Hardware-Assisted Application-Level Access Control

Yu-Yuan Chen and Ruby B. Lee

Princeton University, Princeton NJ 08544, USA
{yctwo,rblee}@princeton.edu

Abstract. Applications typically rely on the operating system to enforce access control policies such as MAC, DAC, or other policies. However, in the face of a compromised operating system, such protection mechanisms may be ineffective. Since security-sensitive applications are most motivated to maintain access control to their secret or sensitive information, and have no control over the operating system, it is desirable to provide mechanisms to enable applications to protect information with application-specific policies, in spite of a compromised operating system. In this paper, we enable application-level access control and information sharing with direct *hardware* support and protection, bypassing the dependency on the operating system. We analyze an originator-controlled information sharing policy (ORCON), where the content creator specifies who has access to the file created and maintains this control *after* the file has been distributed. We show that this policy can be enforced by the software-hardware mechanisms provided by the Secret Protection (SP) architecture, where a Trusted Software Module (TSM) is directly protected by SP's hardware features. We develop a proof-of-concept text editor application which contains such a TSM. This TSM can implement many different policies, not just the originator-controlled policy that we have defined. We also propose a general methodology for *trust-partitioning* an application into security-critical and non-critical parts.

1 Introduction

Access control in a computer system mediates and controls accesses to resources. It is an essential part of the security of a computer system, preventing illegitimate access to sensitive or protected information. Various access control policies exist, e.g. mandatory access control (MAC), discretionary access control (DAC), role-based access control (RBAC), etc. One access control policy that has been hard to achieve is ORCON [1,2], or originator-controlled access. This is neither a MAC nor a DAC policy. It is not specified by a central authority (like DAC), but its subsequent re-distribution by legitimate recipients must be controlled (like MAC). While DAC allows individuals to specify the access policy for their files, it cannot control how a legitimate user re-distributes those files. In this paper, we propose a hardware-software mechanism for achieving flexible access control and information sharing policies, including ORCON-like policies.

P. Samarati et al. (Eds.): ISC 2009, LNCS 5735, pp. 363–378, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Access control mechanisms are usually implemented by the operating system (OS), which also enforces the access control policies. However, if the OS is compromised, then the access control policy enforcement can also be compromised. Applications need some way to protect secret or sensitive information, in spite of a compromised OS, over which they typically have no control. Hence, we examine how an application can be provided a flexible mechanism to achieve an application-level access control or information sharing policy, without depending on the OS.

In addition to conventional access control mechanisms, cryptographic mechanisms have also been used to control access to protected information. For example, Digital Rights Management (DRM) systems[3,4,5] use cryptographic mechanisms for copy protection of digital media. Here, anyone can get access to the encrypted material, but only legitimate recipients may gain access to the plaintext material – at least, that is the goal of DRM systems. Strong cryptography can be used to protect the contents of sensitive files by encrypting them into an unintelligible mass, while decrypting them only when needed or authorized. However, two critical issues arise: how the keys are managed and how the decrypted plaintext is managed. Commercial DRM systems such as Advanced Access Content System (AACS) [3] are broken not because they use weak cryptography (as in the case of Content Scramble System (CSS) [4]), but because of the unsafe storage of the keys used by the application software [5]. Hardware protection mechanisms such as TPM [6] are designed to protect cryptographic keys by *sealing* them in the TPM hardware, and the keys are only retrieved when the system is running in a *verified* condition. TPM offers greater protection to the keys and includes the measurements of the integrity of the operating system in the trust chain to make sure that it has not been compromised. However, TPM's protection model does not consider how the keys are used and where they are stored after they are *unsealed*, therefore the access control of the decrypted sensitive information is still left to the application and the decrypted symmetric keys from the TPM chip can still be obtained by examining the memory contents[7,8]. Hence, it all boils down to the management of the keys and the decrypted plaintext.

The access control to keys is often delegated to the operating system, since it governs all accesses to resources. However, modern operating systems are large and complex, and hence more prone to software vulnerabilities. Further, in the monolithic kernel model of common operating systems such as Linux and Windows, all kernel modules have equal privilege, so that one compromised kernel module can access the memory of another kernel module, which may be security-critical. An attacker can gain control of the operating system by targeting one of the many device drivers, bypass the access control and retrieve the application's secret or sensitive information.

In this paper, we propose the following solution: *A small, verifiable application module that enforces its own policy with direct hardware protection that cannot be bypassed or manipulated by the operating system.* Implementing access control or information sharing policies in the application-space removes the dependency on the operating system and adds the flexibility of incorporating different policies.

Our solution architecture builds on top of the Secret Protection (SP) architecture [9,10], which requires a small addition to the processor hardware to protect a Trusted Software Module (TSM). We provide protection of the application by modifying it slightly to incorporate a TSM directly protected by the hardware to prevent any undesired information leakage. We implement an ORCON-like access control policy for protected documents that is designed to be enforced in a distributed manner.

The contributions of the paper are as follows:

- Proof-of-concept implementation of a distributed access control policy that is difficult to enforce, e.g. ORCON.
- Developing a methodology for *trust-partitioning* of an application.
- Demonstrating the versatility of the SP architecture for implementing different access control or information sharing policies.

Section 2 gives the detailed definition of our target access control policy. Section 3 describes the threat and trust models considered in this paper. Section 4 describes our solution architecture. Section 5 explains the methodology we developed to partition an application into a trusted and an untrusted part. Section 6 gives the security analysis of our solution. Section 7 describes related work in this area and Section 8 concludes the paper.

2 Problem Statement

Information sharing has different requirements in different contexts. For example, confidentiality is of top concern in a military system, whereas integrity is essential in commercial systems. We consider an information sharing policy that could be tailored to work in both environments, to meet the needs of both confidentiality and integrity. Consider the case where a secret document is to be distributed to selected recipients of different clearance levels, while the content of the original document cannot be modified. Further, the re-distribution of the content has to be approved by the content creator. This policy, previously known as Originator-Controlled policy (ORCON) [1,2], was proposed to address such a scenario. Since the control point of the policy is neither entirely centralized nor entirely distributed, it cannot be directly solved by applying Mandatory Access Control (MAC) or Discretionary Access Control (DAC).

In such an information sharing policy, the key players include the *content creator*, the *recipients* and the *trust group* (Figure 1). The recipients can be further categorized as *authorized* recipients, who are within the trust group and are allowed access to the content of the document by the content creator, and *unauthorized* recipients who are outside the trust group. Not all members of the trust group are authorized recipients of a given document. We formalize the problem statements of the information sharing policy as follows.

- **Problem 1: Dissemination to authorized recipients**
  The content creator wants to restrict access to the content to authorized recipients only. In other words, recipients who have not gained explicit approval from the content creator will not have access to the content.

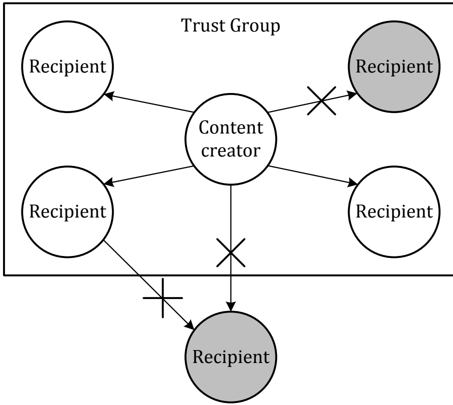

Fig. 1. Players in the information sharing policy. Gray circles represent unauthorized recipients, while white circles represent authorized recipients.

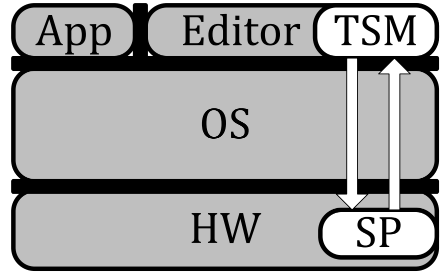

Fig. 2. Using SP architecture for flexible access-control enforcement. Grey parts are the untrusted system, white parts are trusted.

- **Problem 2: Prevent illegitimate re-dissemination**
  After the authorized recipients have gained access to the content, it should not be possible to redistribute or copy the original content to any unauthorized recipients. An unauthorized recipient must ask the content creator for explicit access rights in order to access the protected content.
- **Problem 3: Allow legitimate appending to the content**
  In the case where the content creator allows for appending extra information to the original content, an authorized recipient must be able to append to the original content, while preserving the authorized recipients of the original policy dictated by the content creator, i.e. the protected content may grow but the list of authorized recipients should remain unchanged.

To solve the above problems, we identify the requirements that must be met:

- The policy dictated by the content creator has to be tied to the corresponding protected content.
- The policy has to be enforced regardless of the presence of the content creator, i.e. the enforcement is distributed among the recipients.
- Updating (appending) the content should allow changes only in the content, not the policy. Therefore the policy should be physically separated from the content but logically tied to it.

Our solution architecture, as described in the following sections, adheres to these requirements and hence guarantees that the policy is never violated.

3 Threat and Trust Models

We assume that every recipient uses some type of computing device to access the content in the protected document, where each device has a Central Processing

Unit (CPU) that is trusted. Further, the content of the protected document is accessed by a piece of *editor* software that can read, display or modify the content in the document. For simplicity, we consider the protected content as digital text documents in this paper, although our proposed solution and methodology apply to any digital multimedia contents, e.g. digital photos, video or music.

The goal of an adversary is to gain access to the information in the protected content without explicit approval from the content creator. The adversary may have obtained the file of the protected document and have physical access to the computing device, and he can write his own software to run on the computing device to try to gain as much information as possible. Since the adversary has physical access, memory bus tapping or access to raw bits on disk are considered valid attacks. However, we *do not* consider any analog attacks, e.g. shoulder surfing or social engineering, since these attacks are out-of-band exploits that are not within the control of a computer system.

We divide the editor program into a *trusted* and an *untrusted* part, where the trusted part is guaranteed to perform the desired functions and any tampering with the trusted part will be detected, by means of our hardware protection mechanisms. However, the adversary can modify the untrusted part or the operating system to perform any malicious activities.

On the recipients' computing devices, we assume that a *trusted path* exists between the user input and the trusted CPU, and between the CPU and the display output. Hence, the device user can be assured that the input comes directly from him and that what is displayed is indeed that which is processed by the CPU. Various techniques exist [11,12,13] to support a trusted input path and a trusted display.

4 Architecture

Our solution consists of a combination of CPU hardware and application software, which builds upon the Secret Protection (SP) [9,10] architecture to provide direct hardware protection of the application. In essence, we partition the editor application into a trusted and an untrusted part and provide protection of the trusted part directly by the hardware, as shown in Figure 2.

4.1 SP Architecture

SP Architecture was first proposed [10] to protect the user's secret or sensitive information (*user mode*) and later modified [9] to protect a remote authority's and third parties' secret or sensitive information (*authority mode*). Our solution builds upon the authority mode SP [9]. We highlight the key architectural features of SP below.

The architecture consists of the Trusted Software Module (TSM) in the user-level application and the SP hardware in the microprocessor chip. There are two *hardware trust anchors* in the microprocessor chip: *Device Root Key* (DRK) and *Storage Root Hash* (SRH). The DRK is unique for each chip; it never leaves the

chip and can not be read or written by any software. The only software that can use the DRK is the TSM, via a special instruction that can derive a new key from the DRK given nonces and/or constants. The SRH securely stores the root hash of a secure user-defined storage structure (on disk or on-line storage) accesssible only to the TSM. The SRH is accessible only to the TSM. Other software cannot read or write the SRH, including the operating system.

Hardware *Code Integrity Checking (CIC)* ensures the integrity of the TSM code while executing. Each instruction cache line embeds a MAC (a keyed hash), with the DRK as the key. The hash is verified before the instruction cache line is brought on-chip. Hardware *Concealed Execution Mode (CEM)* protects the TSM's data while it is executing, to guarantee confidentiality and integrity of any temporary data that the TSM uses, whether this is in on-chip registers or caches, or evicted to off-chip memory. During interrupt handling, hardware protects the contents of general registers and the interrupt return address from a potentially corrupted OS. All data cache lines containing protected data are encrypted and hashed when evicted from the microprocessor chip. A hardware encryption and hashing engine accelerates the automatic encryption (or decryption) and hash generation (or verification), reducing cryptographic overhead to the infrequent cache-miss handling of the last level of on-chip caches.

4.2 Distributed Access Control Architecture

The access control required by our information sharing policy is enforced by the new trusted part of the editor application, i.e. the TSM in the user-space. To guarantee the confidentiality and integrity of the protected document while it is opened by the editor, and to simplify the access control mechanism, we dedicate a special TSM buffer for use only by the TSM to store and manipulate any temporary data it uses. All the data in the TSM buffer are tagged as secure data in the processor's on-chip caches. When secure data cache lines are evicted from on-chip caches out to the main memory, the SP hardware mechanism will ensure that they are encrypted and hashed, by a key that is derived from the DRK. The TSM buffer does not interfere with the internal buffer structures of the editor program, so that the editor functions that do not involve the TSM are not modified at all. The TSM buffer is used by the TSM to hold temporary decrypted lines of the protected content. In other words, the protected content remains encrypted inside all internal buffers of temporary files used by the editor, only decrypted by the TSM in the TSM buffer when the TSM is active.

As mentioned in Section 2, the policy and the content should be physically separated but logically tied. We store the policy dictated by the content creator in the secure storage maintained by the SP hardware, and we tie together the policy and the content by a cryptographic hash that is also stored and protected in the secure storage. The root of trust of the secure storage (SRH) is protected on-chip and accessible only by the TSM, and only the TSM can legitimately access or modify the stored policies in the secure storage. Any illegitimate modifications to the stored policies will be caught by the TSM when checking the

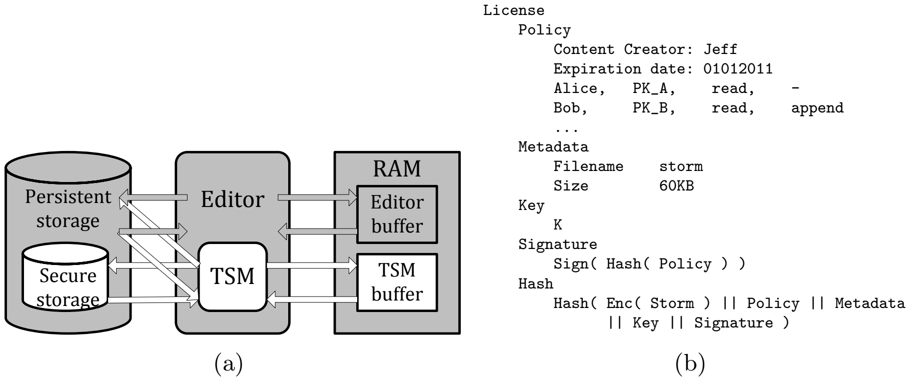

Fig. 3. (a): Partitioning the editor application and the system into untrusted (grey) and trusted (white) parts. The TSM gets its own buffer to work with temporary data, and it can access both the secure storage where the policies are stored and normal storage where the protected (encrypted) content is stored. (b): A license for the document. This contains a policy dictated by the content creator (PK_A represents the public key of Alice).

integrity of the secure storage. Figure 3(a) shows the interaction between the editor application (trusted and untrusted parts), the temporary buffers (SP-protected and unprotected), and the persistent storage (secure and unsecured). Physically separating the data and the policy reduces the amount of information that needs to be directly protected in the secure storage of SP, since a file can be very large.

The sensitive content is protected by encrypting the document with a key that is stored in the secure storage, along with the policy dictated by the content creator. Since the document is encrypted, it can be safely stored in any public storage without additional access control protection. The key to decrypt it is bound by the policy and the policy is enforced by the TSM. The TSM always controls the access to the decryption keys according to the corresponding policies. To ensure compliance with the requirements described in Section 2, in addition to the policy and the key, we store other pertinent information of the protected document in a data structure called a *license*, (see Figure 3(b)), which is stored in SP-protected secure storage. A license contains the access control policy, metadata, the key to decrypt the document, the originator's signature on the policy, and a hash over the encrypted document and all items in the license.

Before the user is allowed access to the content in the document, the TSM first checks the integrity (`Hash`) of the encrypted document and the license, to make sure they have not been tampered with. Then the TSM checks if the policy allows the particular recipient access to the content of the document. After all checks are successfully passed, the TSM decrypts the content of the document and stores it in the temporary TSM buffer and, through the trusted display link, displays the contents to the authenticated recipient.

4.3 TSM Architecture

Figure 4 shows a general structure of the TSM consisting of several modules (libraries) that perform different functionalities required by the TSM. The TSM is not limited to a specific application and a specific access control policy.

Since in our threat model we assume a trusted I/O path exists, a trusted I/O module serves as the gateway for the TSM to receive user input, to display output or to connect with other TSMs. A crypto module that implements symmetric key encryption/decryption, asymmetric key encryption/decryption and cryptographic hash functions, and a random number generation (RNG) are included in the TSM, so that the TSM does not need to depend on the operating system for these functions. The core of the TSM is the policy enforcement module that interacts with the TSM buffer and interprets the policy stored in the secure storage to mediate the I/O of the TSM. The policy enforcement module acts as the TSM resource manager that can be tailored to implement various access control policies. A user authentication module, along with a set of PKI interfaces is included in the TSM to take care of the user authentication required to guarantee that the owner of the public/private key pair specified in the policy is correctly authenticated. User authentication is described in Section 4.5.

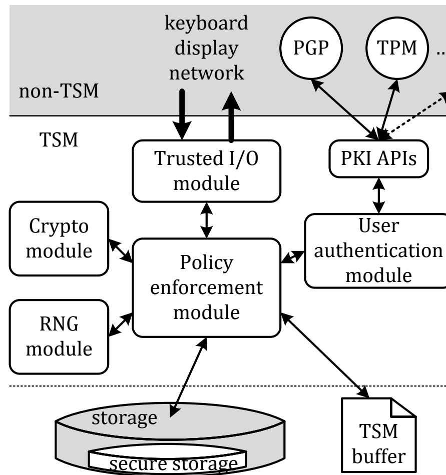

Fig. 4. TSM architecture. The trusted (white) parts of the system are the TSM and the SP-protected secure memory and secure storage.

4.4 Operation

We walk through an example to show how the TSM and the SP hardware protect and enforce the access control of the protected document.

1. The content creator creates the document containing sensitive data using any application he/she chooses.
2. The content creator dictates the policy he/she would like to enforce, e.g. *who* has *what* access to the content.

3. The content creator runs the editor application which contains the TSM, to turn the document into a protected document. A series of steps occur.
   (a) The TSM first randomly generates a new symmetric key.
   (b) The TSM encrypts the contents using the generated key and erases the plaintext.
   (c) The TSM calculates the hash of the policy and asks the content creator to sign the hash.
   (d) The TSM calculates the hash of the encrypted document, policy, metadata, key and the signature, and stores them in a newly created license in the secure storage.
4. The content creator can now distribute the encrypted document to all recipients he/she desires.
5. The TSM on the content creator side encrypts the license using the group encryption scheme [14] for the recipients (group encryption scheme and the trust group are described in Section 4.6).
6. The TSMs of the recipients' devices decrypt the license with their group decryption keys and securely store the license in the secure storage.
7. The TSM on the recipient side authenticates the recipient and checks the policy before granting access to the contents of the protected document.

4.5 User Authentication

User authentication is a difficult problem for the TSM, since we cannot rely on the operating system for existing user authentication mechanisms. To simplify the design of the TSM and not burden it with complex user authentication functions, we propose a public/private key authentication solution. We build a generic API interface that can interact with and make use of different public/private key applications, e.g. OpenPGP or GnuPG, that manage users' private keys. Below, we outline the protocol used by the TSM to authenticate a user utilizing other PKI applications.

When invoked by the user to read a policy-protected document, the TSM prompts the user for identity, for example, *Alice*. The TSM reads the corresponding policy in the secure storage to locate Alice's public key, `PK_A`. The TSM calls the RNG module to generate a new random number and uses `PK_A` to encrypt the random number as a challenge. The TSM sends the random challenge to the PKI application through the PKI interface and asks it to decrypt the random challenge.

The PKI application authenticates the user via its normal mechanisms, e.g. passphrase or TPM [6]. The PKI application returns the decrypted challenge to the TSM. The TSM checks for the validity of the random challenge to determine if the user has been successfully authenticated.

Ideally, the whole PKI application should be included in the TSM, since it is a security-critical function. If we consider the operating system as untrusted, the PKI application could also be compromised. However, our architecture still ensures that the keys that are used to decrypt the document, and the plaintext of the document, are never released outside the TSM.

4.6 Group Encryption and Trust Groups

We use group encryption [14] for distributing the protected license to the authorized recipients. Group encryption is the dual of the well-known group signature scheme [15,16,17]. In a group signature scheme, a member of a group can anonymously sign a message on behalf of the group, without revealing his/her identity. In a group encryption scheme, the sender can encrypt a piece of data and later convince a verifier that it can be decrypted by the members of a group without revealing the identity of the recipient. The authority in both cases is the only entity that can reveal the identity of the signer in the group signature scheme or the recipient of the group encryption scheme. One group in a group encryption scheme has one *group encryption key* and multiple *group decryption keys* associated with it. The group encryption key is public and is used to encrypt messages, while the group decryption keys are private.

In SP architecture [9], a trusted authority installs all TSMs and knows the DRKs of all the SP devices. This is also the authority in the group encryption scheme. In our architecture, the authority that initializes and installs the TSMs creates a group that includes all SP devices, and assigns each SP hardware a unique *group decryption key*, while publishing the *group encryption key* for that group, such that in the secure storage of each SP device a pair of group encryption and decryption keys is stored and tied to the particular SP hardware. Therefore the content creator can be assured that the license can only be decrypted by SP-enabled devices in the same group. For simplicity, we assume that all SP-enabled devices are in the same group, although different groups of SP-enabled devices can be established depending on application requirements. Note that the authority that governs the SP-enabled devices and the trust groups need not be the same as the certificate authority in the PKI systems for user authentication.

In practice we may desire to have *multiple trust groups*, where each group may contain an arbitrary number of SP devices. Since each originator may need to distribute the protected document to a different set of authorized recipients, it is desirable, although not necessary, to have separate groups for each originator. This scheme can be easily incorporated in our solution since we can store multiple group encryption-decryption key-pairs in the secure storage of each SP device, and the TSM is responsible for distinguishing between different trust groups and making sure that there is no information flow between trust groups, unless it is explicitly allowed by the originator. Therefore, a recipient can belong to multiple trust groups without the need to use multiple devices. However, since there is only one authority that knows the DRKs of all devices and hence the only authority that can properly insert group encryption-decryption key-pairs into the devices, we cannot allow multiple authorities in a trust group without extending the SP architecture [9].

5 Trust-Partitioning an Application

We developed a methodology for partitioning an existing application into a trusted and an untrusted part. We chose *vi* [18] as our proof-of-concept application

to implement the application-level information sharing policy, since it is one of the most common text editors in the Unix operating system. Our methodology can also be applied to other applications.

To partition an application, we need to identify the entry and exit points into and out of the TSM. We first categorize the commands available in *vi*. Figure 5 shows the flow chart used to categorize the various commands of *vi* into 5 generic groups.

Table 1 shows the commands in each group. In particular, we are interested in the commands that are relevant to our information sharing policy, e.g. displaying the content of a file or appending new content to the original file, etc. The commands in **bold** (i.e., **ex** and **quit**) are modified *vi* commands and the commands in *italic* are new commands. These commands are the entry and exit points of the TSM and are the only commands that can legitimately manipulate

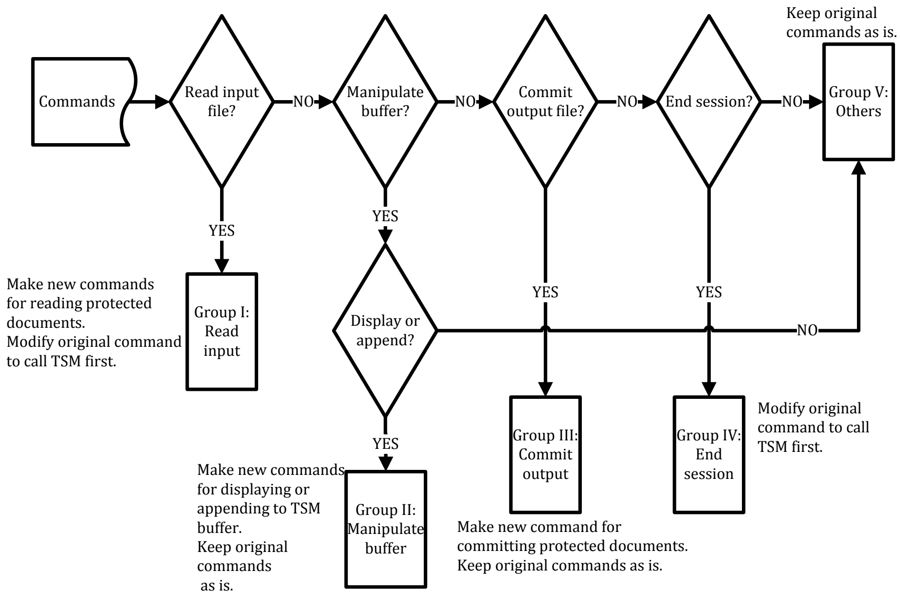

Fig. 5. Categorization of functions within an application for TSM protection

Table 1. The groups of *vi* commands after categorization

Group I Read input	Group II Manipulate buffer	Group III Commit output	Group IV End session	Group V Others
ex	print	write	**quit**	abbreviate
tsm_ex	read	*tsm_write*		args
	tsm_print			cd
	tsm_read			delete
				...
				tsm_create

Table 2. New and modified vi commands

tsm_ex filename	Open a protected document.
tsm_print line_number	Display the contents of a protected document.
tsm_read filename	Append the contents of filename to current protected document.
tsm_write	Automatically re-encrypt the protected document (with any appended data) and update the length and the hash stored in the license. E_K(document || appended_data)
tsm_create filename	Turn a document into a protected document.
quit & **ex**	Erase the plaintext in TSM buffer.

the TSM buffer. They start by bringing the processor into SP CEM mode and finish by exiting CEM mode, hence each of these commands is protected by the SP hardware to ensure they perform the desired functions. All other commands of *vi* remain unchanged. There are a total of 70 commands in the original *vi*, with 2 modified, 5 new ones added and the remaining 68 unmodified. The new and modified commands are described in Table 2.

The above partitioning steps, although applied to *vi* specifically, can also be applied to other applications, with the goal of identifying the entry and exit points of the TSM. We propose the following general methodology for trust-partitioning an application:

1. Identify the security-critical information that needs to be protected.
2. Identify the *liveness* of the information, i.e. transient data or persistent data.
3. Identify the input and output paths leading to and leaving from the protected information.
4. Relocate the information to the TSM buffer (transient data) or the secure storage (persistant data).
5. Rebuild or modify the input and output paths using the new TSM functionalities.

6 Security Analysis

We analyze the security of our proposed solution according to three main security concerns: confidentiality, integrity and availability.

6.1 Confidentiality

In the information sharing policy, the content creator is most concerned with the confidentiality of the sensitive content in the protected document – only the authorized recipients can have access to the decrypted content.

We first consider the case where the adversary is outside the trust group, e.g., the adversary does not have a legitimate SP-enabled device. The adversary

can try to attack the system by intercepting the communication (1) when the content creator is sending the encrypted document over to the recipients, or (2) when the content creator's device is sending the license to the recipients' devices. The adversary does not gain any information in the first case since the document sent over the communication is encrypted, and we assume the use of strong cryptography. Similarly, the communication channel intercepted in the second attack is also encrypted, using the group encryption key, which is known only by an SP device in the same group.

The attacker can also steal one of the recipients' devices and try to impersonate the authorized recipient. In this attack, in order for the adversary to successfully authenticate himself as the authorized recipient, he must know, or have access to, the private key of the authorized recipient.

We now consider the case where the adversary has a legitimate SP-enabled device and belongs to the correct trust group, but is not on the list of authorized recipients. The adversary now is also able to perform the previous three attacks. Further, the adversary can impersonate an authorized recipient and try to communicate with the content creator directly to ask for a legitimate license. However, the most that the adversary can do is to have both the encrypted document and the legitimate license stored in his/her device; the adversary still needs to have the private key of an authorized recipient to authenticate himself.

In the extreme case where the adversary is in the authorized recipient list – an *insider* attack – the adversary can access the contents of the document but has no way of digitally copying the contents to another file, since the plaintext document is only present in the TSM buffer during CEM mode and there is no command that allows direct memory copy of the plaintext from the TSM buffer to unprotected memory. The adversary can take pictures of the displayed content or memorize the content and later re-create it in another file. However, these attacks are not within the control of the computer system and hence, not in our threat model, as stated in Section 3.

Our solutions did not require the application used by the originator to create a new document to be trusted. Although the information could be stolen at this point, this is out-of-scope for this paper, since we are concerned not with the leaking of information when it is being created, but the leaking after it is recognized as important and being distributed.

6.2 Integrity and Availability

The integrity of the protected document and the corresponding policy is enforced by the `Hash` that ties together all the pertinent information of a policy-protected document. The `Hash` is stored in the secure storage, which is itself encrypted and integrity protected by the TSM using the keys accessible only by the TSM. The root of trust of the integrity of the secure storage is stored on the processor chip (SRH). Therefore, there is an integrity trust-chain from the protected content and license to the SRH, that does not depend on the potentially compromised OS.

SP architecture does not directly address denial-of-service attacks, therefore if the adversary modifies or completely deletes the document, or the license in

the secure storage, any access to the protected information is lost. Although it is easy to achieve such denial-of-service attacks, they are not considered detrimental since no security-critical information is leaked by these attacks. In fact, these attacks show the fail-safe nature of the access control implementation. Nevertheless, SP architecture does provide intrinsic support for availability, in terms of the resiliency of the TSM to unrelated attacks. Since the trust chain consists only of the SP hardware and the TSM, attacks on the untrusted part of the application and the OS do not prevent the TSM from enforcing its access control functions.

7 Related Work

Several commercial solutions have been proposed to address the issue of information sharing, both in the context of digital media and digital documents. DRM solutions [3,4,5] focus on the copy-protection of the digital media, with a threat model that assumes that the whole *box* of the computing device is trusted, thus leading to the compromise of the encryption keys as described in Section 1. Cryptolope [19], known as cryptographic envelopes, also decouples the distribution of information and its license (called *superdistribution*) - similar to our solution. Cryptolope enables a commercial platform for the content creator and the publisher to license their content to the customers, by controlling the distribution of the decryption keys. However, Cryptolope assumes the same threat model as other DRM solutions – the device or the software on the device is trusted. Therefore, if an attacker can compromise the operating system or tap the memory bus, the attacker can have access to the decryption keys. Adobe Acrobat [20] has the ability to set permissions to protect sensitive files in the application level, including viewing, printing, changing, copying or commenting. But the password protection employed by Acrobat can be more easily defeated and is vulnerable to a malicious operating system as well. SISA [21] is a recent alliance of several industry companies, aiming to provide a secure end-to-end architecture for information sharing in a distributed environment. It involves several levels of access control, e.g. physical access control, network access control, storage access control, etc. Although the architecture provides extensive defense-in-depth, it still assumes the computing *box* as trusted. Also, the complexity of the architecture may make it more suitable only for large organizations.

Another area of related work is in hardware protection of application software. XOM [22] is another secure processor architecture that protects applications in an untrusted operating system environment. The protected applications running on XOM are kept in different *compartments*, each with its own compartment key. Like SP architecture, XOM has the ability to protect registers and encrypt memory traffic. Therefore, our application-level solution can also be mapped on the XOM processor by executing the TSM in a separate compartment. However, XOM is much more complicated than SP. TPM [6] is an industry solution to support trusted computing. Essentially TPM can be used to provide password protection, disk encryption and, most importantly, a trusted boot-chain. When

employing TPM protection, applications can safely seal a piece of sensitive information inside the TPM chip. In other words, TPM can essentially *bind* a set of files to a particular host. However, since the TPM itself is not designed to provide protection of the decrypted plaintext once it leaves the TPM chip, a malicious operating system or hardware attacker can intercept the decrypted traffic in memory, although he/she cannot obtain the decryption keys in the TPM chip. Flicker [23] employs the newly introduced late launch instructions (both AMD and Intel) together with TPM to achieve a trusted execution environment for the protected part of an application. Like our proposal, it tries to minimize the trusted code base. However, unlike our proposal, Flicker does not consider hardware attacks. Also, our solution can achieve the same level of security without an external TPM chip. Overshadow [24] presented a framework for protecting applications without trusting the operating system. They do not require special hardware (like TPM, XOM or SP) but implement the protection mechanisms in the virtual machine monitor (VMM). They also do not consider hardware attacks and the TCB is larger since it has to include the entire VMM.

8 Conclusion

The SP security architecture provides a simple yet flexible software-hardware mechanism for protecting a Trusted Software Module (TSM) directly by SP hardware. This enables applications to express and enforce different security policies, without depending on the operating system over which they have no control. In this paper, we demonstrated the implementation of an originator-controlled (ORCON) distributed information sharing policy for documents. Such an access control policy is difficult to achieve with only MAC or DAC mechanisms. We achieve this in the user-space *vi* application, without relying on the operating system which can be compromised. The SP protection is rooted in the CPU hardware, defending against both software and hardware attacks. Our modified *vi* application is a proof-of-concept of the effectiveness of the SP hardware-software architecture. We also developed a general methodology for *trust-partitioning* an application, which is useful not only for our information sharing policy, but more generally for separating out the security-critical parts of applications.

References

1. Graubart, R.: On The Need for A Third Form of Access Control. In: 12th National Computer Security Conference Proceedings, October 1989, pp. 296–303 (1989)
2. McCollum, C.J., Messing, J.R., Notargiacomo, L.: Beyond the Pale of MAC and DAC – Defining New Forms of Access Control. In: IEEE Computer Society Symposium on Research in Security and Privacy, pp. 190–200 (1990)
3. Advanced Access Content System (AACS), `http://www.aacsla.com/home`
4. Content Scramble System (CSS), `http://www.dvdcca.org/css/`
5. Leyden, J.: Blu-ray DRM Defeated: Copy-protection Cracked Again (January 23, 2007), `http://www.theregister.co.uk/2007/01/23/blu-ray_drm_cracked/`

6. Trusted Computing Group: Trusted Platform Module, `https://www.trustedcomputinggroup.org/home`
7. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In: SS 2008: Proceedings of the 17th Conference on Security Symposium, Berkeley, CA, USA, pp. 45–60. USENIX Association (2008)
8. Kumar, A.: Discovering Passwords in the Memory, White Paper, Paladion Networks (November 2003)
9. Dwoskin, J.S., Lee, R.B.: Hardware-rooted Trust for Secure Key Management and Transient Trust. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS 2007), October 2007, pp. 389–400 (2007)
10. Lee, R.B., Kwan, P.C.S., McGregor, J.P., Dwoskin, J., Wang, Z.: Architecture for Protecting Critical Secrets in Microprocessors. In: ISCA 2005: Proceedings of the 32nd Intl. Symposium on Computer Architecture, pp. 2–13 (2005)
11. Challener, D., Yoder, K., Catherman, R., Safford, D.: 15. In: A Practical Guide to Trusted Computing, pp. 271–276. IBM Press (2008)
12. Epstein, J.: Fifteen Years after TX: A Look Back at High Assurance Multi-Level Secure Windowing. In: ACSAC 2006, pp. 301–320 (2006)
13. Ocheltree, K., Millman, S., Hobbs, D., Mcdonnell, M., Nieh, J., Baratto, R.: Net2Display: A Proposed VESA Standard for Remoting Displays and I/O Devices over Networks. In: Proceedings of the 2006 Americas Display Engineering and Applications Conference (ADEAC 2006) (October 2006)
14. Kiayias, A., Tsiounis, Y., Yung, M.: Group Encryption. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 181–199. Springer, Heidelberg (2007)
15. Camenisch, J., Stadler, M.: Efficient Group Signature Schemes for Large Groups (Extended Abstract). In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg (1997)
16. Chaum, D., van Heyst, E.: Group Signatures. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 257–265. Springer, Heidelberg (1991)
17. Chen, L., Pedersen, T.P.: New Group Signature Schemes. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 171–181. Springer, Heidelberg (1995)
18. The Traditional vi, `http://ex-vi.sourceforge.net/`
19. Kohl, U., Lotspiech, J., Nusser, S.: Security for the Digital Library - Protectiong Documents Rather Than Channels. In: DEXA 1998: Proceedings of the 9th International Workshop on Database and Expert Systems Applications, p. 316 (1998)
20. Adobe Acrobat Family, `http://www.adobe.com/products/acrobat`
21. Secure Information Sharing Architecture (SISA) Alliance (2007), `http://www.sisaalliance.com/`
22. Lie, D., Thekkath, C.A., Horowitz, M.: Implementing an Untrusted Operating System on Trusted Hardware. In: SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 178–192 (2003)
23. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. In: Eurosys 2008: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 315–328. ACM, New York (2008)
24. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: ASPLOS XIII, pp. 2–13 (2008)