________-----_________-_____-----_________↩
↪------_____________________--------_____-----_____-----_____________--↩
↪_____--__________--------_____-------------------------_________↩
↪---------_________---_______--------_________-----_____------_____--↩
↪_____________________----------_____________________↩
↪----------_____________________--_________--------_________-----↩
↪_____________________-----_____----_________----_____--------_____--_--↩
↪_________---------_____________________-------------------------↩
↪_____________--------_____________________-----------------------↩
↪___---____________________
```

Attack #22 is implemented by class `com.endosoft.privacyattack.GeneratorCPULoadManchester` and `com.endosoft.privacyattack.ReaderCPULoadManchester`. They should be started in an inner and an outer execution environment accordingly. A successful attack can decode the signal modulated on the CPU load and outputs it as described in Section 5.1.3.



(a) CPU load graph of attack #22 with disabled countermeasures of the PGEC.



(b) CPU load graph of attack #22 with enabled countermeasures of the PGEC.

Figure 5.8: Comparison of the CPU load graphs produced by the implementation of attack #22 without and with enabled PGEC countermeasures.

In this implementation, “Hello World” should be printed after a minimum duration of 194 seconds. With enabled countermeasures (uncomment lines 680 and 681 in `PrivacyManager.java`), the attack results in a randomized output. The effectivity of the countermeasure against attack #22 is illustrated by the direct comparison of the CPU load graphs that are produced by the attack. While in Figure 5.8a the transported signal is quite obvious, it can not be found in Figure 5.8b.

Chapter 6

Summary

In this dissertation scenarios have been introduced and analyzed, in which private data arise or are used. Based on those, aspects of privacy protection were identified. The first aspect regards the prevention of generation of private information, such as behavioral traces. Protection of and control over data actually released to others are the second aspect of privacy protection. A third aspect concerns only about the deduction of information from released or sensed private data [2]. It is considered, that fulfillment of the first two aspects obsoletes the third. Hence, this dissertation attended just to these two aspects.

Unsolicited data creation belongs to the first aspect. Such unsolicited creation of private information is gathering of behavioral profiles, such as sensing of data by sensors out of the subject's control [81, 82]. A special case of sensing behavior is to track the usage of access controlled services. In prevention of this kind of tracking, an access control technology that provides k-anonymity during the usage of services has been introduced and described. The approach uses blindly signed certificates, each representing a granted right and an expiration date. The represented rights and expiration dates remain verifiable in spite of the blind signing. The expiration date is kept reasonably low to allow implicit revocation of certificates. Besides these, other information or constraints can be certified as long as the information can be expressed as or mapped to a natural number. This approach is secured against attacks of malicious users trying to gain access, which is not granted. It is further verified, that a malicious Certificate Authority (CA) is not able to break the user's privacy. The presented access control technology guarantees that the identity of the presenter of such certificate cannot be revealed. The only information that can be inferred is, it was one of the k users, to which this particular right was granted. Thereby, a k-anonymity is gained. In order to provide good privacy protection, reasonable scenarios have to involve a large number of users and a low number of possible rights.

The achieved anonymity by the presented access control technology provides a good pri-

vacy protection. However, it requires online connection between the service providers and the CA for any certificate verification. Besides unnecessary network traffic, this introduces a bottleneck in the verification of large numbers of certificates. Further research will have to find ways for offline verification at the services themselves. The signature schemes of Elliptic Curve Cryptography (ECC) differ from those of Rivest-Shamir-Adleman (RSA). It might be possible to release the public keys for verification without the possibility to misuse them for derivation of validly signed certificates reflecting non-granted rights. The application of ECC for digital signing or more sophisticated signature key derivation schemes are potential candidates. It must be ensured that the knowledge of the verification key does not enable to derive another valid signature from any validly signed certificate.

The proposed certificates were designed for scenarios of anonymous access control. Additionally, they can open other interesting application fields, which may require further research. One of those fields can be proxy voting [83] or other electronic voting and election scenarios. The usual access control scenarios do not require protection against double spending, but for proper application in voting and elections, means against double spending must be included. This double spending does not only apply to a single certificate, but rather to a group of certificates. Only one certificate of this group may be used. An idea to accomplish this is to assign each certificate with a point in a vector space. While each user is assigned with a curve in this vector space, which may be uniquely identified by two points, she/he cannot be identified by a single point. That is, if more than one certificate from a group of a particular user was presented, the identity of the user may be revealed. E-cash schemes use similar approaches [13].

The second aspect of privacy protection assumes that generation and use of private information shall not be prevented but are vital for provision of services. That is, private data has to be released to services. In order to prevent profiling or use of these data for any other purpose than the expected, the data must not be copied or persisted in any way. Similar approaches are known from DRM systems. There, data is protected from unauthorized copying and usage within a very limited scope of services, which are merely decoding and playback services for audio and video data. This dissertation presents the Privacy Guaranteeing Execution Container (PGEC), an approach to allow DRM-like data protection in conjunction with arbitrary services. The PGEC provides execution environments for services that ensure access to private data of the service users while strictly controlling and limiting communication and persistence means. To reasonably support distributed services across the Internet, making use of powerful infrastructure at service providers, the PGEC is designed to be used distributed. The privacy protected environments are logically extended over multiple distributed PGEC instances to enable a logical unity, which may be compared to Enterprise Java Bean (EJB) containers known from Java 2 Enterprise Edition (J2EE).

From the investigated privacy protection demands and technical requirements it has been discovered, that runtime environments, such as the Java Runtime Enviroment (JRE) or

.NET, are most suitable for a PGEC implementation. The existing prototype is implemented in Java and adheres to the Java 1.1 security model, which allows for backward compatibility. Here it shall further be noticed, that the .NET runtime provides similar concepts of class loading and security management. The class loader in .NET loads only complete assemblies, that there is only one security manager and both cannot be overridden. Hence, an implementation in .NET, may be not as straightforward as in Java but still feasible.

A number of attacks against the PGEC or the privacy of users were researched in this dissertation and repelled by the existing PGEC implementation. The attacks and the respective countermeasures are presented. Most of them are also implemented and tested. Hence, the protection principles can be shown and the described scenarios can be implemented using the PGEC.

The presented PGEC approach provides means to verify and trust itself and its host runtime environment. Basically, the permitted libraries and their secure hashes are listed and checked at startup. It further describes means for mutual authentication of distributed PGEC instances. The instances share a secret key, which is built-in, to establish mutual trust. The use of Public Key Infrastructure (PKI) keys is proposed to enable revocation of individual instances of the PGEC. In either case, secret or private keys embedded in software are vulnerable to extraction. Hence, the use of hardware dongles storing and processing the keys and the hash values of the permitted libraries is proposed and should be considered for implementation of a productive PGEC. Obviously, the existing PGEC prototype does not provide 100% security. Even a perfect implementation will not be able to do so. Therefore, economical security has been introduced. That is, data are economically secure, if the effort and costs to obtain an unprotected copy are higher than the data's value of benefit. Hardware dongles do not provide 100% protection either, but definitely increase the effort of key extraction, which can improve the economical security.

A number of issues are still open for further research and development. Those include usability and security aspects. Features and concepts of distributed systems, e.g., load balancing, code migration and transparent remote procedure calls, will improve the usability. Hence, an integration with or into existing EJB containers is envisioned for future development.

The security model of Java evolved to enhance flexibility and security. While the PGEC implementation highly relies on the Java security model, it should make use of the most advanced features of this model. In fact, the protection domains introduced by the Java 2 security model are resembled by the PGEC implementation. Further, the Java 2 security model introduced an `AccessController` and `AccessControlContext`s, which cannot be overridden. These are implemented in native code to have deeper access to the call stack, which is annotated with information about the current protection domain. These classes read the respective permissions from a policy file, but the PGEC requires the permissions/prohibitions rather fixed and out of control of the executor of a PGEC

instance. Therefore, these classes must not be used or their native implementation has to be replaced. The feasibility to do so, is subject to further research.
Even though Java has broad acceptance, it is desirable to support multiple platforms. With .NET being a widely accepted platform for managed code, an aim of further research is migration or implementation of the PGEC in the .NET platform. The characteristics of the security model of .NET has to be studied and adapted to match the PGEC's protection requirements. The language J# and its .NET compilers provide language compatibility to Java. The J# compiler introduces a mapping of the Java class loading and security management into those provided by .NET. This might allow for direct compilation of the PGEC in .NET. At least, it promises that a migration of the PGEC to .NET is possible at all.
While the PGEC is functional and provides a good prove of concept, a few issues were not addressed, yet. With full access to the hardware of the machine running the PGEC and services, an attacker might read the contents of the volatile memory. That is, in memory encryption should be applied. With regard to economical security, the ratio between effort and value of the private user data to be gained may not be reasonable. Since the PGEC has not much control over the underlying Operating System (OS) and its memory management, an attacker could force the PGEC's memory to be swapped to the hard-drive, from where it could be read. The OS must further ensure secure demarcation of memory segments. In order to address this memory security, the PGEC shall be rather embedded in the OS, as the Protected Media Path (PMP) in the Protected Environment (PE) since Windows Vista is.
It has further be determined, that application of dongles or Trusted Platform Modules (TPMs) can improve the economical security. Integration of such hardware into a Java implementation of the PGEC is left as an open issue. Here, the possible ways of integration and appropriate partitioning between hard- and software system has to be investigated.
In order to determine the actual resilience of the proposed PGEC, a complete implementation ought to be exposed to the developer and tester community. Besides possibly discovering other attacks or ways to compromise the PGEC, the community will be able to build services in and around it. Useful services, e.g., navigation and Location Based Services (LBSs), social networking or online communication and collaboration tools like Google Wave[1], are essential to receive user acceptance.
Despite the open issues, the existing PGEC prototype provides reasonable protection of provided data, while being able to release them to arbitrary services. As long as the PGEC can be trusted, users do not have to worry about their private data and service providers do not have to worry about their technical assertion of compliance to their own and legal privacy policies. Until a broad acceptance and adoption of PGECs, people should not loose a certain amount of sanity and reason and develop a healthy privacy awareness.

[1] http://wave.google.com

Bibliography

[1] Alan F. Westin. *Privacy and freedom / [by] Alan F. Westin ; foreword by Oscar M. Ruebhausen*. Atheneum, New York :, [1st ed.] edition, 1967. [cited at p. 3]

[2] Steffen Ortmann, Peter Langendörfer, and Michael Maaser. Enhancing Privacy by Applying Information Flow Modelling in Pervasive Systems. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops*, volume 4806 of *Lecture Notes in Computer Science*, pages 794–803, Vilamoura, Algarve, Portugal, November 25-30 2007. International Workshop on Privacy in Pervasive Environments (PiPE07), Springer LNCS. [cited at p. 3, 5, 113]

[3] G. A. Wilkes and W. A. Krebs. *Collins English dictionary : an extensive coverage of contemporary international and Australian English / special Australian consultants, G.A. Wilkes, W.A. Krebs*. HarperCollins, Sydney :, updated 3rd ed. edition, 1995. [cited at p. 3]

[4] Lorrie F. Cranor, Brooks Dobbs, Serge Egelman, Giles Hogben, Jack Humphrey, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, Joseph Reagle, Matthias Schunter, David A. Stampley, and Rigo Wenning. W3C: Platform for Privacy Preferences (P3P) Project. http://www.w3.org/P3P/, November 12 2006. [cited at p. 4, 13]

[5] J. Cuellar, J. Morris, D. Mulligan, J. Peterson, and J. Polk. GEOPRIV requirements. Request for Comments: 3693, February 2004. [cited at p. 4, 14]

[6] Stefan Krempl and Jürgen Kuri. Datenschützer fordert Ende der Datensammelwut. http://www.heise.de/newsticker/Datenschuetzer-fordert-Ende-der-Datensammelwut–/meldung/99342/, November 21 2007. German only. [cited at p. 4]

[7] Annie I. Anton, Qingfeng He, and David. L. Baumer. Inside JetBlue's Privacy Policy Violations. *IEEE Security and Privacy*, 2(6):12–18, Nov./Dec. 2004. [cited at p. 4]

[8] Michael Barbaro and Tom Zeller, Jr. A Face Is Exposed for AOL Searcher No. 4417749. http://www.nytimes.com/2006/08/09/technology/09aol.html?ex=1312776000en=f6f61949c6da4d38ei=5090, August 9 2006. [cited at p. 4]

[9] Jürgen Kuri. Millionen Briten von Datenpanne betroffen. http://www.heise.de/newsticker/Millionen-Briten-von-Datenpanne-betroffen–/meldung/99315
http://news.bbc.co.uk/2/hi/uk_news/politics/7103828.stm, November 21 2007. German and English. [cited at p. 4]

[10] Florian Rötzer. Italienisches Finanzministerium veröffentlichte Einkommenssteuererklärungen aller Bürger. http://www.heise.de/newsticker/Italienisches-Finanzministerium-veroeffentlichte-Einkommenssteuererklaerungen-aller-Buerger–/meldung/107281/, May 1 2008. German only. [cited at p. 4]

[11] Stefan Krempl and Peter-Michael Ziegler. Bundestag nickt Abkommen zur Weitergabe von Fluggastdaten ab. http://www.heise.de/newsticker/Bundestag-nickt-Abkommen-zur-Weitergabe-von-Fluggastdaten-ab–/meldung/99138, November 16 2007. German only. [cited at p. 4]

[12] Stefan Krempl and Jürgen Kuri. EU-Datenschützer gehen gemeinsam gegen SWIFT-Affäre vor. http://www.heise.de/newsticker/EU-Datenschuetzer-gehen-gemeinsam-gegen-SWIFT-Affaere-vor–/meldung/76096, July 28 2006. German only. [cited at p. 4]

[13] Krzysztof Piotrowski, Peter Langendörfer, and Damian Kulikowski. Moneta: An Anonymity Providing Lightweight Payment System for Mobile Devices. In *Proceedings of the 2nd International Workshop for Technology, Economy, Social and Legal Aspects of Virtual Goods*, 2004. [cited at p. 7, 34, 114]

[14] Redaktion intern.de . ...die falschen Bücher gekauft? http://www.intern.de/news/4874.html, October 15 2003. German only. [cited at p. 8]

[15] Shu Wang, Jungwon Min, and Byung K. Yi. Location Based Services for Mobiles: Technologies and Standards. In *IEEE International Conference on Communication (ICC)*, 2008. [cited at p. 8]

[16] Lorrie F. Cranor, Marc Langheinrich, and Massimo Marchiori. W3C: A P3P Preference Exchange Language 1.0 (APPEL1.0). http://www.w3.org/TR/P3P-preferences/, 15 April 2002. W3C Working Draft. [cited at p. 13]

[17] Peter Langendörfer and Rolf Kraemer. Towards User Defined Privacy in location-aware Platforms. In *Proceedings of the 3rd International Conference on Internet Computing*, USA, 2002. 3rd international Conference on Internet computing, CSREA Press. [cited at p. 14]

[18] Marcel Bennicke and Peter Langendörfer. Towards Automatic Negotiation of Privacy Contracts for Internet Services. In *Proceedings of the 11th IEEE Conference on Networks*, pages 312–324. ICON, IEEE Society Press, 2003. [cited at p. 14]

[19] W. Wagealla, S. Terzis, and C. English. Trust-based Model for Privacy Control in Context-aware Systems. In *Proceedings of the 2nd Workshop on Security in Ubiquitous Computing*. 2nd Workshop on Security in Ubiquitous Computing, 2003. [cited at p. 14]

[20] Kåre Synnes, James Nord, and Peter Parnes. Location Privacy in the Alipes platform. In *Proceedings of the Hawai'i International Conference on System Sciences*, Big Island, Hawaii, USA, January 2003. Hawai'i International Conference on System Sciences (HICSS-36). [cited at p. 14]

[21] Marco Gruteser and Dirk Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM/USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003. [cited at p. 14, 16]

[22] Javier López, Antonio Maña, Ernesto Pimentel, José M. Troya, and Mariemma I. Yagüe. Access Control Infrastructure for Digital Objects. In *Proceedings of the International Conference on Information and Communications Security (ICICS'02)*, pages 399–410, Singapore, December 2002. International Conference on Information and Communications Security (ICICS'02), LNCS 2513, Springer-Verlag. [cited at p. 14]

[23] Ricardo Haragutchi, Barry D. Nusbaum, Carlos de Luna Sáenz, Nilson Tenorio Batista, Roberto Morizi Oku, Patrick Schmitt-Heinrich, and Robert Macgregor. *IBM Redbook: Building the Infrastructure for the Internet*, chapter Chapter 12 Networked Applications, page 526. IBM, November 1996. Cryptolope. [cited at p. 14]

[24] Antonio Maña, Javier Lopez, Juan J. Ortega, Ernesto Pimentel, and José M. Troya. A Framework for Secure Execution of Software. *International Journal of Information Security*, 2(4):99–112, November 2004. Springer. [cited at p. 14]

[25] H. Garcia-Molina, S. Ketchpel, and N. Shivakumar. Safeguarding and Charging for Content on the Internet. In *Proceedings of the International Conference On Data Engineering '98*. International Conference On Data Engineering, 1998. [cited at p. 14]

[26] N. Huda, Shigeki Yamada, and Eiji Kamioka. Privacy Protection in Mobile Agent based Service Domain. In *Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05)*, Sydney, July 4th- 7th 2005. [cited at p. 14]

[27] A. Yannopoulos, Y. Stavroulasa, N. Papadakis, D. Halkos, and T. Varvarigou. A method which enables the assessment of private data by an untrusted party using arbitrary algorithms but prevents disclosure of their content. In P. Langendoerfer and V. Tsaoussidis, editors, *Proceedings of the 3rd International Conference on Internet Computing*. 3rd International Conference on Internet Computing, CSREA Press, 2002. [cited at p. 14]

[28] Shigeki Yamada and Eiji Kamioka. Access Control for Security and Privacy in Ubiquitous Computing Environments. *IEICE-Transactions on Communications*, E88-B(3):846–856, March 2005. [cited at p. 14]

[29] Richard Conway and David Strip. Selective partial access to a database. In *ACM 76: Proceedings of the annual conference*, pages 85–89, New York, NY, USA, 1976. ACM. [cited at p. 15]

[30] Georg Treu and Axel Küpper. Efficient Proximity Detection for Location Based Services. In *Proceedings of 2nd Workshop on Positioning, Navigation and Communication 2005*, Hannover Germany, March 2005. WPNC05, SHAKER-Publishing. [cited at p. 16]

[31] Marek Fisz. *Probability Theory and Mathematical Statistics*. John Wiley & Sons, 3rd edition, January 1 1963. [cited at p. 16]

[32] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, page 439450, Dallas, Texas, May 2000. ACM Press. [cited at p. 17]

[33] Hillol Kargupta, Souptik Datta, Qi Wang, and Krishnamoorthy Sivakumar. On the Privacy Preserving Properties of Random Data Perturbation Techniques. In *Proceedings of the Third ICDM IEEE International Conference on Data Mining*, pages 99–107, Melbourne, FL, November 2003. [cited at p. 17]

[34] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, New York, NY, USA, 2002. ACM. [cited at p. 17, 18]

[35] Wenbo He, Xue Liu, Hoang Nguyen, Klara Nahrstedt, and Tarek Abdelzaher. PDA: Privacy-preserving Data Aggregation in Wireless Sensor Networks. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, pages 2045–2053, May 2007. [cited at p. 17, 18, 20]

[36] Maithili Narasimha. Privacy Homomorphisms. http://sconce.ics.uci.edu/docs/ Privacy%20Homomorphisms.pdf, February 13 2003. [cited at p. 19]

[37] Josep Domingo-Ferrer. A New Privacy Homomorphism and Applications. *Information Processing Letters*, 60(5):277–282, 1996. [cited at p. 20]

[38] Ronald L. Rivest, Leonard M. Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–177, 1978. [cited at p. 20]

[39] Ernest F. Brickell and Yacov Yacobi. On Privacy Homomorphisms. In *Advances in Cryptology EUROCRYPT 87*, pages 117–125. Springer Berlin / Heidelberg, 1987. Extended Abstract. [cited at p. 20]

[40] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *Proceedings of the 18th USENIX Security Symposium*, 2009. [cited at p. 22]

[41] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. [cited at p. 22]

[42] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 2005. ACM. [cited at p. 22]

[43] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, February 1 2005. [cited at p. 25]

[44] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). http://www.w3.org/TR/MathML2/, 21 October 2003. W3C Recommendation. [cited at p. 24]

[45] D. Richard Kuhn. Role Based Access Control on MLS Systems Without Kernel Changes. In *Third ACM Workshop on Role Based Access Control*, pages 25–32, 1998. [cited at p. 26]

[46] Ravi S. Sandhu, Edward J. Coynek, Hal L. Feinsteink, and Charles E. Youmank. Role-Based Access Control Models. *IEEE Computer*, 29(2):3847, February 1996. [cited at p. 27]

[47] D.F. Ferraiolo and D. Richard Kuhn. Role-Based Access Control. In *15th National Computer Security Conference*, pages 554–563, October 1992. [cited at p. 28, 158]

[48] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote Authentication Dial In User Service (RADIUS). Request for Comments: 2865, June 2000. [cited at p. 27, 29]

[49] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). Request for Comments: 1510, September 1993. [cited at p. 27]

[50] Peter Langendörfer, Krzysztof Piotrowski, and Michael Maaser. A Distributed Privacy Enforcement Architecture based on Kerberos. In *WSEAS Transactions on Communications*, volume Vol. 5 (2), pages 231–238, 2006. [cited at p. 28]

[51] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments: 2401, November 1998. [cited at p. 29]

[52] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. In *IEEE Transactions on Information Theory*, volume IT-22, page 644654, November 1976. [cited at p. 29, 42, 95]

[53] Frank A. Stevenson. Cryptanalysis of Contents Scrambling System. http://www.cs.cmu.edu/~dst/DeCSS/FrankStevenson/analysis.html, November 8 1999. [cited at p. 30]

[54] John Leyden. Blu-ray DRM defeated - Copy-protection cracked again. http://www.theregister.co.uk/2007/01/23/blu-ray_drm_cracked/, January 23 2007. [cited at p. 30]

[55] Trusted Computing Group Administration . http://www.trustedcomputinggroup.org, 2008. [cited at p. 30, 88, 157]

[56] Microsoft Corporation . Code Signing for Protected Media Components in Windows Vista. http://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/PMP-sign.doc, August 25 2006. [cited at p. 31, 158]

[57] Michael Maaser and Steffen Ortmann. Providing Granted Rights with Anonymous Certificates. In *Proceedings of the 15th IEEE International Conference on Electronics, Circuits, and Systems*. IEEE International Conference on Electronics, Circuits, and Systems, August 31 - September 3 2008. [cited at p. 33]

[58] P. Samarati. Protecting Respondents' Identities in Microdata Release. *IEEE Transactions on Knowledge & Data Engineering*, 13(6):1010–1027, 2001. [cited at p. 34]

[59] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. In *Communications of the ACM 24*, pages 84–88. ACM, February 1981. [cited at p. 34]

[60] David Chaum and E. van Heyst. Group signatures. In *Advances in Cryptology*, pages 257–265. EUROCRYPT '91, Springer-Verlag, 1991. Vol. 547 of LNCS. [cited at p. 34]

[61] Stefan Brands. *Rethinking Public Key Infrastructure and Digital Certificates - Building in Privacy*. PhD thesis, Institute of Technology, Eindhoven, Netherland, 1999. [cited at p. 34]

[62] Endre Bangerter, Jan Camenisch, and Anna Lysyanskaya. A cryptographic framework for the controlled release of certified data. In *Twelfth International Workshop on Security Protocols 2004*. Springer-Verlag, 2004. LNCS. [cited at p. 34]

[63] Jan Camenisch, Dieter Sommer, and Roger Zimmermann. A General Certification Framework with Application to Privacy-Enhancing Certificate Infrastructures. In *Proceedings of the IFIP TC-11 21st International Information Security Conference (SEC 2006)*, Karlstad, Sweden, May 22-24 2006. [cited at p. 34]

[64] David Chaum. Blind signature systems. In *Advances in Cryptology*. CRYPTO '83, 1984. [cited at p. 34, 35]

[65] D. G. Kendall and R. Osborn. Two Simple Lower Bounds for Euler's Function. *Texas Journal of Science*, 17(3), 1965. [cited at p. 38]

[66] Michael Maaser and Peter Langendörfer. Privacy from Promises to Protection: Privacy Guaranteeing Execution Container. *Mob. Netw. Appl.*, 14(1):65–81, 2009. [cited at p. 51]

[67] Michael Maaser and Peter Langendörfer. Automated Negotiation of Privacy Contracts. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, Edinburgh, UK, July 26-29 2005. IEEE Society Press. [cited at p. 53, 58, 59]

[68] Michael Maaser, Steffen Ortmann, and Peter Langendörfer. NEPP: Negotiation Enhancements for Privacy Policies. In *W3C Workshop on Languages for Privacy Policy Negotiation and Semantics-Driven Enforcement*, Ispra, Italy, October 17-18 2006. [cited at p. 53, 58, 59]

[69] Michael Maaser, Steffen Ortmann, and Peter Langendörfer. The Privacy Advocate (PrivAd): A Framework for Negotiating Individual Privacy Contracts. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST)*, Barcelona, Spain, March 3-6 2007. [cited at p. 53, 58, 59]

[70] Michael Maaser, Steffen Ortmann, and Peter Langendörfer. *The Privacy Advocate: Assertion of Privacy by Personalised Contracts*, volume 8 of *Lecture Notes in Business Information Processing*, pages 85–97. Springer, Setúbal, Portugal, 2008. [cited at p. 53, 58, 59]

[71] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thom, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. Cryptology ePrint Archive, Report 2010/006, 2010. [cited at p. 55]

[72] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. In *PKC '00: Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography*, pages 446–465, London, UK, 2000. Springer-Verlag. [cited at p. 55]

[73] Dieter Spaar. Einmal ein Cracker sein. *c't*, (21):212–217, 2007. [cited at p. 55]

[74] Elke Spiegelhalter. Wibu-Systems im internationalen Wettstreit mit Hacker Ergebnis zur CeBIT 2007. http://www.wibu.com/press.php?year=2007&num=01, January 2007. Pressemitteilung. [cited at p. 55]

[75] K. El-Khatib. A Privacy Negotiation Protocol for Web Services. In *Workshop on Collaboration Agents: Autonomous Agents for Collaborative Environments*, Halifax, Nova Scotia, Canada, 2006. NRC Institute for Information Technology; National Research Council Canada. [cited at p. 58]

[76] Sören Preibusch. Implementing Privacy Negotiation Techniques in E-Commerce. In *CEC '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology*, pages 387–390, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 58]

[77] John McHugh. *Handbook for the Computer Security Certification of Trusted Systems*, volume NRL Technical Memorandum 5540:062A, chapter Chapter 8: Covert Channel Analysis. Naval Research Laboratory, Code 5540, Washington, D.C. 20375-5337, February 12 1996. [cited at p. 66]

[78] Idongesit Mkpong-Ruffin, John A. Hamilton, Jr., and Martin C. Carlisle. The New Java Security Architecture. *CrossTalk - The Journal of Defense Software Engineering*, Jul 2006. [cited at p. 72, 160]

[79] . IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, December 26 2008. [cited at p. 86]

[80] Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. Using Software-based Attestation for Verifying Embedded Systems in Cars. In *Proceedings of the Embedded Security in Cars Workshop '04*. Embedded Security in Cars Workshop (escar), November 2004. [cited at p. 88]

[81] Steffen Ortmann, Peter Langendörfer, and Michael Maaser. A Self-Configuring Privacy Management Architecture for Pervasive Systems. In Albert Y. Zomaya and Sherali Zeadally, editors, *Proceedings of the 5th ACM International Workshop on Mobility Management and Wireless Access (MobiWAC)*, pages 184–187, Chania, Crete Island, Greece, October 22 2007. ACM Press. [cited at p. 113]

[82] Steffen Ortmann, Peter Langendörfer, and Michael Maaser. Adapting Pervasive Systems to Multi-user Privacy Requirements. *International Journal of Ad Hoc and Ubiquitous Computing*, 3(4):264–276, 2008. [cited at p. 113]

[83] . *Modernising Democracy: Innovations in Citizen Participation*. M.E.Sharpe, 80 Business Park Drive, Armonk, NY, 10504, 2006. [cited at p. 114]

Appendices

Appendix A

Listings

Listing A.1: Native code gathering information about the current running process and comparing it with a specified set of libraries and modules

```
1  using System;
2  using System.Threading;
3  using System.Diagnostics;
4  using System.Xml;
5  using System.Xml.Schema;
6  using System.IO;
7  using System.Security.Cryptography;
8
9  public class com_endosoft_pgec_PrivacyManager
10 {
11     public com_endosoft_pgec_PrivacyManager() { }
12
13     private void ValidationEventHandler(object sender, ValidationEventArgs e)
14     {
15         if (e.Severity == XmlSeverityType.Warning)
16         {
17             Console.Write("WARNING: ");
18             Console.WriteLine(e.Message);
19         }
20         else if (e.Severity == XmlSeverityType.Error)
21         {
22             Console.Write("ERROR: ");
23             Console.WriteLine(e.Message);
24         }
25     }
26
27     /// <summary>
28     /// CheckSoftwareEnvironment will be the method called from within java.
29     /// </summary>
30     public Boolean CheckSoftwareEnvironment()
31     {
32         XmlReaderSettings readerSettings = new XmlReaderSettings();
33         readerSettings.Schemas.Add("PGECEnvironmentDescriptor", "D:/development/↩
       ↪dissertation/PGEC/JNI/CSharp/EnvironmentDescriptor.xsd");
34         readerSettings.ValidationType = ValidationType.Schema;
35         readerSettings.ValidationEventHandler += new ValidationEventHandler(↩
       ↪ValidationEventHandler);
36
37         XmlReader xmlr = XmlReader.Create("D:/development/dissertation/PGEC/JNI/CSharp/↩
       ↪EnvDescriptor.xml", readerSettings);
38
```

```
39          XmlDocument xmld = new XmlDocument();
40          xmld.Load(xmlr);
41          XmlNode root = xmld.DocumentElement;
42
43          //Instantiate an XmlNamespaceManager object.
44          System.Xml.XmlNamespaceManager xmlnsManager = new System.Xml.XmlNamespaceManager(↩
            ↪xmld.NameTable);
45
46          //Add the namespaces used in books.xml to the XmlNamespaceManager.
47          xmlnsManager.AddNamespace("pgec", "PGECEnvironmentDescriptor");
48          //xmlnsManager.AddNamespace("pub", "urn:Publisher");
49
50          Process myProcess = Process.GetCurrentProcess();
51          Console.WriteLine("I am Process ID=" + myProcess.Id);
52          ProcessThreadCollection ptc = myProcess.Threads;
53          foreach (ProcessThread pt in ptc)
54          {
55              Console.Out.WriteLine("Thread:" + pt.Id);
56          }
57          try
58          {
59              if (!CheckProcessModule(root, xmlnsManager, myProcess.MainModule, true))
60              {
61                  return false;
62              }
63              // Get all the modules associated with 'myProcess'.
64              ProcessModuleCollection myProcessModuleCollection = myProcess.Modules;
65              // Display the properties of each of the modules.
66              for (int i = 0; i < myProcessModuleCollection.Count; i++)
67              {
68                  ProcessModule myProcessModule = myProcessModuleCollection[i];
69                  if (!CheckProcessModule(root, xmlnsManager, myProcessModule, false))
70                  {
71                      return false;
72                  }
73              }
74          }
75          catch (Exception exc)
76          {
77              Console.Error.WriteLine(exc.Message);
78              Console.Error.WriteLine(exc.StackTrace);
79              return false;
80          }
81          return true;
82      }
83
84      private Boolean CheckProcessModule(XmlNode root, System.Xml.XmlNamespaceManager ↩
        ↪xmlnsManager, ProcessModule myProcessModule, Boolean claimsMain)
85      {
86          //TODO cope around with Mainmodule - claimsMain
87          string moduleName = myProcessModule.ModuleName;
88          IntPtr baseAddress = myProcessModule.BaseAddress;
89          IntPtr entryPointAddress = myProcessModule.EntryPointAddress;
90          int memorySize = myProcessModule.ModuleMemorySize;
91          string fileName = myProcessModule.FileName;
92          long fileSize = 0;
93
94          if (File.Exists(fileName))
95          {
96              FileInfo fileInfo = new FileInfo(fileName);
97              fileSize = fileInfo.Length;
98          }
99          else
100         {
101             Console.Error.WriteLine("wie kann ein modul aus einem nicht existierenden ↩
                ↪File geladen werden?");
102             //should never occur
103             //TODO raise privacy exception
104             return false;
105         }
```

```
106         XmlNodeList allowedModules = root.SelectNodes("descendant::pgec:Module[pgec:Name↩
            ↪='" + moduleName + "' and pgec:Size=" + fileSize + "]", xmlnsManager);
107         if (allowedModules != null && allowedModules.Count > 0)
108         {
109             byte[] data = File.ReadAllBytes(fileName);
110 
111             bool processModuleIsAllowed = false;
112             foreach (XmlNode allowedModuleUnderTest in allowedModules)
113             {
114                 XmlNode moduleHash = allowedModuleUnderTest.SelectSingleNode("pgec:Hash",↩
                    ↪ xmlnsManager);
115                 string hashType = moduleHash.Attributes["type"].Value;
116                 string hashHexValue = null;
117                 if (moduleHash.FirstChild != null)
118                 {
119                     hashHexValue = moduleHash.FirstChild.Value;
120                 }
121                 else
122                 {
123                     hashHexValue = "NONE";
124                 }
125 
126                 byte[] hash;
127 
128                 if ("SHA1".Equals(hashType, StringComparison.OrdinalIgnoreCase))
129                 {
130                     SHA1 sha = new SHA1CryptoServiceProvider();
131                     // This is one implementation of the abstract class SHA1.
132                     hash = sha.ComputeHash(data);
133                 }
134                 else if ("MD5".Equals(hashType, StringComparison.OrdinalIgnoreCase))
135                 {
136                     MD5 md5 = new MD5CryptoServiceProvider();
137                     // This is one implementation of the abstract class SHA1.
138                     hash = md5.ComputeHash(data);
139                 }
140                 else
141                 {
142                     hash = new byte[0];
143                 }
144                 StringWriter sw = new StringWriter();
145                 for (int j = 0; j < hash.Length; j++)
146                 {
147                     sw.Write(hash[j].ToString("X2"));
148                 }
149                 if (hashHexValue.Equals(sw.ToString()))
150                 {
151                     //OK fits the hash
152                     //TODO may be check baseaddresse and entrypoint as well
153                     processModuleIsAllowed = true;
154                     break;
155                 }
156                 else
157                 {
158                     // does not fit the hash try another
159                     Console.Error.WriteLine("this process module [" + moduleName + "] is ↩
                        ↪has hash " + hashType + ":" + sw.ToString());
160                 }
161             }
162             if (processModuleIsAllowed)
163             {
164                 Console.Out.WriteLine("this process module [" + moduleName + "] is OK");
165                 Console.Out.WriteLine(" it was loaded from "+fileName);
166                 return true;
167             }
168             else
169             {
170                 //the module in the process does not fit the hashes
171                 //it is likely to be manipulated
```

```
172                 Console.Error.WriteLine("this process module [" + moduleName + "] is ↩
                    ↪manipulated");
173                 //TODO raise privacy exception
174                 return false;
175             }
176         }
177         else
178         {
179             Console.Error.WriteLine("this process module [" + moduleName + "] is not ↩
                    ↪allowed");
180 
181             //a module was not listed hence it is not allowed
182             //TODO this should raise a privacy exception
183             Console.Error.WriteLine("               its File name is: "
184                 + fileName);
185             Console.Error.WriteLine("               its File size is: "
186                 + fileSize);
187             Console.Error.WriteLine("               its base address is: "
188                 + baseAddress);
189             Console.Error.WriteLine("               its Entry point address is: "
190                 + entryPointAddress);
191             Console.Error.WriteLine("               its MemorySize is: "
192                 + memorySize);
193             return false;
194         }
195     }
196 
197     public static int Main(String[] args)
198     {
199         new com_endosoft_pgec_PrivacyManager().CheckSoftwareEnvironment();
200         return 0;
201     }
202 }
```

Listing A.2: XML schema for environment descriptors.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="EnvironmentDescriptor" targetNamespace="PGECEnvironmentDescriptor" elementFormDefault="qualified" xmlns="PGECEnvironmentDescriptor" xmlns:mstns="PGECEnvironmentDescriptor" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="PGEC" type="PGECT">
  </xs:element>
  <xs:complexType name="PGECT">
    <xs:sequence minOccurs="1" maxOccurs="unbounded">
      <xs:element name="Module" type="ModuleT">
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ModuleT">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Name" type="xs:string">
      </xs:element>
      <xs:element name="Size" type="xs:positiveInteger">
      </xs:element>
      <xs:element name="Hash" type="HashT" />
      <xs:element name="BaseAddress" type="xs:long" minOccurs="0" maxOccurs="1">
      </xs:element>
      <xs:element name="EntryPoint" type="xs:long" minOccurs="0" maxOccurs="1">
      </xs:element>
      <xs:element name="MemorySize" type="xs:long" minOccurs="0" maxOccurs="1">
      </xs:element>
    </xs:sequence>
    <xs:attribute name="canBeMain" type="xs:boolean" use="optional" />
    <xs:attribute name="forDebugOnly" type="xs:boolean" use="optional" />
  </xs:complexType>
  <xs:complexType name="HashT">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" type="HashAlgorithms" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:simpleType name="HashAlgorithms">
    <xs:restriction base="xs:string">
      <xs:enumeration value="MD5" />
      <xs:enumeration value="SHA1" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Listing A.3: Overridden class loader spanning execution environments that are unable to access static references of each other

```
1  package com.endosoft.pgec;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.FileNotFoundException;
6  import java.io.IOException;
7  import java.lang.Thread.UncaughtExceptionHandler;
8  import java.lang.reflect.InvocationTargetException;
9  import java.lang.reflect.Method;
10 import java.util.HashMap;
11 import java.util.StringTokenizer;
12 import java.util.Vector;
13
14 import org.apache.commons.logging.Log;
15 import org.apache.commons.logging.LogFactory;
16
17 import com.endosoft.pgec.XMLConfigParser.Class.entrytypes;
18
19 /**
20  * <p>
21  * berschrift:
22  * </p>
23  *
24  * <p>
25  * Beschreibung:
26  * </p>
27  *
28  * <p>
29  * Copyright: Copyright (c) 2005
30  * </p>
31  *
32  * <p>
33  * Organisation:
34  * </p>
35  *
36  * @author Michael Maaser
37  * @version 1.0
38  */
39 public abstract class ExecutionEnvironment extends ClassLoader {
40
41     private ExecutionEnvironmentID ID = null;
42     protected Log log = LogFactory.getLog("ExecSpace");
43     private XMLConfigParser.Environment environmentDescription = null;
44     private Vector<Thread> environmentThreads = new Vector<Thread>();
45
46     // these are used when class is loaded from remote container
47     private ContainerID remoteClassSource = null;
48     private String name = null;
49     private XMLConfigParser.Class classDescription = null;
50     private byte[] classData = null;
51     private HashMap<String, Class<?>> loadedclasses = new HashMap<String, Class<?>>();
52
53     public ExecutionEnvironment(
54             XMLConfigParser.Environment environmentDescription) {
55         synchronized (this) {
56             ID = new ExecutionEnvironmentID(
57                     this instanceof InnerExecutionEnvironment, ""
58                             + this.hashCode()); /*
59                                                  * TODO what happens if this is
60                                                  * accidentially the same as of
61                                                  * another existing execution
62                                                  * environment started in
63                                                  * another container; refer to
64                                                  * ExceutionEnvironment(Class,
65                                                  * ExecutionEnvironment)
66                                                  */
67             // Permissions all = new Permissions();
```

```
68              // all.add(new AllPermission());
69              // ProtectionDomain outerDomain = new ProtectionDomain(null, all,
70              // this, null);
71              // AccessControlContext outerACC = new AccessControlContext(
72              // new ProtectionDomain[] { outerDomain });
73              // AccessControlContext currentACC = AccessController.getContext();
74              // final XMLConfigParser.Class desc = entryPointDescription;
75              // AccessController.doPrivileged(new PrivilegedAction<Object>() {
76              // public Object run() {
77              // startServiceInEnvironment(desc);
78              // return null;
79              // }
80              // }, outerACC);
81              this.environmentDescription = environmentDescription;
82              startServiceInEnvironment();
83          }
84      }
85
86      public ExecutionEnvironment(
87              XMLConfigParser.Environment environmentDescription,
88              ExecutionEnvironmentID id) {
89          synchronized (this) {
90              if (id.isInner() && (this instanceof InnerExecutionEnvironment)) {
91                  this.ID = id;
92              } else if (id.isOuter()
93                      && (this instanceof OuterExecutionEnvironment)) {
94                  this.ID = id;
95              } else {
96                  throw new PrivacyException("attempt to give an "
97                          + (id.isInner() ? "inner" : "outer")
98                          + " execution environemnt the id of an "
99                          + (id.isInner() ? "outer" : "inner") + " one");
100             }
101             this.environmentDescription = environmentDescription;
102             startServiceInEnvironment();
103         }
104     }
105
106     public ExecutionEnvironment(ContainerID remotecontainer,
107             ExecutionEnvironmentID id, String name,
108             XMLConfigParser.Class clientClassDescription, byte[] clientclassdata) {
109         synchronized (this) {
110             if (id.isInner() && (this instanceof InnerExecutionEnvironment)) {
111                 this.ID = id;
112             } else if (id.isOuter()
113                     && (this instanceof OuterExecutionEnvironment)) {
114                 this.ID = id;
115             } else {
116                 throw new PrivacyException("attempt to give an "
117                         + (id.isInner() ? "inner" : "outer")
118                         + " execution environemnt the id of an "
119                         + (id.isInner() ? "outer" : "inner") + " one");
120             }
121             this.environmentDescription = null;
122             this.name = name;
123             this.remoteClassSource = remotecontainer;
124             this.classDescription = clientClassDescription;
125             this.classData = clientclassdata;
126         }
127     }
128
129     public void start() {
130         this.startServiceInEnvironment(this.classDescription, this.classData);
131     }
132
133     /**
134      * @param entryPointDescription
135      */
136     private void startServiceInEnvironment() {
137         ThreadGroup execEnvThreadGroup = new ThreadGroup(
```

```
138                 (this instanceof InnerExecutionEnvironment) ? "Inner Execution ↩
                    ↪Environment Threads ("
139                         + environmentDescription.getName() + ")"
140                         : "Outer Execution Environment Threads ("
141                                 + environmentDescription.getName() + ")");
142         for (XMLConfigParser.Class entryPointDescription : environmentDescription
143                 .getClasses()) {
144             switch (entryPointDescription.getType()) {
145             case RUNNABLE:
146                 try {
147                     Class<?> entrypointClass = null;
148                     entrypointClass = Class.forName(entryPointDescription
149                             .getClassName(), true, this);
150                     Runnable runnable = (Runnable) entrypointClass
151                             .newInstance();
152                     Thread environmentThread = new Thread(execEnvThreadGroup,
153                             runnable);
154                     if (this.getUncaughtExceptionHandler() != null) {
155                         environmentThread.setUncaughtExceptionHandler(this
156                                 .getUncaughtExceptionHandler());
157                     }
158                     environmentThread.setContextClassLoader(this);
159                     environmentThreads.add(environmentThread);
160                     environmentThread.start();
161                 } catch (Exception e) {
162                     throw new PrivacyException(
163                             "Environment could not be started, due to", e);
164                 }
165                 break;
166             case MAIN:
167                 Thread environmentThread = new Thread(
168                         execEnvThreadGroup,
169                         new MainRunner(entryPointDescription, this),
170                         (this instanceof InnerExecutionEnvironment) ? "Inner Main Thread ↩
                            ↪("
171                                 + entryPointDescription.getClassName() + ")"
172                                 : "Outer Main Thread ("
173                                         + entryPointDescription.getClassName()
174                                         + ")");
175                 environmentThread.setContextClassLoader(this);
176                 environmentThreads.add(environmentThread);
177                 environmentThread.start();
178                 break;
179             }
180         }
181     }
182
183     /**
184      *
185      * @param entryPointDescription
186      */
187     private void startServiceInEnvironment(
188             XMLConfigParser.Class entryPointDescription, byte[] classdata) {
189         ThreadGroup execEnvThreadGroup = new ThreadGroup(
190                 (this instanceof InnerExecutionEnvironment) ? "Inner Execution ↩
                    ↪Environment Threads ("
191                         + getName() + ")"
192                         : "Outer Execution Environment Threads (" + getName()
193                                 + ")");
194         Class<?> entrypointClass = defineClass(entryPointDescription
195                 .getClassName(), classdata, 0, classdata.length);
196         Thread environmentThread = null;
197         if (entrytypes.RUNNABLE.equals(entryPointDescription.getType())) {
198             try {
199                 Runnable runnableClient = (Runnable) entrypointClass
200                         .newInstance();
201                 environmentThread = new Thread(execEnvThreadGroup,
202                         runnableClient);
203                 if (this.getUncaughtExceptionHandler() != null) {
204                     environmentThread.setUncaughtExceptionHandler(this
```

```
205                             .getUncaughtExceptionHandler());
206                 }
207             } catch (Exception e) {
208                 throw new PrivacyException(
209                         "Environment could not be started, due to", e);
210             }
211         } else {
212             environmentThread = new Thread(
213                     execEnvThreadGroup,
214                     new MainRunner(entryPointDescription, this, entrypointClass),
215                     (this instanceof InnerExecutionEnvironment) ? "Inner Main Thread ("
216                             + entryPointDescription.getClassName() + ")"
217                             : "Outer Main Thread ("
218                                     + entryPointDescription.getClassName()
219                                     + ")");
220         }
221         if (environmentThread != null) {
222             // environmentThread.setContextClassLoader(this);
223             environmentThreads.add(environmentThread);
224             environmentThread.start();
225         }
226     }
227 
228     public synchronized boolean isActive() {
229         if (environmentThreads == null) {
230             return false;
231         } else {
232             for (Thread envThread : environmentThreads) {
233                 if (envThread.isAlive()) {
234                     return true;
235                 }
236             }
237             return false;
238         }
239     }
240 
241     protected UncaughtExceptionHandler getUncaughtExceptionHandler() {
242         return null;
243     }
244 
245     private class MainRunner implements Runnable {
246 
247         private XMLConfigParser.Class entryPointDescription;
248         private ExecutionEnvironment myEnvironment;
249         private Class<?> entrypointClass;
250 
251         private MainRunner(XMLConfigParser.Class entryPointDescription,
252                 ExecutionEnvironment environment) {
253             this(entryPointDescription, environment, null);
254         }
255 
256         private MainRunner(XMLConfigParser.Class entryPointDescription,
257                 ExecutionEnvironment environment, Class<?> classdata) {
258             this.entryPointDescription = entryPointDescription;
259             myEnvironment = environment;
260             if (classdata == null) {
261                 try {
262                     entrypointClass = java.lang.Class.forName(
263                             entryPointDescription.getClassName(), true,
264                             myEnvironment);
265                 } catch (ClassNotFoundException e) {
266                     throw new PrivacyException(
267                             "Environment could not be started, due to", e);
268                 }
269             } else {
270                 entrypointClass = classdata;
271             }
272         }
273 
274         @Override
```

```
275         public void run() {
276             switch (entryPointDescription.getType()) {
277             case RUNNABLE:
278                 break;
279             case MAIN:
280                 try {
281                     Thread.currentThread().setUncaughtExceptionHandler(
282                             myEnvironment.getUncaughtExceptionHandler());
283                     Method mainMethod = entrypointClass.getMethod("main",
284                             new Class<?>[] { String[].class });
285                     mainMethod
286                             .invoke(null, new Object[] { entryPointDescription
287                                     .getArguments() });
288                 } catch (InvocationTargetException ite) {
289                     throw new RuntimeException(ite.getTargetException());
290                     /*
291                      * if (this instanceof InnerExecutionEnvironment) {
292                      * System.err.println("don't tell anything"); } else { if
293                      * (ite.getCause() instanceof PrivacyException) {
294                      * ite.getCause().printStackTrace(); } else {
295                      * ite.printStackTrace(); } }
296                      */
297                 } catch (IllegalArgumentException iae) {
298                     iae.printStackTrace();
299                 } catch (IllegalAccessException iae) {
300                     iae.printStackTrace();
301                 } catch (PrivacyException pe) {
302                     pe.printStackTrace();
303                 } catch (SecurityException se) {
304                     se.printStackTrace();
305                 } catch (NoSuchMethodException nsme) {
306                     nsme.printStackTrace();
307                 }
308                 break;
309             }
310         }
311     }
312
313     /**
314      * Finds the class with the specified <a href="#name">binary name</a>. The
315      * classes that explicitly belong to the container may be loaded via the
316      * parent class loader. Other wise we get a lot of class cast exceptions,
317      * because the PrivacyManager loaded by the system class loader appears to
318      * be something different from a PrivacyManager loaded by one of the
319      * execution environments.
320      *
321      * @param name
322      *            The <a href="#name">binary name</a> of the class
323      * @return The resulting <tt>Class</tt> object
324      * @throws ClassNotFoundException
325      *             If the class could not be found
326      */
327     protected Class<?> findClass(String name) throws ClassNotFoundException {
328         // Class<?> loadedC = this.findLoadedClass(name);
329         for (Class<?> partOfPGEC : PrivacyManager.getClassesBelongingToPGEC()) {
330             if (partOfPGEC.getName().equals(name)) {
331                 return partOfPGEC;
332             }
333         }
334         Class<?> defined = getLoadedClass(name);
335         if (defined != null) {
336             return defined;
337         }
338         byte[] b = loadClassData(name);
339         if (b != null) {
340             defined = defineClass(name, b, 0, b.length);
341         } else {
342             // TODO darf nicht einfach wenn class nicht geladen werden kann
343             // den system classloader benutzen SICHERHEITSRISIKO
344             defined = this.getParent().loadClass(name);
```

```
345             //TODO new PrivacyManager.InfiniteLock(defined);
346         }
347         addLoadedClass(name, defined);
348         return defined;
349         // } else {
350         // // some classes explicitly named should be loaded by original class
351         // // loaders to allow correct casting
352         // return this.getParent().loadClass(name);
353         // }
354     }
355 
356     private String getClassPath() {
357         if (remoteClassSource != null) {
358             return null;
359         } else {
360             if (getEnvironmentDescription().getClasspath() != null) {
361                 return getEnvironmentDescription().getClasspath() + ";"
362                         + System.getProperty("java.class.path");
363             } else {
364                 return System.getProperty("java.class.path");
365             }
366         }
367     }
368 
369     /**
370      * TODO load from JAR file or other URL or source
371      *
372      * @param name
373      *            String
374      * @return byte[]
375      */
376     protected final byte[] loadClassData(String name) {
377         // load the class data from the connection
378         log.debug("loading class " + name);
379         if (remoteClassSource != null) {
380             if (name.startsWith("java.")) {
381                 // since system classes in package java have to be loaded via
382                 // the system class loader anyway
383                 // I don't need to load it from remote
384                 return null;
385             }
386             if (name.equals("com.sun.xml.internal.stream.XMLOutputFactoryImpl")) {
387                 // if I try to load this from remote I need an instance of just
388                 // this class, which I don't have yet, it's a hen-egg problem
389                 // solution: load this class locally
390                 return null;
391             }
392             if (name.equals("com.sun.xml.internal.stream.XMLInputFactoryImpl")) {
393                 // if I try to load this from remote I need an instance of just
394                 // this class, which I don't have yet, it's a hen-egg problem
395                 // solution: load this class locally
396                 return null;
397             }
398             return PrivacyManager.getPrivacyManager()
399                     .loadClassDataFromOtherContainer(remoteClassSource, name);
400         }
401         String filename = name.replace('.', '/') + ".class";
402         String classpath = getClassPath();
403         StringTokenizer cpTokenizer = new StringTokenizer(classpath, ";");
404         while (cpTokenizer.hasMoreTokens()) {
405             String directory = cpTokenizer.nextToken();
406             if (!directory.equals("")) {
407                 File classToLoad = new File(directory, filename);
408                 if (classToLoad.exists()) {
409                     int classLength = (int) classToLoad.length();
410                     byte[] out = new byte[classLength];
411                     try {
412                         FileInputStream fis = new FileInputStream(classToLoad);
413                         int ptr = 0;
414                         while (ptr < classLength) {
```

```
415                             ptr = fis.read(out, ptr, classLength);
416                         }
417                     } catch (FileNotFoundException fnfe) {
418                         out = null;
419                     } catch (IOException ioe) {
420                         out = null;
421                     }
422                     return out;
423                 }
424             }
425         }
426         // System.out.println(this.getClass().getResource("."));
427         // System.out.println("CLASSPATH="+System.getProperty("java.class.path"));
428         return null;
429     }
430 
431     /**
432      * Loads the class with the specified <a href="#name">binary name</a>.
433      *
434      * @param name
435      *             The <a href="#name">binary name</a> of the class
436      * @return The resulting <tt>Class</tt> object
437      * @throws ClassNotFoundException
438      *              If the class was not found
439      */
440     public Class<?> loadClass(String name) throws ClassNotFoundException {
441         return loadClass(name, false);
442     }
443 
444     /**
445      * Loads the class with the specified <a href="#name">binary name</a>.
446      *
447      * @param name
448      *             The <a href="#name">binary name</a> of the class
449      * @param resolve
450      *             If <tt>true</tt> then resolve the class
451      * @return The resulting <tt>Class</tt> object
452      * @throws ClassNotFoundException
453      *              If the class could not be found
454      */
455     protected synchronized Class<?> loadClass(String name, boolean resolve)
456             throws ClassNotFoundException {
457         Class<?> c = findClass(name);
458         if (resolve) {
459             resolveClass(c);
460         }
461         return c;
462     }
463 
464     protected final void addLoadedClass(String name, Class<?> clazz) {
465         synchronized (loadedclasses) {
466             loadedclasses.put(name, clazz);
467         }
468     }
469 
470     protected final Class<?> getLoadedClass(String name) {
471         synchronized (loadedclasses) {
472             if (loadedclasses.containsKey(name)) {
473                 return loadedclasses.get(name);
474             } else {
475                 return null;
476             }
477         }
478     }
479 
480     protected final ExecutionEnvironmentID getID() {
481         return ID;
482     }
483 
484     protected final String getName() {
```

```
485         if (this.name != null) {
486             return this.name;
487         } else {
488             return getEnvironmentDescription().getName();
489         }
490     }
491
492     protected final XMLConfigParser.Environment getEnvironmentDescription() {
493         return environmentDescription;
494     }
495
496 }
```

Listing A.4: Support class enumerating via reflection all objects that can be reached from a given object reference and starting a thread holding a lock on each of their monitors

```
280      *         environment.
281      */
282     public static final Class<?>[] getClassesBelongingToPGEC() {
283         return classesBelongingToPGEC;
284     }
285 
286     public static final class InfiniteLock extends Thread {
287         private final HashSet<Object> listOfObj = new HashSet<Object>();
288         private Object locksEngagedNotify = new Object();
289         private static final HashSet<Object> lockedObjects = new HashSet<Object>();
290         private static final HashSet<Object> criticalObjects = new HashSet<Object>();
291 
292         static {
293             criticalObjects.add(Window.class);
294             criticalObjects.add(SecurityManager.class);
295             //criticalObjects.add(System.err);
296             //criticalObjects.add(System.out);
297         }
298 
299         private boolean hasStaticSynchronizedMethods(Class<?> clz) {
300             for (Method meth : clz.getDeclaredMethods()) {
301                 if (Modifier.isStatic(meth.getModifiers())
302                         && Modifier.isSynchronized(meth.getModifiers())) {
303                     return true;
304                 }
305             }
306             return false;
307         }
308 
309         private boolean hasSynchronizedMethods(Object obj) {
310             for (Method meth : obj.getClass().getDeclaredMethods()) {
311                 if (Modifier.isSynchronized(meth.getModifiers())) {
312                     return true;
313                 }
314             }
315             return false;
316         }
317 
318         private InfiniteLock() {
319             super("locking boxes");
320             lockBooleanBoxes();
321             lockByteBoxes();
322             lockIntegerBoxes();
323             lockShortBoxes();
324             lockCharacterBoxes();
325             this.start();
326             synchronized (locksEngagedNotify) {
327                 try {
328                     locksEngagedNotify.wait();
329                 } catch (InterruptedException e) {
330                 }
331             }
332         }
333 
334         /**
335          * Due to boxing mechanism the cast of a primitive data type into its
336          * object results in the same object instance for a memory and
337          * implementation limited number of values. I lock on all these objects.
338          */
339         private void lockBooleanBoxes() {
340             prepareForLocking((Boolean) true);
341             prepareForLocking((Boolean) false);
342         }
343 
344         /**
345          * Due to boxing mechanism the cast of a primitive data type into its
346          * object results in the same object instance for a memory and
```

```
347          * implementation limited number of values. I lock on all these objects.
348          */
349         private void lockByteBoxes() {
350             for (byte i = Byte.MIN_VALUE;; i++) {
351                 prepareForLocking((Byte) i);
352                 if (i == Byte.MAX_VALUE) {
353                     break;
354                 }
355             }
356         }
357 
358         /**
359          * Due to boxing mechanism the cast of a primitive data type into its
360          * object results in the same object instance for a memory and
361          * implementation limited number of values. I lock on all these objects.
362          */
363         private void lockIntegerBoxes() {
364             int i = -1;
365             while (i < Integer.MAX_VALUE) {
366                 i++;
367                 Object A = (Integer) i;
368                 Object B = (Integer) i;
369                 if (A == B) {
370                     prepareForLocking(A);
371                     // new InfiniteLock(A);
372                 } else {
373                     break;
374                 }
375             }
376             i = 0;
377             while (i > Integer.MIN_VALUE) {
378                 i--;
379                 Object A = (Integer) i;
380                 Object B = (Integer) i;
381                 if (A == B) {
382                     prepareForLocking(A);
383                     // new InfiniteLock(A);
384                 } else {
385                     break;
386                 }
387             }
388         }
389 
390         /**
391          * Due to boxing mechanism the cast of a primitive data type into its
392          * object results in the same object instance for a memory and
393          * implementation limited number of values. I lock on all these objects.
394          */
395         private void lockShortBoxes() {
396             short i = -1;
397             while (i < Short.MAX_VALUE) {
398                 i++;
399                 Object A = (Short) i;
400                 Object B = (Short) i;
401                 if (A == B) {
402                     prepareForLocking(A);
403                     // new InfiniteLock(A);
404                 } else {
405                     break;
406                 }
407             }
408             i = 0;
409             while (i > Short.MIN_VALUE) {
410                 i--;
411                 Object A = (Short) i;
412                 Object B = (Short) i;
413                 if (A == B) {
414                     prepareForLocking(A);
415                     // new InfiniteLock(A);
416                 } else {
```

```
417                     break;
418                 }
419             }
420         }
421
422         /**
423          * Due to boxing mechanism the cast of a primitive data type into its
424          * object results in the same object instance for a memory and
425          * implementation limited number of values. I lock on all these objects.
426          */
427         private void lockCharacterBoxes() {
428             char i = '\0';
429             while (i < Character.MAX_VALUE) {
430                 i++;
431                 Object A = (Character) i;
432                 Object B = (Character) i;
433                 if (A == B) {
434                     prepareForLocking(A);
435                     // new InfiniteLock(A);
436                 } else {
437                     break;
438                 }
439             }
440         }
441
442         private InfiniteLock(Object obj) {
443             super("locking " + obj);
444             prepareForLocking(obj);
445             this.start();
446             synchronized (locksEngagedNotify) {
447                 try {
448                     locksEngagedNotify.wait();
449                 } catch (InterruptedException e) {
450                 }
451             }
452         }
453
454         @SuppressWarnings("unused")
455         private InfiniteLock(Object... objects) {
456             super("locking " + objects[0] + " ...");
457             for (Object obj : objects) {
458                 prepareForLocking(obj);
459             }
460             this.start();
461             synchronized (locksEngagedNotify) {
462                 try {
463                     locksEngagedNotify.wait();
464                 } catch (InterruptedException e) {
465                 }
466             }
467         }
468
469         InfiniteLock(Class<?> clz) {
470             super("locking " + clz);
471             // this.obj = clz;
472             prepareForLocking(clz);
473             this.start();
474             synchronized (locksEngagedNotify) {
475                 try {
476                     locksEngagedNotify.wait();
477                 } catch (InterruptedException e) {
478                 }
479             }
480         }
481
482         /**
483          * @param clz
484          * @throws SecurityException
485          */
486         private void prepareForLocking(Class<?> clz) throws SecurityException {
```

```
487             synchronized (doPrivilegedActionByMyselfLock) {
488             synchronized (lockedObjects) {
489                 if (!lockedObjects.contains(clz)
490                         && !criticalObjects.contains(clz)) {
491                     if (hasStaticSynchronizedMethods(clz)) {
492                         // there are synchronized static methods in Thread
493                         if (clz.isPrimitive()) {
494 
495                         } else {
496                             System.err.println("1:critical object " + clz);
497                             criticalObjects.add(clz);
498                         }
499                     } else {
500                         lockedObjects.add(clz);
501                         listOfObj.add(clz);
502                         // this.start();
503                     }
504                     for (Field objField : clz.getDeclaredFields()) {
505                         if (Modifier.isStatic(objField.getModifiers())
506                                 && !Modifier.isPrivate(objField.getModifiers())
507                                 && !objField.getType().isPrimitive()) {
508                             try {
509                                 Object field = objField.get(null);
510                                 if (field != null) {
511                                     prepareForLocking(field);
512                                     // new InfiniteLock(field);
513                                 }
514                             } catch (IllegalArgumentException e) {
515                                 // TODO Auto-generated catch block
516                                 e.printStackTrace();
517                             } catch (IllegalAccessException e) {
518                                 // TODO Auto-generated catch block
519                                 // e.printStackTrace();
520                             }
521                         }
522                     }
523                     for (Class<?> objInnerClass : clz.getDeclaredClasses()) {
524                         if (Modifier.isStatic(objInnerClass.getModifiers())
525                                 && !Modifier.isPrivate(objInnerClass
526                                         .getModifiers())) {
527                             try {
528                                 prepareForLocking(objInnerClass);
529                                 // new InfiniteLock(objInnerClass);
530                             } catch (IllegalArgumentException e) {
531                                 // TODO Auto-generated catch block
532                                 e.printStackTrace();
533                             }
534                         }
535                     }
536                     Class<?> superclass = clz.getSuperclass();
537                     if (superclass != null) {
538                         prepareForLocking(superclass);
539                         // new InfiniteLock(superclass);
540                     }
541                     for (Class<?> implementedInterface : clz.getInterfaces()) {
542                         prepareForLocking(implementedInterface);
543                         // new InfiniteLock(implementedInterface);
544                     }
545                 }
546             }
547             }
548         }
549 
550         /**
551          * @param obj
552          * @throws SecurityException
553          */
554         private void prepareForLocking(Object obj) throws SecurityException {
555             synchronized (doPrivilegedActionByMyselfLock) {
556             if (obj instanceof Class<?>) {
```

```
557                 prepareForLocking((Class<?>) obj);
558             } else {
559                 synchronized (lockedObjects) {
560                     if (!lockedObjects.contains(obj)
561                             && !criticalObjects.contains(obj)) {
562                         if (hasSynchronizedMethods(obj)) {
563                             // there are synchronized static methods in Thread
564                             if (obj.getClass().isPrimitive()) {
565 
566                             } else {
567                                 System.err.println("2:critical object " + obj);
568                                 criticalObjects.add(obj);
569                             }
570                         } else {
571                             lockedObjects.add(obj);
572                             listOfObj.add(obj);
573                             // this.start();
574                         }
575                         Class<?> clz = obj.getClass();
576                         for (Field objField : clz.getDeclaredFields()) {
577                             if (!Modifier.isPrivate(objField.getModifiers())
578                                     && !Modifier.isTransient(objField
579                                             .getModifiers())
580                                     && !objField.getType().isPrimitive()) {
581                                 try {
582                                     Object field = objField.get(obj);
583                                     if (field != null) {
584                                         prepareForLocking(field);
585                                         // new InfiniteLock(field);
586                                     }
587                                 } catch (IllegalArgumentException e) {
588                                     // TODO Auto-generated catch block
589                                     // e.printStackTrace();
590                                 } catch (IllegalAccessException e) {
591                                     // TODO Auto-generated catch block
592                                     // e.printStackTrace();
593                                 }
594                             }
595                         }
596                         for (Class<?> objInnerClass : clz.getDeclaredClasses()) {
597                             if (!Modifier.isPrivate(objInnerClass
598                                     .getModifiers())) {
599                                 try {
600                                     prepareForLocking(objInnerClass);
601                                     // new InfiniteLock(objInnerClass);
602                                 } catch (IllegalArgumentException e) {
603                                     // TODO Auto-generated catch block
604                                     e.printStackTrace();
605                                 }
```

Listing A.5: Transmitting “Hello World” using Manchester Encoding over memory allocation.

```
/**
 *
 */
package com.endosoft.privacyattack;

import java.util.LinkedList;

/**
 * It produces high and low memory load in Manchester code. Unfortunately it does not ↩
 ↪allocate the CPU in
 * multi-core environments predictively.
 * @author Michael Maaser
 *
 */
public class GeneratorMemoryLoadManchester {

    public static int duration = 600;

    LinkedList<byte[][]> dummy = new LinkedList<byte[][]>();
    private void produceLow() {
        if (dummy.size()>0) {
            dummy.clear();
            System.gc();
        }
        try {
            Thread.sleep(duration);
        } catch (InterruptedException e) {
        }
    }
    private void produceHigh() {
        if (dummy.size()==0) {
            for (int i = 0; i < 128; i++)  {
                if (Runtime.getRuntime().freeMemory() > 1024 * 1024 * 2) {
                    dummy.add(new byte[1024][1024]);
                }
            }
        }
        try {
            Thread.sleep(duration);
        } catch (InterruptedException e) {
        }
    }

    private void produce(int bit) {
        switch (bit){
        case 0: produceHigh();produceLow();
        break;
        case 1: produceLow();produceHigh();
        break;
        }
    }

    private void produce(char byt) {
        for (int i = 0; i < 8; i++) {
            if (byt >= 128) {
                produce(1);
            } else {
                produce(0);
            }
            byt &= 0x7F;
            byt <<= 1;
        }
    }

    private void produce(String text) {
        for (int i =0; i < text.length(); i++) {
            produce(text.charAt(i));
```

```
67          }
68      }
69
70      private static long med =0;
71      private static long low =0;
72      private static long high =0;
73      /**
74       * @param args
75       */
76      public static void main(String[] args) {
77          GeneratorMemoryLoadManchester me = new GeneratorMemoryLoadManchester();
78          me.produceHigh();
79          me.produceHigh();
80          me.produceHigh();
81          me.produceHigh();
82          high = Runtime.getRuntime().freeMemory();
83          me.produceLow();
84          me.produceLow();
85          me.produceLow();
86          me.produceLow();
87          low = Runtime.getRuntime().freeMemory();
88          med = (high +low)/2;
89          System.out.println("Mem: low:"+low+" med:"+med+" high:"+high);
90
91          me.produce("Hello World");
92      }
93
94  }
```

Listing A.6: Receiving information using Manchester Encoding over memory allocation.

```
/**
 *
 */
package com.endosoft.privacyattack;

/**
 * @author Michael Maaser
 *
 */
public class ReaderMemoryLoadManchester extends Thread {

    private static int duration = GeneratorMemoryLoadManchester.duration;
    private static long med =0;
    private static long low =0;
    private static long high =0;

    public void run() {
        boolean currentstate = false; //low
        int samplingintervall = duration/4;
        try {
            Thread.sleep(0 * (samplingintervall+ 2 * duration));
        } catch (InterruptedException e) {
        }
        long lastStateChange = System.currentTimeMillis()- duration;
        int bitcounter = 0;
        char byt = 0;
        low = Runtime.getRuntime().freeMemory();
        high = low - 90000000;
        med = (high +low)/2;
        int eta = 2;
        for (int i = 0; i <1000; i++) {
            long fm = Runtime.getRuntime().freeMemory();
            long ts = System.currentTimeMillis();
            boolean previousstate = currentstate;
            currentstate = fm>med;
            System.out.print(currentstate?"_":"-");
            if (currentstate != previousstate) {
                if (ts - lastStateChange > duration * 3.5) {
                    eta --;
                    if (eta == 0) {
                        lastStateChange = ts - duration;
                    } else {
                        lastStateChange = ts;
                    }
                }
                if (eta == 0) {
                    if (ts - lastStateChange > duration * 1.6) {
                        byt <<=1;
                        if (!currentstate) {
                            byt +=1;
                        }
                        if (++bitcounter == 8) {
                            System.err.println(byt);
                            bitcounter = 0;
                            byt = 0;
                        }
                        lastStateChange = ts;
                    }
                }
            }
            if (currentstate) {
                high = fm;
            } else {
                low = fm;
            }
            med = (high +low)/2;
            try {
```

```
68                  Thread.sleep(samplingintervall);
69              } catch (InterruptedException e) {
70              }
71          }
72      }
73  }
```

Listing A.7: Overloading information encoded in memory allocation with random noise.

```
/**
 *
 */
package com.endosoft.pgec;

import java.util.LinkedList;
import java.util.Random;

/**
 * @author Michael Maaser
 *
 */
public class CovertChannelCounterMeasureMemory extends Thread {

    private LinkedList<byte[][]> dummy = new LinkedList<byte[][]>();
    private Random rand = new Random();
    private boolean running = false;

    public CovertChannelCounterMeasureMemory() {
        super("CovertChannelCounterMeasureMemory");
        this.start();
    }

    private void produceLow() {
        if (dummy.size()>0) {
            dummy.clear();
            System.gc();
        }
        try {
            Thread.sleep(rand.nextInt(4000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    private void produceHigh() {
        long fm = Runtime.getRuntime().freeMemory() / 1024 /1024;
        if (dummy.size()==0) {
            for (int i = 0; i < fm; i++)  {
                if (Runtime.getRuntime().freeMemory() > 1024 * 1024 * 2) {
                    dummy.add(new byte[1024][1024]);
                }
            }
        }
        try {
            Thread.sleep(rand.nextInt(4000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void finish() {
        running = false;
    }

    @Override
    public void run() {
        running = true;
        while(running) {
            produceHigh();
            produceLow();
        }
    }
}
```

Listing A.8: Transmitting “Hello World” using Manchester Encoding over CPU load.

```
/**
 *
 */
package com.endosoft.privacyattack;

import java.util.Random;

/**
 * It produces high and low CPU load in Manchester code. Unfortunately it does
 * not allocate the CPU in multi-core environments predictively.
 *
 * @author Michael Maaser
 *
 */
public class GeneratorCPULoadManchester {

    public static int duration = 1000;
    private static int numberofcores = 2;

    private void produceLow() {
        try {
            Thread.sleep(duration);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    private boolean highend = false;

    private void produceHigh() {
        // Object moni = new Object();
        // synchronized (moni) {
        highend = false;
        @SuppressWarnings("unused")
        int dummy = 0;
        final Random r = new Random();
        Thread[] th = new Thread[numberofcores + 1];

        th[0] = new Thread() {
            public void run() {
                try {
                    Thread.sleep(duration);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                highend = true;
            }
        };
        for (int i = 1; i <= numberofcores; i++) {
            th[i] = new Thread() {
                public void run() {
                    @SuppressWarnings("unused")
                    int dummy = 0;
                    while (!highend) {
                        dummy = r.nextInt();
                    }
                }
            };
        }
        for (int i = 0; i <= numberofcores; i++) {
            th[i].setName("CPU load generator ["+i+"]");
            th[i].start();
        }
        try {
            for (int i = 0; i <= numberofcores; i++) {
                th[i].join();
            }
```

```
70          } catch (InterruptedException e) {
71          }
72          // }
73      }
74
75      private void produce(int bit) {
76          switch (bit) {
77          case 0:
78              produceHigh();
79              produceLow();
80              break;
81          case 1:
82              produceLow();
83              produceHigh();
84              break;
85          }
86      }
87
88      private void produce(char byt) {
89          for (int i = 0; i < 8; i++) {
90              if (byt >= 128) {
91                  produce(1);
92              } else {
93                  produce(0);
94              }
95              byt &= 0x7F;
96              byt <<= 1;
97          }
98      }
99
100     private void produce(String text) {
101         for (int i = 0; i < text.length(); i++) {
102             produce(text.charAt(i));
103         }
104     }
105
106     /**
107      * @param args
108      */
109     public static void main(String[] args) {
110         try{
111             numberofcores = Integer.parseInt(args[0]);
112         }catch(Exception e){}
113         long start = System.currentTimeMillis();
114         GeneratorCPULoadManchester me = new GeneratorCPULoadManchester();
115         for (int i = 0; i < 8; i++) {
116             me.produceLow();
117         }
118         for (int i = 0; i < 8; i++) {
119             me.produceHigh();
120         }
121         me.produceLow();
122
123         me.produce("Hello World");
124         System.err.println("finished in "+((System.currentTimeMillis()-start)/1000)+"↩
            ↪seconds");
125     }
126
127 }
```

Listing A.9: Receiving information using Manchester Encoding over CPU load.

```
1  package com.endosoft.privacyattack;
2
3  import java.util.Random;
4
5  public class ReaderCPULoadManchester implements Runnable {
6
7      public static void main(String[] args) {
```

```
8           Thread th = new Thread(new ReaderCPULoadManchester(), "CPU load reader");
9           th.setPriority(Thread.MIN_PRIORITY);
10          th.start();
11      }
12
13      private Random r = new Random();
14      private static int duration = GeneratorCPULoadManchester.duration;
15      private int state = 0;
16      private char byt = 0;
17      private int bitcounter = 0;
18
19      private void appendBit(boolean bit) {
20          byt <<= 1;
21          if (bit) {
22              byt += 1;
23          }
24          if (++bitcounter == 8) {
25              System.err.print(byt);
26              bitcounter = 0;
27              byt = 0;
28          }
29      }
30
31      private void decode(boolean lowsignal, int duration) {
32          switch (state) {
33          case 0:
34              // initial low
35              if (!lowsignal && duration > 6) {
36                  state = 1;
37              }
38              break;
39          case 1:
40              // initial high
41              if (lowsignal && duration > 6) {
42                  state = 2;
43              }
44              break;
45          case 2:
46              if (duration == 2) {
47                  appendBit(!lowsignal);
48              } else {
49                  state = 3;
50              }
51              break;
52          case 3:
53              appendBit(!lowsignal);
54              state = 2;
55              break;
56          }
57      }
58
59      @Override
60      public void run() {
61          @SuppressWarnings("unused")
62          int dummy = 0;
63          long end = 0;
64          long count = 0;
65          boolean laststate = false;
66          long start = System.nanoTime();
67          long laststatechangetime = start;
68          boolean currentstate = false;
69          while (true) {
70              start = System.nanoTime();
71              for (count = 1; count <= 50000
72                      && (end = System.nanoTime()) < (start + 1000 * 1000 * 500); count++) ↩
                        ↪{
73                  dummy = r.nextInt();
74              }
75              // end = System.nanoTime();
76              currentstate = (count * 1000 * 1000 / (end - start)) > 200; // low
```

```
77                                                                          // CPU
78                                                                          // load
79              if (currentstate != laststate) {
80                  laststate = currentstate;
81                  decode(currentstate, (int) Math
82                          .round((start - laststatechangetime) / 1000.0 / 1000
83                                  / duration));
84                  laststatechangetime = start;
85              }
86          }
87      }
88  }
```

Listing A.10: Overloading information encoded in CPU load with random noise.

```
/**
 *
 */
package com.endosoft.pgec;

import java.util.Random;

/**
 * @author Michael Maaser
 *
 */
public class CovertChannelCounterMeasureCPU extends Thread {

    private Random rand = new Random();
    private boolean running = false;

    public CovertChannelCounterMeasureCPU() {
        super("CovertChannelCounterMeasureCPU");
        this.start();
    }

    private void produceLow() {
        try {
            Thread.sleep(rand.nextInt(4000));
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    private boolean highend = false;
    private void produceHigh() {
        Object moni = new Object();
        synchronized (moni) {
            highend = false;
            @SuppressWarnings("unused")
            int dummy = 0;
            Random r = new Random();
            new Thread() {
                public void run() {
                    try {
                        Thread.sleep(rand.nextInt(4000));
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    highend = true;
                }
            }.start();
            while (!highend) {
                dummy = r.nextInt();
            }
        }
    }

    public void finish() {
        running = false;
    }

    @Override
    public void run() {
        running = true;
        while(running) {
            produceHigh();
            produceLow();
        }
    }
}
```

List of Symbols and Abbreviations

3G third generation
4G fourth generation
AACS Advanced Access Content System
AES Advanced Encryption Standard
API Abstract Programming Interface
APPEL A P3P Preference Exchange Language
CA Certificate Authority
CD Compact Disc
CGMS-A Copy Generation Management System-Analog
CPDA Cluster-based Private Data Aggregation
CPU Central Processing Unit
CSS Content Scrambling System
DAC Discretionary Access Control
DHT Distributed Hash Table
DRM Digital Rights Management
DVD Digital Versatile Disc
ECC Elliptic Curve Cryptography
EJB Enterprise Java Bean
EPR Electronic Patients Record
GeoPriv Geographic Location/Privacy
GPS Global Positioning System
GSM Global System for Mobile communications (originally from Groupe Spécial Mobile)
GUI Graphical User Interface
HD-DVD High-Definition/Density DVD
HDCP High-bandwidth Digital Content Protection
IETF Internet Engineering Task Force
IPsec Internet Protocol Security

J2EE Java 2 Enterprise Edition
JDBC Java Database Connectivity
JRE Java Runtime Enviroment
JVM Java Virtual Machine
KAS Kerberos Authentication Service
LAN Local Area Network
LBS Location Based Service
MAC Mandatory Access Control
MathML Mathematical Markup Language
MIDI Musical Instrument Digital Interface
MP3 MPEG-1 Audio Layer 3
MPEG Moving Picture Experts Group
OASIS Organization for the Advancement of Structured Information Standards
OS Operating System
P3P Platform for Privacy Preferences Project
PACS Picture Archiving and Communication System
PC Personal Computer
PDA Personal Digital Assistant
PE Protected Environment
PET Privacy Enhancing Technologies
PGEC Privacy Guaranteeing Execution Container
PKI Public Key Infrastructure
PMP Protected Media Path
PRIME PRivacy and Identity Management for Europe
PSK Pre-shared key
PUMA Protected User Mode Audio
PVP Protected Video Path
PVP-OPM Protected Video Path - Output Protection Management
PVP-UAB Protected Video Path - User-Accessible Bus
RADIUS Remote Authentication Dial-In User Service
RAM Random Access Memory
RBAC Role Based Access Control
RFC Request For Comments
RSA Rivest-Shamir-Adleman (cryptosystem named by its inventors Ronald L. Rivest, Adi Shamir und Leonard Adleman)
SMART Slice-Mix-AggRegaTe
SQL Structured Query Language
SSL Secure Sockets Layer
SWIFT Society for Worldwide Interbank Financial Telecommunication
TAN Transaction Authentication Number
TCP Transmission Control Protocol

TGS Ticket Granting Service
TLS Transport Layer Security
TPM Trusted Platform Module[55]
UDP User Datagram Protocol
UMTS Universal Mobile Telecommunications System
VOD Video On Demand
WiMax Worldwide Interoperability for Microwave Access
WEP Wired Equivalent Privacy
WLAN Wireless LAN
WPA Wi-Fi Protected Access
WSN Wireless Sensor Network
XACML eXtensible Access Control Markup Language
XML eXtensible Markup Language
XSD XML Schema Definition

List of Figures

List of Tables