Enabling Fairer Digital Rights Management with Trusted Computing[⋆]

Ahmad-Reza Sadeghi, Marko Wolf
Horst-Görtz-Institute for IT-Security, Ruhr-University Bochum, Germany
[sadeghi,mwolf]@crypto.rub.de

Christian Stüble
Sirrix AG security technologies, Germany
stueble@sirrix.com

N. Asokan, Jan-Erik Ekberg
Nokia Research Center, Helsinki, Finland
[n.asokan,jan-erik.ekberg]@nokia.com

Abstract. Today, digital content is routinely distributed over the Internet, and consumed in devices based on open platforms. However, on open platforms users can run exploits, reconfigure the underlying operating system or simply mount replay attacks since the state of any (persistent) storage can easily be reset to some prior state. Faced with this difficulty, existing approaches to Digital Rights Management (DRM) are mainly based on preventing the copying of protected content thus protecting the needs of content providers. These inflexible mechanisms are not tenable in the long term since their restrictiveness prevents reasonable usage scenarios, and even honest users may be tempted to circumvent DRM systems.

In this paper we present a security architecture and the corresponding reference implementation that enables the secure usage and transfer of stateful licenses (and content) on a virtualized open platform. Our architecture allows for openness while protecting security objectives of both users (flexibility, fairer usage, and privacy) and content providers (license enforcement). In particular, it prevents replay attacks that is fundamental for secure management and distribution of stateful licenses. Our main objective is to show the feasibility of secure and fairer distribution and sharing of content and rights among different devices. Our implementation combines virtualization technology, a small security kernel, trusted computing functionality, and a legacy operating system (currently Linux).

Keywords. Trusted Computing, security architectures, stateful licenses

[⋆] Full version appears as a technical report HGI-TR-2007-002 in [24].

1 Motivation

Timo was about to board a train home when he noticed an advertisement for a wireless kiosk selling the first album from a new band. He took out his music phone, connected the kiosk which was already visible in his music gallery application, and with a few clicks downloaded a preview copy of the lead song in the album. While on board the train, Timo listened to the song and liked it so much that he listened to it once more. When he tried to listen a third time, the phone told him that he had finished the free previews, but can buy a full license. He bought the full license with a few more clicks and could listen to the song with no constraints. When he got home, he transferred the song to his home stereo system. When Anna visited Timo, he played the new song to her. She wanted a copy of her own. Timo used the remote control of his stereo system to lend a copy to Anna's music phone for a week. Timo's copy of the song remained disabled for a week while Anna was enjoying the song.

This and other similar scenarios for trading and using digital goods involve policies whose enforcement requires the enforcement mechanism to securely maintain state information about past usage or environmental factors. They can be enforced by using *stateful licenses.* Some e-business applications already deploy such (mostly proprietary) stateful licences to sell certain digital goods (online video, music tracks, software), for limited use (number of copies or trials, etc.) [2, 17, 29].

However, managing and enforcing stateful licenses on open platforms is difficult. 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. Existing enforcement mechanisms have been defeated in various ways [32, 33]. An attacker can easily record the platform state (e.g., hard-disks) and revert the platform to this state at a future point in time. This way he can reset a stateful license to a prior state and consequently circumvent license conditions. This can be done for instance by ordinary backup mechanisms or by applying software tools [9] that log all storage modifications to easily revoke these modifications for reuse of a license. Consequently, content providers tend to provide inflexible static licenses, which prevent users from any kind of transfer of licenses, including moving to other devices, lending or selling to other users. This approach is not tenable in the long term because its restrictiveness prevents reasonable usage scenarios like the above from being realized. Even honest users, frustrated at not being able to do what they consider reasonable, will be tempted to circumvent the license enforcement mechanisms.

Some systems attempt to augment open platforms with tamper-resistant hardware devices such as dongles [7] or smartcards [4]. Others have used closed systems [12] that consider only the providers needs The use of additional, external devices, however, cannot guarantee the integrity of the operating system and a proper behavior of applications since manipulations of the operating system or corresponding applications frequently allow users to bypass the security mechanisms.

Main Contribution: In this paper we present a security architecture and the corresponding reference implementation that enables secure enforcement of stateful licenses on open computing platforms as well as secure license transfers among platforms. Our proposed architecture allows for protecting the security objectives of providers (license enforcement) and users (flexibility, fairer usage, and privacy). Our main goal is to show the feasibility of the legal and fairer usage allowing for transfer of licenses. 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 that allow to establish multilateral security. We show how our architecture can efficiently be implemented using existing virtualization and trusted computing technology.

1.1 Related Work

Shapiro and Vingralek [27] 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 and secure one-way counters realized by battery-backed SRAM or special on-chip EEPROM/ROM functions. Tygar and Yee [37] elaborate on enforcement of static and dynamic licenses without centralized servers. They present a secure bootstrap process and protocols for sealing of data to a local and remote platforms. The proposed architecture relies on a microkernel which is running in a *physical* security partition provided by a secure coprocessor. This is different to our approach which is based on a virtualization layer offering *logical* security partitions ("compartments").

Marchesini et al. [15] 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 and binding of 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 minor and the overall security improves, since the virtualization layer is much less complex than a monolithic OS kernel. Baek and Smith [6] build on Marchesini's work and implement a prototype for enforcing QoS policies on open platforms.

Publicly available documentation for common rights management systems from Microsoft [18], Authentica [5], or, Apple [2] do not mention how they resist replay attacks for their (proprietary) dynamic license implementations. Moreover, most of these solutions are closed software and cannot be verified for inherent security flaws. Some existing solutions affect the entire host security or violate user privacy [22], while others could be broken [32, 33], and provide license transfers only to some selected devices. This point clearly contradicts the *first sale* concept: the licensor should be allowed to securely transfer legally obtained digital content without permission or interaction of the licensee. Other approaches [14, 20] use small-value or short-term sublicenses based on a single source license to transfer rights. 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 [26], 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. However, this approach, reveals the user's overall platform configuration to the content provider, conflicting with the privacy principle of *least information*. Also, the content provider will only attest the last platform configuration given and is not able to predict future configuration. And even if periodic attestation was compelled, a client could still apply replay attacks between two measurements.

The *Enforcer* project [16] considers freshness by using the (non-volatile) *data integrity register* (DIR) of the TCG (Trusted Computing Group) specification version 1.1b [36]. Writing to a DIR requires owner authorization, reading can be done by anyone. Since the platform owner can still backup and restore the DIR storage, this is not secure against replay attacks.

2 System Model

2.1 Terms and Definitions

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. These parties have only limited trust in each other. As shown in Figure 1, the provider distributes digital content (e.g., software, media files) together with the corresponding *license*, which defines the *usage-rights* (e.g., copy, play, print) applicable to the content. The user consumes content according to the license where the consumption is managed by the underlying platform. We distinguish two types of licenses, immutable *static licenses* and *stateful licenses* where the internal license state may change when it is used. This allows for many use cases where content consumption is somehow limited (e.g., n days or n times), or for transfer of licenses among devices.

Furthermore, we define a *compartment* as a software component that is logically isolated from other software components. *Isolation* means that these components can communicate or access each others data only over specified interfaces. The *configuration* of a compartment unambiguously describes the compartment's I/O behavior. We call the process of deriving the configuration of a compartment *measurement* according to a well-defined metric. We distinguish secure and trusted communication channels between compartments. *Secure* channels ensure confidentiality and integrity of the communicated data as well as the authenticity of the endpoint compartment. A *trusted* channel is a secure channel that is bound to the configuration of the endpoint. More concretely, the channel additionally allows each endpoint compartment to (i) validate the configuration of the other endpoint compartment and (ii) to bind data to the configuration of the endpoint compartment such that solely and exclusively this compartment with this configuration can access the data. We define the *Trusted Computing Base* (TCB) as the set of all system components whose failure would allow to breach the security policy defined for the platform (e.g., as agreed by the involved parties). Note that the main design goal is to minimize the TCB.

2.2 Architecture Overview

Figure 1 gives a general overview of our architecture. The Trusted Computing Base (TCB) for our purpose (application) includes the following compartments: the Trust Manager (TM), the Storage Manager (SM), the Compartment Manager (CM), the Secure I/O (SO) compartment, and the DRM Controller (DC). Note that these components are in general distributed since all compartments communicate over trusted channels, and hence, there is no restriction on their actual physical location. In the following we briefly describe the compartments and core security properties of our architecture.

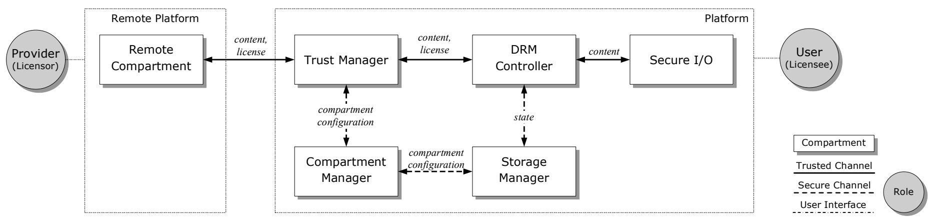


Fig. 1. System architecture.

Compartment Manager (CM) initializes and closes compartments as well as *measures* compartments' configurations during initialization. Furthermore, CM enables a mapping between temporary compartment identifiers[1] and persistent compartment configurations.

Trust Manager (TM) offers basic trusted computing services and a functionality that can be used by other compartments to, e.g., establish trusted channels between compartments.

Storage Manager (SM) provides persistent storage for other compartments while preserving integrity, confidentiality, authenticity (by binding data to the compartment configuration and/or user secrets), and freshness of the stored data. Since a complete tamper-resistant storage unit would be very costly and inflexible, we used untrusted storage, i.e., a regular harddisk, with the help of TM.

Secure I/O (SO) renders (e.g., displays, plays, prints) content while preventing content leakage into untrusted compartments. Thus SO incorporates all functionality required for rendering a certain content, e.g., all corresponding drivers,

[1] A compartment identifier unambiguously identifies a compartment during runtime.

rendering engines, and decoders. Moreover, access control in our architecture allows SO to communicate only with devices essential for the rendering process.

DRM Controller (DC) is a compartment that enforces the policy according to a 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.[2] DC interprets the license and initiates content rendering. Moreover, DC is the core component for license transfers (cf. Section 2.3). Available content and licenses are internally indexed by DC while the index, content and licenses are persistently stored using the Storage Manager SM.

Trusted Channels as mentioned in Section 2.1, allow the involved communication end-points (compartments) to validate the configuration of the other endpoint for integrity and consequently allow determining the trustworthiness (as specified by the underlying security policy). The data sent over a trusted channel is exclusively bound to the configuration of the endpoint compartment as measured by the CM. In contrast to other approaches such as [26], which report the whole platform configuration, our architecture provides trusted channels between single compartments reducing the amount of information disclosed about the platform (privacy aspects).[3] Trusted channels can be established using the functionality offered by the Trust Manager TM. Note, we call trusted channels between compartments running on the same platform as *local* trusted channels.

Strong Isolation means runtime isolation of compartments as well as data isolation in persistent storage. Runtime isolation is provided by the underlying virtualization layer (cf. Section 3.1), and the isolation of compartments' persistent state is provided by the Storage Manager (SM).

2.3 Usage and Transfer of Licenses

In the following, we define the basic mechanisms for secure license usage and license transfers. For this we assume the following to be given, and explain in Section 3 how they are implemented: (i) strong compartment isolation (cf. Section 3.1), (ii) the proper initialization of the TCB (cf. Section 3.2), and (iii) the availability of trusted channels with freshness detection (cf. Section 3.3).

On startup, DC loads its actual content/license index i_{DC} from the Storage Manager SM using a (local) trusted channel. To provision licenses the provider establishes a trusted channel to DC. Over this channel the content and licenses are sent to DC and locally stored by SM. For *using (stateful) licenses* the user invokes DC, which loads the corresponding license, checks if all conditions for the corresponding usage-rights are fulfilled, and opens a (local) trusted channel to the secure I/O compartment SO. On the execution of the usage-right, DC

[2] DC's decision is based on either a approved configuration described in the license or on the platform security policy associated with the actual TCB configuration.

[3] Further advantages of our approach are scalability and flexibility: it need not to verify the integrity of all compartments executed on the platform and the integrity verification remains valid even if the user installs or modifies other compartments since the verification is independent of other compartments running in parallel.

updates the state of the license, synchronizes its internal state i_{DC} with the one stored by SM, decrypts the corresponding content, and invokes SO to securely render it. For *transferring stateful licenses* from a source controller DC_s to a destination controller DC_d the following steps are be taken:

1. The user requests DC_s to transfer a license L to DC_d. DC_s uses TM to establish a fresh trusted channel to DC_d to send the license (and corr. content).
2. TM establishes a trusted channel with freshness detection to DC_d allowing DC_s to verify that the configuration of DC_d is conforming to the security policy of L. Note that DC_d does *not* need the equivalent verification for DC_s since the overall security architecture protects and enforces any license once accepted into any DC_s, regardless of the source of the license.[4]
3. Once the decision to transfer L to DC_d is made, DC_s invalidates L locally while synchronizing its internal state i_{DC} with SM where the identity of DC_d (e.g., a public key) is stored together with the license identity to handle possible further requests from DC_d (e.g., when the channel was disconnected for some reason). Note that freshness detection (cf. Section 3.3) will ensure that DC_d will accept L only once.
4. DC_s sends L (and corr. content) to DC_d over the fresh trusted channel. To handle transmission failures, DC_s allows retransmissions requests to DC_d.

The procedure for lending a license is similar to a license transfer: if the license allows lending DC_s generates a license for DC_d valid for the loan period, and updates the state of its own license so that it remains disabled during the loan period. This assumes the availability of secure time.

2.4 Security Objectives

We consider the following security objectives of users and providers.

(O1) *License integrity*: Unauthorized alteration of licenses must be infeasible. This is required by both provider and user.

(O2) *License enforcement*: The license can only be used according to the usage-rights prescribed by the license and to the security policy defining requirements on DC.

(O3) *Freshness*: Replay of licenses must be infeasible. Received and retrieved data is the last one sent or stored even in the case of a platform re-installation.

(O4) *Privacy:* Usage or transfer of licenses must not violate privacy policies. This concerns in particular the least information policy such that components not under full control of the user shall be able to collect, store, and reveal user's private information only to the extent required for license enforcement.

The system is not limited to a specific set of license issuers, and is capable of enforcing the terms of any license accepted by the user. Requirements like license issuance and unforgeability is considered out of the scope of this paper. Distributed authorship proofs and rights management (e.g., as in [1]) can still effectively be built based on our architecture.

[4] As fallback solution, e.g., in case of a broken DC or a dishonest provider, DC_d may also verify the validity of DC_s.

3 Reference Implementation

3.1 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 strong isolation between processes. Virtualization technology enables reutilization of legacy operating systems and existing applications whereas TC technology serves as root of trust. In our architecture a compartment maps to a running application or operating system, whereas a compartment configuration maps to a hash value of the software binary including all initialization information. The architecture of our implementation is shown in Figure 2 whose layers we describe below.

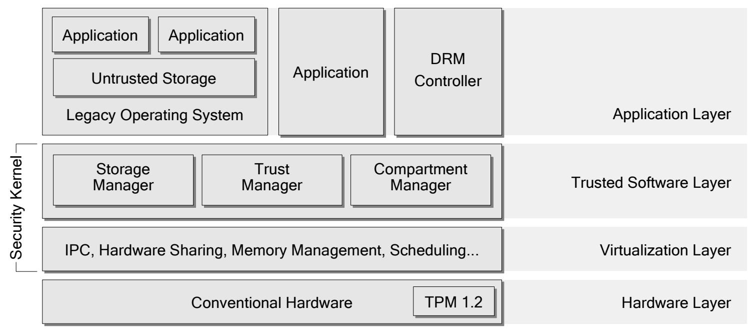


Fig. 2. Security architecture.

Hardware Layer. The hardware layer consists of commercial off-the-shelf PC hardware enhanced with trusted computing technology as defined by the Trusted Computing Group (TCG) [35]. TCG has published several specification for extending the common computing platforms with cryptographic and security features in hardware and software. The main TCG specification is Trusted Platform Module (TPM) [36], which is currently implemented as dedicated cost-effective crypto chip mounted on mainboards of computing devices[5]. Many vendors already ship their platforms with TPMs (mainly laptop PCs and servers) providing the following features: A hardware-based random number generator (RNG), a cryptographic engine for encryption and signing (RSA) as well as a cryptographic hash function (SHA-1, HMAC), read-only memory (ROM) for firmware and certificates, volatile memory (RAM), non-volatile memory (EEPROM) for internal

[5] TPMs are assumed tamper-evident and will only provide a limited protection against hardware based attacks, due to the trade-off between costs and tamper protection. Nevertheless, at least rudimentary tamper precautions and tampering-detection sensors are included in the design and manufacturing process.

keys, monotonic counter values and authorization secrets, and optionally, sensors for tampering detection. Security critical operations (e.g., key generation and decryption) are performed on-chip and security critical information (e.g., secret keys) never leave the TPM unencrypted. The TPM's most important keys were the endorsement key EK, an asymmetric key that uniquely identifies each TPM; and the Storage Root Key SRK, an asymmetric key used to encrypt all other keys created by the TPM. Note that neither EK nor SRK can be read-out from the TPM. The TPM provides further a set of registers called *Platform Configuration Registers* (PCR) that can be used to store hash values.[6] During system startup, a chain of trust is established by cryptographically hashing each boot stage before execution. These hash values are also called *measurements* (in the TCG terminology) and are stored in PCRs. The set of PCR values provides is an evidence for the system's state after boot. This state is called the platform *configuration.* Based on this PCR set, among others, the two functions *sealing* resp. *binding* can be provided to relate data to a platform configuration, sealing additionally relating the data to the specific TPM instance using the TPM's endorsement key EK.

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 these resources. The current implementation is based on microkernels[7] of the L4-family [13]. It implements hardware abstractions such as threads and logical address spaces as well as inter-process communication. 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 layers generic to support also other virtualization technologies (e.g., Xen [25]). However, we decided to employ a L4-microkernel that allows for isolation between processes without the need to create a full OS instance in every compartment in contrast to Xen.

Trusted Software Layer. The trusted software layer, based on the PERSEUS security architecture [19], uses the functionality offered by the virtualization layer to provide security functionalities on a more abstract level. It provides elementary security properties such as trusted channels, platform policy control and compartment isolation. These realize security critical services independent of and protected from application layer compartments. The main services of the trusted software are described in Section 2.2.

Application Layer. On top of the security kernel, several instances of legacy operating systems (here Linux) as well as security-critical applications (here the DRM controller and Secure I/O) are executed in strongly isolated compartments. Unauthorized communication between compartments and unauthorized

[6] The hardware ensures that the value of a PCR can only be *extended* as follows: $PCR_{i+1} \leftarrow \mathsf{hash}[PCR_{\mathsf{i}}|\mathsf{x}]$, with the previous register value PCR_i, the new register value PCR_{i+1}, and the input value x (e.g., again a hash value).

[7] A microkernel is an OS kernel that minimizes the amount of code running in privileged processor mode [21].

I/O access is prevented. The proposed architecture offers an efficient migration of existing legacy operating systems. We are currently running a para-virtualized Linux [11]. 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.

In our reference implementation[8], DC manages, based on XrML, license interpreting and license transfers for several audio formats. Using a Linux multimedia library [10], our SO implementation provides the corresponding audio rendering and play-back.

3.2 Verifiable Initialization

For verifiable bootstrapping of the Trusted Computing Base (TCB), a TCG-enabled BIOS, called the *Core Root of Trust for Measurement* (CRTM), measures the the Master Boot Record (MBR), before passing control to it. A secure chain of measurements is then established: Before a program code is executed it is measured by a previously (measured and executed) component. For this purpose, we have modified the GRUB bootloader (cf. `www.prosec.rub.de/tgrub.html`) to measure the integrity of the TCB. The measurement results are securely stored in the PCRs of the TPM. All further compartments, applications and legacy OS instances are then subsequently loaded, measured, and executed by the Compartment Manager CM.

3.3 Trust Manager and Trusted Channels

Our Trust Manager (TM) implementation is based on the open-source TCG software stack [34]. Trusted channels can be established online or offline. The former requires a direct connection between user and provider whereas the latter does not. Examples are the online purchase of content and licenses at a provider website or obtaining content offline via DVD or as indirect copy by a third party.

Figure 3 gives a description of the protocol for establishing a trusted channel. The protocol can be decomposed into two major phases, namely issuing and verifying a target certificate, and establishing a secret key whose usage is bound to the configuration of the endpoint compartment and the underlying TCB.

If a remote compartment RC requests a trusted channel to a local compartment LC, LC passes this request to TM. TM maps LC's compartment identifier to its compartment configuration $comp\_conf_{\mathsf{LC}}$ using CM. TM then uses, by the means of TPM_CreateWrapKey[], the TPM to create a binding key pair (PK_{BIND}, SK_{BIND}) where usage of SK_{BIND} is restricted to the current TCB configuration TCB_conf measured during initialization (cf. Section 3.2). The TPM then returns PK_{BIND} and the SRK-encrypted[9] secret part ESK_{BIND}.

[8] Most of the corresponding source code is available at `www.emscb.org`.

[9] The Storage Root Key (SRK) is a non-migratable key contained in the TPM as the root key for protected storage.

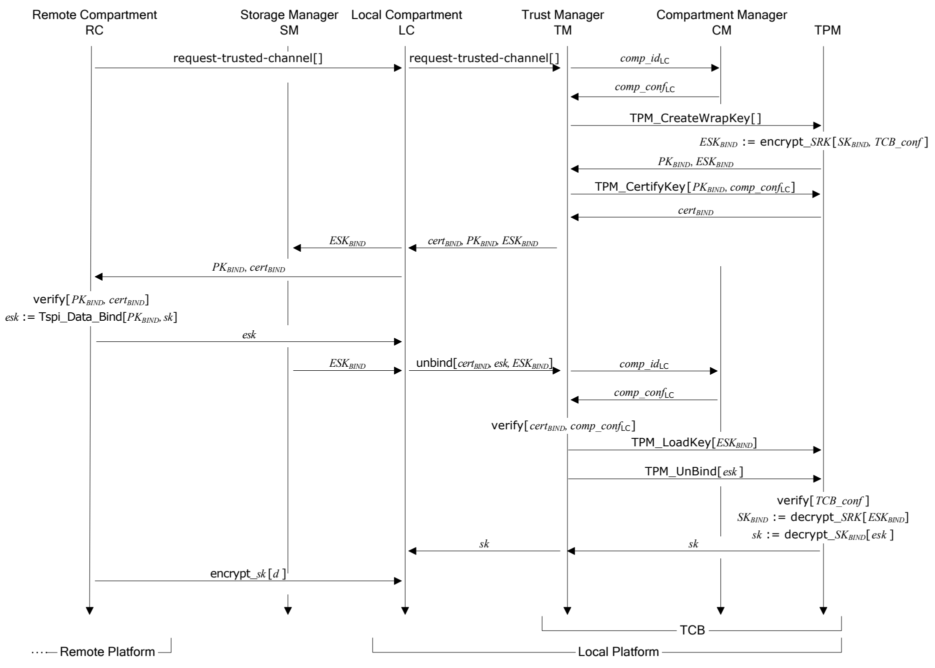

Remote Compartment RC
Storage Manager SM
Local Compartment LC
Trust Manager TM
Compartment Manager CM
TPM
request-trusted-channel[]
request-trusted-channel[]
comp_id_LC
comp_conf_LC
TPM_CreateWrapKey[]
ESK_BIND := encrypt_SRK[SK_BIND, TCB_conf]
PK_BIND, ESK_BIND
TPM_CertifyKey[PK_BIND, comp_conf_LC]
cert_BIND
ESK_BIND
cert_BIND, PK_BIND, ESK_BIND
PK_BIND, cert_BIND
verify[PK_BIND, cert_BIND]
esk := Tspi_Data_Bind[PK_BIND, sk]
esk
ESK_BIND
unbind[cert_BIND, esk, ESK_BIND]
comp_id_LC
comp_conf_LC
verify[cert_BIND, comp_conf_LC]
TPM_LoadKey[ESK_BIND]
TPM_UnBind[esk]
verify[TCB_conf]
SK_BIND := decrypt_SRK[ESK_BIND]
sk := decrypt_SK_BIND[esk]
sk
sk
encrypt_sk[d]
TCB
Remote Platform
Local Platform

Fig. 3. Protocol for establishing a trusted channel.

Then TM invokes the TPM to certify PK_{BIND} and hence to certify $comp\_conf_{\mathsf{LC}}$ and TCB_conf using an Attestation Identity Key (AIK)[10]. We denote the result by $cert_{BIND}$:= TPM_CertifyKey[PK_{BIND}, $comp\_conf_{\mathsf{LC}}$]. Finally, TM returns $cert_{BIND}$ together with PK_{BIND} and ESK_{BIND} to LC. LC then stores ESK_{BIND} using SM and sends ($cert_{BIND}$, PK_{BIND}) to RC. RC verifies $cert_{BIND}$ and then the configurations (TCB_conf, $comp\_conf_{\mathsf{LC}}$) by comparing them with reference values (conforming to its security policy). If positive, RC generates a secret key *sk* and encrypts it using PK_{BIND}. The result is denoted by *esk* := Tspi_Data_Bind[PK_{BIND}, *sk*][11] and sent back to LC. Upon receipt of *esk*, LC loads ESK_{BIND} from SM and requests TM to unbind *sk*. For this, TM again requests CM for mapping LC's compartment identifier to $comp\_conf_{\mathsf{LC}}$. Having successfully verified that $comp\_conf_{\mathsf{LC}}$ matches the configuration denoted in $cert_{BIND}$, TM requests the TPM to unbind *sk*. The TPM first compares the actual PCR values to those SK_{BIND} was restricted to, before returning *sk* to TM. Finally, TM passes *sk* to LC that can now decrypt the data *d* (license and content) received from RC. For online trusted channels, *sk* is used as session key to establish a secure channel inside a subsequent server-authenticated TLS connection be-

[10] An AIK is a special, non-migratable, anonymized key that has been attested to come from a TCG conform platform.

[11] Tspi_Data_Bind[] is a TCG software stack function that does not require any TPM hardware (functionality).

tween RC and LC[12] whereas for offline trusted channels sk is used for encryption of data before being transferred using a indirect connection between RC and LC.

Freshness extension. To tackle replay attacks we extend our trusted channels with freshness. In case of an online trusted channel, freshness can be mutually provided by the underlying TLS handshake protocol by binding the TLS channel to LC (channel binding). This can be done in various ways, e.g., by including $cert_{BIND}$ in regular TLS certificates [31]. In case of offline trusted channels (or without TLS) this can be provided by a slight protocol extension and/or measures at LC. Here different approaches are possible. A simple approach is to require LC to memorize all licenses it has received (i.e., even expired ones) to easily detect license replays. Eventually, this may amount to a huge license list, and one solution is to update (PK_{BIND}, SK_{BIND}) from time to time. Another solution is to let LC also send a nonce N together with $(cert_{BIND}, PK_{BIND})$. In the last protocol step, RC encrypts N together with corresponding data d, so that LC can verify N (and thus freshness) of d and delete N after decryption. An alternative solution to nonces is to let LC create a different public key pair (PK_L, SK_L) for each license, store SK_L in SM, and send $(PK_L$, $cert_{BIND}$, $PK_{BIND})$ to RC. Then RC encrypts data d as before using sk, but encrypts sk with PK_L *and* PK_{BIND}, i.e., $esk :=$ Tspi_Data_Bind$[PK_{BIND},$ encrypt_$PK_L[sk]]$, and sends both quantities to LC. LC now can detect replays of already known licenses by identifying PK_L. Recall that there is a unique relation between PK_L and a license. Once a license has been expired or transferred, SK_L can be deleted. In all scenarios, all secret keys and freshness verification information is persistently stored in trusted storage managed by SM (cf. Section 3.4). All solutions can defeat replays even if the platform is completely re-installed since in this case also all keys and freshness information (contained in SM) are deleted making the the corresponding licenses and content inaccessible.

We have implemented this protocol on TPMs of some major vendors (cf. [24] for more details). The TPM computation dominates the overall computation time. Hence, depending on the efficiency requirements of the underlying application, we have forseen a service (e.g., as part of the TM) that performs the related TPM tasks in software (e.g., generating binding keys). This service is clearly a part of the TCB and is included in the measurements during the verifiable initialization. In this case the trust assumptions of the TCB become stronger since the secret binding key is now in software and not in the TPM security module.

3.4 Storage Manager

The main interfaces of Storage Manager SM (cf. Figure 4) are the trusted channels load[] and store[] for loading/storing data for requesting compartments, and plain channels read[] and write[] for reading/writing data from/to an untrusted storage compartment (e.g., a hard disk drive) to persistently write respectively

[12] Alternatively, PK_{BIND} directly can be integrated into the TLS handshake, e.g., to encrypt RC's pre-master-secret.

read data. Internally, SM maintains an index i_{SM} for metadata of all managed data objects. The main entries in this index are: the configuration *comp_conf* of the requesting compartment, the data object identifier d_{ID}, its freshness detection information f, possible further access restrictions *rest* (e.g., user id, group id or date of expiry), a monotonic counter c_{SM} verifying the freshness of i_{SM}, and a sealed k_{SM} used to seal i_{SM} to SM's configuration.

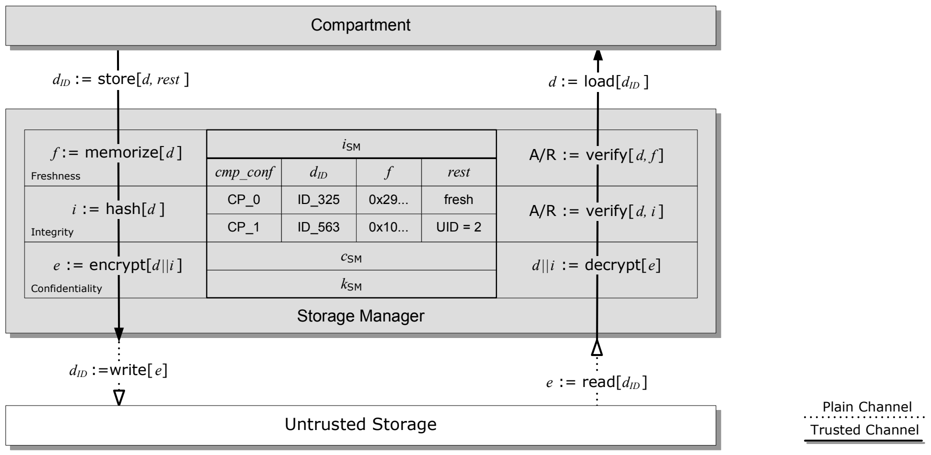


Fig. 4. Implementation of SM.

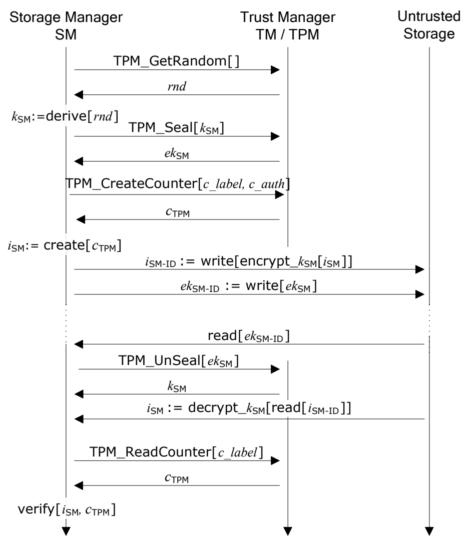
To ensure freshness of the metadata the index i_{SM} itself, SM manages an internal software counter c_{SM} that is incremented synchronously with a TPM 1.2[13] monotonic hardware counter c_{TPM} each time SM updates its index[14]. A mismatch means outdated data which will be handled according to the underlying security policy. In order to employ TPM's monotonic counters, SM has to be initialized correctly. Figure 5 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. At initial setup SM uses the TPM to create its internal cryptographic key k_{SM}, which is sealed to the current TCB configuration. To enable freshness detection and thus trusted storage, SM creates a monotonic counter c_{TPM} with a label *c_label* for identification and an authentication *c_auth* (e.g., a randomly chosen secret password). The initial setup finishes with the generation of i_{SM} and the sealed key ek_{SM} and writing i_{SM} (that includes c_{SM}, *c_label* and *c_auth*) encrypted on untrusted storage using k_{SM}.

After a platform reboot, SM reads the ek_{SM} from the untrusted storage and asks the TPM to unseal ek_{SM} to its internal key k_{SM}. The TPM is able to unseal k_{SM} if the platform has the same configuration as it had at the sealing process,

[13] TPM version 1.1b cannot be used for fresh storage [24].

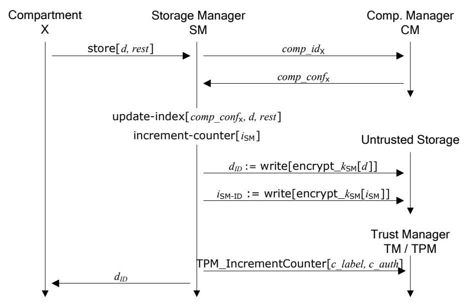
[14] As specified in [36] version 1.2, a TPM supports monotonic counters with an increment rate of at least once every 5 seconds other at least 7 years.

thus preventing a modified SM to access k_{SM}. Then SM uses k_{SM} to decrypt i_{SM} and verifies freshness of i_{SM} by comparing the decrypted counter value c_{SM} of i_{SM} with the actual counter value of the corresponding hardware counter c_{TPM}.



Storage Manager SM
Trust Manager TM / TPM
Untrusted Storage
TPM_GetRandom[]
rnd
k_{SM}:=derive[rnd]
TPM_Seal[k_{SM}]
ek_{SM}
TPM_CreateCounter[c_label, c_auth]
c_{TPM}
i_{SM}:= create[c_{TPM}]
$i_{SM\text{-}ID}$:= write[encrypt_k_{SM}[i_{SM}]]
$ek_{SM\text{-}ID}$:= write[ek_{SM}]
read[$ek_{SM\text{-}ID}$]
TPM_UnSeal[ek_{SM}]
k_{SM}
i_{SM} := decrypt_k_{SM}[read[$i_{SM\text{-}ID}$]]
TPM_ReadCounter[c_label]
c_{TPM}
verify[i_{SM}, c_{TPM}]

Fig. 5. SM initialization.



Compartment X
Storage Manager SM
Comp. Manager CM
store[d, rest]
comp_id$_X$
comp_conf$_X$
update-index[comp_conf$_X$, d, rest]
increment-counter[i_{SM}]
Untrusted Storage
d_{ID} := write[encrypt_k_{SM}[d]]
$i_{SM\text{-}ID}$:= write[encrypt_k_{SM}[i_{SM}]]
Trust Manager TM / TPM
TPM_IncrementCounter[c_label, c_auth]
d_{ID}

Fig. 6. SM's store protocol.

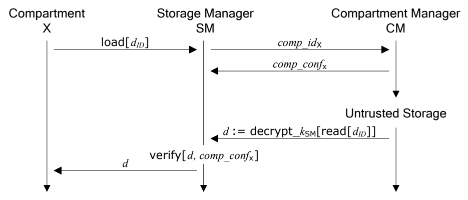


Fig. 7. SM's load protocol.

Figure 6 depicts the protocol steps required to bind a compartment's data object (e.g., i_{DC}) persistently to its actual configuration. After the mapping of compartment identifier to the actual compartment configuration (e.g., $comp\_conf_{\mathsf{DC}}$) using CM, SM updates i_{SM} with the corresponding metadata as well as the incremented software counter c_{SM} to enable freshness detection for i_{SM}. SM encrypts both the data object and the updated index i_{SM} using k_{SM} and writes them to untrusted storage. Finally, SM synchronizes its software counter c_{SM} with the TPM's monotonic hardware counter c_{TPM} (using *c_label* and *c_auth*) and returns the data object identifier.

Figure 7 depicts the protocol steps required to load a compartment's data object. Again after a mapping of compartment identifier to the actual compartment configuration using CM, SM reads the requested data object from untrusted storage and decrypts it using k_{SM}. Before returning d to the corresponding compartment, SM verifies all existing access restrictions (e.g., freshness, or a certain user id) given on store via *rest* based on the corresponding metadata in i_{SM} and verifies that the requesting compartment has the same configuration as used on store.

4 Security Considerations

In this section we sketch the security aspects of our implementation. First we consider the core security properties (verifiable initialization, strong isolation, trusted channels, trusted storage) provided by our implementation. Based on these properties we consider the individual security objectives (cf. Section 2.4).

Verifiable Initialization. It ensures that the TCB bootstrap is measured and securely stored in the TPM (cf. Section 3.2). Other compartments can then use TPM functionality to securely query the actual TCB configuration. Note that subsequent modifications at runtime are not reflected by the initialization measurements. However, a TCB configuration that would allow arbitrary alternation/patches of core security components cannot be considered as trustworthy.

Strong Isolation. Runtime isolation is provided by the small virtualization layer that implements only logical address spaces, inter-process communication 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 small[15] and can, in contrast to monolithic operating system kernels[16] such as Linux or MS Windows, be easier validated for 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.

Trusted Channels. The establishment of a trusted channel is described detailed in Section 3.3. The inter-process communication provided by the virtualization layer enables secure channels between local compartments. Secure channels between local and remote compartments can be provided either by using the secret key *sk* to establish a secret channel inside a tunnel created by standard security protocols such as TLS [8] (online trusted channel) or by using *sk* to encrypt content at RC before sending it indirectly (e.g., via DVD or CD-ROM) to LC (offline trusted channel). As mentioned in Section 3.3 trusted channel enables access to data only by an authorized compartment (trustworthy configuration). The configuration of a compartment and the underlying TCB are securely measured during the initialization (cf. Section 3.2). Replay attacks on trusted channels can be defeated using one of the freshness solutions described in Section 3.3.

Trusted Storage. SM provides integrity, authenticity, confidentiality and freshness of data as described in Section 3.4. The integrity and confidentiality are achieved by using standard cryptographic mechanisms whereas monotonic hardware counters are used for freshness detection. We have improved common approaches

[15] A microkernel-based approach can be realized with around 50.000 SLOC [28].

[16] The sources lines of code (SLOC), e.g., for Windows XP are around 40 million and around 6 million for a regular Linux 2.6 kernel [30].

while taking advantage of the strong isolation capability of our architecture that prevents the exposure of cryptographic secrets to unauthorized or malicious processes. Our SM enables compartments to persistently bind their local state to their actual configuration. The verifiable initialization (cf. Section 3.2) verifies whether the TCB components booted are trustworthy, i.e., conform to the underlying security policy.

Given these properties we sketch the analysis of the security objectives. License integrity (O1): Trusted channels ensure that only mutually trusted compartments can modify a license, whereas strong isolation and trusted storage prevent unauthorized alteration of licenses at runtime and while persistently stored. License enforcement (O2): License and content are sent only to a local compartment whose configuration matches that of DC. Further, the isolation property prevents any other malicious code from accessing the content or modifying the license. Freshness (O3): The freshness extension (cf. Section 3.3) and SM ensure that any data loaded is the last one stored. Privacy (O4): The properties of our architecture such as isolation and binding and the fact that security policy defined by the platform owner restricts the I/O behavior of every application imply that even if third party applications, like DC, can locally enforce their own security policy, they cannot bypass the defined overall security policy. In particular, the information revealed to third parties (content providers) is restricted following the least privilege policy, e.g., only the configuration of the TCB and DC essential for transferring licenses are revealed. However, if it is required not to reveal the TCB configuration a possible extension to our architecture would be to add property-based attestation service [23] to TM and CM to hide both the (binary) configuration of the TCB and DC.

5 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. 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).

The building blocks needed for stateful licenses can also enable offline superdistribution [3]. For example, in our motivating scenario, Timo could generate a *new* license for Anna's device. The DRM controller will record this fact in its stateful license until Timo pays for the new copy. Allowing copies to be made while still retaining the ability for proper metering and reporting of new copies will enable rapid *legal* spread of popular content. We plan to describe this extension more fully in a forthcoming paper.

Finally, copyright itself is a strongly debated topic. In course of time, the world may develop alternative business models that do not require protection of copyright in its current form. However, the type of platform security described is also useful in many other applications like copy-protected ticketing, and electronic money. In fact, the same techniques that are used to protect the interests of a third party from a malicious device owner can also help protect the device owner from a thief who stole the device.

Acknowledgments

We thank Michael Scheibel for his contributions in jointly developing and implementing the concept of trusted channels.

References

1. Adelsbach, A., Sadeghi, A.-R., and Rohe, M. Towards multilateral secure digital rights distribution infrastructures. In *Proceedings of the ACM Workshop on Digital Rights Management* (2005).
2. Apple Computer, Inc. FairPlay DRM. www.apple.com/itunes/.
3. Asokan, N., and Ekberg, J.-E. Mobile digital rights management. In *Professional Mobile Internet Technical Architecture – Visions & Implementations.* 2002.
4. Aura, T., and Gollmann, D. Software license management with smart cards. In *Proceedings of the First USENIX Workshop on Smartcard Technology* (1999).
5. Authentica, Inc. Authentica active rights management. www.authentica.com.
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. Council, N. R. *The Digital Dilemma, Intellectual Property in the Information Age.* National Academy Press, 2000.
8. Dierks, T., and Allen, C. RFC2246 - the TLS protocol version 1.0. www.ietf.org/rfc/rfc2246.txt, January 1999.
9. Epsilon Squared, Inc. InstallRite Version 2.5. www.www.epsilonsquared.com.
10. Freitas, M., Roitzsch, M., Melanson, M., and Mattern, T. The xine free multimedia player. www.xinehq.de.
11. Hohmuth, M. Linux-Emulation auf einem Mikrokern. Master's thesis, Dresden University of Technology, Dept. of Computer Science, 1996.
12. Koenen, R., Lacy, J., MacKay, M., and Mitchell, S. The long march to interoperable digital rights management. *Proceedings of the IEEE 92* (2004).
13. Liedtke, J. Towards real microkernels. *Communications of the ACM 39* (September 1996).
14. 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.
15. Marchesini, J., Smith, S., Wild, O., Barsamian, A., and Stabiner, J. Open-source applications of TCPA hardware. In *20th Annual Computer Security Applications Conference* (2004).
16. 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, Dartmouth College, 2003.
17. Microsoft Corporation. 60 days trial program. us1.trymicrosoftoffice.com.
18. Microsoft Corporation. Windows media rights manager 10. www.microsoft.com/windows/windowsmedia/drm/default.aspx.

19. 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.
20. Pruneda, A., and Travis, J. Metering the use of digital media content with Windows Media DRM 10. `http://msdn.microsoft.com/library/en-us/dnwmt/html/meteringcontentusage10.asp`.
21. 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* (2000).
22. Russinovich, M. Sony, rootkits and digital rights management gone too far. `http://blogs.technet.com/markrussinovich/`, October 2005.
23. Sadeghi, A.-R., and Stüble, C. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *The New Security Paradigms Workshop* (2004).
24. Sadeghi, A.-R., Wolf, M., Stüble, C., Asokan, N., and Ekberg, J.-E. Enabling Fairer Digital Rights Management with Trusted Computing. Tech. Rep. HGI-TR-2007-002, Horst-Görtz-Institute for IT-Security, Ruhr-University Bochum, June 2007.
25. 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, 2005.
26. 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* (2004).
27. 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* (2002), vol. 2320 of *LNCS*.
28. Singaravelu, L., Pu, C., Helmuth, C., and Härtig, H. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Eurosys Conference Proceedings* (Leuven, Belgium, 2006).
29. Starz Entertainment Group. Video on demand service. `www.vongo.com`.
30. Tanenbaum, A. Keynote at linux.conf.au, January 2007.
31. TCG Infrastructure Workgroup. Tcg infrastructure workgroup subject key attestation evidence extension specification version 1.0 revision 7.
32. The Hymn Project. Free your iTunes Music Store purchases from their DRM restrictions. `www.hymn-project.org`, March 2007.
33. The Register. DVD Jon hacks Media Player file encryption. `www.theregister.co.uk/2005/09/02/dvd_jon_mediaplayer/`, October 2005.
34. TrouSerS. The open-source TCG software stack. `trousers.sourceforge.net`.
35. Trusted Computing Group. `www.trustedcomputinggroup.org`.
36. Trusted Computing Group. TPM main specification. Tech. rep. `www.trustedcomputinggroup.org/specs/TPM/`.
37. Tygar, J., and Yee, B. Dyad: a system using physically secure coprocessors. In *Technological Strategies for Protecting Intellectual Property in the Networked Multimedia Environment* (1994).