Play it once again, Sam - Enforcing Stateful Licenses on Open Platforms

Ahmad-Reza Sadeghi
Horst-Görtz-Institute for IT-Security
Ruhr-University Bochum
sadeghi@crypto.rub.de

Michael Scheibel
Sirrix AG Security Technologies
Bochum, Germany
m.scheibel@sirrix.com

Christian Stüble
Horst-Görtz-Institute for IT-Security
Ruhr-University Bochum
stueble@acm.org

Marko Wolf
Horst-Görtz-Institute for IT-Security
Ruhr-University Bochum
mwolf@crypto.rub.de

ABSTRACT
Various applications and business models for distributing digital content over open networks demand for licenses to control usage of the content and restrict access to it by authorized entities only. Of particular interest are stateful licenses that allow usage for a fixed time period or fixed number of times.

However, existing solutions using stateful licenses are vulnerable to various attacks, particularly on open computing platforms that are under consumers' control who can run exploits as well as reconfigure the underlying operating system. In this context replay attacks play an important role, since the state of common storage (e.g., hard-disks and flash memory) can easily be reset to some prior state bypassing access control mechanisms or breaking cryptographic protocols that keep state. Hence content providers tend to inflexible static licenses and *closed* DRM systems that mainly provide *unilateral* security, i.e., protect the needs of content providers and not of consumers.

In this paper we present a security architecture that enables the secure deployment and transfer of stateful licenses on open computing platforms while protecting the security objectives of both users and providers. We show how to implement this security architecture efficiently by means of virtualization technology, a security kernel, trusted computing functionality, and a legacy operating system (currently Linux). Moreover, our system extends the TCG chain of trust concept to arbitrary composed (trusted) domains, i.e., our architecture measures and reports the configuration of only those software components that are security-critical for a certain operation at certain time.

Keywords
Digital Rights Management, stateful licenses, freshness, security architectures, trusted computing

1. MOTIVATION
E-commerce applications for trading digital goods over open networks are becoming increasingly appealing. In this context techniques for secure distribution and usage of digital goods, where a license defines the owner's rights to consume (access, use) the data linked to this license, are crucial.

A particular license type are *stateful licenses* which allow the use of rights for a fixed time period, e.g., for n days, or for a fixed quantity, e.g., for n times.

A few e-business applications already employ such (mostly proprietary) stateful licences to sell certain digital goods for limited use. Important references are video-on-demand services or online video rentals [8, 47, 29] that use stateful licences to enable flexible pay-per-view scenarios. Various digital music stores [3] use stateful licences to control the maximum number of analogue copies allowed. Moreover, some software vendors already employ stateful licenses to offer trial versions that allow users to test a software for a limited time or allow a limited number of executions [25, 59]. Stateful licences enable new promising pay-per-use software business models.

In addition, one can think of other interesting applications using stateful licences to enforce policies: For instance sensitive user data such as email correspondence or identity information is being stored on remote servers today. Often the users are not fully aware of the data traces they leave on remote servers. In other cases users have to provide some personal information in order to use a service. However, users must have the right and be able to decide and limit the usage of this information by any third party (e.g., service provider). Another application is when digital music stores allow their customers for instance to hear a track two times for free before offering the acquisition of a license for unlimited access. Stateful licences can also enforce one-time access to sensitive data. Thus, for instance a company can prevent their employees from making unauthorized copies or forwarding of sensitive content that could leak information to its competitors.

Another important issue beside the secure usage of stateful licenses is the secure transfer of licenses among different platforms. This includes also secure lending or selling (sub-)licenses to other individuals without requiring the interaction of the licensor. In this context the license itself describes the conditions under which a transfer of the content, it is attached to, is authorized. For example, the licensee would not be allowed to freely *copy* the content, but would instead be allowed to *move* it to certain devices without Internet connectivity. Such resale and sub-licensing is commonly considered acceptable use, yet unlike the "offline" transaction, usually requires interaction with the licensor.

However, managing and enforcing stateful licenses on open platforms is particularly vulnerable to various threats such as unauthorized access, misuse, and illegal redistribution of the content to be protected [24, 49, 50]. Open platforms are under the control of their owners, who can attack and circumvent even sophisticated protection mechanisms by running exploits and reconfiguring the underlying operating system. Particularly replay attacks set the platform state (e.g., hard-disks and flash memory) and thus a stateful license to a prior state and circumvent security mechanisms. This can be done for instance by ordinary backup mechanisms or by applying software tools [11] that log all storage modifications to easily revoke these modifications for reuse of a license[1].

Consequently, content providers use tamper-resistant hardware devices like dongles [30] or smartcards [4] to securely store a small amount of data to protect their assets. The use of external devices, however, cannot guarantee the integrity of the operating system and a proper behavior of applications since debugging utilities and other manipulations of the operating system or applications frequently allow users to bypass security mechanisms. [2]

Thus, content providers currently tend to inflexible static licenses and *closed* DRM systems. The problem with closed DRM systems, such as [16, 28], is that they mainly provide *unilateral* security protecting the needs of content providers and usually not consumers[3] Moreover, common DRM systems do not provide adequate stateful licenses and thus heavily restrict users' rights, e.g., by preventing them from transferring licenses (that includes license moving, resale or renting).

1.1 Main Contribution & Outline

In this paper we present a security architecture that enables secure enforcement of stateful licenses on open computing platforms and secure license transfers among platforms while protecting the security objectives of users and providers. To the best of our knowledge there currently exists no solution that is capable of enforcing stateful licenses on open platforms while providing security functionalities allowing to establish multilateral security. We show how our architecture can efficiently be implemented using existing virtualization and trusted computing technology. In contrast to existing solutions, our system architecture measures and reports the configuration of only those software components that are security-critical for a certain operation, instead of reporting the configuration of *all* currently running software components that clearly affect user's privacy.

[1]Cryptographic measures like digital signatures, encryption and even cryptographic file systems [7, 57] cannot protect stateful licenses, since a complete backup can still be replayed.

[2]In particular, dongles turned out to be impractical for the mass market because of missing consumer friendliness and high costs [2].

[3]This is conform to the legislative trend (see [31]) of putting more restrictions on consumers' rights on using digital content.

Our paper is organized as follows. In Section 1.2 we summarize related work. We then define in Section 2 an ideal system model for distributed content access that satisfies our stated security objectives of the involved parties. Going towards the real world we replace the ideal model in Section 3 by several logical components that are implemented by real software components of a first prototype implementation based on a small security kernel, virtualization technology and trusted computing technology (Section 4).

1.2 Related Work

Shapiro and Vingralek [45, 56] identified the replay problem in client platforms that are completely under the control of the user. The authors proposed to manage persistent states using external *locker services* or assumed a small amount of secure memory (i.e., that cannot be read or written by an attacker) and secure one-way counters realized by battery-backed SRAM or special on-chip EEPROM/ROM functions.

Tygar and Yee [55] elaborate on applications of physically secure coprocessors, including enforcement of static and dynamic licenses without centralized servers. They show how to protect and attest the integrity of their system with the help of a secure coprocessor and a secure bootstrap process. In addition, protocols for sealing of data to a local platform and binding of data to a remote platform are presented. They identify the replay problem in the context of electronic currency and propose "two-phase" commits to ensure atomic transfers to remote platforms. The proposed architecure relies on a microkernel which is running in a *physical* security partition provided by the secure coprocessor. This is different to our approach which is based on a virtualization layer offering *logical* security partitions ("compartments").

Marchesini et al. [22] use OS hardening to create "software compartments" which are isolated from each other and cannot be accessed by a "root spy". Based thereon, their design provides "compartmentalized attestation", i.e. attestation of and binding data to single compartments. Our approach does not employ OS hardening techniques to secure a complex monolithic legacy OS. Instead we put the legacy OS in a compartment which is then run on top of a virtualization layer. The performance loss is negligible, but the increase in security is not, since the virtualization layer is much less complex than a monolithic OS kernel.

Baek and Smith [6] build on this work and implement a prototype for enforcing QoS policies on open platforms.

Publicly available documentation on both DRM implementations of Microsoft Windows Rights Management Services [27] and Authentica Active Rights Management [5] do not mention how they resist replay attacks. Once a client application is authorized to access a document, it can backup and restore its state to entirely access all documents at backup time.

The same holds for common DRM implementations for digital contents (audio, video, ebooks, software), e.g., Microsoft's Windows Media Rights Management [26], Apple's FairPlay [3] and Real Network's Helix DRM [36], all providing proprietary stateful licenses.

Moreover, most of these solutions are closed software and cannot be verified for inherent security flaws. Some affect the entire host security [24] or violate user privacy [46]. Many could be continuously broken [50, 49], and provide license transfers only to some selected devices owned by the user. This point clearly contradicts the *first sale* doctrine: The licensor should be allowed to transfer legally obtained digital content without permission or interaction of the licensee.

Another approach uses small-value or short-term sub-licenses based on a single source license to transfer rights. Examples are transient licenses [35], rechargeable tokens [17], or tracking files [20]. Since users of these systems always have full control over the platform storage, they can easily backup their (sub-)licenses and restore them after expiration.

In [42, 44], the authors propose an operating system extension that attests an integrity measurement (a SHA-1 digest over all executed content) based on a cryptographic coprocessor. The proposed architecture allows a content provider to remotely verify the integrity of software and data of a client platform. Since this approach measures all executed content, i.e., also all non-security-critical and private content, this procedure gives a content provider user's overall platform configuration. Since delivering the complete platform configuration reveals a lot of additional information not required for license enforcement this would clearly conflict with the least privilege security property, thus affecting users privacy. On the other hand, the content provider can attest always only the last platform configuration given and is not able to predict future configuration. To detect potential replay attacks the content provider would furthermore have to request and store client measurement logs repeatedly. Besides the necessary online connectivity, a client could still apply replay attacks between two measurements.

The *Enforcer* project alias *The Bear* [21, 23] tried to realize freshness using the (non-volatile) *data integrity register* (DIR) of the TCG specification version 1.1b [54]. Writing to a DIR requires owner authorization, reading can be done by anyone. However, this approach cannot be used to enforce stateful licenses since the platform owner can still backup and restore the DIR storage.

New processor architectures like AEGIS [48] and XOM [18] provide secure in-processor storage that cannot be reset by unauthorized entities. Although it seems possible to use these processor architectures as a basis for protecting the freshness of information, we chose another solution which builds on (cheaper) commercial-of-the-shelf components.

2. SYSTEM MODEL AND OBJECTIVES

We start our consideration with an ideal system model for distributed content access[4]. It represents the desired environment in which the security objectives of all involved parties are satisfied by definition. In later sections we go towards real world by replacing this ideal system in Section 3 with several logical components followed by the realization of these components by software components in Section 4.

2.1 Terms and Definitions

We define a *compartment* as a software component that is logically isolated from other software components. The *configuration* of a compartment unambiguously describes the compartment's I/O behavior based on its initial state S_0 and its set of state transactions that convey a compartment from state S_i to state S_{i+1}. Moreover, we distinguish secure, trusted, and plain communication channels between compartments. *Plain channels* transfer data without providing any security property. *Secure channels* ensure confidentiality and integrity of the communicated data as well as the authenticity[5] of the endpoint compartment. *Trusted channels* are secure channels that additionally validate the configuration of the endpoint compartment. Finally, *integrity* of information obtained from a channel or compartment is provided, if any modification is at least detectable.

2.2 Ideal System Model

The main parties involved are *providers* (licensors) and *users* (licensees). We consider a provider as the representative party for rights-holders whereas the user represents consumers of digital content. As depicted in Figure 1 the provider distributes digital content (e.g., software, media files, etc.) and the corresponding license. The *license* defines the *usage-rights* (e.g., copy, play, print, etc.) applicable to the content. *license* represents a certificate issued by an authorized instance (licensor) confirming non-repudiability that certain usage-rights on certain contents are granted to some party[6]. Here, a license describes the usage-rights that its owner holds and the prerequisites to consume (access, use) the contents linked to this license. The user consumes the content according to the license where the consumption is managed by the underlying platform as shown by the dashed lines in Figure 1. In the ideal model the platform is an abstract black box which is trusted by all other parties. The usage-rights can be defined in rights expression languages such as XrML[7] or ODRL[8] and are digitally signed by the licensor. We distinguish two types of licenses, *static licenses* and *stateful licenses.* While the state of a static license remains unmodified when used, that of a stateful license may change during its utilization.

The involved parties have only limited trust in each other. In our ideal system model the platform is fully trusted by both user and provider to act correctly.

In Figure 1 we also made the next step by replacing the abstract platform by two logical compartments the *Trusted Policy Enforcer* and the *Trusted Storage Provider* instead of the platform as an abstract. These compartments are strongly isolated from each other and communicate over trusted channels.

[4]We do not consider payment channels or content distribution details such as content provision or license generation here. System models for complete DRM systems can be found in [12, 38].

[5]A compartment's authenticity could be an alias or a temporary compartment identifier.

[6]A formal treatment of rights, licenses and transactions on rights can be found, e.g., in [1].

[7]www.xrml.org

[8]www.odrl.net

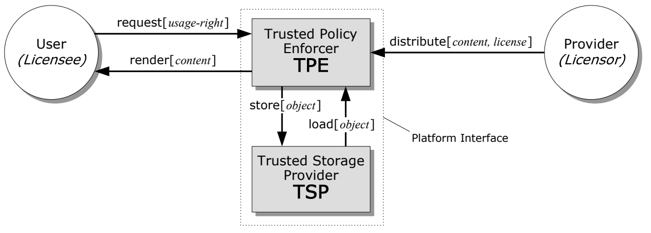

Figure 1: The ideal system model.

The *Trusted Policy Enforcer* (TPE) incorporates the usage-rights management and content rendering. The provider uses the trusted channel distribute[] to transfer content and licenses to TPE. On request of the user via request[], a usage-right is retrieved and the content is rendered according to this right using the trusted output channel render[]. TPE correctly maintains its state using the trusted channels store[] and load[] provided by the trusted storage.

The *Trusted Storage Provider* (TSP) provides the interfaces store[] and load[] to store and load data objects persistently ensuring confidentiality, integrity, availability and particularly freshness. Note, user and provider trust TSP only indirectly, i.e., by establishing a trusted channel to TPE. TPE will only be trusted if it has verified TSP to be trusted by establishing a trusted channel to TSP.[9]

2.3 Security Objectives

In the following we define the overall security objectives of users and providers that our system architecture has to achieve for our distributed content access.

- ***(O1)*** **License integrity:** Unauthorized alteration of licenses must be infeasible. Both user and provider require this, since a license defines a contract between user and provider that must not be altered without mutual approval.
- ***(O2)*** **Licenses unforgeability:** Unauthorized issuance of licenses must be infeasible. Only authorized parties (rights-holder) are allowed to issue new usage-rights. This prevents creation of illegal, unauthorized licenses.
- ***(O3)*** **License enforcement:** The license must be enforced upon acceptance. This is required by the licensor that the user can access and use the content only according to user-rights provided by the license. Otherwise, users could violate a contract with a provider. However, a license should be enforced only when the user as accepted it.
- ***(O4)*** **License availability:** Legally obtained licenses can be used at any time. This especially requires precautions with regard to physical and/or logical failures, as a result of the poor experiences with hitherto existing dongles and smartcards solutions.
- ***(O5)*** **Privacy:** Usage of licenses must not violate privacy policies. The user's privacy policy must be protected when performing transactions on licenses. This includes that the overall system enforces least privilege such that components not under full control of the user, collect, store, and redistribute user's private information only to the extent required for license enforcement and with user's consent.
- ***(O6)*** **Freshness:** Any information obtained should be new, i.e., received or retrieved information is the last one sent or stored.

[9]Since TPE communicates with TSP over a trusted channel, it can be central or distributed, e.g., located at user-side, at provider-side, or realized by a trusted third party TTP.

2.4 Required Security Properties

In order to fulfill the overall security objectives we consider in the following the essential properties which are assumed to be given in the ideal system model (in Section 2.2). In Section 3 and Section 4 we show how one can provide respectively implement these properties.

- ***(R1)*** **Trusted channels:** The underlying platform provides trusted channels between compartments to enable a party to verify a compartment's configuration in order to determine the compartment's trustworthiness.
- ***(R2)*** **Strong isolation:** The underlying platform ensures that compartments are isolated from each other, i.e., compartments cannot access the states of other compartments.
- ***(R3)*** **Trusted storage:** A trusted storage provider TSP provides confidentiality, integrity, freshness and availability of data persistently stored.

2.5 Usage and Transfer of Licenses

Based on our ideal system model and the assumptions, this section describes the general functionality of our architecture with regard to obtaining, usage and transferring of stateful licenses. The main platform components involved in these transactions are illustrated in Figure 1.

Obtaining licenses: The first scheme describes how a license l (and the corresponding content) is obtained by the platform component TPE responsible for enforcing it. We assume that the license negotiation phase has already been completed (outside our model).

1. The user requests the provider the (negotiated) license l from the provider and the respective content (if necessary, i.e., if the user does not already has the protected content).
2. The provider establishes a remote trusted channel to TPE (to verify that the configuration of TPE is conform to his security policy.). Then the provider distributes l and the respective content (if necessary) to TPE.
3. TPE stores l and the corresponding content on TSP using a local trusted channel (thus verifying trustworthiness of TSP.).

Using stateful licenses: The following scheme is an abstract description of how the content and the corresponding license are securely managed by TPE.

1. The user requests TPE for a usage-right on a content.
2. TPE loads from TSP a license l that covers the requested usage-right.
3. If all conditions for the corresponding usage-right are fulfilled, TPE updates the corresponding subset of state variables within l, stores the changed l using TSP again, and invokes the content rendering.

Transferring licenses: We describe a secure transfer protocol of a license (and the corresponding content) between a source platform (transferor) TPE_s to a destination platform (transferee) TPE_d.

1. The user requests TPE_s to transfer a license l to TPE_d.
2. TPE_s establishes a trusted channel to TPE_d to verify that the configuration of TPE_d is conform to the security policy of l needed for a transfer to correctly take place. Note that TPE_d does *not* need this verification for TPE_s since the overall security architecture would not allow a platform to use a license if it does not provide the trust properties required by the licensor.
3. After a trusted channel is established to TPE_d, TPE_s sends l (and the corresponding content) to TPE_d using the previously established trusted channel.
4. After an approved transfer process based on a cryptographic receipt, TPE_s updates or invalidates l, while TPE_d stores the new license (and content) synchronized with TPE_s. In order to handle transmission failures, TPE_s allows arbitrary retransmissions requests to TPE_d while the corresponding usage-right remains invalidated.

The procedure to loan a license is similar to the license transfer: In case the license of allows the user to generate sublicenses TPE_s generates a sublicense l_d of the master license l for TPE_d and invalidates the respective usage-rights in the corresponding master license l locally, i.e, disables the respective usage-rights for the loan period or decreases the respective state variables.

A secure implementation of these protocols is described in Section 3.5.

3. SYSTEM DESIGN

In this section we describe the high-level design of our architecture. The ideal system model in Section 2.2 is decomposed into several smaller compartments. We describe how trusted channels, strong isolation, and trusted storage are realized. Finally, we consider in more detail how an application, i.e., the DRM controller, can obtain, use and transfer a stateful license based on these features.

3.1 Architectural Overview

Figure 2 gives an overview of the compartments into which our ideal system model presented in Section 2.2 can be decomposed. The resulting architecture consists of the trusted computing base (TCB) including a User Manager, a Trust Manager, a Storage Manager, and a Compartment Manager. Another compartment, the DRM Controller, is the example application that uses the TCB to realize the use cases discussed above. Note since all compartments communicate with each other using trusted channels, there are no requirements on their actual physical location.

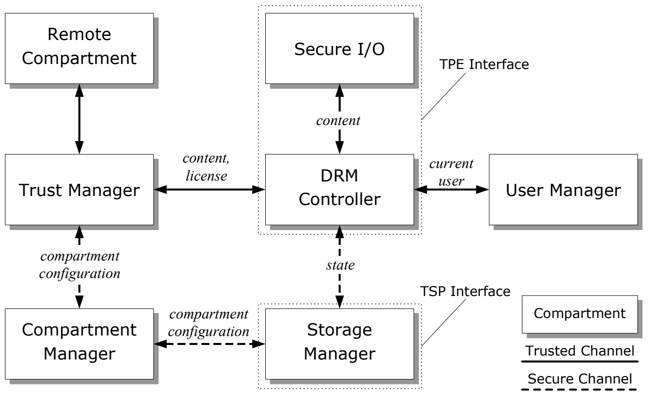

Figure 2: Compartments of the system model.

User Manager: The User Manager (UM) maps between real user names and system-internal user identifiers. Moreover, it performs user authentication and manages a secret attached to each users, e.g., to allow the Storage Manager to bind data to a user. The programming interface offered by the User Manager hides the concrete user model. Thus, it is possible to use a UNIX-like user model, or a role-based model without modifications of other system components.

Storage Manager: The Storage Manager (SM) provides persistent storage for the other compartments while preserving integrity, confidentiality, availability and freshness of the stored data. Moreover it enforces strong isolation by binding the stored data to the compartment configuration and/or user secrets[10]. The Storage Manager has access to the configuration of its clients, since it communicates with them over trusted channels. For a more detailed description of the implementation, see Section 3.3 and Section 3.4.

Compartment Manager: The Compartment Manager (CM) manages creation, update, and deletion of compartments. It controls which compartments are allowed to be installed and enforces the mandatory security policy. During installation of compartments, it derives its configuration to be able to offer a mapping between temporary compartment identifiers[11] and persistent compartment configurations.

Trust Manager: The Trust Manager (TM) offers functions that can be used by application-level compartments to establishing trusted channels between remote and local compartments.

Secure I/O: The Secure I/O (SO) renders (e.g., displays, plays, prints, etc.) content while preventing unauthorized information flow. Thus SO incorporates all compartments

[10]Since SM does not provide sharing of data between compartments, it does not realize a regular file system.
[11]A compartment identifier unambiguously identifies a compartment during runtime.

that are responsible for secure output of content (e.g., drivers, trusted GUI, etc.).

DRM Controller: The DRM controller (DC) is an application that enforces the policy according to the given license attached to digital content. DC enforces security policies locally, e.g., it uses trusted channels to decide whether a certain SO is trusted for rendering the content, i.e., whether it matches the configuration described in the license. More details of the implementation of the DRM controller can be found in Section 3.5.

With the architectural overview in mind, we explain in the following sections how this architecture is used to provide the necessary security properties, i.e., privacy, trusted channels, secure storage, and fresh storage.

3.2 Trusted Channels

According to the definition in Section 2.1, trusted channels allow the involved communication end-points to determine their configuration and thus to derive their trustworthiness. Other integrity measurement architectures, however, [42, 44], report the integrity of the whole platform configuration including *all* currently running compartments to remote parties, and thus violating user privacy. In contrast, our architecture supports to establish trusted channels between single compartments, and not only between platforms whole platforms. This has the following advantages:

- *Privacy:* A remote party now only needs to know the configuration of the appropriate compartment including its trusted computing base, and not the configuration of the whole platform.

- *Scalability:* Remote parties do not have to derive the trustworthiness of all compartments executed on top of the platform, to determine the trustworthiness of the appropriate compartment.

- *Usability:* Since a compartment's trustworthiness can be determined independent of other compartments running in parallel, the derived trustworthiness keeps valid even if the user installs or modifies other compartments.

Trusted channels can be established using the functions offered by the Trust Manager and the Compartment Manager, while the Compartment Manager, which is responsible for installation and manipulation of compartments, provides the mapping from compartment identifiers into configurations. Thus, trusted channels can be established assuming that the TCB including the Compartment Manager and the Trust Manager is trustworthy. In Section 4, we will explain how remote parties can determine the trustworthiness of the TCB, but now we continue describing the establishment of trusted channels on this design level.

We distinguish between trusted channels between compartments running on the same platform (local trusted channels) and trusted channels between a remote and a local compartment (remote trusted channels).

Local Trusted Channels: Since both the sender and the receiver are executed on top of the same TCB, an explicit verification of the TCB's trustworthiness does not make sense in this case. Therefore, trusted channels can easily be established using secure channels offered by the underlying TCB, and the functions provided by the Compartment Manager: The sending compartment first requests the configuration of the destination compartment from the Compartment Manager. On successful validation that the destination configuration conforms to its security policy, the source compartment establishes a secure channel to the destination compartment.

Remote Trusted Channels: The required steps to establish a remote trusted channel from a remote compartment to the local compartment are as follows (cmp. Figure 3):

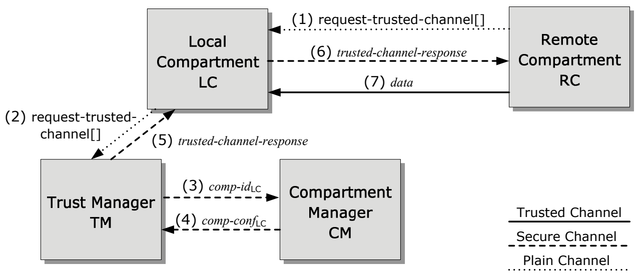

Figure 3: A remote trusted channel depends on local and remote secure channels.

If a local compartment receives a request (1) from a remote compartment, the local compartment requests the Trust Manager (2) to receive a credential including its own configuration. Then the Trust Manager generates the credential based on both the compartment configuration provided by the Compartment Manager (4) and the configuration of the platform's TCB. The resulting credential is returned to the invoking local compartment (5) that forwards it (6) to the remote compartment. That can now verify the trustworthiness of the local compartment and, on success, using the credential to open a trusted channel.

Section 4.2 describes the realization of the TCB credential and Section 4.3 describes the proposed protocol in more detail. Moreover, it shows how to realize the credentials based on X.509 certificates.

3.3 Strong Isolation

In order to strongly isolate compartments from each other, isolation at runtime as well as isolation in persistent storage is required. On this design level, we assume that runtime isolation is provided by the underlying layer (see Section 4.1.2).

Isolation of the persistent states of compartments, however, is provided by the Storage Manager (SM).

The Storage Manager binds all of the compartment's data to the corresponding compartment configuration while preserving integrity and confidentiality. In this context, *bind* means that access to bound data is only possible under the terms defined on storage, e.g., a certain compartment configuration or a user ID.

Since the Storage Manager communicates with its clients over trusted channels, it can be located at user-side, at provider-side, or realized by a trusted third party.[12] Our security architecture uses a local Storage Manager for the following reasons: First, access to the storage is needed every time a stateful license is used. Using a remote storage requires an instant online access and limits the frequency of possible state updates. Second, maintaining a trusted channel to an external storage clearly increases overall complexity and failure probability of the system. An external storage is a single point of failure. A denial of service (DoS) attack, for instance, violates the availability requirement of all stateful licenses.

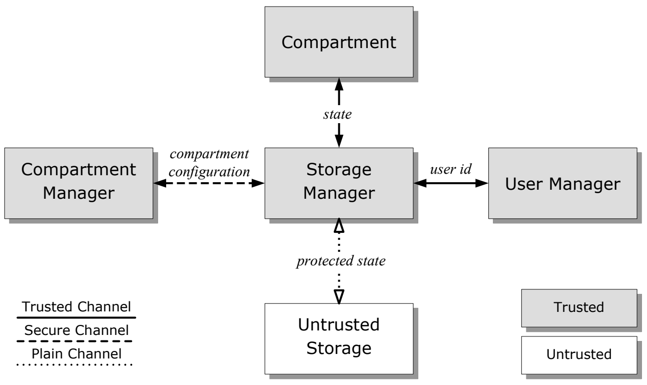

Figure 4: The Storage Manager enforces strong offline isolation.

Figure 4 depicts the involved compartments and dependencies to realize offline isolation of compartments. The Storage Manager uses the Compartment Manager to retrieve the origin compartment configuration, the User Manager to bind compartment's data to a certain user (if requested) and an untrusted storage compartment to persistently write and read plain data[13]. Internally, Storage Manager uses cryptographic functions to preserve confidentiality and integrity of data before it is committed to untrusted storage.

3.4 Trusted Storage

The following section describes how the trusted storage provider TSP can be realized. Providing a completely tamper-resistant trusted storage compartment would clearly raise costs and limit flexibility. Hence, we look for a more efficient approach. To keep the high-level architecture independent of a concrete instantiation of the underlying hardware platform, the design decision is to provide a logical service that protects the freshness of arbitrary data. More concretely, we extended SM that already provides isolated secure storage (see Section 3.3) by a freshness property. In order to realize freshness detection, SM has exclusive access to a small tamper-resistant memory location.

[12]Microsoft's Media Rights Manager [26], for instance, applies the provider-side approach where a local storage client regularly connects to an external *content protection server* to enforce freshness.

[13]For the realization of availability we suggest common solutions based on high redundancy, i.e., utilization of multiple distributed storage locations (e.g., USB sticks or online sites) assisted by an appropriate RAID system. In case of failure of a particular storage device, it is still possible to retrieve data from alternative storage mirrors.

3.5 DRM Controller

The DRM controller DC consists of a license interpreter and a content access arbitration. It is the core component for the secure usage and transfer of licenses (see Section 2.5) where licenses are defined by an XrML license file. All available contents and licenses are internally indexed to provide all necessary information about the available contents and licenses to the user. The index itself, the contents, and the licenses are persistently stored using the Storage Manager (cf. Section 3.3) that enforces the storage security requirements of both user and provider (cf. Section 2.4).

The prerequisites for usage and transfer of licenses is a proper initialization of the platform and the DRM Controller. In the following, we assume that (i) the TCB has been loaded properly, (ii) the Trust Manager contains the appropriate credential, (iii) the DRM Controller has been measured and started by the Compartment Manager, and (iv) the components for mandatory security policy that relate to the DRM Controller are part of its configuration. On startup, the DRM Controller loads its actual content/license index from the Storage Manager over a local trusted channel.

To obtain licenses the provider establishes a remote trusted channel to the DRM Controller, and if successful, the content and the license are sent to the DRM Controller compartment over this channel. The DRM Controller updates its index. On shutdown, it stores the license and the corresponding content using the Storage Manager. Since the communication is performed over a trusted channel, the DRM Controller can verify whether the Storage Manager is trustworthy for to the given license.

For using stateful licenses the user invokes the DRM Controller. An example implementation would be to use a communication client that enables requests from the legacy Linux to the DRM Controller. The DRM controller loads the corresponding license and checks if all conditions for the corresponding usage-rights are fulfilled. It then verifies the trustworthiness of an applicable output device, e.g., the secure user interface, by opening a trusted channel to it. On a successful license coverage, the DRM Controller updates the corresponding subset of state variables within the license, synchronizes its internal state with that stored by the Storage Manager, loads the corresponding content, and invokes the output device to securely render the given content.

For the transfer of stateful licenses, again the user invokes a locally running DRM Controller to transfer a certain license to a remote DRM Controller on the destination platform. The source platform uses the Trust Manager to establish a remote trusted channel to the destination platform to send the license (and the corresponding content) to it. In case this is successful the source platform updates resp. invalidates its index and synchronizes its internal state with the Storage Manager. The destination platform stores the new license (and content) using its own storage Manager.

The security of the realization discussed above depends on certain assumptions, i.e., a secure channel between compartments, compartment isolation during runtime, and the availability of credentials. The following section describes how our architecture provides them.

4. IMPLEMENTATION

In this section we describe details of our implementation. We first give an overview to describe the operational basis providing the security properties demanded in Section 2.4. Furtheron, we briefly explain each the layers our implementation, the initialization process as well as the implementation of the core components, namely the Storage Manager and the DRM Controller.

4.1 Implementation Overview

Our implementation primarily relies on a small security kernel, virtualization technology, and Trusted Computing technology. The security kernel, located as a control instance between the hardware and the application layer, implements elementary security properties like trusted channels and isolation between processes. Virtualization technology enables reutilization of legacy operating systems and present applications whereas Trusted Computing technology serves as root of trust.

Our abstract definitions from Section 2.1 can be mapped to real world implementation. Thus a compartment maps to an application process, while a compartment configuration maps to a software binary including the initial state of all variables and the instruction set.

The more detailed architecture of our realization is depicted in Figure 5. The bottom layer is conventional hardware with additional Trusted Computing (TC) support. Above the hardware layer resides our security kernel consisting of a virtualization layer and a trusted software layer providing sharing of hardware resources and realizing elementary security and management services that are independent and protected from a legacy OS. On top of the security kernel, a para-virtualized legacy operating system (currently Linux) including legacy applications, the DRM controller, and the Secure I/O are executed in strongly isolated compartments running *in parallel* as user processes.

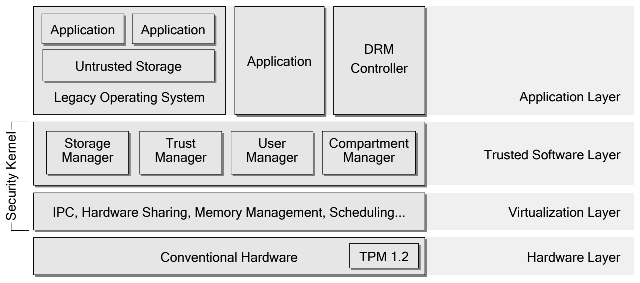

Figure 5: The PERSEUS security architecture.

In the following, we briefly describe each implemented layer in more detail.

4.1.1 Hardware Layer

The hardware layer consists of commercial off-the-shelf PC hardware with additional Trusted Computing technology as defined by the Trusted Computing Group (TCG) [52] in form of a security chip known as Trusted Platform Module (TPM). The TPM is considered to be a tamper-evident hardware device similar to a smart-card and is assumed to be securely bound to a computing platform. It is primarily used as a root of trust for platform's integrity measurement and reporting. During system startup, a chain of trust is established by cryptographically hashing each boot stage before execution. The measurement results are stored protected in *Platform Configuration Registers* (PCRs). Based on this PCR configuration, two basic functions can be provided: *Remote Attestation* allows a TCG enabled platform to attest the current measurement and *Sealing* resp. *Binding* to locally resp. remotely bind data to a certain platform configuration. Our implementation uses such a TCG Trusted Platform Module in the present version 1.2 [53] since previous TPM versions cannot be used to provide fresh storage as we will elaborate on in Section 4.4).

4.1.2 Virtualization Layer

The main task of the virtualization layer is to provide an abstraction of the underlying hardware, e.g., CPU, interrupts, devices, and to offer an appropriate management interface. Moreover, this layer enforces an access control policy based on this resources. The current implementation is based on microkernels[14] of the L4-family [15, 19]. It implements hardware abstractions such as threads and logical address spaces as well as an inter-process communication (IPC). Device drivers and other essential operating system services, such as process management and memory management, run in isolated user-mode processes. In our implementation, we kept the interfaces between the layers generic to support also other virtualization technologies. Thus, the interface offered by the virtualization layer is similar to those offered by virtual machine monitors resp. hypervisors like sHype and Xen [33, 43, 10]. However, we actually decided to employ a L4-microkernel that easily allows isolation between single processes without creating a new full OS instance in each case such as when using Xen.

4.1.3 Trusted Software Layer

The trusted software layer, based on the PERSEUS security architecture [32, 39, 41], uses the functionality offered by the virtualization layer to provide security functionalities on a more abstract level. It provides elementary security properties like trusted channels and strong compartment isolation as well as several elementary management compartments (e.g., I/O access control policy) that realize security critical services independent and protected from compartments of the application layer. The main services of the trusted software layer to enable stateful licenses and license transfers are, as described in Section 3.1, the Trust Manager (cf. Section 4.3), the User Manager, Compartment Manager, and particularly the Storage Manager (cf. Section 4.4).

4.1.4 Application Layer

On top of the security kernel, several instances of the legacy operating system as well as security-critical applications – in our case the DRM controller and Secure I/O – are executed in strongly isolated compartments such that unauthorized

[14]A microkernel is an operating system kernel that minimizes the amount of code running in privileged ("ring 0") processor mode [37].

communication between applications or unauthorized I/O access is prevented.[15] The proposed architecture offers an efficient migration of existing legacy operating systems. We are currently running a para-virtualized Linux [14]. The legacy operating system provides all operating system services that are not security-critical and offers users a common environment and a large set of existing applications. If a mandatory security policy requires isolation between applications of the legacy OS, they can be executed by parallel instances of the legacy operating system.

4.2 Secure Initialization

The security of the whole architecture relies on a secure bootstrapping of the trusted computing base. A TPM-enabled BIOS, the Core Root of Trust for Measurement, measures the integrity of the Master Boot Record (MBR), before passing control to it. A secure chain of measurements is then established: Before program code is executed it is measured by a previously measured and executed component. For this purpose, we have modified the GRUB bootloader[16] to measure the integrity of the core compartments, i.e., the virtualization layer, all compartments interacting directly with the TPM – Compartment Manager, Trust Manager and Storage Manager – as well as the TPM device driver. The measurement results are securely stored in the PCRs of the TPM. All other compartments (including the legacy OS) are subsequently being loaded, verified, and executed by the Compartment Manager according to the effectual platform security policy.

Upon completion of the secure initialization, an authorized compartment (such as the Trust Manager) can instruct the TPM to generate a credential for the Trusted Computing Base. This credential consists of all PCR values reflecting the configuration of the TCB and a key pair which is bound to these PCR values. Together with an I/O access policy management service that is of course also part of the TCB, the private key can only be used by compartments that are both part of the TCB and are authorized to access the TPM.

4.3 Trust Manager

Our implementation of the Trust Manager is based on the open-source TCG Software Stack *TrouSerS* [51]. In order to provide remote trusted channels, the Trust Manager creates on request of a local compartment a private binding key whose usage is bound to the requesting compartment's configuration and the configuration of the platform's TCB (including the Trust Manager itself). The appropriate certificate of the public binding key has to be extended such that remote parties can verify both configurations. To access content that is remotely decrypted with the public binding key, the Trust Manager checks whether the configuration of the compartment that want to use the corresponding private binding key *matches* the configuration of the compartment that has initiated the creation of that binding key. Note that, by extending this 'match' function, one can easily provide property-based attestation/sealing [40, 34, 13] on top of the Trust Manager.

According to Figure 6, in the following, we give a detailed description of the protocol for establishing a remote trusted channel. The protocol can be decomposed into three major steps, namely *certificate generation*, *encryption of a session key*, and *decryption of the session key*.

Certificate Generation: The request of the remote compartment RC for a trusted channel to the local compartment LC reaches TM via LC. After the mapping of LC's compartment identifier to his actual compartment configuration *comp-conf*$_{\mathsf{LC}}$ using CM, TM invokes the TPM to create a asymmetric binding key bound to the actual TCB configuration.[17] The TPM then returns the public binding key PK_{BIND} and the encrypted secret part SK'_{BIND} using TPM's storage root key (SRK). Then TM invokes the TPM to sign over the actual TCB configuration, the binding key, and the configuration of LC using an attestation identity key (AIK).[18] Finally, TM embeds the received TPM certificate within an X.509 certificate for use in the TLS handshake, which will be sent together with PK_{BIND} to RC.

TCB configuration *TCB-conf*
Public binding key PK_{BIND}
Local compartment configuration *comp-conf*$_{\mathsf{LC}}$
TPM Signature = $sign_{AIK}$ (*TCB-conf*, PK_{BIND}, *comp-conf*$_{\mathsf{LC}}$)

Table 1: Structure of the TPM certificate $cert_{BIND}$.

Encryption of Session Key: RC verifies the certificate signature and validates the two embedded configurations *TCB-conf* and *comp-conf*$_{\mathsf{LC}}$ by comparing them with reference values known to be trustworthy. On success, RC encrypts a symmetric session key to *esk* using PK_{BIND} and acknowledges the TLS handshake with *esk*, that can be unbound by *LC* only if it provides the stated compartment and TCB configuration.

Decryption of Session Key: Upon receipt of the encrypted session key *esk*, LC requests TM to unbind the session key. Therefore, TM again maps LC's compartment identifier to his actual compartment configuration *comp-conf*$_{\mathsf{LC}}$ using CM, to validate the compartment configuration stated in the certificate with the one requesting the unbind process. On success, TM invokes the TPM to unbind the session key using the encrypted private part of the binding key SK'_{BIND}. The TPM first compares the actual PCR values with ones SK_{BIND} is bound to, before returning the decrypted session key to TM. TM finally, passes the decrypted session key back to LC which uses it for the completion of the TLS handshake to establish a (one-way) SSL-based trusted channel from compartment RC to LC.

Performance Measurements: We have implemented the described protocol and run it on TPMs of different vendors. The measurement results with maximum asymmetric key lengths (2048 bits) are shown below. Note that the TPM

[15]However, covert channels are still feasible.

[16]`www.prosec.rub.de/tgrub.html`

[17]The actual TCB configuration *TCB-conf* was measured during secure initialization (cf. Section 4.2).

[18]The attestation identity key (AIK) is a non-migratable key that has been attested by a privacy-CA to come from a TCG conform platform. An AIK (in contrast to the general signature key) can be used only to sign other TPM keys or PCR values.

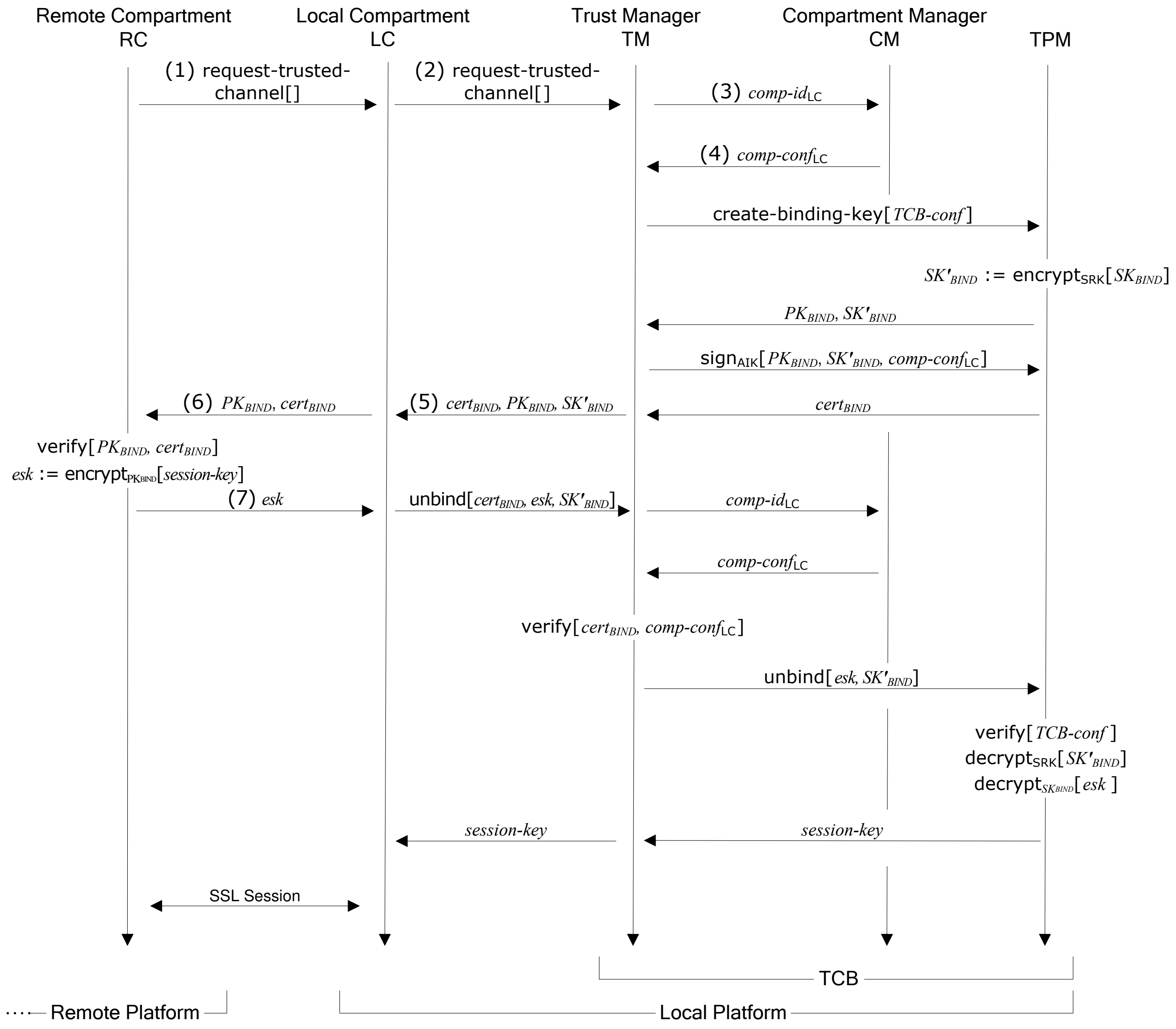

Figure 6: Protocol for establishing a remote trusted channel. The numbers (X) on the arrows refer to the protocol steps of Figure 3.

calculations dominate the overall computation and network transfer times.

	Atmel 1.1b	NSC 1.1b
Certificate generation	30 – 80 s	52 – 55 s
Session key encryption (w/o TPM)	< 1 s	< 1 s
Session key decryption	2 – 3 s	23 – 24 s

Table 2: Trust Manager performance measurement results.

4.4 Storage Manager

The following section describes the implementation of the Storage Manager SM, that enables other compartments to persistently bound their local states to their actual configuration while preserving integrity, confidentiality and freshness. We first give an short overview and then describe the realization of secure storage that will be extended by an additional freshness layer to provide also trusted storage. At the end of this section, we briefly describe the protocols to init SM as well as for storing to and loading from trusted storage using SM.

Overview: The Storage Manager is invoked by a compartment to store a data object persistently preserving confidentiality and integrity – optional with additional restrictions *rest* (e.g., freshness, certain user id). SM invokes the Compartment Manager to retrieve the actual configuration of the respective compartment to bind the data object to that origin compartment configuration *cmp-conf*. As shown in Figure 7, SM creates/updates a metadata entry for the corresponding data object *data* with the data object identifier $data_{ID}$, its freshness detection information f, i.e., the actual cryptographic hash value, and all relevant access restrictions *rest*[19] within its index $index_{\mathsf{SM}}$. SM extends the data object with an integrity verification information, synchronizes its monotonic counter cnt_{SM}, encrypts the data object and the updated index and writes it on untrusted persistent storage using key_{SM}. Since $index_{\mathsf{SM}}$ is the base of security for SM, $index_{\mathsf{SM}}$ is sealed to SM's configuration via the sealed key_{SM}. Thus only the same, trusted Storage Manager configuration is able to unseal and use the key again. On a load request, SM again uses the Compartment Manager to compare the invoking compartment configuration with the one that afore stored the respective data object. On a successful verification, SM reads and decrypts the data object from the untrusted persistent storage and verifies its integrity. Before the data object is committed to the requesting compartment, SM also verifies possibly existing additional restrictions such as freshness or a certain user id.

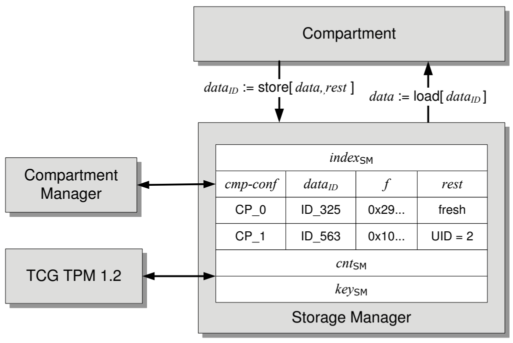

Figure 7: SM's metadata index.

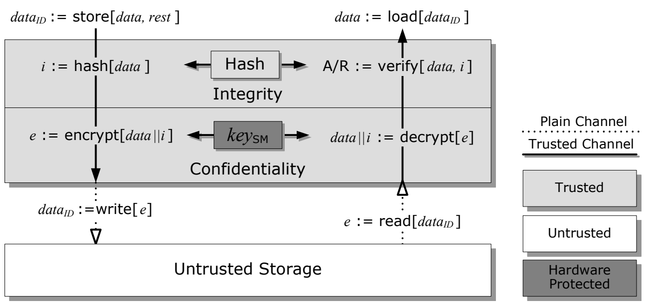

Figure 8: Compartment view of SM's secure storage implementation.

Secure Storage: Figure 8 depicts our secure storage implementation. Thus, our secure storage compartment basically offers two trusted channels namely load[] and store[] while itself uses two *untrusted* channels namely read[] and write[] from an untrusted storage compartment to persistently write respectively read data while providing at least availability.[20] If SM receives a data object *data* via store[*data*, *rest*], SM internally creates or updates object's metadata[21] and calculates its hash value i to verify integrity. Then *data* together with i is encrypted with the internal cryptographic secret key_{SM} using the function $e := \mathsf{encrypt}[data||i]$ (to provide confidentiality). The encrypted data e will afterwards be written on untrusted storage using $data_{ID} := \mathsf{write}[e]$ that returns the object identifier $data_{ID}$. Conversely, if e is read from the untrusted storage via $e := \mathsf{read}[data_{ID}]$ it will be decrypted to *data* and i via $\mathsf{decrypt}[e]$ using the internal cryptographic secret key_{SM}. Before returning *data* to load[], SM verifies the integrity of *data* and further access restrictions (e.g., a certain user id) based on the corresponding metadata in SM's index using the function $\mathsf{verify}[data, \mathrm{i}]$.

Trusted Storage: In order to provide trusted storage, we enhance SM by an additional layer for managing freshness of data objects. Figure 9 depicts SM's extension where the (currently abstract) function $f := \mathsf{memorize}[data]$ updates the internal data structure *FRESH* with the freshness value f. Afterwards, *data* will be stored persistently ensuring confidentiality and integrity using secure storage. On load from secure storage, the function $\mathsf{verify}[data, \mathrm{f}]$ additional verifies that the received data object *data* is the last one being stored.

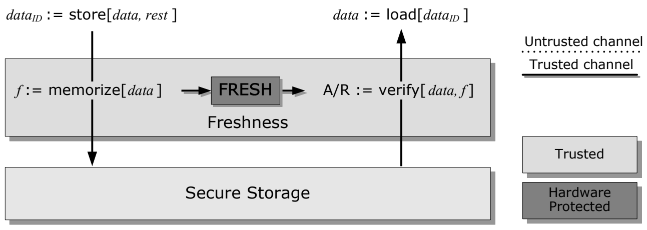

Figure 9: Compartment view of SM's trusted storage implementation.

To provide such freshness detection, SM uses an additional metadata field (cf. Figure 7) to store the cryptographic hash value $Hash(data)$ that defines the last stored version of *data*. On load, SM calculates $Hash(data)$ again and checks if it matches the hash value on last store. In order to ensure freshness of these metadata, the index of SM itself has to be stored fresh.

We therefore analyzed to what extend TPMs of version 1.1b and 1.2 can be used to realize a fresh index for SM.

- *DI-Register:* TPMs version 1.1b provide a Data Integrity Register (DIR) that can persistently store a 160 bit value [21, 23]. Unfortunately, access to this register is only authorized by the TPM-Owner secret implying that the TPM-Owner can always perform replay attacks. The only solution would be to distribute platforms with an activated TPM and an owner authorization secret that is unknown to the user. This solution does not conform to the TCG specification that demands that TCG-enabled platforms have to be shipped with no owner installed (see [54], page 139).
- *SRK Recreation:* An alternative way to prevent replay attacks based on TPMs version 1.1b would be to create

[19]Further access restrictions can be a certain user id, group id or date of expiry.

[20]For the realization of availability we suggest solutions based on high redundancy, i.e., by the utilization of multiple distributed storage locations (e.g., USB sticks or online sites) assisted by an appropriate RAID system. In case of failure of a particular storage device, it is still possible to retrieve data from alternative storage mirrors.

[21]More details on storage metadata at the end of this section.

a new Storage Root Key (SRK) before the system is shut down. Recreation of the SRK would prevent that previously created TPM encryption keys can be used any more. Unfortunately, a SRK can only be renewed by the TakeOwnership function which itself requires a previously OwnerClear that itself disables the TPM. Therefore, an online-recreation of the SRK seems to be impossible.

- *NV-RAM:* TPMs version 1.2 provide a limited amount of non-volatile (NV-) RAM to which access is restricted to authorized entities. So-called NV-Attributes define which entities are authorized to write to and/or read from the NV-RAM. Thus, data integrity can be preserved by storing a hash value of the data into the NV-RAM and ensuring that only the Storage Manager can access the authorization secret.

- *Secure Counter:* A TPM version 1.2 supports at least four monotonic counters. Based on this functionality, the freshness of data can be detected by securely concatenating it with the actual counter value.

A result of our previous analysis we showed that TPMs version 1.1b cannot be used to provide fresh storage as required to enforce stateful licenses and/or to transfer licenses. Therefore we decided to realize trusted storage based on the monotonic counter functionality of TPMs version 1.2.

A monotonic hardware counter allows us to securely maintain versioning of an arbitrary data component, by keeping a software counter synchronized with one (of four guaranteed) hardware counters of the TPM. As depicted in Figure 7, SM manages an internal software counter that, every time SM updates its index, is incremented synchronously with the monotonic hardware counter. If both mismatch at any time, a outdated data is detected, that will be handled according to the actual security policy.

However, in order to employ TPM's monotonic counters, SM has to be initialized correctly. Figure 10 depicts the steps needed for the first initialization of SM on a new platform together with the initialization necessary for instance after rebooting the platform. On the initial setup SM uses the TPM to create its internal cryptographic key key_{SM} that then will be sealed to the actual platform configuration. To enable freshness detection and thus trusted storage, SM creates a monotonic counter *cntid* with a authentication *auth*, e.g., a secret password. The initial setup finishes with the creation of SM's internal metadata index $index_{\mathsf{SM}}$ and the saving of the sealed key blob and the encrypted index on untrusted storage.

After a platform reboot, SM reads the key blob from the untrusted storage and asks the TPM to unseal its internal key. The TPM is able to unseal key_{SM} if the platform has the same configuration as it was at the sealing process, thus preventing a modified SM to access key_{SM}. Then SM uses key_{SM} to decrypt its metadata index read from the untrusted storage. Finally, SM verifies freshness of $index_{\mathsf{SM}}$ by comparing the decrypted counter of $index_{\mathsf{SM}}$ with the actual counter value of the corresponding TPM counter *cntid*.

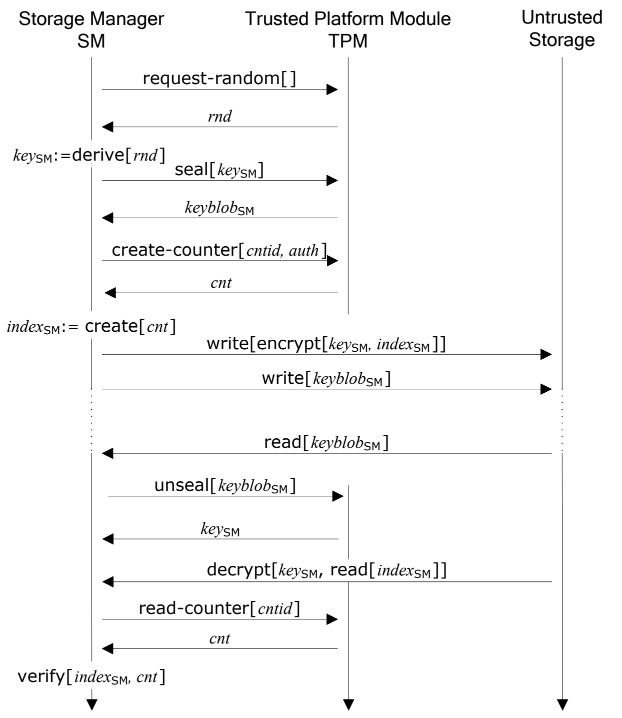

Figure 10: Protocol view of SM's initialization.

Figure 11 depicts the protocol steps required to bind a compartment's data object persistently to its actual configuration. After the mapping of compartment identifier to the actual compartment configuration using CM, SM updates $index_{\mathsf{SM}}$ with the corresponding metadata as well as the incremented software counter to enable freshness detection for $index_{\mathsf{SM}}$. Afterwards, SM writes both, the data objects and the updated index, encrypted on the untrusted storage using key_{SM}. Finally, SM synchronizes its software counter with the TPM's monotonic hardware counter and returns the data object identifier.

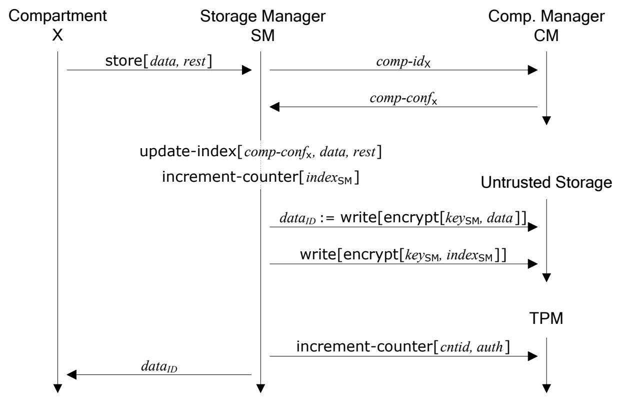
Compartment X
Storage Manager SM
Comp. Manager CM
store[data, rest]
comp-idX
comp-confX
update-index[comp-confX, data, rest]
increment-counter[indexSM]
Untrusted Storage
dataID := write[encrypt[keySM, data]]
write[encrypt[keySM, indexSM]]
TPM
increment-counter[cntid, auth]
dataID

Figure 11: Protocol view of SM's store implementation.

We complete the scenario with Figure 12 that depicts the protocol steps required to load a compartment's data object again. After the mapping of requesting compartment identifier to the actual compartment configuration using CM, SM reads the requested data object from untrusted storage and decrypts it using key_{SM}. Before returning *data* to the corresponding compartment, SM verifies all access restrictions (e.g., freshness, or a certain user id) given on store via *rest* based on the corresponding metadata in $index_{\mathsf{SM}}$ and verifies that the requesting compartment has the same configuration as it was on store.

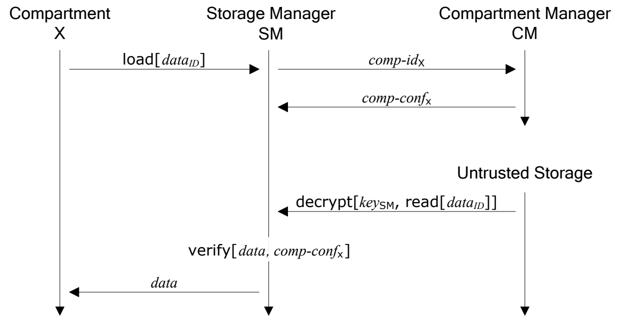

Figure 12: Protocol view of SM's load implementation.

4.5 Secure I/O

The Secure I/O compartment receives protected content in plain for rendering. Thus the SO is a security critical compartment that has to be trusted by user and provider. Therefore SO is executed in parallel, isolated from the actual legacy OS and has to be verified by the DRM controller for trustworthiness. In order to provide a flexible efficient implementation, we used a para-virtualized Linux operating system reduced to the essential functionality to render the decrypted content[22] from DC. Moreover, our whole system architecture enforces that SO is allowed to communicate only with devices essential for the rendering process and in turn receives communication only from DC so that decrypted content cannot leak into untrusted compartments.

5. SECURITY CONSIDERATIONS

In this section we first show why our implementation realizes the security properties *(R1) – (R3)* demanded in Section 2.4. We then shortly analyze how we can fulfill the overall security objectives *(O1) – (O6)* demanded in Section 2.3.

5.1 Trusted Channels

The inter-process communication (IPC) provided by the virtualization layer enables secure channels between local compartments that enforce confidentiality and integrity protection. To provide secure channels also between local and remote compartments, we suggest the application of common established mechanisms for communication security such as SSH [58] and TLS [9]. In order to extend secure channels to trusted channels that enable a party to verify a compartment's configuration, we have implemented the Trust Manager (TM) and the Compartment Manager (CM). Both together allow local and remote compartments to determine the configuration of their communication contacts and thus to derive their trustworthiness. Moreover, our architecture enforces that information bound to the determined configuration cannot be accessed by an unauthorized (and potentially untrusted) configuration based on the TCG mechanisms sealing and binding. The secure initialization process (cf. Section 4.2) however enforces the trustworthiness of TM, CM and the underlying TCB.

5.2 Strong Isolation

In order to strongly isolate compartments from each other, isolation at runtime as well as isolation in persistent storage is required. Runtime isolation is provided by the small virtualization layer that implements only logical address spaces, inter-process communication (IPC) and an appropriate interface to enforce an access control management for the underlying hardware. Device drivers and other essential operating system services, such as process management and memory management, run in isolated user-mode processes. Thus, the amount of code running in privileged ("ring 0") processor mode, is clearly minimized and can, in contrast to monolithic operating system kernels such as Linux or MS Windows, properly be verified for its correctness. Moreover, a failure in one of these services cannot directly affect the other services, especially the code running in privileged mode. Thus, malicious device drivers cannot compromise core operating system services as they are all executed in user-mode. Isolation in persistent storage is provided by our Storage Manager (SM) implementation and the usage of trusted channels. Since conventional computer architectures cannot provide a trusted channel to the persistent storage device, an adversary can always arbitrarily change the state of the storage or access the communication to and from the corresponding controller. We prevent such offline manipulations and replay attacks while establishing a trusted channel to SM during the secure initialization (cf. Section 4.2) process that enables the platform to verify the trustworthiness of SM.

5.3 Trusted Storage

Our architecture provides secure storage, i.e., storage providing integrity and confidentiality, using established cryptographic mechanisms. However, we improved common approaches while taking advantage of the strong isolation capability of our architecture that prevents the exposure of cryptographic secrets to unauthorized or malicious processes. We also extended the secure storage by a hardware-based freshness detection mechanism that detects outdated persistently stored information, i.e., information that indeed could be decrypted and verified for integrity but that was not the information written at last. Having a freshness detection mechanism for persistent storage, our architecture is able to manage for instance stateful licenses while preventing the corresponding replay attacks. In order to provide Trusted Storage, i.e., storage that enables other compartments to persistently bound their local states to their actual configuration while preserving integrity, confidentiality and freshness, we employ the Storage Manager (SM) together with the Compartment Mananger (CM). As SM innately enforces integrity, confidentiality and freshness, CM provides trustworthy measurement of compartments configuration used by SM to return information requested on load only to compartments with the same configuration as provided on the preceding storage request. The secure initialization (cf. Section 4.2) however, again enforces the trustworthiness of SM,

[22]Our exemplary SO implementation provides rendering of several audio formats

CM and the underlying TCB.

5.4 Security Objectives

In the following we shortly analyze how we fulfill the overall security objectives *(O1) – (O6)* of users and providers demanded in Section 2.3 using the implemented security properties *(R1) – (R3)*.

(O1) **License integrity:** Trusted channels ensure that only mutually trusted compartments can modify a license, whereas strong isolation and trusted storage prevents unauthorized alteration of licenses at runtime and while persistently stored.

(O2) **Licenses unforgeability:** Trusted channels enable a party to verify a compartment's configuration for correct sublicensing and license transfers.

(O3) **License enforcement:** Trusted channels enable a party to verify a compartment's configuration for correct license enforcement whereas strong isolation prevents malicious impacts on the license enforcing compartment.

(O4) **License availability:** Using trusted storage ensures availability of licenses.

(O5) **Privacy:** User's privacy is realized by ensuring that the security policy defined by the platform owner restricts the I/O behavior of every application. Thus, even if third party applications, like the DRM controller, can locally enforce their own security policy, they cannot bypass the platform's security policy while accessing non authorized information or devices. At the same time we ensure that the platform owner cannot directly access the state of applications to bypass security policies locally enforced by the applications. Moreover, our architecture is able to attest (resp. seal to) single compartments such that content providers only know the configuration of the TCB and their DRM controller. The controller itself can check if all other required services such as user management, local policy enforcement, or storage are according to the content provider's security policy. While not delivering the user's overall platform configuration to the content provider, our approach reveals only information essential to attest the DRM controller and thus ensuring the security property of least privilege. The user in turn can locally attest[23] the DRM controller and enforce that it cannot reveal any additional information not essential for license enforcement. A possible extension would be to add the concept of property-based attestation [40] to the Trust Manager and the Compartment Manager to hide both the (binary) configuration of the TCB and the DRM controller.

(O6) **Freshness:** Using trusted storage ensures freshness of arbitrary information, i.e., trusted storage ensures retrieved information is the last one stored.

[23]The user can attest third party applications for instance by comparing the attestation results with known good values provided by an institution trusted by the user that has sufficient expertise and possibly further resources.

6. SUMMARY

In this paper, we introduced the design, the realization and implementation of an open security architecture that is capable to enforce stateful licences on open platforms. Particularly, it allows the transfer of stateful licences, while preventing replay attacks. Further, the security architecture provides security properties such as strong isolation that are used to enforce the user's policy, e.g., protecting against spyware. We have shown how to implement this security architecture by means of virtualization technology, an (open source) security kernel, trusted computing functionality, and a legacy operating system (currently Linux).

7. REFERENCES

[1] Adelsbach, A., Sadeghi, A.-R., and Rohe, M. Towards multilateral secure digital rights distribution infrastructures. In *ACM DRM 2005* (2005).

[2] Anderson, R. J. *Security Engineering — A Guide to Building Dependable Distributed Systems.* 2001.

[3] Apple Computer, Inc. FairPlay DRM. `www.apple.com/itunes/`.

[4] Aura, T., and Gollmann, D. Software license management with smart cards. In *Proceedings of the First USENIX Workshop on Smartcard Technology* (Chicago, Il, May 1999), USENIX. ISBN 1-880446-34-0.

[5] Authentica, Inc. Authentica active rights management. `www.authentica.com/products/overview.aspx`.

[6] Baek, K.-H., and Smith, S. W. Preventing theft of quality of service on open platforms. *IEEE/CREATE-NET Workshop on Security and QoS in Communications Networks* (September 2005).

[7] Blaze, M. A cryptographic file system for UNIX. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security* (New York, NY, USA, 1993), ACM Press, pp. 9–16.

[8] Deutsche Telekom AG. T-online: Video on demand. `http://vod.t-online.de`.

[9] Dierks, T., and Allen, C. RFC2246 - the TLS protocol version 1.0. `www.ietf.org/rfc/rfc2246.txt`, January 1999.

[10] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[11] Epsilon Squared, Inc. InstallRite Version 2.5. `www.www.epsilonsquared.com`.

[12] Guth, S. *A Sample DRM System.*, vol. 2770 of *LNCS.* Springer, 2003, pp. 150–161.

[13] Haldar, V., Chandra, D., and Franz, M. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*

(May 2004). also Technical Report No. 03-20, School of Information and Computer Science, University of California, Irvine; October 2003.

[14] Hohmuth, M. Linux-Emulation auf einem Mikrokern. Master's thesis, Dresden University of Technology, Dept. of Computer Science, 1996.

[15] Jaeger, T., Liedtke, J., Panteleenko, V., Park, Y., and Islam, N. Security architecture for component-based operating systems. In *8th ACM SIGOPS European Workshop* (Sintra, Portugal, Sept. 1998).

[16] Koenen, R., Lacy, J., MacKay, M., and Mitchell, S. The long march to interoperable digital rights management. *Proceedings of the IEEE 92*, 6 (June 2004), 883–897.

[17] Kwok, S. H. Digital rights management for the online music business. *SIGecom Exch. 3*, 3 (2002), 17–24.

[18] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)* (Cambridge, MA, USA, Nov. 2000), ACM Press, pp. 168–177. Appeared as 34.5.

[19] Liedtke, J. Towards real microkernels. *Communications of the ACM 39*, 9 (September 1996), 70–77.

[20] Liu, Q., Safavi-Naini, R., and Sheppard, N. P. A license-sharing scheme in digital rights management. Tech. rep., Cooperative Research Centres - Smart Internet Technology, Australia, 2004.

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

[22] Marchesini, J., Smith, S., Wild, O., Barsamian, A., and Stabiner, J. Open-source applications of TCPA hardware. In *20th Annual Computer Security Applications Conference* (Dec. 2004), ACM.

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

[24] Mark's Sysinternals Blog. Sony, rootkits and digital rights management gone too far. `www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html`, October 2005.

[25] Microsoft Corporation. Visio 2003 30-day software trial. `www.microsoft.com/office/visio/prodinfo/trial.mspx`.

[26] Microsoft Corporation. Windows media rights manager 10. `www.microsoft.com/windows/windowsmedia/drm/default.aspx`.

[27] Microsoft Corporation. Windows rights management services. `www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx`.

[28] Microsoft Corporation. Intoduction to Network Access Protection. `http://www.microsoft.com/windowsserver2003/techinfo/overview/napoverview.mspx`, June 2004. Updated April 2005.

[29] Movielink, LLC. Movielink video on-demand service. `www.movielink.com`.

[30] National Research Council. *The Digital Dilemma, Intellectual Property in the Information Age*. National Academy Press, 2000.

[31] Office, U. C. Copyright law of the United States of America. Title 17 of the United States Code U.S.C, June 2003.

[32] Pfitzmann, B., Riordan, J., Stüble, C., Waidner, M., and Weber, A. The PERSEUS system architecture. Tech. Rep. RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, Apr. 2001.

[33] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[34] Poritz, J., Schunter, M., Van Herreweghen, E., and Waidner, M. Property attestation—scalable and privacy-friendly security assessment of peer computers. Tech. Rep. RZ 3548, IBM Research, May 2004.

[35] Pruneda, A., and Travis, J. Metering the use of digital media content with Windows Media DRM 10. `msdn.microsoft.com/library/en-us/dnwmt/html/meteringcontentusage10.asp`.

[36] RealNetworks, Inc. Helix DRM. `www.realnetworks.com/products/drm/`.

[37] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium* (Denver, Colorado, Aug. 2000), USENIX.

[38] Rosenblatt, W., Mooney, S., and Trippe, W. *Digital Rights Management: Business and Technology.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[39] Sadeghi, A.-R., and Stüble, C. Bridging the gap between TCPA/Palladium and personal security. Tech. rep., Saarland University, Germany, 2003.

[40] Sadeghi, A.-R., and Stüble, C. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *The 2004 New Security Paradigms Workshop* (Virginia Beach, VA, USA, Sept. 2004), ACM SIGSAC, ACM Press.

[41] SADEGHI, A.-R., STÜBLE, C., AND POHLMANN, N. European multilateral secure computing base - open trusted computing for you and me. 548–554.

[42] SAILER, R., JAEGER, T., ZHANG, X., AND VAN DOORN, L. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington, DC, USA, Oct. 2004), ACM Press.

[43] SAILER, R., VALDEZ, E., JAEGER, T., PEREZ, R., VAN DOORN, L., GRIFFIN, J. L., AND BERGER, S. sHype: Secure hypervisor approach to trusted virtualized systems. Tech. Rep. RC23511, IBM Research Division, Feb. 2005.

[44] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004), USENIX.

[45] SHAPIRO, W., AND VINGRALEK, R. How to manage persistent state in DRM systems. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management* (London, UK, 2002), vol. 2320 of *LNCS*, pp. 176–191.

[46] SPIEGEL ONLINE. Datenschutzproblem: itunes funkt nach hause. `www.spiegel.de/netzwelt/politik/0,1518,394740,00.html`, January 2006.

[47] STARZ ENTERTAINMENT GROUP. Video download service for portables. `www.vongo.com`.

[48] SUH, G., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the Annual USENIX Technical Conference* (2003).

[49] THE HYMN PROJECT. Free your itunes music store purchases from their drm restrictions. `www.hymn-project.org`, May 2006.

[50] THE REGISTER. Dvd jon hacks media player file encryption. `www.theregister.co.uk/2005/09/02/dvd_jon_mediaplayer/`, October 2005.

[51] TROUSERS. The open-source TCG software stack. `http://trousers.sourceforge.net`.

[52] TRUSTED COMPUTING GROUP. `www.trustedcomputinggroup.org`.

[53] TRUSTED COMPUTING GROUP. TPM main specification. Main Specification Version 1.2 rev. 85, Trusted Computing Group, Feb. 2005.

[54] TRUSTED COMPUTING PLATFORM ALLIANCE (TCPA). Main specification, Feb. 2002. Version 1.1b.

[55] TYGAR, J., AND YEE, B. Dyad: a system using physically secure coprocessors. Tech. rep., 1993.

[56] VINGRALEK, R., MAHESHWARI, U., AND SHAPIRO, W. TDB: A database system for digital rights management. Technical Report STAR-TR-01-01, STAR*Lab, InterTrust Technologies Corporation, 2001.

[57] WRIGHT, C., MARTINO, M., AND ZADOK, E. Ncryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference* (2003), pp. 197–210.

[58] YLONEN, T., KIVINEN, T., SAARINEN, M., RINNE, T., AND LEHTINEN, S. IETF draft - SSH transport layer protocol. `www.openssh.org/txt/draft-ietf-secsh-transport-14.txt`, March 2002.

[59] ZDNET. Trail license filter. `http://downloads.zdnet.com`.