Efficient Isolation of Trusted Subsystems in Embedded Systems

Raoul Strackx, Frank Piessens, and Bart Preneel

IBBT-Distrinet, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{raoul.strackx,frank.piessens}@cs.kuleuven.be,
bart.preneel@esat.kuleuven.be

Abstract. Many embedded systems have relatively strong security requirements because they handle confidential data or support secure electronic transactions. A prototypical example are payment terminals. To ensure that sensitive data such as cryptographic keys cannot leak, security-critical parts of these systems are implemented as separate chips, and hence physically isolated from other parts of the system.

But isolation can also be implemented in software. Higher-end computing platforms are equipped with hardware support to facilitate the implementation of virtual memory and virtual machine monitors. However many embedded systems lack such hardware features.

In this paper, we propose a design for a generic and very lightweight hardware mechanism that can support an efficient implementation of isolation for several subsystems that share the same processor and memory space. A prototypical application is the software implementation of cryptographic support with strong assurance on the secrecy of keys, even towards other code sharing the same processor and memory. Secure cohabitation of code from different stakeholders on the same system is also supported.

Keywords: software security, memory protection, isolation.

1 Introduction

Many embedded systems, including for instance payment terminals and other terminals supporting secure electronic transactions, have relatively strong security requirements. In order to meet these requirements, security-critical parts of these systems, like the cryptographic processor, are implemented as separate chips, and hence physically isolated from other parts of the system [1,2,3,4]. This increases the assurance that sensitive data such as cryptographic keys cannot leak.

But isolation can also be implemented in software. Many of the hardware security features of today's higher-end computing platforms – including memory protection hardware and hardware to support virtualization – were designed to enable the software implementation of efficient isolation of components that

S. Jajodia and J. Zhou (Eds.): SecureComm 2010, LNICST 50, pp. 344–361, 2010.

share the computing platform. Based on these hardware building blocks, efficient software implementations of virtual memory, virtual machine monitors or hypervisors are feasible, and these in turn make it possible to have high assurance isolation between several software components sharing the same physical hardware [5,6,7,8].

Several trends in embedded system design make it interesting to investigate to what extent such isolation mechanisms can play a role in secure embedded systems.

First, the desire to minimize cost pushes towards the reuse of one single processor for tasks that were traditionally divided over physically separated hardware. A prime example are the hardware security modules or cryptographic coprocessors mentioned above: the increased computational power of general purpose processors combined with an increased support for high assurance isolation of software components sharing the processor makes it feasible to design software-based cryptographic coprocessors. Obviously, this raises security concerns that need to be investigated. In particular, one would like to maintain the strong assurance on key secrecy that separate hardware security modules provide.

Second, the co-location of different applications owned by different stakeholders on the same embedded system makes it important to provide high-assurance isolation between these applications. Prime examples are third-party applications on mobile phones, multi-application smartcards, or shared sensor networks. While some of the security requirements of such multi-stakeholder embedded platforms are similar to these of high-end multi-user computing platforms, there are also essential differences, and hence it is necessary to re-evaluate and where needed re-design the security mechanisms to provide secure isolation.

This paper proposes *self-protecting modules (SPM)*: based on a minimal form of hardware support for memory access control, we show how trusted subsystems can share the same processor and memory space, while still maintaining strong security properties including strong isolation guarantees between two such subsystems, and high assurance on the confidentiality of subsystem-private data.

More specifically, the contributions of this paper are the following:

- a novel memory access control model, where access to memory locations can also depend on the value of the program counter,
- based on this access control model, the design of self-protecting modules: software modules that can provide strong security guarantees both for the data they handle as well as for how they can be invoked by other modules,
- a proof sketch of the security of this design,
- and a discussion of several application examples.

The remainder of this paper is structured as follows: in the next Section the threat model will be presented as well as the security properties provided. Section 3 will present SPM's in more detail including an overview, layout of an SPM and the required hardware modifications. We also discuss a proof sketch of the security properties of SPM's. In Section 4 we discuss possible applications. Finally we discuss related work and offer a conclusion.

2 Problem Statement

2.1 Threat Model

We assume that an attacker has the ability to inject machine code of his choice into the memory space of the system under attack. This is a realistic assumption: there are several ways in which an attacker can achieve this. First, the attacker can exploit a software vulnerability such as a buffer overflow in one of the applications on the system, and perform a code injection attack [9,10].

Second, the attacker could be one of the stakeholders in a system where several mutually distrusting stakeholders cohabit the same platform. Third, the attacker may have compromised the software layer below (for instance the OS kernel).

We also assume that the attacker does *not* have the ability to perform a physical attack: he can for instance not disconnect memory from the processor, place probes on the memory bus, or perform a hard reset of the system. An example of such an attack is discussed by Halderman [11]: memory chips containing sensitive information are placed in a machine that is under total control of the attacker, and the secrets can be extracted relatively easy. Such attacks are not considered in this paper. If such physical attacks are an important concern, the software-based implementation of security proposed in this paper is not appropriate.

2.2 Security Properties

Under the threat model discussed above, we want to support the execution of software modules that share the same memory address space guaranteeing the following security properties:

- *Restriction of entry points.* Software modules can securely restrict how they can be invoked. In other words, the entry points into the module can be defined by the module provider. An attacker can not jump to an arbitrary location within the module.
- *Security of module data.* Sensitive information, such as keys, managed by the module can only be read or modified by code from the module.
- *Authentication of modules.* Modules have a secure mechanism of identifying other modules in memory.
- *Secure communication between modules.* Modules can communicate efficiently with other modules they have authenticated. Moreover, the integrity and confidentiality of messages passed over this communication channel can be assured.
- *Minimal Trusted Computing Base (TCB).* The correct and secure execution of a module depends only on (1) the hardware, (2) a small part of the boot process of the system (see Section 3.7), and (3) the correct behavior of the code of the module itself and any third-party modules that it calls. In particular, the operating system kernel is excluded from the TCB.

Note that we do not aim to protect a module against vulnerabilities in its own implementation: if a module contains a logical fault (e.g. a faulty API design [12]),

or an implementation-level vulnerability (e.g. a buffer overwrite [9], a buffer overread [13] or other low-level vulnerabilities [10]), then sensitive data may leak. We only protect the module against attacks that are a consequence of malicious code sharing the memory space of the module. Protection against vulnerabilities in the module itself can be provided by other countermeasures [14].

Note also that we do not protect against denial-of-service: malicious code running on the machine can go into an infinite loop, or can install any number of additional modules thus exhausting CPU-time or memory space. In Section 3.10 we discuss some possible mitigations.

3 Self-Protecting Modules

3.1 Overview

A self-protecting module (SPM) is an area of memory with a particular layout and with particular memory protection settings. Many SPM's as well as other code or data can share the same memory address space. Any code outside the SPM, including code in other SPM's, could be potentially hostile. Here is an overview of how SPM's operate.

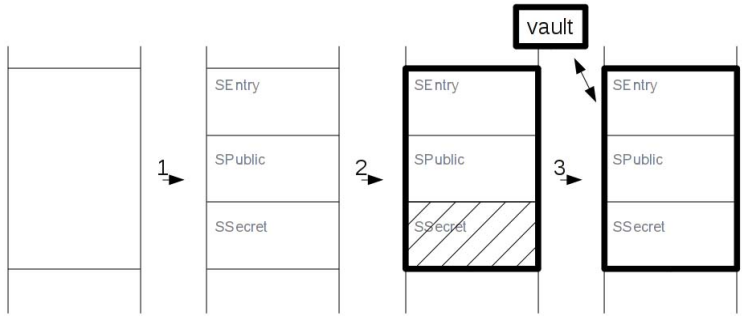
First, an SPM is structured in three sections. Each section is a contiguous range of memory. The SSecret section will contain data that untrusted code should not be able to access directly. The SPublic section will contain data that can be accessed in a read-only manner, as well as the code of the module. Finally, the SEntry section defines the entry points into the module's code: this is a list of pointers into the SPublic section, and the only way to call the SPM is by jumping to an address in this list.

Second, memory access control restricts the rights to read, write or execute memory locations, based on both the value of the program counter (PC), as well as on the address being accessed. For instance, the SSecret memory will only be readable while the PC is in the SPublic section, and the SPublic section is read-only accessible when the PC is outside the SPM. We discuss the access control rules in more detail further on.

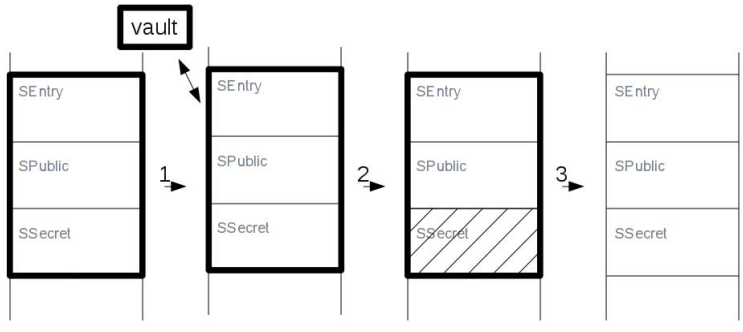
The creation and initialization of an SPM takes several steps and is displayed graphically in Fig. 1a. First, the operating system loads the SPublic and SEntry sections into memory (step 1). This part of the initialization does not need to be trusted: if an attacker interferes with the loading, it will be detected later on.

Second (step 2), a new hardware instruction, `setProtected`, to create the SPM. This instruction defines the boundaries of the three sections of the SPM, enables memory access control, and clears the SSecret section to all zeros. Memory protection enforces, from this point on, that only the SPM itself can destroy itself, or modify its contents. As a consequence, the identity of SPM's can be securely authenticated from this point on: SPublic and SEntry sections are world-readable, and together they define the identity of an SPM.

Third, loading the secret data of the module in the SSecret section requires the assistance of another trusted SPM that we call the *vault* (step 3). This SPM



(a) Initialization



(b) Destruction

Fig. 1. The life of an SPM from initialization (1a) to destruction (1b)

will authenticate the identity of the newly loaded SPM and then provision it with its initial secret data.

Of course, the question then remains how the vault itself gets initialized. For that, we trust (a part of) the boot process: the vault gets installed and provisioned with secret data at boot time, and is never unloaded.

Once SPM's are loaded and initialized, they can securely call functionality of other SPM's. That is: an SPM can call an entry point of another SPM with the following guarantees: (1) it is calling into an SPM with the correct identity, and (2) the integrity and confidentiality of parameters and return values is protected.

Destruction of an SPM is similar to initialization. Fig. 1b displays the steps graphically. First, the vault is used to store secret data securely on untrusted storage. In step two, the secret data is overwritten. Finally, access control on the SPM is disabled and the SPM becomes unprotected memory again.

We now discuss several aspects of this design in more detail.

3.2 Layout of an SPM

An SPM is structured in three memory areas or sections (see Fig. 2) with different access control settings. Table 1 gives a schematic overview and should be read as

follows: the "from" index is determined by the current value of the PC, and the "to" index is determined by the memory address being accessed. For instance, if the PC is in the SPublic section, then memory locations in the SSecret section of the *same* SPM can be read and modified. An identical instruction issued from any other location will be prevented. Note that access to a section originating from a *different* SPM is treated in the same way as access originating from unprotected memory.

SSecret. Sensitive data of the SPM is stored in this section. This includes cryptographic keys and application-level data such as credit card numbers, but also a return stack to implement an SPM's functionality and other control flow data.

In contrast with the other sections, *any* attempt to access this section from outside the SPM will fail. This provides complete isolation of secret data. Data can only flow out or into this section using the functionality provided by the self-protected module.

In our design, execution of instructions stored in the SSecret section is prevented for the following reasons: (1) the SSecret section is not a part of the identity of the SPM that can be authenticated by other SPM's, and (2) making any the only writable section non-executable has important security advantages from the point of view of protecting against vulnerabilities in the SPM itself [15,16].

SPublic. Contrary to the SSecret section, the SPublic section can be read from any location, including from unprotected memory locations. The instructions implementing the functionality of the SPM are placed in this section, as well as constant non-secret data such as security certificates. The hardware implemented access control will prevent write instructions to this section from *any* location. Therefore, it can't be modified after access control on the module is enabled and SPM's can be authenticated easily (see Section 3.5).

The code in the SPublic section is assumed to be trustworthy. It is responsible for instance to prevent undesired leaking of secret information to untrusted memory locations or to untrusted SPM's. It should also make sure that an attacker cannot inject false data.

SEntry. It is very hard to guarantee good properties of a piece of machine code if one cannot restrict the possible entry points into the code [17]. By carefully choosing the destination of a jump instruction, security sensitive code, such as encryption functions, could be skipped.

To prevent such attacks, direct calls to the SPublic section from outside the SPM are prevented (see Table 1). But jumping to the SEntry section is allowed. This section contains a list of jump instructions to valid locations in the SPublic section. By making SEntry executable from outside the secured section, and SPublic not, entry points are effectively restricted to those listed in the SEntry section. Note that jumping from SEntry to SPublic is allowed by the memory access control model.

Modifications of the SEntry section are prevented by marking it only read and executable, both from within as from outside the secured section.

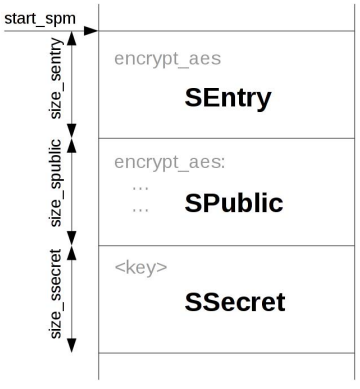


Fig. 2. The layout of an SPM in memory

Table 1. The memory access control matrix

from\to	SEntry	SPublic	SSecret	unprotected
SEntry		x		
SPublic	rx	rx	rw	rwx
SSecret				
Unprotected/other SPM	rx	r		rwx

3.3 Hardware Modifications

In order to use the proposed solution, some hardware modifications are required. Besides the access control model, three instructions need to be supported.

setProtected. Installation of an SPM starts with loading the content of its sections into memory. Up to this point this content is not protected but any modification will be detected later on. Only after successful execution of the `setProtected` instruction with the correct parameters, access control is enabled and the SPM is protected from hostile code stored at any location outside the SPM, running at any privilege level.

To simplify checks executed before protection is enabled, the `setProtected` instruction assumes a fixed ordering of the SPM's sections in memory. The SEntry section is always placed at the lower memory locations immediately followed by the SPublic and SSecret sections, respectively. Using this fixed layout, the instruction only requires 4 arguments; `start_spm`, `size_sentry`, `size_spublic` and `size_ssecret` (see Fig. 2). The first argument, `start_spm`, provides the address of the lowest memory location that will be protected, the base of the SEntry section. The other arguments provide the length of each section as they are placed in memory.

Before access control can be safely enabled, a check needs to be performed that the new SPM will not overlap with an existing one[1].

When the check succeeds, the content of the SSecret section is blanked with zeros to prevent an attacker from injecting false data. Finally, the SPM is protected by enabling access control on each section.

isProtected. Before secret data can be passed between SPM's securely, they should be able to authenticate one another. The ability to read the code and public data part of an SPM is not sufficient. Its correct installation must be proven. This not only includes that the access control is enabled, but also that the layout of the SPM is as expected.

The `isProtected` instruction takes a memory location as an argument and returns the layout of the surrounding SPM in the same format as expected by the `setProtected` instruction. In case the memory location is not protected an error value is returned.

resetProtected. Once an SPM is created, its protection cannot be disabled from outside the SPM, not even by code running at the processor's highest privilege level. Only the SPM can remove it by executing the `resetProtected` instruction.

To keep data stored in the SSecret section secret from attackers, it should be destroyed before access control is disabled. Since we need to trust the SPM code to correctly clean up for other purposes as well (see Section 3.6), we require the SPM to overwrite the data explicitly rather than blanking it automatically when the `resetProtected` instruction is issued.

3.4 Initialization of SPM's

Initialization of SPM's takes three steps (see Fig. 1a). First, the content of its SEntry and SPublic sections is loaded in unprotected memory (step 1). Next, its SSecret section is blanked by the `setProtected` instruction and access control on all sections is enabled (step 2). Finally, the SPM should initialize its internal data structures (step 3). For example, a new return stack should be created within the SSecret section as control flow data of the SPM should never be stored at an unprotected location. There are two approaches for the initialization of SPM's.

First, it may be possible to initialize it using only public data. This situation occurs when only secure execution is an issue, not secrecy. When the provided data stays the same over time, it could be shipped and placed alongside the code in the SPublic section. To allow its modification by the SPM, it can be copied to the SSecret section. In case the public data changes repeatedly over time and only integrity needs to be protected, the SSecret section could be created and cryptographically signed by a trusted third party and sent to the SPM. After checking the signature, the provided data can be used to initialize the SPM.

[1] In principle the SEntry and SPublic sections could be shared by multiple instances of the same SPM to reduce memory consumption. This optimization and its security issues are considered to be out-of-scope for this paper.

Second, the SSecret section could be initialized using secret data stored in a cryptographically encrypted and signed file. As a secret key must be available for decryption and since prior to initialization, an SPM can not contain secret data, help of another trusted SPM is required. Our design proposes a special SPM called *vault* to provide such functionality. We discuss this in Section 3.7.

3.5 Authentication of SPM's

Previous sections described how an SPM could be loaded into memory from an untrusted source. Even when a software module has been received correctly, it may have been modified while it was stored on disk. Before it can be trusted with secret data, its trustworthiness must be validated.

For this purpose, each SPM is shipped with a *security report*. It states that the correct implementation of the SPM has been verified by its issuer. In case that third party is trusted, so can the SPM when the security report is valid. Recall that our threat model assumes that the SPM does not contain logical faults nor implementation-level vulnerabilities (see Section 2.1).

By placing the security report in the SPublic section, it can be accessed easily and efficiently as access control of the SPM allows read access from any location.

Each security report contains following information:

- *Hash of SEntry and SPublic sections*: To be able to establish trust in an SPM, it must be identical to the SPM certified by the trusted third party. By providing a hash result of the SEntry and SPublic sections[2], any modification will be detected.
- *The layout of the SPM*: When incorrect parameters are supplied with the `setProtected` instruction, the SPM may use unprotected memory locations to store secret data. To avoid such situations, the layout of the SPM is included in the security report. Using the results of the `isProtected` instruction, the layout of the newly installed SPM can be validated.
- *Cryptographic signature*: The security report is signed with its issuer's private key. An SPM that wishes to verify the trustworthiness of another, has a list of trusted certificate authorities (CA's). When a chain of trust can be built from a CA to the public key of the issuer, the security report can be trusted.

If an SPM A wishes to authenticate SPM B, it should (1) verify the signature of B's security report, (2) verify the hashes of the SEntry and SPublic sections, and (3) verify the SPM layout using the `isProtected` instruction.

3.6 Secure Communication

The ability to communicate securely between two mutually trusted SPM's does not only result in a more modular system, it is also required to bootstrap the system. Section 3.7 describes how an SPM loaded from an untrusted location

[2] The hash of the SPublic section implies knowledge of the security report. To break this circular dependency, the security report is replaced with zeros during calculation.

can be authenticated and provided its secret data. This Section presents how two SPM's can communicate with one another while preserving secrecy and integrity of the exchanged messages. Injection of false data is prevented as well.

Each of the presented protocols assumes that the SPM's know each others location and implemented functionality. In practice this can be accomplished by requiring each SPM to register itself to a centralized service. As this service does not have to be trusted, its inner workings are not considered in this paper.

One-Way Authentication. Some applications only require that one endpoint of the communication channel is authenticated. Consider for example an SPM *SecureRandom*, providing the service of secure random number generation. For obvious security reasons, the client needs to authenticate the service. SecureRandom in turn, has no need to verify the trustworthiness of its client as it does not leak any secrets, it only creates unpredictable random numbers.

The protocol described in this Section offers the following security guarantees for communication between SPM's: authentication of one endpoint, secrecy and integrity of the messages sent and received.

Fig. 3 displays the protocol. In the first step, the client authenticates SecureRandom. It does so by fetching its security report from its SPublic section and validating it. This operation can be performed without leaving the client SPM as SEntry and SPublic sections are world-readable (see Section 3.2).

Next, the generation of a new random number is requested. This request can be made similar to an ordinary function call; by jumping to the correct location in the SEntry section of SecureRandom and passing arguments in registers. However, unlike a function call, execution cannot return directly to the instruction following the call instruction, as this would provide SecureRandom with a way of (re-)entering the calling SPM at an address that is not in the SEntry list, thus enabling return-into-libc-like attacks [17]. The access control on the SPM will prevent such jump instructions into the SPublic section as it originated from outside the SPM. Instead, returning from a service call is implemented by creating a new entry point, `client_entry`, in the SEntry section and sending it as an argument with the request. After the random number is created, SecureRandom will then issue a jump instruction to the specified entry point. There control flow is directed to the correct, fixed, location, as allowed by the access control. This is similar to continuation-passing-style programming.

In the third step of the protocol, the random generator returns the random number `k`, by placing it in a register and jumping to the return entry point specified by the client.

Placing sensitive information in registers is an inexpensive solution as it does not require encryption and signing. Unfortunately it can only be used when a small amount of data needs to be transferred from one SPM to another. When bulk data needs to be exchanged, it can either be divided and transported using multiple jump instructions, or it can be communicated in untrusted, unprotected memory after appropriate encryption and signing. The keys used can than be exchanged securely in registers.

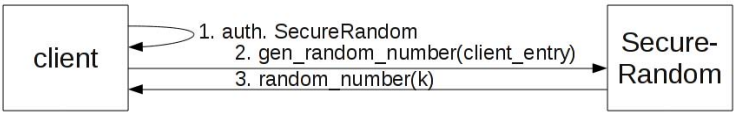


Fig. 3. One-way authentication between SPM's. A client authenticates the random number generator before requesting a new random number.

Mutual Authentication. The protocol presented in Section 3.6 can be modified easily to authenticate both communication endpoints (see Fig. 4). As before, the client initiates the protocol and authenticates the vault. Next, a message is sent requesting the secret data. The entry point to be used to return the data, `client_entry`, is added as well as the location of the security report of the client, `client_sec_rep`.

At the reception of the message, vault must verify that the security report of the client is valid and trusted. To prevent sending the secret to an unprotected location or to an incorrect SPM, vault also has to check that the given entry point is located within the SPM described by the security report. Only when both tests are valid, the secret information, `k`, will be returned.

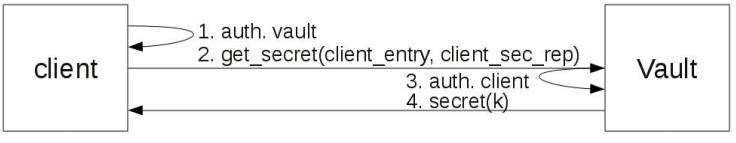


Fig. 4. Two-way authentication between SPM's

Mutual Authentication with Support of Both Endpoints. The previous protocol is inefficient in case multiple authenticated communication events between the same SPM's occur. With support of both endpoints, performance overhead can be reduced by avoiding repeated checks of the security reports.

Fig. 5 displays the protocol where SPM's A and B wish to communicate. The protocol establishes a persistent secure channel in only two passes.

First, A authenticates endpoint B. After trust is established, the `notify_destruction` entry point of B is called providing an entry point of A, $notify_A$ that should be called when B is about to be destroyed. A freshly generated cryptographic nonce N_{BA} is also added to the request. As only B has knowledge of the nonce, any message containing N_{BA} must be sent by B. This avoids repeated authentication of B.

In the second part of the protocol endpoint B performs identical steps, providing A with the entry point $notify_B$ and the fresh nonce N_{AB}. Now A and B are able to communicate securely without repeated authentication events. As before, entry points of the other SPM can be called passing secret data in registers. Providing the received nonce with each communication event, proves the origin of the message.

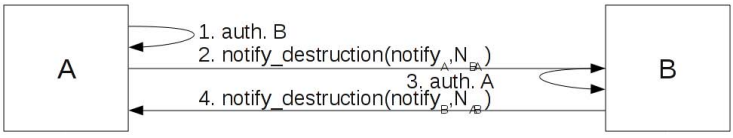


Fig. 5. Two-way authentication between SPM's with support

3.7 Vault: Bootstrapping Trust

After loading the SEntry and SPublic sections into unprotected memory and calling the `setProtected` instruction (step 1 and 2 in Fig. 1a respectively), the SSecret section of newly created SPM's is blank. As described in Section 3.4, SPM's can be easily initialized using public data (step 3). However, in some cases secret data, for example a cryptographic key, of a previous instance of an SPM needs to be restored. A special SPM called *vault* provides such functionality. It is able to store secret data securely in persistent but untrusted memory and guarantees that the secret data will only be returned to the same SPM that requested its storage.

Requesting storage goes as follows. First, an SPM establishes an one-way authenticated channel to the vault. Next, the secret data is transferred to the vault where it is appended with the security report of the requesting SPM, encrypted and signed with the vault's cryptographic keys and stored in persistent storage. Note that vault only stores secret data from other SPM's. As it does not provide any of its own secrets, it does not have to trust its clients.

A different instance of the same SPM, for example after the system is rebooted, is now able to retrieve the stored secrets from the vault. First, it establishes a two-way authenticated channel with the vault. The secret data is fetched from persistent storage[3], its signature checked and decrypted by the vault. Only when the stored security report matches the requesting SPM's, the secret data is passed over the secure channel.

This leaves the problem of how the vault itself gets initialized with its keys. For this we trust (a part of) the boot process: the system is modified to create the vault as early in the boot sequence as possible. Unlike any other SPM, its secret data is provided directly by hardware by copying it from protected memory that is only accessible at boot time.

3.8 Destruction

Before protection on the module is disabled, the SPM's destruction should be prepared. In general two cases need to be considered. First, the stored secret data in the SPM. When access control is disabled, it can be accessed from any location. To avoid disclosing it in unprotected memory, it needs to be overwritten.

[3] How the secret data is found on persistent storage is not relevant from a security point of view and is omitted for clarity. However, the vault could return an identifier, for example a filename, when storage is requested and stored unprotected.

Second, other SPM's may assume the presence of the trusted, protected module at a certain memory location. These SPM's need to be notified of the imminent destruction to prevent them from issuing jump instructions to unprotected and/or untrusted code while passing secret data in registers.

3.9 Discussion

Limitations of the Current Design. The proposed solution reduces the TCB to only the SPM's used, a small part of the boot process and the hardware. By eliminating the kernel from the TCB, its correct behavior can no longer be trusted upon and access in *any* way conflicting with the access control model presented in Section 3.2 needs to be prevented. As a result, support of many advanced features, such as interrupts, virtual memory and others, must be implemented in the SPM's, in collaboration with the hardware. Supporting these features is considered to be out of the scope of this paper.

- *interrupts*: when an interrupt occurs during the execution of an SPM, sensitive information stored in registers will be accessible to the kernel. Therefore SPM's should be executed in the highest interrupt level, preventing interrupts from being handled during execution. Hence, SPM implementors should make sure that SPM calls return within a reasonable amount of time.
- *swapping*: in case more memory is required than is available on the machine, the kernel will swap chunks of memory to disk. This may not only prevent the correct memory location from being called, as secret data is stored at an unprotected location, confidentiality and integrity may be compromised. Therefore, swapping of SPM's should be prevented.
- *direct memory access (DMA)*: peripheral devices often use DMA to access memory locations directly; protected memory locations must be excluded.
- *paging*: paging allows the same physical page to be mapped to different address spaces, even at different addresses. In itself this does not pose a security problem as long as access control remains correctly enforced. In practice this may be difficult, as an SPM may span multiple pages and additional pages may be injected at runtime. Support for paging is considered to be out-of-scope, SPM's are currently expected to use physical addresses.
- *concurrent execution of SPM's*: When multiple SPM's exchange secret data, authentication of endpoints does not suffice. In the limited amount of time between authentication and a communication event, one of the endpoints may have been removed. Issuing jump instructions in that case, may leak secret data stored in registers. For this reason, concurrent execution of SPM's is currently not supported.

Note that these assumptions are only made during the execution of self-protecting modules. Execution of unprotected code, including concurrent execution on a different core, are not restricted as long as the access rights presented in Section 3.2 are enforced. By using a multi-core processor to execute the kernel on a different core than the SPM's, the system will remain responsive.

Proof Sketches of Security Guarantees

Security of module code. After installation, an SPM contains all the code that implements its functionality. It cannot be modified nor influenced, not from outside nor from within the SPM, and can only be called using the entry points into the module as defined by the module's provider.

This property almost directly follows from access control enforced on SPM's. Only the SEntry section is executable from outside the module. A user has no other option than to use these specified entry points. After calling an entry point, the SPM is entered and control flow is directed to the SPublic section. Both SPublic as SEntry sections are now executable. As both sections are not writable from any location, an attacker is not able to modify the stored instructions.

Only modification of control data still needs to be considered. In the SSecret section, a return stack may be built to allow easy implementation of the SPM's functionality. Implementation vulnerabilities may allow this data to be overwritten, for example by exploiting a buffer overflow vulnerability [9,10]. Modification of a return address, frame pointer, or other control data may result in modified behavior of the SPM as defined by the SPM's provider. However, it is assumed that code stored in the SPM does not contain such vulnerabilities.

While an attacker is able to modify the code of the SPM before access control is enabled, such modification will be detected during authentication.

Security of module data. Sensitive data such as keys stored in the module must be protected. Access must be restricted to the SPM and unless explicitly specified, secret data must not leak. For example, a key may only leave the SPM when it is securely passed to another, trusted, SPM or when it is encrypted and signed.

Isolation of data stored in the SSecret section is directly provided by the access control model enforced upon it. Any access attempt from outside the SPM is prevented. In contrast, instructions within the SPM are allowed to read and modify data stored in the SSecret section. Considering that the code of the module is secure and cannot be modified nor influenced after installation, as stated by the previous security guarantee, we conclude that secret data can only leave the SPM as implemented by the SPM's provider.

Finally the destruction of an SPM needs to be considered. As access control only allows an SPM to disable its own protection and the SPM is able to enforce the conditions under which the `resetProtected` instruction is issued, any secret data can be overwritten prior to the destruction of the SPM.

Secure communication between modules. Combining the strong isolation of data and code with a secure communication scheme between SPM's, will result in a modular and secure subsystem. Only the existence of such a secure communication mechanism still needs to be argued. In order to pass secret data between modules securely, (mutual) authentication and a secure channel are required.

To authenticate an SPM, its implementation needs to be verified. Access control restricts execution of code within the SPM to the SEntry and SPublic sections. As these sections are world-readable, the functionality of an SPM can be

checked easily. The presence of self-modifying code or code injection attacks does not have to be considered; in a previous paragraph it is already argued that after installation the code of an SPM cannot be modified. Finally, the `isProtected` instruction can be issued to check the correct setup of the SPM's protection.

Authentication can only consider the SEntry and SPublic sections as the SPM's protection will prevent access to the SSecret section. Without proper security measures, an attacker may still carefully craft an SSecret section. This could, for example, trick the trusted code into storing received secret data at an unprotected location encrypted using a key under the attackers control. Such spoofing attacks are prevented by automatically clearing the entire SSecret section when the `setProtected` instruction is issued. After initialization the SPM's are responsible to prevent injection of false data.

Next, a secure channel between two modules can be established. Authenticity and confidentiality of the exchanged messages must be provided. SPM's can be called like ordinary functions; by directly modifying the program counter of the processor. Messages can be passed using registers. It is assumed that the execution of modules cannot be interrupted. Therefore, the secrecy of data passed in registers cannot be breached and both requirements are met.

Finally, it must be assured that the secure channel is set up between the correct SPM's. Between authentication and the first message, an endpoint may be destroyed. In that situation, secrecy of the data stored in registers cannot be ensured. An attacker may have replaced the SPM with malicious code.

To prevent such situations, control flow must not leave the SPM between authentication and the communication event. Because SPM's can only be destroyed by themselves, the authenticated SPM must have been entered between the two events. However, it is enforced that at any time no two SPM's can be executing simultaneous. Therefore such attacks are prevented.

Minimal trusted computing base (TCB). The hardware and implementation of *vault* forms the root of trust of the systems. Because modules are able to control the flow of secret data to authenticated modules or to unprotected memory under specific conditions, a chain of trust can be built. This trust does *not* include the kernel. As a result, the TCB only consists of the trusted modules, a small part of the boot process creating a root of trust and the hardware.

3.10 Extensions

The current design does not allow SPM's to be interrupted during execution or swapped to disk. Leveraging these limitations, an attacker is able to execute a denial-of-service attack (DoS), for example, by installing an SPM that goes into an infinite loop. On devices that support multiple privilege levels, the chances of such an attack can be reduced easily. By restricting the `setProtected` instruction to kernel mode, an attacker with only user privileges must request the kernel for the protection of an SPM. Before this request is validated, security checks can be performed. For example, only SPM's of trusted issuers could be allowed. Note that an attacker who compromised the kernel to avoid these restrictions, is also able to power the system down, executing a similar denial-of-service.

4 Applications

Many embedded systems, for instance payment terminals and mobile phones running third-party applications including m-banking applications, have strong security requirements. For example, sensitive data such as cryptographic keys must not leak. To assure these requirements, secret data is stored and computed on a physically separated co-processor and memory, packaged together on a single chip called a *hardware security module (HSM)*. Many modern PC's are already being shipped with such a chip [1]. Similar hardware for mobile devices is being developed.

However, isolation can also be guaranteed by SPM's in software. This reduces manufacturing cost as a separate co-processor and memory is no longer required. For the same reason power consumption is reduced, making secure isolation possible for low-end devices or improve mobility. Prime examples are multi-application smartcards and shared sensor networks.

However, there are differences between HSM's and SPM's. First, a Trusted Platform Module (TPM) [1], the HSM found on many desktop PC's, can only execute the cryptographic algorithms installed on the chip when it was manufactured[4]. Other algorithms on secret data still need to be executed in unprotected memory under a huge TCB. In contrary, SPM's are able to isolate *any* module.

Second, many HSM's do have advantages over SPM's. Secret data can also be protected against physical attacks. However, in many situations the user with physical access to the device can be trusted. A user trying to access his banking account, for example, is only interested to keep his/hers login data secure.

Third, HSM's could also be used to improve performance. As they are built with a specific purpose, they can be more easily optimized for performance. However, when the HSM chips can be omitted, it could be replaced by an additional, general-purpose processor core. This would allow a performance improvement of any process, not just cryptographic algorithms.

5 Related Work

Many security measures have been proposed to increase the security of computing devices. Early work proposed hardware support for multiple privilege modes in the processor to separate trusted from untrusted code [18]. From these added hardware features, a balance between secure and performant architectures have been investigated, leading to a whole design-space ranging from micro-kernels to large monolithic kernels [19].

The Dyad HW[2] and other architectures [3,4] uses hardware features more extensively. Executing trusted code on a co-processor provides strong isolation, even protecting against physical attacks.

Hardware security modules such as the TPM [1], allow integrity measurements during boot process. While it is able to attest a trusted boot sequence, it

[4] However, there exist HSM chips that are able to execute custom algorithms within the secured boundaries.

relies on the correctness of the entire code base [6]. An infeasible secure solution considering the millions lines of code of modern monolithic kernels.

Recently, virtualization techniques, with or without hardware support, are considered to provide isolation between trusted and untrusted code. For example, Nizza [5], uses a minimal, trusted kernel to run both trusted AppCores and a legacy operating system running untrusted processes. However, its TCB still consists of hundred of thousands lines of code.

Oslo [6] takes advantage of virtualization instructions found in recent AMD™ and Intel® processors to establish a dynamic root of trust, providing more flexibility. Flicker [7,8] also takes this approach. Running trusted code as virtualized machines, called PAL's and taking advantage of the functionality of a TPM, strong isolation is provided with a small TCB. However, maintaining state between a PAL's executions incurs a large performance overhead. In addition, the requirement of both a TPM as hardware supported virtualization make it ill-equipped for mobile and embedded devices.

6 Conclusion

Many embedded systems, for instance payment terminals and mobile phones running third-party applications including m-banking applications, have relatively strong security requirements.

We propose a novel access control model where access to memory locations also depends on the value of the program counter. Using this approach a secure subsystem can be built and isolated in software instead of hardware, reducing manufacturing cost and offering strong security guarantees to low-end devices such as multi-party smartcards and sensor-networks.

Acknowledgments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Trusted Computing Group: Tpm main specification, `http://www.trustedcomputinggroup.org/`
2. Yee, B.: Using secure coprocessors. PhD thesis (1994)
3. Smith, S., Weingart, S.: Building a high-performance, programmable secure coprocessor. Comput. Networks 31(8), 831–860 (1999)
4. Chen, B., Morris, R.: Certifying program execution with secure processors. In: USENIX HotOS Workshop, pp. 133–138 (2003)
5. Singaravelu, L., Pu, C., Härtig, H., Helmuth, C.: Reducing TCB complexity for security-sensitive applications: Three case studies. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. ACM, New York (2006)
6. Kauer, B.: OSLO: improving the security of trusted computing. In: Proceedings of 16th USENIX Security Symposium, pp. 1–9. USENIX Association (2007)

7. McCune, J., Parno, B., Perrig, A., Reiter, M., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008. ACM, New York (2008)
8. McCune, J., Perrig, A., Reiter, M.: Safe passage for passwords and other sensitive data. In: Proceedings of NDSS (2009)
9. Aleph1: Smashing the stack for fun and profit. Phrack 49 (1996)
10. Younan, Y., Joosen, W., Piessens, F.: Code injection in c and c++: A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven (2004)
11. Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: Cold boot attacks on encryption keys. In: USENIX Security Symposium, pp. 45–60 (2008)
12. Longley, D., Rigby, S.: An automatic search for security flaws in key management schemes. Computers & Security 11(1), 75–89 (1992)
13. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: EUROSEC 2009: Proceedings of the Second European Workshop on System Security, pp. 1–8. ACM, New York (March 2009)
14. Erlingsson, Ú.: Low-level software security: Attacks and defenses. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2007. LNCS, vol. 4677, pp. 92–134. Springer, Heidelberg (2007)
15. Microsoft Corporation: Changes to functionality in microsoft windows xp service pack 2,
`http://www.microsoft.com/downloads/details.aspx?FamilyID=7bd948d7-b791-40b6-8364-685b84158c78`
16. The PaX Team: Documentation for the pax project,
`http://pax.grsecurity.net/docs/pax.txt`
17. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and communications security, p. 561. ACM, New York (2007)
18. Corbato, F., Vyssotsky, V.: Introduction and overview of the Multics system. In: Proceedings of the fall joint computer conference, part I, November 30-December 1, pp. 185–196. ACM, New York (1965)
19. Liedtke, J.: Toward real microkernels. Communications of the ACM 39(9) (1996)