KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

A Security Kernel for Protected Module Architectures

Alexandru - Madalin Ghenea

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science, specialisation
Secure Software

Thesis supervisor:
Prof. dr. ir. Frank Piessens

Assessors:
Dr. A. Hovsepyan
Dr. J.T. Mühlberg

Mentors:
Dr. J.T. Mühlberg
Ir. J. Van Bulck

Academic year 2016 – 2017

Preface

I would like to start by thanking my advisors Jan Tobias Mühlberg and Jo Van Bulck for their patience, guidance, mentoring and continuous support through the year. I also want to thank them for the weekly meeting, without which I wouldn't have been able to learn so much and improve the quality of the thesis. I would like to thank my promoter prof. Piessens for allowing me to work on such an up-to-date and interesting topic. I would also like to thank the jury for taking time to review the text. My sincere gratitude also goes to my family and friends for their continuous support and encouragements.

Alexandru - Madalin Ghenea

Contents

Abstract

More and more embedded devices are being connected to a network. Network connectivity adds more attack vectors, making them more susceptible to security threats. While for high-end systems there are many security solutions, most of these solutions rely on features such as virtual memory and CPU privilege levels, which are not available for low-end embedded systems. Because of this, finding security solutions for low-end embedded systems is still an active area of research. Therefor, security solutions are required to make embedded systems more secure. This need has lead to the creation of the concept of Protected Module Architecture (PMA), which can provide strong security guarantees. There are many implementations of PMAs, both for high-end devices and for low-end devices. One PMA implementation which focuses on low-end devices is Sancus [21]. Sancus provides strong security guarantees by relying on a small Trusted Computing Base made only of hardware.

This master's thesis studies the trade-offs and the security properties that can be obtained by adding a security kernel. The security kernel prototype developed for this thesis is composed of two components which try to tackle two main goals. One component is designed to protect against call-stack shortcutting attacks, through the use of a shadow call-stack. Based on the results obtained, the component that protects against call-stack shortcutting adds a computational overhead that should be acceptable for many applications.

The other component is designed to replace the hardware cryptographic module with a software implementation while trying to maintain the same security guarantees. A software implementation would reduce hardware costs, would allow for more flexibility, by being able to change the implementations after deployment and would be a step towards interruptibility since the hardware implementation cannot be interrupted. The prototype developed provides only local attestation under the same conditions as the hardware implementation. Remote attestation, however is restricted by more assumptions. It is left for future work, to provide a solution for remote attestation under the same assumptions as the original version of Sancus. The measurements that were done show that the software implementation adds significant computational overhead and that this overheard depends heavily on the algorithm used for implementing the cryptographic operations.

The developed components can be used either together or independently, depending on the requirements of the application for which the security kernel is used.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
EA-MPU	Execution-Aware Memory Protection Unit
IoT	Internet of Things
IPC	Inter-Process Communication
ISMC	Inter Software Module Communication
LOC	Lines of Code
MAC	Message Authentication Code
MPU	Memory Protection Unit
PCBMAC	Program Counter Based Memory Access Control
PMA	Protected Module Architecture
ROM	Read Only Memory
RTM	Root Of Trust for Measurement
PROM	Programmable Read Only Memory
PRV	Prover
TCB	Trusted Computing Base
SD	Standard Deviation
SM	Software Module
SMART	Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust
SPM	Self Protecting Module
VRF	Verifier

Chapter 1

Introduction

In recent years there has been a growing trend towards increasing the network connectivity of embedded computing devices and towards an Internet of Things (IoT) [33, 13].A significant part of these devices are used in safety-critical and security-sensitive applications, such as automotive applications, industrial control systems, critical infrastructure, medical applications and e-payment systems [16, 5]. By adding network connectivity, the platforms are more susceptible to remote security threats. Studies have shown varied threats and vulnerabilities, such as using code injection to build self-propagating worms [11, 10], remote car hijacking [18], infecting network connected appliances with botnets [25].

While for high-end systems, connectivity and software extensibility problems are well studied, for resource-constrained embedded systems, finding efficient security solutions is still a very active area of research [16, 5, 20, 27, 8, 1]. For high-end systems, there are many security solutions, depending on the requirements. For example, privilege levels [3, 24], support for virtual memory, secure co-processors [26], firmware security services, verifiable execution of measured code [12]. These solutions require extra hardware, which in turn increases the costs of the system. Because of the increased costs, those solutions are not available for low cost embedded systems. In recent years a new concept called Protected Module Architectures (PMAs) has emerged, that provide strong isolation. PMAs are suitable for both high-end devices, providing extra security in combination with the solutions presented earlier and for low-end devices, providing strong isolation without the need of more expensive mechanisms. In PMAs the security-critical components are separated into smaller protected modules, which are isolated by hardware or a small software layer from all other software on the system such that they cannot be tampered with. The separation allows for the security critical modules to be rigorously verified for bugs and vulnerabilities.

There is a wide array of implementations for PMAs, on different levels of architecture going from complete hardware implementation (SMART [8], Sancus [20]) to a mix between hardware and software (TrustLite [16], TyTAN [5], Intel SGX [17]) to OS kernel implementations (Salus [28]) or hypervisor-based solutions (Fides [30]), providing different security guarantees based on the implementation.

1.1 Goals

The goal of this thesis is to create a security kernel for Sancus [20], a Protected Module Architecture specifically designed for resource constrained networked embedded devices and study some of the security properties and trade-offs that can be obtained by this addition. Additionally, the security kernel is designed to replace some of the hardware mechanisms with a software implementation and study the effects of this change.

1.2 Summary of results

The security kernel developed for this thesis is composed of two components, which can be used independently or together. First, a component which protects against call-stack shortcutting attacks. If used independently, it adds significant overhead in terms of cycles during the first calls. Subsequent calls add only a small overhead. The second component provides a software solution for local attestation using a software implementation of a hashing algorithm. This reduces hardware costs and is a step towards full real-time compatibility. If SPONGENT, the same hashing algorithm as the one implemented in hardware, is used, the overhead in terms of CPU cycles obtained is very large, making the kernel unsuitable for almost any application. However if SHA-2 algorithm is used, the overhead is significantly reduced. Given additional assumptions, the software local attestation mechanism is sufficient for remote attestation as well. The source of the security kernel developed can be found at: https://github.com/nighthawk017/sancus_kernel .

1.3 Outline

The remainder of this thesis is organized as follows:
Chapter 2: Background and related work This chapter introduces the relevant notions required for understanding the contributions of this thesis. More specifically, it describes in detail the concept of Protected Module Architecture and several implementations developed for embedded systems.
Chapter 3: Problem statement and Solution Outline This chapter discusses several short-comings of the Protected Module Architecture *Sancus* and introduces a the concept of a security kernel for Sancus with the purpose of tackling the disadvantages described in this chapter.
Chapter 4: Mitigating Call-stack shortcutting This chapter presents the motivation for a software implementation for inter-software communication mechanism. After that, the design of this mechanism is explained in detail and evaluated.
Chapter 5: Secure linking and Software Attestation This chapter presents the motivation for a software implementation for software local attestation. After that, the design of this mechanism is explained in detail and evaluated.
Chapter 6: Conclusion This chapter presents the final conclusions based on the

work of this thesis. After that, possible applications for the security kernel are discussed and finally the future work for the security kernel is proposed.

Chapter 2

Background and related work

2.1 Protected Module Architectures

Most devices have multiple software programs deployed from mutually distrusting software providers. Because the software programs deployed are mutually distrusting, there is a need for an isolation mechanism, such that software programs cannot influence each other in undesired ways. The conventional software isolation mechanism relies on the Operating System to allocate virtual address spaces for each software program. The main drawback of this approach is that the security and integrity of the programs rely on the kernel of the Operating System, which can be very large and hard to check for vulnerabilities. That is why instead of trying to guarantee the security of large components, another approach is to minimize the security critical components.

Trusted Computing Base (TCB) refers to the hardware and software components of the security architecture that provide the security guarantees. Exploiting components outside the TCB cannot affect the expected behavior of the TCB. Because the TCB is so critical to the security and integrity of the whole platform it is important for it to be as small as possible to allow exhaustive checks for bugs and other vulnerabilities.

PMAs are security architectures that provide strong isolation for software programs without the need of an operating system. PMAs usually also provide other security features such as local and/or remote attestation, sealing and data and/or code confidentiality [29]. While PMAs can be used for both high end systems in order to strengthen security even more and low end systems where PMAs can provide strong security guarantees without the need for more expensive mechanisms, the focus in this paper will be on PMAs that are targeted for low end embedded systems. Many of the PMAs designed for embedded systems use Program Counter Based Memory Access Control (PCBMAC) to enforce software isolation [31, 2]. The PCBMAC is a mechanism that enforces memory access control rules that can take the program counter into consideration. While PCBMAC has a lower hardware cost than virtual memory and does not influence the critical path [21], it can guarantee even stronger isolation between software, thus making it suitable for low-end embedded systems

[21, 8].

As previously mentioned there is a large spectrum of PMA implementations, from complete hardware implementation on the one end to complete software implementations on the other. Hardware implementations can withstand attackers that can manipulate any deployed software, but cannot be upgraded or modified afterwards, thus being less flexible than software implementations which can provide upgradability, but require mechanisms to protect TCB code from attackers. The remainder of this section will revolve around three implementations: Sancus, TrustLite and TyTAN. Sancus relies only on a hardware TCB and is on the hardware end of the PMA implementation spectrum. TrustLite has both hardware and software in its TCB, positioning it somewhere in the middle of the PMA implementation spectrum. TyTAN is an extension of TrustLite and has a significantly larger software TCB than TrustLite, situating it more towards the software end of the PMA implementation spectrum. First an attacker model that is common for these PMAs will be described. Afterwards, several security properties that can be offered by these PMAs are presented. Finally, each of the three architectures are discussed in terms of design choices, architecture and properties offered.

2.2 Sancus

Sancus is a Protected Module Architecture tailored for low-end networked embedded systems that provides software isolation, local attestation, remote attestation and confidential deployment with only a hardware TCB [21]. Also, a special C compiler has been developed that compiles C modules to Sancus SMs.[21]

Sancus is based around a setting in which a single infrastructure provider, IP, owns and administers a set of microprocessor-based systems that referred to as nodes N_i. These nodes are utilized by mutually distrusting third-party software providers SP_j, which develop and deploy software modules $SM_{j,k}$ on the nodes [21].

Each SM has an layout which consists of the start and end of the addresses of the text and data sections of that SM. Each SM also has an identity, which consists of the layout and a hash based on the content of the text section. Adding layout to the identity allows for multiple instances of the same SM on the same platform, as the layout and thus the identity would be different for each instance [21]. Also, the layout is used to easily verify that an SM that is being loaded does not overlap with other SMs.

Next, the main guarantees offered are discussed in terms of how they are implemented.

2.2.1 Software Module Isolation

The SM isolation is enforced by a hardware Program Counter Based Memory Access Control or PCBMAC. More specifically, the PCBMAC ensures that the protected data section is only accessible when the program counter is in the corresponding code section of the same module [21]. Also, each SM can only be accessed through a single entry point, making it impossible for attackers to misuse specific code chunks.

From/to	Entry	Text	Protected	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/ Other SM	r-x	r--	---	rwx

FIGURE 2.1: "Memory access control rules enforced by Sancus using the traditional Unix notation. Each entry indicates how code executing in the "from" section may access the "to" section." [21]

These access control rules can be seen in Figure 2.1. After an SM is loaded by an untrusted loader, its protection can be activated using a hardware instruction, called *protect*. This instruction verifies that the layout of the SM being protected does not overlap with other SMs already deployed and then saves the layout into a protected storage that is not accessible from software. Afterwards, it enables the access rules as presented earlier and a key based on the Node, SP and SM is created and stored in protected storage. This key is used for attestations and authentications and can be generated by the Software provider as well. If the code has been tampered with, the key generated would not match and the attestations/authentications would fail. The instruction also assigns a sequential ID to the protected SM. This ID is guaranteed to be unique within a boot cycle and it's not reused even after the module is unloaded. Note that the ID is different from the identity of the SM. Once the protect instruction has finished, the code of the protected SM cannot be modified anymore [21]. The memory layout of a node with a software module loaded can be seen in Figure 2.2.

To disable protection, the *unprotect* instruction is used. This instruction can be called only by an SM on itself. Note that the instruction clears the code and data section in order to prevent leaking confidential data [21].

Furthermore, for ensuring strong isolation, each module has its own call-stack to prevent leaking of stack allocated variables. This is done with the help of the compiler, for each SM a stack is created in the protected memory. Whenever a call outside the SM is made, the stack pointer is saved in the protected memory and after the return from the call the stack pointer is restored [21].

Sancus can also provide confidential loading. This is done by using the *protect* instruction with the following signature:

protect layout, SP, MAC

This variant of *protect* behaves similarly to the original *protect*. The only difference is that this protect, decrypts the text section of the module being loaded using a key derivated from the Node and SP. If the Message Authentication Code (MAC) obtained after decrypting is the same as the MAC given as parameter, the *protect* instruction continues with the computation of the SM key. Otherwise the instruction fails, clearing the text section and disabling the protection [21].

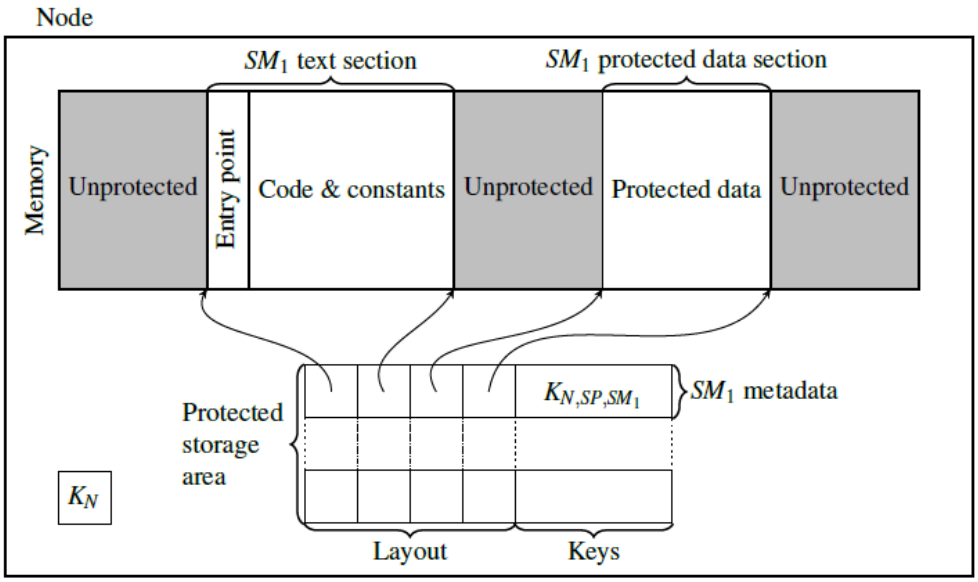

Figure 2.2: "A node with a software module loaded. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions." [21]

2.2.2 Remote Attestation and Secure Communication

As stated previously Sancus can provide remote attestation and secure communication. This is done using the following instructions:

encrypt plaintext, associated data, ciphertext (output), tag (output) [,key]

decrypt ciphertext, associated data, tag, plaintext (output),[,key]

This *encrypt* instruction takes as input plaintext and associated data, and based on the input and a key, computes a ciphertext and a tag. The tag is a Message Authentication Code (MAC) corresponding to both the plaintext and the ciphertext [21]. The parameter key used for the *encrypt* and *decrypt* instructions is optional. If no key is given, the SM key generated after enabling protection is used. Using these two instructions an SM can securely communicate with Software Provider (SP). Whenever an SM would want to send a message to the SP, it would first encrypt the data using the *encrypt* instruction. Then, the SM would send the ciphertext and the corresponding tag to SP. Using a ciphertext instead of plaintext provides confidentiality between the SM and SP. The SP, would then decrypt the ciphertext and verify the authenticity and integrity of the ciphertext using the associated tag. To satisfy remote attestation, a freshness guarantee must be added which can be in the form of a nonce [21].

2.2.3 Local attestation

Sancus provides both callee and caller authentication. The solution for callee authentication is explained first. In order for an SM1 to verify that an SM2 has not been tampered with, the following instruction is used:

attest address, expected hash

This instruction checks that a module is loaded and protected at the provided address, computes the hash based on the text section and the layout, compares it with the *expected hash* parameter and if the hashes match it returns the ID of SM2, otherwise it returns 0. In order to use this instruction for local attestation, SM1 must be deployed with the expected hash of SM2. Because computing the hash is an expensive operation, after identifying SM2, the instruction *get-id address* can be used for further attestations, which returns the ID of the module located at the address given as parameter. Indeed, this is sufficient because as long as SM2 protected, it cannot be tampered with. It is important to note that *get-id* is significantly cheaper than *attest* and does not use any cryptographic primitives [21]. It is important to note that the ID of an SM is different from its identity.

Caller authentication can be obtained in different ways, depending on the application requirements with the help of the *attest-caller* and *get-caller-id* instructions. Both instructions have similar behaviors as *attest* and *get-id*, but using the caller SM as parameter implicitly. One solution would be by first using *attest-caller* to identify them. This is needed only once. Afterwards, the *get-caller-id* instruction is used which returns the ID of the previous running SM. This ID is compared to the IDs obtained from calling *attest-caller*, thus identifying the caller. Again, *get-caller-id* does not use the cryptographic module and is much faster than *attest-caller*. Another less expensive solution, similar to the client-server model would be for the callee to directly use *get-caller-id* to get the id of the caller, without checking its integrity. This can be useful in scenarios in which the callee is not interested in integrity of the caller, but the value of the data returned by the callee is affected by the previous calls of the caller to the callee.[21]

2.3 Related work

2.3.1 TrustLite

TrustLite is security architecture that enforces isolation of software modules and provides local attestation with a small hardware and software TCB [16]. Because TrustLite has a smaller hardware TCB it also has smaller hardware extension cost compared to Sancus.

In TrustLite, the code, associated data and meta-data is known as a *program*. A task describes the runtime state of a program, including its CPU state, call stack and other volatile data. Tasks that are designed to implement a particular security mechanism are called *trusted tasks* or *trustlets*. [16]

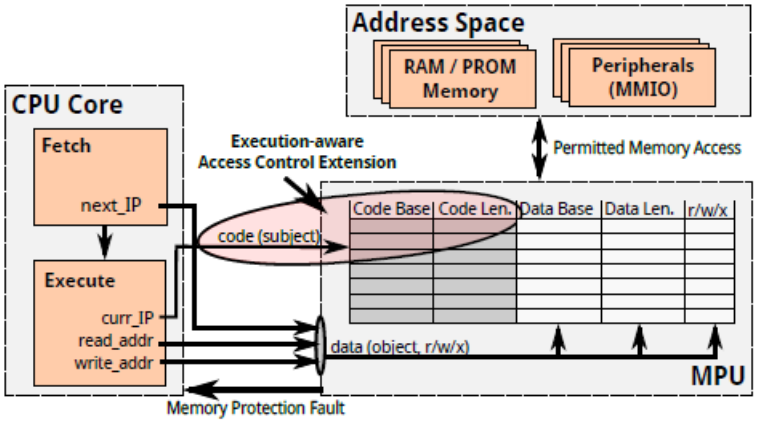

Figure 2.3: "Block diagram of EA-MPU" [16]

Software module isolation Isolation is ensured using an Execution-Aware Memory Protection Unit (EA-MPU). The EA-MPU organizes the physical memory into regions, each with associated access control rules. These rules are kept in local registers only accessible to the MPU. The regions can be of two types: code regions or data regions. EA-MPU offers the means to link code regions to data regions. This means that both the instruction address of the executing instruction and the address of the data are taken validated. This is similar to the PCBMAC, in both cases data is accessible based on the address of the instruction that is being executed. The EA-MPU scheme can be seen in Figure 2.3 and an example of the EA-MPU control rules can be seen in Figure 2.4. The access rules are more fine-grained in TrustLite than in Sancus. More specifically, in TrustLite, the EA-MPU can allow for several code regions to have access to a protected data region, while in Sancus, without modifications, a protected data region can only be accessed by the corresponding protected code region (giving exclusive control to it) [16]. In [32] a solution for Sancus that grants shared access to data is presented.

The initialization of the EA-MPU and the loading of programs, including trustlets, is done at startup by a Secure Loader. The Secure Loader first clears out the MPU registers, then parses the meta-data of the trustlets present in PROM and allocates memory regions for each of them and initializes each trustlet. Afterwards, the Secure Loader configures the MPU according to the access control rules required by the loaded trustlets. Finally, the Secure Loader can optionally load an untrusted OS [16]. A schematic of the steps that the Secure Loader makes at startup can be seen in Figure 2.5.

The EA-MPU registers cannot be modified after completing the configuration step. This implies, trustlets cannot be loaded or unloaded dynamically at runtime, which makes TrustLite less flexible than Sancus which can load/unload SMs at runtime.

TrustLite, also provides an Exception Engine that preserves memory isolation even when software or hardware exceptions are used. Whenever an exception is

		Object \ Subject		MPU Access Control Rules		
				TL-A (0x00-0A)	TL-B (0x0A-0B)	OS (0x0B-0F)
PROM / Flash	0x00..	Trustlet A	entry	rx	rx	rx
			code	rx	r	r
	0x0A..	Trustlet B	entry	rx	rx	rx
			code	r	rx	r
	0x0B..	OS	entry	rx	rx	rx
			code	r	r	rx
SRAM / DRAM	0x10..	Trustlet A	data	rw	-	-
			stack	rw	-	-
	0x1A..	Trustlet B	data	-	rw	-
			stack	-	rw	-
	0x1B..	OS	data	-	-	rw
			stack	-	-	rw
Peripherals	0x20..	MPU	flags	r	r	r
			regions	r	r	r
	0x2A..	Timer	period	r	r	rw
			handler(ISR)	r	r	rw
	0x2B..	...				

FIGURE 2.4: "Example of MPU access control rules in a scenario with 2 trustles and an OS" [16]

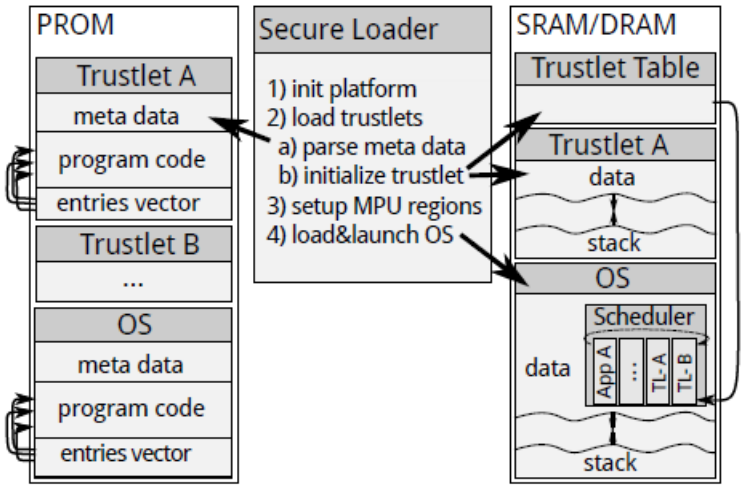

FIGURE 2.5: "Boostrapping Trustlets and OS from PROM" [16]

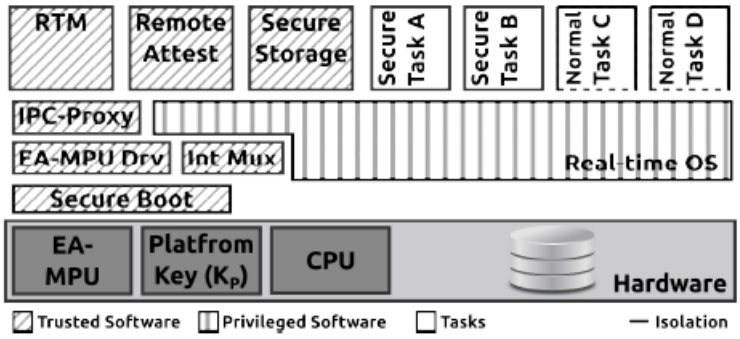

Figure 2.6: "TyTAN architecture" [5]

triggered, the stack and instruction pointer and the CPU registers are saved and then the CPU registers are cleared before jumping to the exception handler. This provides protection against leaking data to the exception handler. When returning from an exception handler, the first thing done should be to restore the stack pointer, otherwise if another exception is triggered, the CPU may be stored to the wrong stack. Finally, similar to Sancus to protect against code reuse, each trustlet can only be accessed through entry vectors [16].

Local attestation TrustLite provides callee authentication. A trustlet T1 can check that a trustlet T2 has been correctly loaded by verifying that the MPU registers are configured correctly and can compute a hash of the code to verify if T2 has been tampered with before loading. The Secure Loader can also be modified to do these measurements for all the trustlets and provide this data to trustlets [16].

TrustLite can provide caller authentication through the use of a handshake protocol. The initiator, trustlet that initiates communication, first does a check that the MPU of the desired callee is configured correctly. After that, it may also compute a measurement of the intended callee to verify the integrity. After finishing the measurement, the initiator sends a *syn* message to the intended callee, containing the identifier of the initiator, the intended callee and a nonce. The callee, then does the same verifications on the initiator and sends back an acknowledge message containing the parameters from the *syn* message as well as another nonce. The identities of the initiator and the callee and the two nonces can be used as basis for a cryptographic session token which can be used to authenticate messages by both parties [16].

2.3.2 TyTAN

TyTAN is an extension of TrustLite that provides strong isolation, local and remote attestation, dynamic software loading at runtime and real-time guarantees [5]. The hardware TCB of TyTAN is smaller than that of TrustLite, while the software TCB of TyTAN is significantly larger than that of TrustLite. The architecture of TyTAN can be seen in Figure 2.6

TyTAN is based around a similar setting to Sancus, in which there is a device manufacturer M, a device owner O, and multiple task providers Pi that provide programs called tasks. The tasks deployed on the device are mutually distrusted.

Isolation Isolation is enforced again using the EA-MPU. The EA-MPU is initialized by a Secure boot which loads all the other tasks that are part of the TCB, such as the EA-MPU driver. The EA-MPU driver controls the EA-MPU configuration at runtime, allowing for dynamic loading and unloading of tasks at runtime [5]. This design choice makes TyTAN more flexible than TrustLite, which does not provide dynamic loading of programs.

Remote attestation The Root of Trust for Measurement (RTM) task is responsible for local and remote attestation. RTM computes a hash based on the code of for each task loaded and the hash is then used as the identity for the task and can only be modified by RTM [5].

Remote attestation is obtained using Message Authentication Codes (MACs), in a similar fashion to Sancus. The MAC is computed by the Remote attest task, using an attestation key derived from a platform key and the identity of the task. The obtained MAC along with the identity of the task can be sent to a remote party to attest the task. The platform key is accessible only to several tasks that are part of the TCB and the attestation key is accessible only to the Remote Attest task [5].

Because RTM and Remote attest are tasks, they can be interrupted, including during the computation of the hash making TyTAN compatible with real-time applications. During the computation of the id, the corresponding task cannot be unloaded [5].

Local attestation As mentioned previously, the RTM task is also responsible for local attestation. The identity can be used to attest the integrity of the task to other tasks.

TyTAN provides also caller and callee authentication. Communication between tasks is done via the Inter-Process Communication (IPC) Proxy task. Whenever a task T1 wants to call a task T2, it calls IPC Proxy and sends the message and the identity of T2 using the CPU registers. The IPC searches for the location T2, which can be seen as callee authentication and writes the message and the identity of T1 into T2's memory. EA-MPU ensures that only the IPC Proxy can write in T2's memory. Because of this the presence of the id of T1 in T2's memory can be seen as caller authentication [5].

2.3.3 SMART

SMART or **S**ecure and **M**inimal **A**rchitecture for (Establishing a Dynamic) **R**oot of **T**rust is the first embedded implementation of a Protected Module Architecture [8]. It is based on the Self Protecting Module (SPM) architecture described in [31]. SMART provides remote attestation, remote authentication and trusted code execution using a small TCB and minimal hardware changes [8]. It is developed on

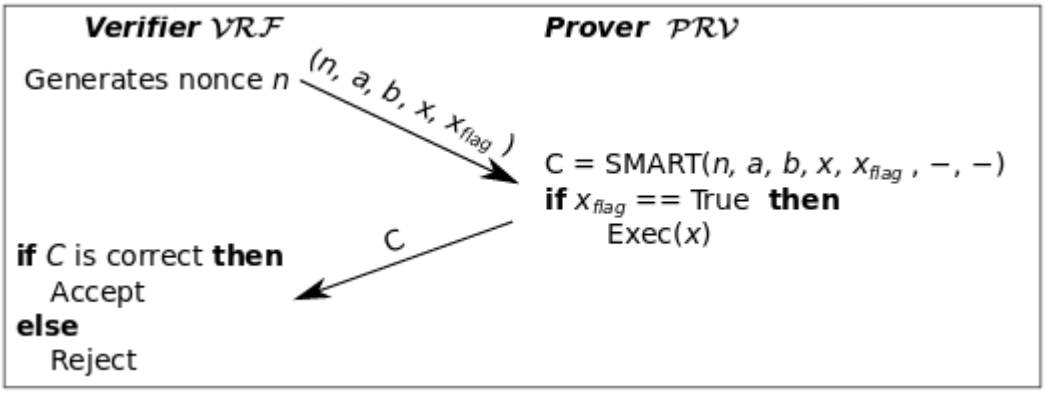

Figure 2.7: "SMART challenge-based protocol" [8]

two low-end embedded microcontrollers, Texas Instruments MSP 430 and Atmel AVR.

Compared to newer PMAs, such as Sancus, TrustLite and TyTAN implementation SMART provides less security guarantees. More specifically, as it will be seen later in this subsection, SMART provides only partial isolation and does not provide local attestation.

The design of SMART is based on four main components [8]. First, the attestation code is deployed on read-only memory (ROM). Second, the key K used for attestation is stored in a memory region inside the CPU and can only be accessed from the attestation code present in ROM. Third, the microcontroller ensures that the key K can only be accessed from the attestation code [8].

SMART provides guaranteed execution of code on the SMART platform, which is called *Prover (PRV)*, to an external party called *Verifier (VRF)*, by using a challenge-based protocol. The protocol is initiated by the *Verifier*, which sends a message to the *Prover*. This message contains a nonce n, the address range that needs to be attested, which is described by the addresses a and b, an address x which specifies where the *Prover* could jump to after completing the attestation and a flag x_{flag} , that specifies whether to jump to the address described by x after attestation. After receiving the message, the *Prover*, first disables interrupts and then executes the *SMART* procedure, which computes a MAC on the memory region describe by addresses a and b using a pre-shared key between the *Prover* and the *Verifier*. It is important to note that the SMART procedure is stored in ROM memory and that the pre-shared key can only be accessed from within the SMART procedure. After computing the MAC, the interrupts are enabled and if the x_{flag} is true, the *Prover* executes the code starting from address x. Finally, after finishing the SMART procedure and the potential execution, the *Prover* sends the cryptographic checksum (the MAC) to the *Verifier*, which can check whether or not it is correct [8]. The protocol is described in Figure 2.7.

The interrupts are disabled for the duration of the checksum computation in order to guarantee atomic execution of the attestation code. In order to protect

against code reuse attacks, the program counter can only jump at the SMART initial address and can only leave at the last SMART address. Any attempt to do otherwise would force an immediate hardware reset. The hardware access control logic that enforces the correct execution of SMART is also responsible for enforcing the access to the pre-shared key from inside the SMART procedure. Furthermore, in order to ensure that no data is leaked due to an incomplete SMART execution or even a power loss, memory cleanup is performed by the processor after each reset [8]. Also, the SMART procedure is not dependent on the pre-shared key, thus not leaking any information about it. Finally mentioned earlier, the SMART procedure is stored in ROM making it impossible for attackers to tamper with its content [8].

2.3.4 XEBRA

XEBRA is another PMA implementation based on SMART and the XEN hypervisor. XEBRA is a software solution that provides the same services as SMART, namely: remote attestation, remote authentication and trusted code execution [1]. Next, the XEN hypervisor is described briefly in order to understand XEBRA better and then XEBRA is explained in detail.

XEN hypervisor XEN is a light, bare-metal hypervisor that allows for multiple virtual machines (domains) to run on the same physical machine. XEN is small, compared to other hypervisors, containing less than 150,000 lines of code. XEN runs in a more privileged CPU state than any other software on the machine and is responsible for the CPU scheduling, memory management, interrupts and for launching the first virtual machine, which is called the Control Domain or Domain 0. However, XEN has no knowledge of any I/O such as storage and networking. The I/O is managed by the Control Domain, which has the capability to access the I/O functions. The Control Domain can launch and interact with other Virtual Machines and provides a control interface. Without the Control Domain, the hypervisor cannot be used [9]. The XEN architecture can be seen in Figure 2.8.

XEBRA implementation XEBRA uses the Xen hypervisor to enforce a similar protocol as SMART in order to provide remote attestation, remote authentication and trusted code execution. More specifically, it requires at least 2 Virtual Machines (domains): the Control Domain and an Application Domain. The Control Domain is used to store the attestation code. On top of this, the Xen hypervisor is used to enforce isolation between the Control Domain and the Application Domain enforcing, by extension, isolation between the attestation code which is stored in the Control Domain and the Application Domain. The isolation between the attestation code and applications is a key component of both SMART and XEBRA, even though the way it is enforced is very different. Xen also acts as a secure loader for the Control Domain, ensuring that it is loaded correctly. [1]

In order to minimize the attack surface, the Control Domain is firewalled and is can only communicate with the remote Verifier and the Application Domain. Because the Control Domain is the only domain that has direct access to the network, it shares

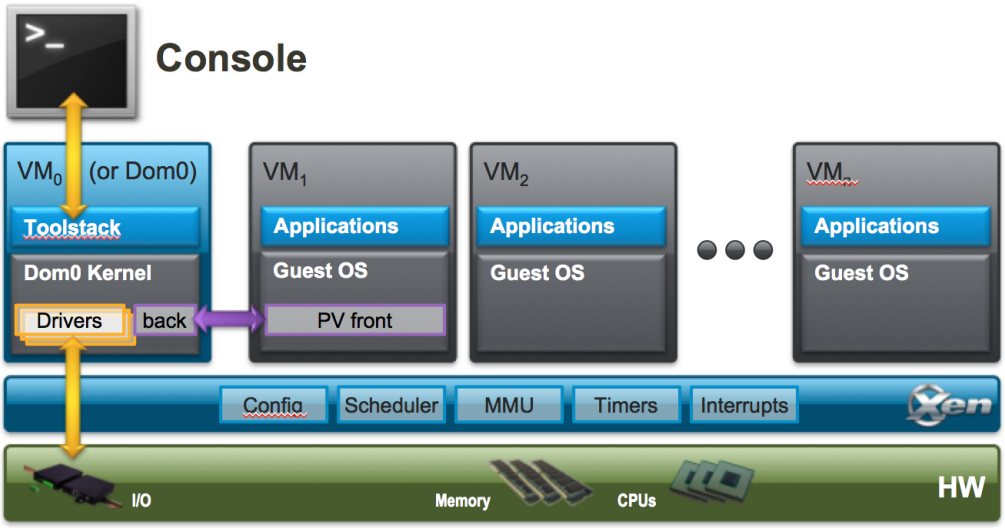

Figure 2.8: "XEN hypervisor Architecture" [9]

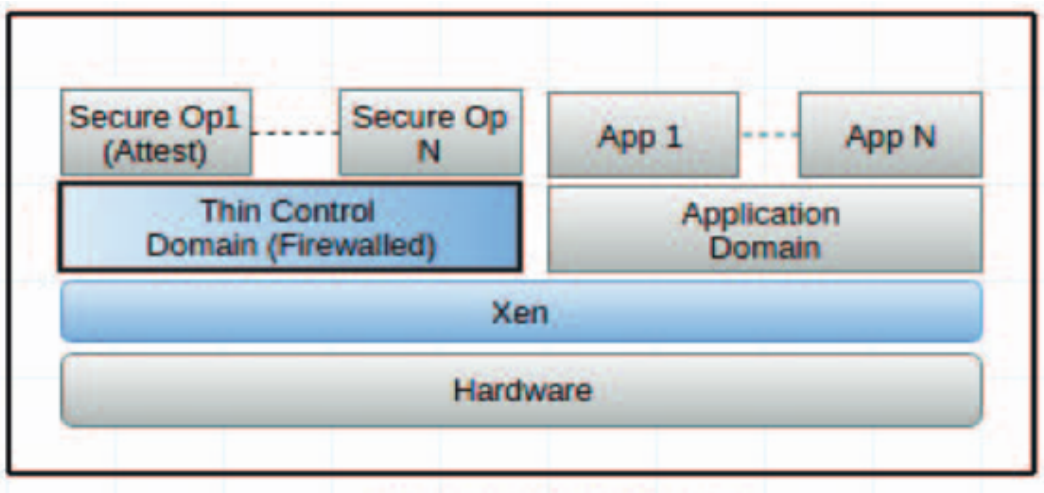

Figure 2.9: "XEBRA Device Structure" [1]

network connectivity with the Application Domain through a bridged connection, allowing the Application Domain to communicate with the outside world. [1]

XEBRA device structure can be seen in Figure 2.9. As it can be seen, XEBRA could be extended with other secure operations besides the attestation code that are isolated from the Application Domain. The communication model is similar to SMART and can be observed in Figure 2.10. The communication model works as follows: Whenever the remote Verifier wants to attest a memory region it sends a message to the Application Domain. The message contains x and$y$, which represent

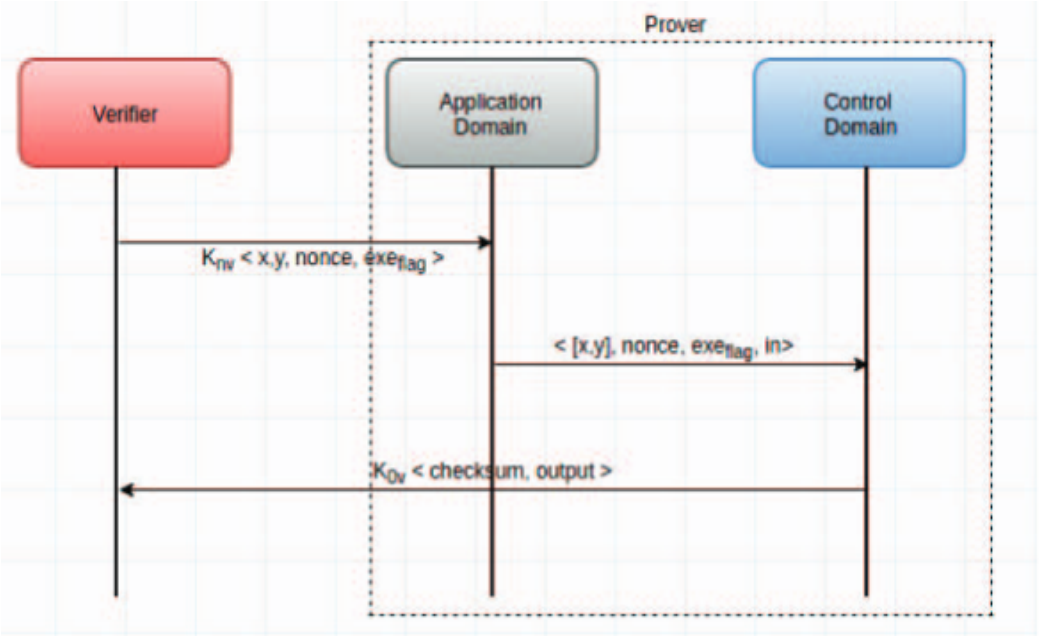

FIGURE 2.10: "XEBRA Communication Model" [1]

the start and end address of the memory region which is requested to be attested, a *nonce* which is used to protect against replay attacks, and finally, the exe_{flag} , which if set to true executes the code starting from address x. The message is encrypted with a pre-shared symmetric key, shared between the Verifier and the Application Domain. Encrypting the message with this key ensures the integrity and origin of the message. After receiving the message, the Application Domain sends another message message to the Control Domain. This message forwards all the parameters received from the Verifier and also sends the actual contents of the memory region between addresses x and y. Next, the Control Domain computes a checksum based on the contents of the desired memory region. After computing the checksum, the Control Domain executes the code if the exe_{flag} is set and finishing the potential execution, it sends the checksum and the output resulted from the code execution to the Verifier through a message. This final message is encrypted with another key, that is pre-shared between the Control Domain and the Verifier, in order to provide message integrity and proof of origin [1].

Even though XEBRA requires more advanced features such as CPU privilege levels, the implementation is light enough to run on higher-end embedded systems such as Intel Galileo. Another advantage of XEBRA is that being a complete software solution, it can be deployed on legacy systems. However, the TCB is very large, Xen having less than 150,000 LOC [9]. On top of that, the Control Domain implemented by a Linux distribution such as Alpine Linux has a TCB of over 20,000,000 LOC [15]. Because XEBRA has such a large TCB it is very difficult to thoroughly check it for bugs and vulnerabilities.

	XEBRA	TyTAN	TrustLite	SMART	Sancus
Local Attestation	No	Yes	Yes	No	Yes
Remote Attestation	Yes	Yes	No	Yes	Yes
Modifiable Security Primitives	Yes	Yes	Yes	No	No
Software Module Isolation	Partial	Yes	Yes	Partial	Yes
Interruptible applications	No	Yes	Yes	No	No
Dynamic Software Loading	Yes	Yes	No	Yes	Yes

Figure 2.11: Comparison based on the features offered by different Protected Module Architectures

2.4 Discussion

Next, the trade-offs of the PMAs presented are discussed. The PMAs reviewed in this Chapter are examples of the large spectrum of possible implementations, ranging from complete hardware implementations such as Sancus [21], to implementations that rely on both hardware and software components such as SMART [8], TrustLite [16] and TyTAN [5] and finally to complete software implementations such as XEBRA [1].

A comparison between the different implementations presented in this chapter can be observed in Figure 2.11. Next, each of the features compared in Figure 2.11 are discussed.

2.4.1 Local attestation

Many embedded systems are deployed with multiple mutually distrusting applications. Because of this, local attestation solutions are needed in order for an application to detect whether or not other applications deployed on the same device have been tampered with. TrustLite provides both callee and caller authentication. Callee authentication means that an application can attest another application. This is done by verifying that Memory Protection Unit registers that correspond to the application being attested are correctly configured and then computing a hash based on the content of the application that is being attested. TrustLite can also provide caller authentication, meaning that an when an application is called it can identify its caller [16]. This is done through the use of a handshake protocol explained in Section 2.3.1. TyTAN provides both caller and callee authentication through the use of the RTM task combined with the IPC proxy task [5].

Sancus provides both callee and caller authentication. callee authentication is provided through the use of the *attest* primitive, which computes a hash based on the layout and contents of an SM and compares the hash with an expected one. If the

attestation is successful, the id of the attested SM is returned. For caller attestation, the *attest-caller* primitive is used which has a similar behavior with *attest*. For subsequent attestations it is sufficient to check that the id of the SM that is being attested has not changed. This is done with the use of the *get-id* and *get-caller-id* primitives [21].

2.4.2 Remote attestation

With the exception of TrustLite which does not guarantee remote attestation all other 4 PMAs presented guarantee remote attestation. All 4 implementations use measurements to provide proof that the integrity of the software to remote parties. Sancus provides remote attestation through the use of *protect* primitive which guarantees that the content of the SM cannot be modified combined with *encrypt* primitive which computes a MAC on the data present in a memory region [21]. TyTAN uses a similar approach as Sancus and uses a secure task called Root of Trust for Measurement (RTM) which computes the measurement for tasks which are protected by the EA-MPU [5]. The main difference between Sancus and TyTAN is the way software protection and measurement computation is implemented. Sancus implements both primitives in hardware while TyTAN provides software isolation through a combination of hardware and software. Also, RTM is software component, while *attest* is implemented in hardware.

SMART and XEBRA rely on a challenge-based protocol which can be seen in Figure 2.7 for SMART and Figure 2.10 for XEBRA. These 2 PMAs do not provide software isolation for the application, that is why the additional steps are required. For SMART, the interrupts are disabled during the measurement, to guarantee that the content cannot be tampered with during the measurement [8]. For XEBRA, the actual code that is to be measured is sent to the attestation code on a separate virtual machine, to guarantee that the code is not modified during the measurement [1].

2.4.3 Modifiable security primitives

Another interesting feature is the ability of being able to modify the security primitives available without any hardware changes. This feature allows for easily updating primitives if bugs or vulnerabilities are found. Also, it allows for the primitives to be tailored on the application requirements. For example, the measurement implementation used for attestations can be selected based on the requirements of an application. TrustLite, TyTAN and XEBRA are more flexible from this point of view and support this, having most of the security primitives implemented in software. On the other side SMART has the attestation code written in ROM. Depending on the technology used for ROM the attestation code can or cannot be modified. Sancus, being a PMA which has only hardware TCB does not support modifiable security primitives, making it less flexible than other implementations from this point of view.

2.4.4 Software Module Isolation

Complete software module isolation is guaranteed for Sancus, TrustLite and TyTAN. In Sancus it is enforced in hardware, through the use of the protect primitive, which explained in detail in Section 2.2. TrustLite and TyTAN both use an hardware memory protection unit which contains memory regions and grants read/write/execution rights based for each SM based on a policy that is configured by a secure initializer at startup. In TrustLite these rights are static, meaning that after they are set they can only be changed at reset by the secure initializer. In TyTAN this is not the case, these rights can be changed at runtime through the use of a memory protection unit driver.

SMART and XEBRA, provide only partial software module isolation. Both in SMART and XEBRA, the attestation code is isolated from the rest of the applications present on the device. However, the applications themselves are not isolated from each other. For SMART, the isolation between the attestation code and the applications is enforced in hardware, as described in Section 2.3.3. Even though XEBRA draws inspiration from SMART, the way it enforces isolation is completely different. XEBRA uses the Xen hypervisor to provide isolation between Control Domain, where the attestation code is deployed and the Application Domain, where the rest of the applications reside.

2.4.5 Interruptible applications

Interruptibility is essential for compatibility with real-time applications. Only TrustLite and TyTAN support interruptible applications [16, 5]. SMART, XEBRA and Sancus, on the other side, do not support interruptible applications. SMART requires interrupts to be disabled while computing the measurement or when executing the measured code [8]. For XEBRA, it is assumed that the applications are self-contained and can run without interrupts [1]. For Sancus, the main limitation in this regard is that the cryptographic operations cannot be interrupted [21].

2.4.6 Dynamic Software Loading

Dynamic software loading refers to the capability of loading and unloading software at runtime. With the exception of TrustLite, all other 4 PMAs support this. Sancus, SMART and XEBRA can have dynamic software loading with the help of an untrusted loader. TrustLite is less flexible from this point of view and does not allow dynamic software loading. That is because memory access right are written in the EA-MPU at startup and these rights cannot be changed after boot. TyTAN allows dynamic software loading through the use of an Memory Protection Unit driver which can change EA-MPU configuration at runtime.

2.5 Conclusion

The need for secure solutions for embedded systems has lead to the concept of Protected Module Architecture. This topic is still extensively researched, that is why there is a wide variety of implementations using different approaches to provide similar security guarantees. In this chapter, 5 PMA implementations were discussed. The PMAs presented highlight the solution diversity, ranging from solutions that rely on software to solutions that rely strictly on hardware to provide the security guarantees. Out of these 5 implementations, Sancus was discussed in greater detail because, this is the PMA which on which the security kernel is implemented. Based on the comparison with other PMAs done in Section 2.4 Sancus is a very capable PMA. However, Sancus still has drawbacks, some of which are tackled by the security kernel which will be introduced in the next chapters.

Chapter 3

Problem statement and Solution Oultine

3.1 Motivation

Even though Sancus provides strong security guarantees [21] discussed in detail in Chapter 2 it also has some potential disadvantages. Being a full hardware TCB implies that the platform has increased hardware costs, compared with the a standard MSP430 platform. The increase of hardware cost may be undesirable, especially for low end embedded applications, where production costs need to be low. A significant part of this additional hardware is due to the cryptographic module required for the local and remote attestation security guarantees [21]. A possible solution for reducing the hardware costs then is to provide a software solution for local and remote attestation instead of a hardware one.

Another possible downside of the current implementation of Sancus is that the cryptographic operations such as *attest*, *attest-caller*, *encrypt* and *decrypt* are not interruptible. This may make Sancus incompatible with real-time applications in which specific hard deadlines must be met. More specifically, if the cryptographic operation requires a larger amount of time than the a deadline, the application has no means to guarantee the satisfaction of that deadline. A possible solution for this is to change the hardware implementation into one that can be securely interrupted and still provide the same security guarantee. This approach would further increase hardware costs, which as mentioned earlier is undesirable for low-end embedded devices. Another solution is to have software implementations of the cryptographic operations which could be interrupted and still maintain the same security guarantees instead of the hardware implementation.

Yet another potential disadvantage of Sancus is that the developers have to be careful to write the Software such that it is not vulnerable to call-stack short-cutting attacks. The call-stack shortcutting attacks refer to attacks in which the integrity of the call-stack is not maintained and inter-SM calls can happen in unpredictable ways. These attacks are described in detail in Section 4.1. A possible solution for this would be a software mechanism that protects against such attacks and which

the developers can use with ease.

3.2 Problem statement

This chapter introduces a security kernel for Sancus which offers an Inter-SM Communication (ISMC) mechanism which protects against call-stack shortcutting attacks. The Secure Kernel also provides a software solution for secure linking between SMs. Both services build on the *protect* primitive of Sancus without using any hardware cryptographic primitives.

The rest of the Chapter is organized as follows. In Sect. 3.3 the attacker model and the assumptions are explained. Next, the desired security properties that this security kernel should offer are presented in Sect. 3.4. Finally, in Sect. 3.5 an overview of the security kernel architecture is discussed.

3.3 Attacker model and assumptions

The attacker model is similar to the one described for the original Sancus platform [21] with a few differences. The security kernel is loaded correctly and protection is enabled for it using a secure boot or similar mechanism. The attacker cannot tamper with the security kernel during the secure boot process. The attacker can manipulate all the software (including Software modules and the Operating System) on the node at any given time with the exception of the security kernel during loading and the potential software mechanism used to load the Kernel correctly (e.g. secure boot).

Similar to the Sancus [21], it is assumed that the Dolev-Yao attacker model [7] is used: attackers cannot break cryptographic primitives, but they can perform protocol-level attacks. Remote communication (e.g. with the Software Provider) is considered secure. Finally, hardware attacks are out of scope. Similar to Sancus, it is assumed that the attacker does not have physical access to the hardware [21].

3.4 Security properties

In this section, the security properties that the security kernel should provide are explained. The kernel should provide similar properties to the original Sancus platform.

- **Software module isolation.** Software modules are isolated from each other, meaning that the code of an SM cannot be modified by other SMs and its state cannot be read or modified by other SMs [21]. The hardware isolation mechanism is actually kept and used by the secure kernel to enforce the other security properties;
- **Local attestation.** An SM can detect whether or not a specific SM has been loaded correctly on the same device even if the loading process is not trusted [21]. The security kernel replaces the hardware local attestation mechanism with a software implementation;

- **Secure Linking.** An SM can detect that it is calling a specific SM. Also, an SM can detect that it has been called by a specific SM [21]. The security kernel uses the software implementation to provide secure linking;
- **Remote attestation.** An outside party (that is not deployed on the same device) such as the software provider can detect with high assurance that a specific SM is loaded correctly on the device even if the loading process is not trusted [21]. The security kernel provides remote attestation using a software implementation under specific assumptions;
- **Call-stack shortcutting protection.** Inter-SM calls happen in a predictable order maintaining *Call Flow Integrity*. More specifically, a called SM can only return to the caller SM. The security kernel provides a software mechanism to protect against these attacks.

3.5 Design overview

To tackle the disadvantages presented in the previous section, a security kernel for Sancus is introduced. The security kernel is composed of two major components. First, the Inter-Software Module Communication component which protects against call-stack short-cutting attacks. And Second, the Software Cryptographic Module, which replaces the Hardware implementation, with the purpose of reducing the hardware costs and being a step towards real-time application compliance. This solution would still provide local attestation and remote attestation given the assumptions made in Section 3.3.

3.5.1 Inter-Software Module Communication

The Inter-Software Module Communication (ISMC) provides protection against call stack shortcutting attacks. ISMC behaves as a proxy for inter-SM communication. Whenever an software module SM1 wants to call another software module SM2, it does so using the ISMC component of the kernel which saves the caller information in an internal stack and proceeds to call the intended SM2. After SM2 returns, the kernel pops the caller information saved previously and returns control to SM1.

In order to enforce protection against attacks that can change the call flow, all SMs should verify that they are only called by the kernel and they should only call other SMs through the kernel. This component is discussed in detail in Chapter 4.

3.5.2 Kernel Secure Linking and Attestation

The security kernel provides secure linking between Software Modules building only the protect primitive. After being loaded, the Software Modules register to the security kernel which saves the layout of the SM in a data structure called *registered_sm_data*. After saving the necessary data in the data structure, the kernel enables protection on the SM. Next, the kernel computes a hash of the SM which is used to determine whether or not that SM has been tampered with. If the

Kernel detects that the SM has been tampered with, the registration fails, the kernel removes the SM's data from the *registered_sm_data* data structure and disables protection for the SM.

After successfully registering to the Kernel an SM can call other registered SMs through the Kernel as explained in Subsection 3.5.1.

The component described in this Subsection can also be used without the ISMC component described in in Subsection 3.5.1. After successfully registering to the Kernel an SM needs to first obtain the id of the intended callee with the use of the *get-id* primitive and then check whether the SM with this id is registered in the Kernel. If it is, it can safely call the intended callee. For subsequent calls, the SM must verify that the id of the intended callee has not changed.

Both methods described, with and without ISMC, provide local attestation and secure linking for the registered SMs. Provided the assumptions described in Section 3.3 the mechanisms described in this Subsection should be sufficient for remote attestation as well. The component described in this Subsection is explained in more detail in Chapter 5.

3.6 Conclusion

This chapter presented a few potential disadvantages of Sancus, such as increased hardware costs, lack of interruptible cryptographic operations and the necessity of special care regarding protection against call-stack shortcutting attacks. The concept of a security kernel for Sancus is introduced in this Chapter. The security kernel is designed to tackle these disadvantages using two components. The ISMC component which tackles the call-stack shortcutting attack and a component that provides a software implementation of the cryptographic operations with the purpose of reducing hardware costs and being a step towards interruptible cryptographic operations. The assumptions made for this kernel and attacker model are also discussed in this chapter.

Chapter 4

Mitigating Call-stack shortcutting

4.1 Call-stack shortcutting attack scenarios

While Sancus is a very powerful architecture, care has to be taken to guard against potential call-stack shortcutting attacks. The security kernel offers a solution for Sancus to protect against *call-stack shortcutting attacks*. This means that the inter-SM calls happen in a predictable way. The called SM can only return to the caller SM. This minimizes the ability of malicious SMs to affect the behavior of other SMs. In the current version of Sancus there are scenarios in which the Call Flow Integrity of the SMs is not maintained. The remainder of this section is split into several subsections. First, a couple of attacks that change the control flow of Sancus are presented and afterwards the defense mechanisms that can solve protect against these potential attacks are discussed.

In this chapter, the Inter-Software Module Communication mechanism introduced in the previous section is discussed in detail.

A first real world scenario in which this can happen is a fire alarm system deployed on a Sancus platform. The fire alarm application contains 3 SMs: a Control Module SM which controls the sound alarm and contacts the fire department, a Smoke Sensor SM, which gathers data from the smoke sensors and a Sensor Data Logger SM that logs the data from the smoke sensors. The normal call flow would be as follows. At periodic amounts of time, the Control Module requests sensor data from the Smoke sensor. The Smoke sensor calls the Sensor Data Logger to log the data. The control returns to the Smoke Sensor which returns the sensor data to the Control Module. This flow can be seen in Figure 4.1. However, if the Sensor Data Logger has a bug or is malicious, it can return directly to Control Module with possibly incorrect data, starting false alarm or even worse not starting alarms when it should. The malicious behavior can be seen in Figure 4.2.

Another more complex scenario in which the inter SM call flow is affected is when SMs have to call each other several times. In this scenario, 3 SMs are deployed: A, B and C. The normal call flow is as follows: A calls B, which calls C. In order for

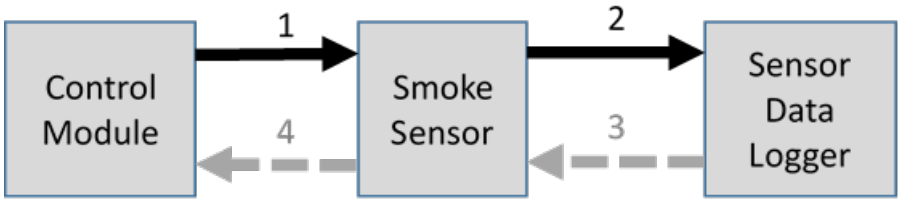

Figure 4.1: Normal Call Flow (Black arrows represent inter SM calls, gray arrows represent the returns from these calls and the number above these arrows represent the order in which the operations happen.)

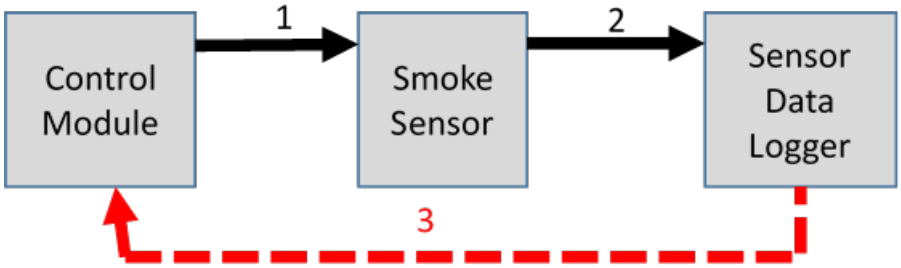

Figure 4.2: Fire alarm - Malicious Call Flow (The red arrow represents the malicious return call)

C to give a result to B it needs extra information from B and it calls back to B. B returns the extra information to C, C finishes its computation and returns back to B and, finally, B returns to A. Again, if B has a bug or is malicious, it can change the call flow of the application. After C calls back to B for more data, B can return to A without waiting for the response from C. In this situation C becomes unavailable, because it is waiting for a return from B for a potential infinite amount of time. The normal, and the malicious call flow can be seen in Figure 4.2.

The rest of the Chapter is organized as follows. First, Section 4.2 presents in detail the implementation of an ISMC prototype. After that, Section 4.3 discusses the security properties of the ISMC prototype, its trade-offs and evaluates it. Finally, Section 4.4 presents the conclusions.

4.2 ISMC implementation

For the first scenario, the most straightforward solution would be for each SM to check that the return call is indeed from the same SM that was called. This can be done by using the get_caller_id primitive that returns the ID of the previous SM, which in this case is the SM that used the return call. This can be enforced either at compiler level, or manually by the developers. The trade-off between the two approaches is that the compiler approach would give a default behavior for this kind of violations (e.g. system restart), while the manual approach would allow for

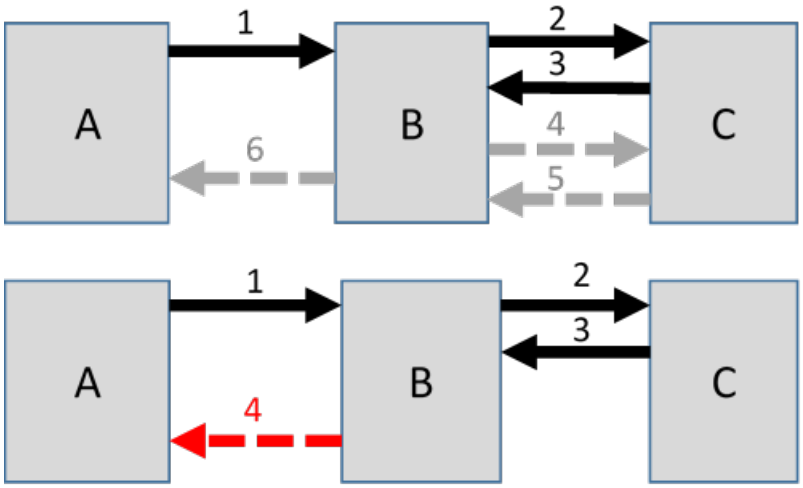

FIGURE 4.3: Call flow scenario - Normal call flow (Top modules) versus malicious call flow (Bottom modules)

specific behavior based on the implementation of each SM. However, the manual approach adds extra burden on the developers, by making sure that all calls are checked.

For the second scenario, the method described in the previous paragraph is not sufficient. Because of this, another solution that also covers the first scenario is proposed. The solution consists of a small security kernel, which would basically be an SM responsible for all inter-SM communication. This SM is called Inter Software Module Communication (ISMC) and contains an internal shadow call stack in order to ensure call flow integrity.

More precisely, when an SM1 wants to call another SM2 it does so by calling the ISMC and sending as a parameter the entry address of SM2. ISMC pushes the address of the caller, SM1, in the shadow call stack and calls the intended SM2. When SM2 finishes, it returns to ISMC. Then ISMC checks that it is the same SM that it has called and then proceeds to pop the address of SM1 from the shadow stack and returns to SM1. The solution for scenario 1 using ISMC can be seen in Figure 4.4.

To ensure that all communication is done via ISMC all SMs should check whenever they are called that they are called by ISMC and that when they get a return that it is from ISMC. This can be done by using the primitive *get_caller_id*. More concretely, whenever a benign SM1 is called by another SM2, it checks with *get_caller_id* that ISMC is the caller. If another the caller is another SM then it simply ignores the call and returns back to the caller. Also, after each return the former caller SM2 should first verify the identity of the SM from which it has returned and if the identity does not match the ISMC it should have a default behavior (e.g. system restart). In the current implementation, ISMC relies only on the *get_caller_id* and *get_id* instructions and does not use any cryptographic instructions.

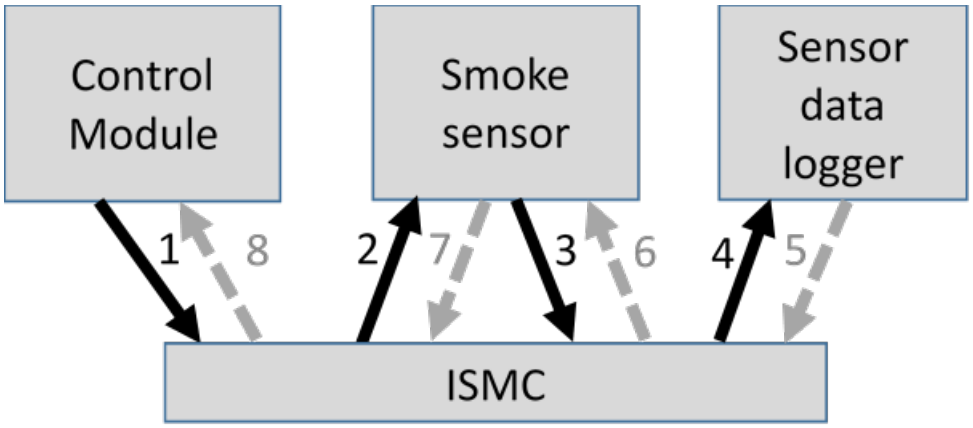

Figure 4.4: ISMC based solution for fire alarm scenario.

The ISMC can be deployed on Sancus in at least two ways. First, a secure boot similar to that of TyTAN [5] can be used to load ISMC in a secure way with a predefined ID before the other SMs. Because ISMC is loaded securely, the other SMs do not need to verify if ISMC has loaded correctly. An SM only needs to verify the intended callee and then call it via ISMC. A second approach is that ISMC is loaded by untrusted software similar to the rest of the SMs. In this case the SMs need to verify ISMC before calling it.

While this design protects against hijacking the call flow between SMs at least in the scenarios presented, it also adds computational overhead due to the extra calls and checks required by this design.

4.3 Evaluation

4.3.1 Macro benchmarks

Three benchmarks have been measured to study the impact on performance made by the addition of the ISMC. First benchmark is composed of two protected SMs, A and B, in which A makes a simple call to B. The Second benchmark consists of three protected SMs, A, B and C, in which A calls B and B then calls C. The last benchmark is again comprised of three protected SMs, which behave as presented in Scenario 2 from Subsection 4.1.

The tests were run on an Sancus platform with 41 KB ROM, 16 KB of RAM, 512 B allocated for each SM stack and a hardware cryptographic module with 64bit security.

The baseline for these scenarios is measured using a Sancus without a security kernel and compared against two implementations of Sancus with the security kernel (ISMC). The first implementation of the security kernel, the ISMC is assumed to be loaded by a secure boot before any other SM and has a predefined ID. In the second implementation, it is assumed to be loaded by untrusted software.

TABLE 4.1: CPU cycles benchmark measurements comparing the Baseline (Sancus without ISMC) with versions of Sancus with ISMC with and without the assumption of being loaded by trusted software.

Benchmark	Init. cycles	First run cycles	Additional run cycles	Total cycles
Baseline	32,553	11,873	244	44,670
Call between two SMs via ISMC with Secure Boot	83,424	15,047	949	99,420
Call between two SMs via ISMC without Secure Boot	86,400	92,665	973	180,038
Baseline	48,456	27,090	484	76,030
Cascade call with 3 SMs via ISMC with Secure Boot	115,509	47,838	2,160	165,507
Cascade call with 3 SMs via ISMC without Secure Boot	116,439	159,585	2,180	278,204
Baseline	54,594	50,441	1,048	106,083
Scenario 2 via ISMC with Secure Boot	132,249	98,370	4,746	235,365
Scenario 2 via ISMC without Secure Boot	132,807	245,758	4,786	383,351

The results of these benchmarks measured in CPU cycles and milliseconds can be seen in Table 4.1 and Table 4.2. The added overhead of the versions of Sancus with ISMC compared to the Baseline can be seen in Table 4.3. The measurements are split into 4 columns. The first column shows the number of cycles required for initialization, the computation of the keys for each SM and enabling the protection. The second column shows the number of cycles required for completing the benchmark for the first run, when the SMs use the expensive cryptographic instruction *attest*. The third column displays the number of cycles required for the benchmarks to complete when *attest* has already been used for attesting the callers and now the much less expensive instruction *get-id* is used. Because the number of cycles needed for the completion of a benchmark can be calculated deterministically, there is no need for more than two runs. Finally, the last column showcases the total number of cycles, the sum of all the other three columns.

As it can be seen, the addition of the ISMC adds a significant amount of cycles in all columns, compared to the version without ISMC. The initialization adds a large overhead due to the fact that ISMC is significantly larger than the other SMs which are just dummy SMs.

For the first run, the implementation in which ISMC is loaded securely takes longer than the baseline. This is because, as mentioned in Section 2.2, each SM has

Table 4.2: Benchmark measurements in milliseconds comparing the Baseline (Sancus without ISMC) with versions of Sancus with ISMC with and without the assumption of being loaded by trusted software.

Benchmark	Init. ms	First run ms	Additional run ms	Total ms
Baseline	1.62	0.59	0.01	2.23
Call between two SMs via ISMC with Secure Boot	4.17	0.75	0.04	4.97
Call between two SMs via ISMC without Secure Boot	4.32	4.63	0.04	9.00
Baseline	2.42	1.35	0.02	3.80
Cascade call with 3 SMs via ISMC with Secure Boot	5.77	2.39	0.10	8.27
Cascade call with 3 SMs via ISMC without Secure Boot	5.82	7.97	0.10	13.91
Baseline	2.72	2.52	0.05	5.30
Scenario 2 via ISMC with Secure Boot	6.61	4.91	0.23	11.76
Scenario 2 via ISMC without Secure Boot	6.64	12.28	0.23	19.16

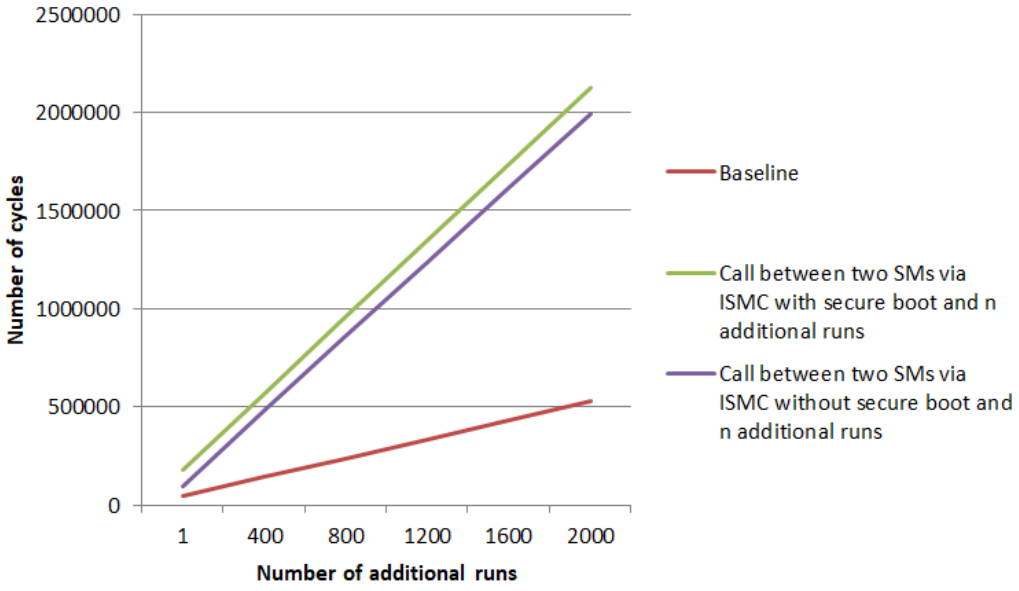

Figure 4.5: Dependence between cycles and the number of additional runs for the Call between two SMs benchmark

TABLE 4.3: Overhead (in CPU cycles and in percentage) added by the versions of Sancus with ISMC compared to the Baseline, based on the measurements presented in Table 4.1.

Benchmark	Init. overhead	First run overhead	Additional run overhead	Total overhead
Call between two SMs via ISMC with Secure Boot	50,871 (156%)	3,174 (26%)	705 (288%)	54,750 (122%)
Call between two SMs via ISMC without Secure Boot	53,847 (165%)	80,792 (680%)	729 (388%)	135,368 (222%)
Cascade call with 3 SMs via ISMC with Secure Boot	67,053 (138%)	20,748 (76%)	1,676 (346%)	89,477 (117%)
Cascade call with 3 SMs via ISMC without Secure Boot	67,983 (140%)	132,495 (489%)	1,696 (350%)	202,174 (265%)
Scenario 2 via ISMC with Secure Boot	77,655 (142%)	47,929 (95%)	3,698 (352%)	129,282 (121%)
Scenario 2 via ISMC without Secure Boot	78,213 (143%)	195,317 (387%)	3,738 (356%)	277,268 (261%)

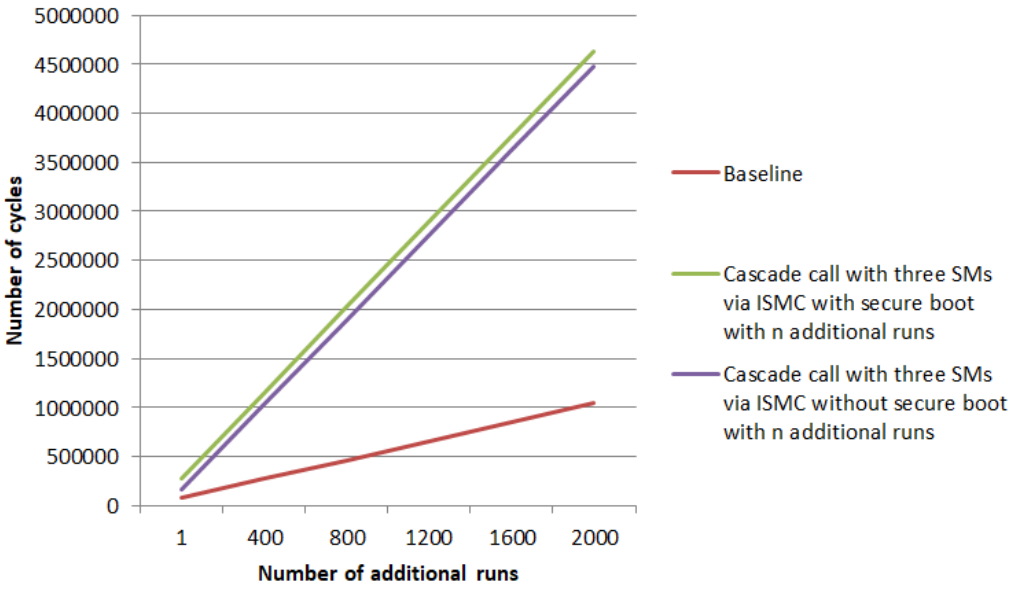

FIGURE 4.6: Dependence between cycles and the number of additional runs for the Cascade call with 3 SMs benchmark

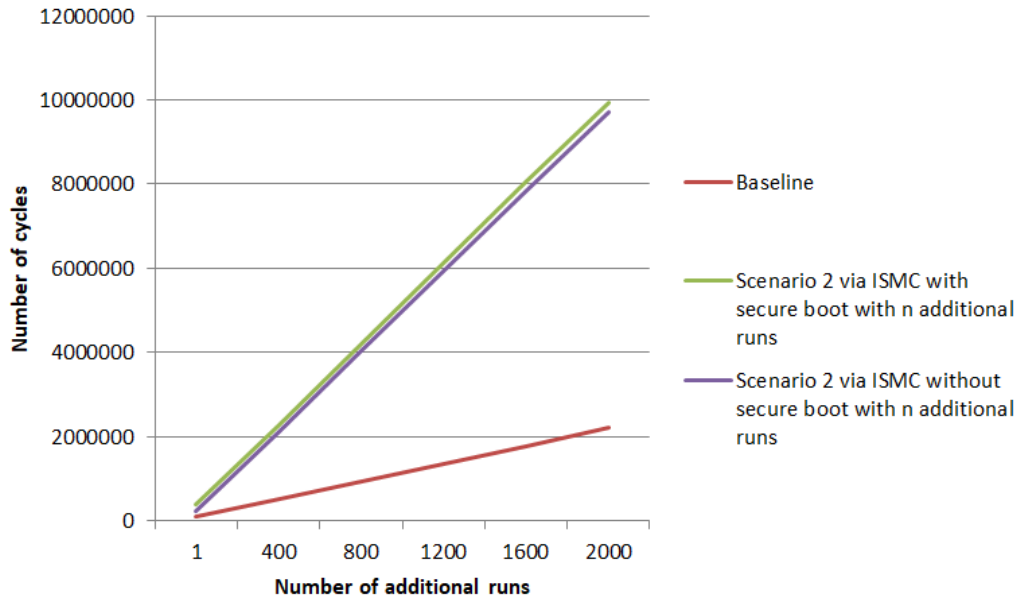

Figure 4.7: Dependence between cycles and the number of additional runs for the Scenario 2 benchmark

to check that is only called by ISMC and calls return only from ISMC. These extra checks increase the amount of code of each SM and thus amount of time required to compute the MACs of SMs. Another reason is that each inter-SM call is done through the ISMC, adding additional calls, therefore more cycles. Regarding the implementation that is assumed to be loaded by untrusted software, the amount of cycles is significantly larger than the other ISMC with secure loading. Because ISMC is not loaded by trusted software, each SM has to attest ISMC with the help of *attest* instruction. Also, SMs need to attest their intended callee before the actual call via ISMC. These verifications require a significant amount of cycles and are the main reason why the overhead is much larger than the other two implementations.

Additional runs after the first run do not use any cryptographic operations, since all the attestations have been done in the first run. Because of this the amount of cycles in which each benchmark is completed is small. The overhead seen in for the additional run is due to the usage of the shadow call stack. The overhead in this case appears to be large also because the other SMs, besides ISMC, are just dummy SMs without any other purpose other than calling a specific SM as explained earlier.

The dependence between the total number of cycles and the number of additional runs can be seen in Figures 4.5, 4.6 and 4.7. The overhead increase between the baseline and the ISMC implementation is proportional with the overhead of the additional runs.

Based on the measurements from Table 4.2, the ISMC is can be deployed without hindering most applications. More specifically, for a Smart Metering application, the hard deadline is 30 seconds[19]. In the worst case scenario, when ISMC is deployed

TABLE 4.4: Source LOC and binary size of ISMC compared to LOC of average SM

Software Module	Source LOC	Binary size (B)
ISMC	177	4966
Average SM	291	2156

on Sancus without Secure boot, a call between two SMs takes 4.63 milliseconds, being safely bellow the hard deadline. Even though, the overhead more the amount of cycles required for calls, the ISMC can still be used by any applications that that have the deadlines of the order of milliseconds.

4.3.2 Code and binary size

Another important aspect is the size of the ISMC component in terms of Lines of Code (LOC) of the source and binary size. The values of these two properties are compared to the values of an average SM, which is considered to be 300 LOC and 2156 Bytes, the values of the application described in [19]. The results can be seen in Table 4.4. The LOC of ISMC is very compact, being smaller than an average SM. The small size allows for the thorough verification in order to detect bugs and vulnerabilities, making the component more secure. Even though the LoC of ISMC is smaller than that of an average SM, the binary size is significantly bigger. That is mainly because it has been compiled without any size optimizations, due to bugs in the implementation of the compiler.

4.3.3 Memory consumption

Finally, the memory size required for ISMC is discussed. The memory size of the ISCM component is dependent on two parameters:

- *TOTAL_SM*: the maximum number of SMs that can be deployed besides itself;
- *MAX_STACK_SIZE*: the maximum depth of the shadow call-stack.

The maximum depth of the shadow call-stack defines how many consecutive inter-SM calls can be done through the ISMC before returning. The minimum configuration for ISMC is *TOTAL_SM* equal to 2 and *MAX_STACK_SIZE* equal to 1. This configuration has a total memory size of 14 Bytes. Each additional SM increases the required memory by 4 Bytes and each additional shadow call-stack level increases the required stack size by 4 bytes. The ISMC that was used for benchmarking had *TOTAL_SM* equal to 4 and *MAX_STACK_SIZE* equal to 10. The required memory size for that configuration is 58 Bytes.

4.4 Conclusions

The addition of the ISMC component reduces the risks of call-stack shortcutting attacks. This allows for developers to focus on creating the desired applications without the need for special care to protect against these kind of attacks. However, the ease of use offered by ISMC comes at a cost. Namely, the ISMC adds significant overhead to the initialization process and to the first calls between the SMs and the ISMC. As explained in Section 4.3, the increased overhead is mainly due to the cryptographic operations required for computing the additional key of the ISMC SM and for local attestations. Further calls add only a small overhead.

Chapter 5

Secure linking and Software Attestation

The component of the Secure Kernel which provides local attestation is discussed in detail in this chapter.

This section is organized as follows. First, Section 5.1 shortly presents the motivation for a software approach for local attestation. Then, Section 5.2 discusses the design choices in detail. After that, in Section 5.2.2, the implementation of a prototype which provides software local attestation is presented. Next, in Section 5.3 evaluates the prototype. In Section 5.4, the prototype is analyzed in term of area and what utility. Finally, Section 5.5 presents the conclusions.

5.1 Motivation

As mentioned in Chapter 3, the current version of Sancus has increased hardware costs due to the additional hardware required to provide the strong security guarantees. A solution for reducing these costs is to implement the functionality of some of these hardware mechanisms in software while maintaining the same security guarantees. A good candidate for a software replacement is the hardware cryptographic module. Implementing it in software would decrease hardware costs. Additionally, a software implementation of the cryptographic primitives would allow for the possibility of modifying and/or upgrading the implementation of the cryptographic primitives after deploying the platform, which is not possible in the current implementation of Sancus. Having the capability to change the implementation of the cryptographic primitives allows for the possibility of tailoring the implementation based on the requirements of the application, fixing potential bugs, replacing implementations if they become vulnerable or if more efficient/secure implementations become available.

Another disadvantage of the current version of Sancus is that the cryptographic primitives are not interruptible. This aspect is important especially for real-time application, where hard deadlines must be met. Because the cryptographic operations require significant amounts of cycles to complete, the current version of Sancus may be incompatible with some real-time applications in which the deadlines are shorter than

the number of cycles required for executing the cryptographic primitives. Replacing the cryptographic primitives with a software implementation is also a step towards solving this problem and offering full real-time compatibility.

5.2 Solution design and implementation

As mentioned previously, the purpose of this component is to replace the hardware cryptographic module. Because of this, the attestation component builds only on the primitives that do not use hardware cryptographic operations. More specifically, it uses the primitives *protect*, *unprotect*, *get_id* and does not use the primitives *attest*, *attest-caller encrypt* or *decrypt*.

5.2.1 Solution design

In order to design a software solution for local attestation several decisions have to be taken. These decisions are described in detail in this Subsection. The local attestation mechanism is based on obtaining a measurement based on the layout and the content of an SM and then verify whether or not the measurement is correct. To guarantee that the measurement is correct, a solution is required in order to ensure that an attacker cannot tamper with the content of the SM that is being measured during the computation of the measurement or after the measurement without the Kernel and/or the other SMs being able to detect this tampering. A possible solution is to enable protection before starting the measurement. The only way an attacker can change the contents of the SM that is being measured is to disable protection for it. This can be detected by checking that the ID of the measured SM has not changed between the start of the measurement and the end. Additionally, other SMs can trust the measurement as long as the ID of the measured SM has not changed since it was measured.

Attestation mechanism An important decision that has to be taken is the choice of local attestation mechanism. Two options are analyzed. The first option is a software implementation of the hardware attestation mechanism, meaning that the kernel provides an API to other SMs that is used to compute the measurement of a desired SM. The kernel simply provides the computed measurement and it is the responsibility of the SM that requested the measurement to determine whether or not it is correct. It is important to note that the measurement, the verification that the desired SM has not been tampered with and the actual call to the desired SM need to be executed consecutively or even atomically if interrupts are enabled. Otherwise time of check to time of use attacks might be possible, in which the attacker could tamper with the desired SM after the measurement computation but before the call.

The second option is more centralized, the kernel computes the measurement of an SM and also has the responsibility of verifying whether the measurement is correct or not. This approach is based around a registration mechanism in which if an SM wants to be used by other SMs, it has to register to the kernel. Through registration,

the kernel computes the measurement and verifies whether or not the measurement is the expected one. Whenever an SM1 wants to call another SM2, it first checks whether that SM2 is registered by asking the kernel. The kernel checks if the SM2 is registered and that the ID of SM2 has not changed since the registration. If these checks pass, SM1 can call SM2 with high assurance that SM2 has not been tampered with. Similar to the first design option, verification and the call of SM2 need to be executed consecutively or even atomically if interrupts are enabled. Otherwise SM2 would be susceptible to time of check to time of use attacks, in which the attacker could tamper with SM2 between the check and the call. Depending on the requirements, the registration approach can be configured in a more restrictive configuration such that only SMs that are registered can call other registered SMs.

Measurement computation For both design options, the measurement is computed based on the content and the layout of the SM that is being measured. To ensure that that the measurement is trustworthy, several steps need to be taken. Before computing the measurement, the kernel first checks whether the addresses from the layout are protected and belong to the same SM. This can be done by obtaining the ID of the start and end addresses described by the layout with *get-id* instruction and checking that all of them have the same value and are not equal to 0. Without these checks, the kernel cannot guarantee that the SM on which it has computed the measurement is loaded or protected correctly. More specifically, if protection is not enabled or enabled incorrectly, an attacker might modify the contents of the SM before, during or after the measurement and/or may be able to leak confidential data.

However, checking that the addresses described by the layout belong to the same protected SM is not enough. An attacker can load and protect an SM that contains malicious code adjacent to the untampered code as presented in Figure 5.1 **b)**. The malicious code can be used to alter and or leak the protected data. The verifications described so far are not sufficient to detect this attack. To solve this, the kernel can check that the addresses before the start and after the end of the code and data sections do not belong to the SM being measured. This naive approach leads to another vulnerability in which if the data and code section of the SM are adjacent an attacker can load and protect the SM maliciously by enabling protection as described in Figure 5.1 **a)**. This would expose most of the protected data section and would not be detected by the kernel. A possible solution would be for the kernel to also verify the whole memory ranges described by the layout belonging to the same SM. It is important to note that the current version of Sancus *get_id* provides the ID only for the code section, not for the data section. In order to obtain the corresponding ID of addresses that belong to the data section of an SM, the hardware implementation of *get_id* needs to be changed.

A more efficient way instead of verifying whole memory regions, which would not require any changes to the *get_id* instruction is to make the kernel responsible for enabling protection for the SMs. The kernel would accept for registration only SMs that are unprotected and would enable protection based on the layout sent as

parameters before starting the measurement. This means that registration would fail for SMs that have protection enabled apriori to the registration. The latter solution was selected for the actual implementation.

Also, as mentioned in the assumptions discussed in Section 3.3, the security kernel is loaded correctly and has protection enabled before any SMs use it.

Validating the measurement In order to know whether or not an SM is tampered with, we need to verify whether the measurement computed based on the contents and the layout of the SM is correct. To do this, an expected hash is needed. This expected hash can be stored in at least two different ways which are complementary with the attestation mechanisms described in the previous paragraph. One option is to store it in a distributed fashion, with each SM having the expected hashes of other SMs that it wants to call. This would correspond to the first attestation mechanism described. Another option is to store all the expected hashes in the security kernel, creating a more centralized design, corresponding to the registration based attestation mechanism. The advantage of storing all the expected hashes in the kernel is that if an SM is updated, the expected hash corresponding to it needs to be updated in a single place, the kernel. On the other side, if the distributed option is used, whenever an SM is updated all the other SMs that are calling the updated SM need to change the corresponding expected hash. This would lead to a more complicated design than if all the expected hashes are stored in the kernel.

Both attestation designs and their corresponding measurement storage options have advantages and disadvantages. The first design offers more flexibility. This approach has the advantage that the security kernel is not required to have knowledge of all the SMs that can be deployed on the platform. The security kernel only provides the means for computing a measurement. The SMs are responsible for actually attesting their intended callees by comparing the expected measurements with the actual measurements obtained from the security kernel. However, if the code of an SM is updated, then the corresponding hash must be updated in all other SMs that can call the updated SM.

The second implementation provides a more centralized approach, in which the measurement is computed only once for each SM. An advantage of the second approach is that if the code of an SM is updated, the corresponding expected hash needs to be updated only in the kernel. Another advantage is that if this attestation mechanism is paired with the ISMC component described in Section 5.2.2, stricter policies can be enforced in which only registered SMs can communicate with other registered SMs. The main disadvantage of the second approach is that the security kernel needs to have the expected measurements of all the SMs that can be deployed on the platform. This means that new SMs or SMs with code section updated, both which do not have their corresponding expected measures stored in the security kernel would not be able to register to the kernel. A solution for this would be to provide a mechanism through which the new expected measurements can be sent securely from a remote party to the security kernel. The addition of this mechanism would complicate the design even more. Based on the trade-offs presented, the second

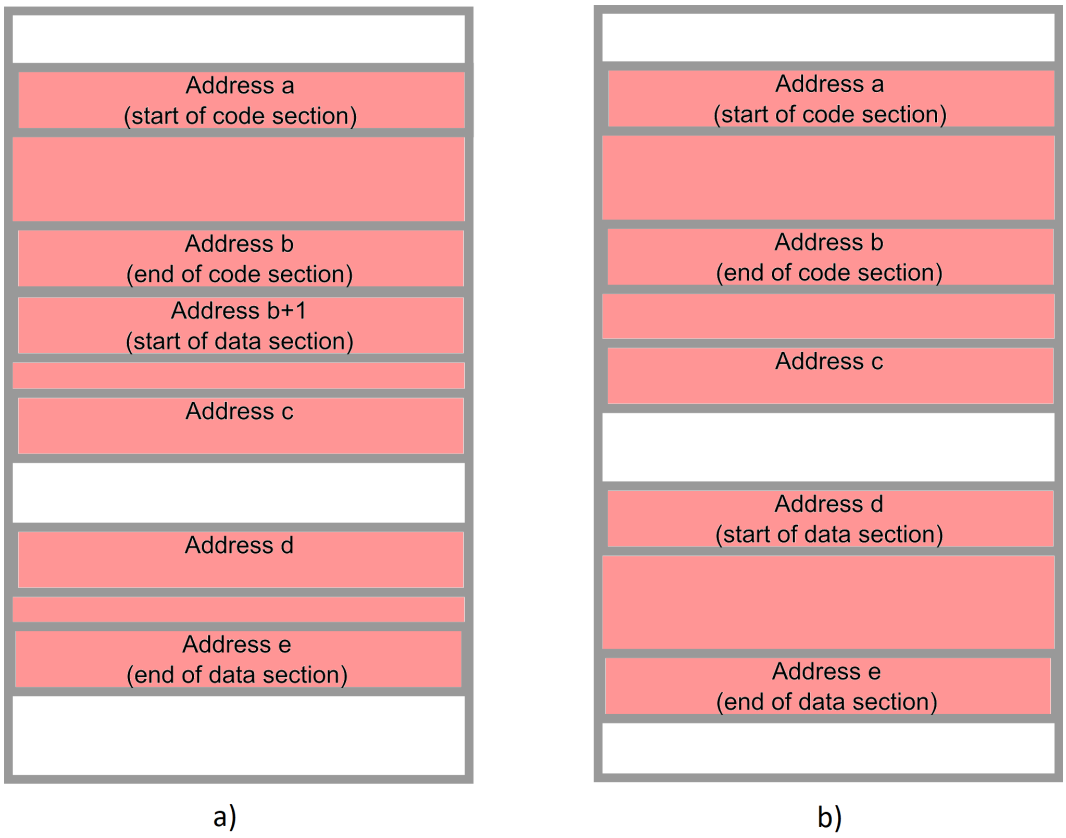

FIGURE 5.1: Two different attacks based on protection enabled maliciously. On the left, an attack that exposes part of the data section when the code and data section are adjacent. On the right, an attack which has malicious code at the end of the benign code section.

design was chosen.

Software acceleration Computing a measurement is an expensive operation. Because of this, it is desirable to compute as few measurements as possible. Similar to the original Sancus local attestation mechanism, for both designs described in this Subsection, an SM only needs to measure intended callee only once. For subsequent calls, just checking that the ID of the intended callee remains unchanged is sufficient. Also, in order to reduce the number of measurements done, the Kernel can save the value of a computed measurement, the corresponding layout and id. If another measurement is requested on an SM that was already measured and its protection has not been disabled since the time of the measurement, then the security kernel can simply return the saved measurement. The kernel can verify that the protection of an SM has not been disabled between measurement requests by checking that the corresponding ID has the same value.

```
save the layout of the SM in the registered_sm_data data structure
if protection is enabled for the SM then:
      registration fails
else :
      compute measurement based on SM
      if measurement is the same as the expected measurement:
            registration successful
      else:
            disable protection of SM and detele the contents of
                  registered_sm_data
            registration fails
```

Figure 5.2: The registration algorithm used by the software local attestation component of the kernel

5.2.2 Implementation description

Next, the implementation of the software attestation mechanism is explained in detail. Whenever a Software Module, wants to securely link with another Software Module, it first needs to register to the kernel. The registration is done by calling the kernel function *register_sm*, which has as parameters the layout of the SM that is going to be registered. The kernel saves the layout data in an internal data structure named *registered_sm_data*. After that protection is enabled for the SM based on the layout data provided. This would provide the guarantee that the protection is enabled correctly and saves the SM ID obtained in the *registered_sm_data*. If the SM is already enabled, the registration fails. Next, the kernel computes a measurement in the form of a checksum based on the content of the code section and the layout. The checksum is obtained using a hashing algorithm and the result is used to detect whether or not the SM that is now registering has been tampered with. After calculating the checksum, the kernel compares it with an expected hash of the SM. If the two hashes match it finishes registration and returns to the SM with a successful return code 0. If the two hashes do not match, the kernel unregisters the SM, by deleting its entry from the *registered_sm_data* and afterwards returns to the SM with an unsuccessful return code 1. In this version if the registration is unsuccessful the control returns to the tampered SM, but it can be also modified to return to a predefined SM. The algorithm for the registration function can be seen in Figure 5.2.

This implementation can be used for both callee and caller authentication. For callee authentication, the caller, can check whether the intended callee is registered. This can be done in two ways. It can call the kernel function, *is_registered*, passing the ID of the intended callee. If the an SM with that ID is registered, it means that

there is an untampered SM loaded. The function checks if there is an SM with that ID registered and returns the ID of the registered SM or 0 if there is no SM with that ID registered. This is sufficient only when no SM has any overlapping layouts. This is quite restrictive, especially because this implementation is tailored towards, low-end devices where there might not be sufficient memory such that all SMs could be deployed without overlapping. To solve this problem, the kernel provides a second function, called *is_registered_with_layout*, which has as parameters the ID and the layout of the intended callee. This function works in a similar way as the *is_registered* function. It searches for an registered SM with the same ID and layout as the ones given as parameters. If one is found, it returns the ID of registered SM, otherwise 0. The same kernel functions can be used by the callee, for caller authentication, either by checking with the *is_registered* function that the caller ID belongs to an SM that is registered or by checking with the *_registered_with_layout* function that the caller ID belongs to an SM that is registered with a specific layout.

For computing the checksum of the SM two hashing algorithm have been tested with the prototype. First, the SPONGENT [4] hash has been used. This is an algorithm designed for hardware implementations and is deployed in hardware in the original version of Sancus. SPONGENT is not optimized for software implementation and requires large amounts of cycles to complete, as it can be seen in the evaluation section (Sect. 5.3). While SPONGENT does not seem to be the best option, it was chosen in order to compare the performance of the hashing algorithm hardware implementation versus a software implementation running on the same platform. For the prototype a SPONGENT with 128 bit security was used. Because of the poor performance of SPONGENT, a second prototype was tested, which used SHA-2 [22] with significantly better results. The SHA-2 implementation used for this prototype had a 256 bit security. Note that the hashing algorithm can be easily replaced with the algorithm that is best suited for the purpose for which the Sancus platform is used. Even if we would not take into consideration the performance of SPONGENT, SHA-2 would be a good candidate to study as well because its implementation has less lines of code than SPONGENT. This is important especially in embedded systems where the available memory is limited. Also, a more compact implementation makes it easier to do thorough verifications in order to remove bugs and vulnerabilities.

After registration, the SM can call other SMs that have been already registered, using a slightly modified version of ISMC component of the kernel. Whenever an SM1 calls an SM2, it does so through ISMC which verifies that the caller, SM1, and the callee, SM2, are registered. This is done by checking if the ids of SM1 and SM2 are in the *registered_sm_data* array with all the SMs registered. The absence of the caller or callee from the array means that it either that SM was not registered or it protection was disabled at some point, not guaranteeing the integrity of the SM. If the callee or the caller are not present in the array, the ISMC returns to the caller with unsuccessful return code 0. Also, if the caller or the callee have different ids from the ones in their corresponding entries in the *registered_sm_data* array, it implies that protection has been disabled before the ISMC call. Because of this, the kernel cannot guarantee that the SMs with different ids have not been tampered with and removes the corresponding entries by unregistering them. Removing these

Table 5.1: The fields that compose the registered_sm_data data structure

Type and name of the field	Description of the field
void* pub_start_addr;	The address where the public section starts.
void* pub_end_addr;	The address where the public start section ends.
void* secret_start_addr;	The address where the secret section starts.
void* secret_end_addr;	The address where the secret section ends.
void* expected_hash[HASH_SIZE];	The expected hash.
char name[NAME_SIZE];	The name of the registered SM.
sm_id id;	The ID of the SM obtained after enabling protection.

entries protects against potential situations in which all the entries are filled and SMs cannot register anymore.

Because a registered SM is protected, other SMs cannot tamper its code or access its state. The only way to modify the code section is for the SM to first disable protection. Doing this will cause the SM to have a different ID even after re-enabling protection.

The kernel is preloaded with all the expected hashes of all legitimate SMs that can be deployed on the node. This option has the advantage that, if an SM is updated, the expected hash needs to be updated only in the kernel instead of all the SMs that are using the updated SM.

Any registered SM can call the kernel function *unregister* to unregister itself. The function searches for the caller ID in the array *registered_sm_data* and removes the corresponding entry.

If a secure connection between Sancus and the third party is created, the mechanism presented earlier is sufficient for remote attestation as well. The remote party can reliably call a specific SM using the security kernel, which guarantees the callee runs untampered.

5.3 Evaluation

5.3.1 Micro benchmarks

Secure Kernel was first tested with a 128 bit security, software implementation of the SPONGENT hashing algorithm [4] in order to compare it with the hardware implementation in terms of cycles required to hash a message. We expected that the software implementation would be slower than the hardware implementation mainly due to the fact that the SPONGENT is designed for hardware implementations. The algorithm uses arithmetic operations such as modulo and multiplication. The hardware implementation of SPONGENT contains hardware implementations for these operations, while the MSP430 does not have specialized instructions for these operations [14]. MSP430 contains a hardware implementation for multiplication [14],

which can be used by loading the operands in special registers. MSP430 does not contain any hardware implementation for modulo. The software implementation of SPONGENT uses the hardware multiplier for multiplication and relies on a software algorithm for modulo which is based on basic operations such as shifting, addition and subtraction. The measurements were done on a Sancus enabled system running at 20 MHz. The results for both the software implementation can be seen in Table 5.2 and Table 5.3. Because the prototype that was used was synthesized with 64 bit security, the measurements for the hardware implementation of SPONGENT were approximated based on the evaluation done in [21]. The results, in milliseconds, were obtained by dividing the number of cycles by the frequency on which Sancus is running. As expected the software implementation is slower than hardware implementation. However, the difference between the two is massive, which was unexpected.

The SPONGENT software implementation is slow even when running on high end CPUs such as Intel x86 processors. To support this claim, the same benchmarks are done on a machine with an i7 4710HQ processor running at 2.5 GHz. The benchmark is run in a Linux 16.04 environment. While Sancus is based around a simple processor on which the cycles required to run a program can be computed in a deterministic way, not the same can be said about modern x86 processors. Because modern x86 processors such as the one used for benchmarking is very complex, containing features such as dynamic frequency scaling, pipeline caches and out of order execution, the number of cycles required for running a program cannot be computed in a deterministic way. In order to measure the number of cycles required for the benchmarks accurately, special steps were taken as described in [23]. We created a kernel module for Linux in which the measurements were done, in order to be able to have full control over the CPU. Before measuring the benchmark, we obtained exclusive ownership over the CPU by disabling hard interrupts and preemption, then we used the CPUID instruction to ensure that all the instruction up to that point are executed. Next, RDTPSCP instruction was used to read the number of cycles since reset. The RDTPSCP is used twice, before starting the benchmark and after finishing it. Subtracting the values obtained from RDTPSCP we obtain the number of cycles required for the benchmark. In order to maximize accuracy, each benchmark is run 10000 times. The average number of cycles for one run, the corresponding standard deviation, the minimum number of cycles for one run and the maximum deviation from the minimum number of cycles were computed over 10000 runs. The results of these benchmarks can be seen in Table 5.4. The SPONGENT software implementation for x86 is significantly faster than the software implementation for Sancus. This is mainly because the x86 processor has hardware support for all the arithmetic operations used for SPONGENT. However even on an x86 processor 128 bit SPONGENT is over 5 orders of magnitude slower than the corresponding 256 bit SHA-2 implementation.

Due to the poor results, another hashing algorithm has been chosen for the security kernel, namely SHA-2. The same benchmarks used for the 128 bit SPONGENT are used for the 256 bit SHA-2 implementation as well. The results in terms of cycles can be observed in Table 5.2 and in terms of milliseconds in Table 5.3. It can be seen that the SHA-2 implementation is much faster compared to the SPONGENT

Table 5.2: Number of cycles required for computing an 64-bit SPONGENT hash using Sancus hardware, 128-bit SPONGENT hash and 256-bit SHA-2 hash using software implementations based on data of different sizes

	Cycles		
Data size	SPONGENT Hw. Sancus	SPONGENT Sw. Sancus	SHA-2 Sancus
256 Bytes hash	25,000	3,574,090,309	211,739
512 Bytes hash	47,500	6,941,412,200	377,387
1024 Bytes hash	92,000	13,676,055,952	708,732

Table 5.3: Milliseconds required for computing an 64-bit SPONGENT hash using Sancus hardware, 128-bit SPONGENT hash and 256-bit SHA-2 hash using software implementations based on data of different sizes

	Milliseconds		
Data size	SPONGENT Hw. Sancus	SPONGENT Sw. Sancus	SHA-2 Sancus
256 Bytes hash	1.25	178,704	11
512 Bytes hash	2.37	374,070	19
1024 Bytes hash	4.6	683,802	35

Table 5.4: Average number of cycles, standard deviation, minimum number of cycles and maximum deviation from minimum cycles based on 10000 runs computing a 128-bit SPONGENT hash based on data of 256, 512 and 1024 Bytes size on an Intel x86 processor

	Cycles					
	SHA-2 256 Bytes hash	SHA-2 512 Bytes hash	SHA-2 1024 Bytes hash	SPONGENT 256 Bytes hash	SPONGENT 512 Bytes hash	SPONGENT 1,024 Bytes hash
Average number of cycles	5,138	8,527	16,507	56,767,672	100,519,359	197,657,301
Standard deviation	1,527	1,987	4,658	4,949,069	1,953,531	2,052,386
Minimum number of cycles	4,728	8,224	15,214	51,128,956	99,395,878	195,930,934
Maximum deviation from minimum	29,520	35,386	98,023	10,678,085	17,083,543	109,526,975

TABLE 5.5: Number of cycles required for computing macro benchmark for 1, 2, 4, 8, 16, 32 clients, with small and average large sized server SM, using attestation with and without the security kernel

Benchmark	Cycles					
	Nb. of Clients					
	1	2	4	8	16	32
Benchmark using Kernel Sw. attestation on small server SM	537,081	537,645	538,773	541,029	545,541	554,565
Benchmark using Hw. attestation on small server SM	20,892	41,784	83,568	167,136	334,272	668,544
Benchmark using Kernel Sw. attestation on average server SM	1,796,480	1,797,044	1,798,172	1,800,428	1,804,940	1,813,964
Benchmark using Hw. attestation on average server SM	84,504	169,008	338,016	676,032	1,352,064	2,704,128
Benchmark using Kernel Sw. attestation on large server SM	4,340,906	4,341,470	4,342,598	4,344,854	4349366	4,358,390
Benchmark using Hw. attestation on large server SM	275,154	550,308	1,100,616	2,201,232	4,402,464	8,804,928

implementation. Even though it is approximately 8 times slower than the hardware implementation of SPONGENT, the SHA-2 implementation can still be used for many applications.

5.3.2 Macro benchmarks

Another way of evaluating the Software Local Attestation is by running macro benchmarks. An interesting use case is based around the client-server concept, where multiple SMs, which we call client SMs, call one or more other SMs, which we call server SMs. Using the original Sancus attestation mechanism, each client would have to compute the hash of the server SM at least once, in the best case scenario. Using the security kernel the hashes would be computed once per server SM in the best scenario. Best case scenario refers to the scenario where after loading and protecting the n SMs they are not unprotected or unloaded for the duration of the scenario. A scenario example based on the use case described earlier would be an server SM controlling a resource (such as storage) that needs to be shared between client SMs. This type of scenario is discussed in depth in [32]. If a version without the security kernel is used, each client SM that calls the server SM must first attest it. However, if the security kernel is used, the same hash needs to be computed only

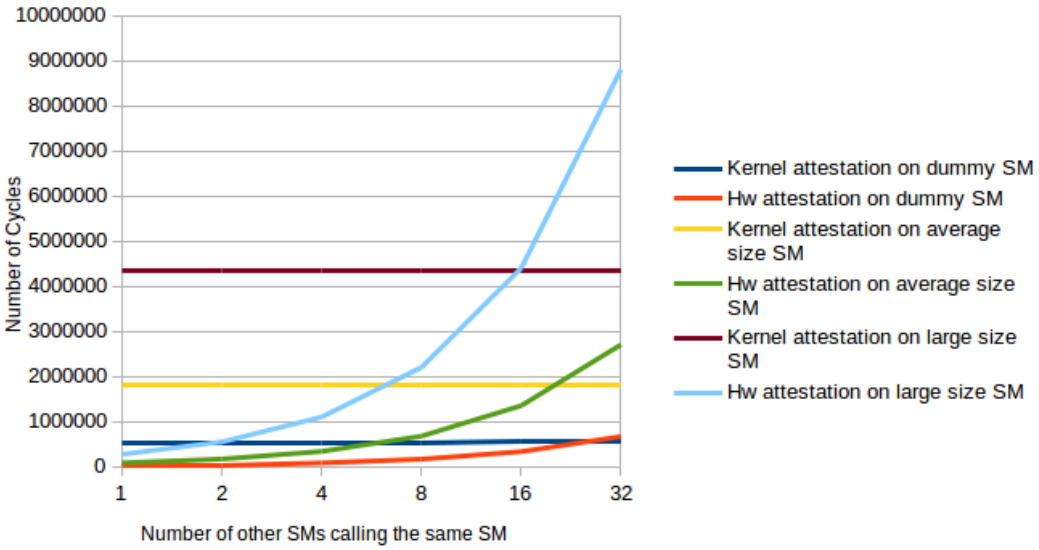

Figure 5.3: Number of cycles required for computing macro benchmark for 1,2,4,8,16,32 clients, with small and average sized server SM, using attestation with and without the security kernel

once, independent of how many client SMs are calling the server SM.

In order to evaluate the attestation mechanism a simple macro benchmark is used. We measured the number of cycles required for multiple client SMs call one server SM and compared the results using the hardware attestation mechanism versus using the security kernel. The macro benchmark has two parameters, the server size and the number of clients calling the server. Regarding the server size, 3 options were used: first, a small dummy server of only 536 Bytes and then. The second option is an average size server 2156 Bytes. This value was selected based on the average size of an SM based on the implementation of the Smart Meter application described in [19]. Finally, a large server of 8624 Bytes was tested which represents approximately 4 times the size of the average sized server. In order to obtain the desired server sizes, the code section of the server SM contains 2 functions: a function which registers the SM when called the first time. Additional calls to this function return immediately. The second function is filled with *nop* operations added until the expected size is reached.

Regarding the number of clients, the benchmark was computed for 1, 2, 4, 8, 16 and 32 clients. The prototype on which the experiments were done, could contain a maximum of 4 simultaneous SPMs, out of which 1 would be used by the security kernel. Because of this, the benchmark could only run on the prototype for 1 and 2 clients. The rest of the measurements were computed based on the results obtained from the scenario in which one client calls the server. The results can be computed reliably because Sancus is deterministic, meaning in this case that the amount of cycles required to computed a hash of a specific size will always be the same and the

amount of cycles required for calling an SM is also fixed. Using this information we computed the results for the rest of the scenarios in the following way. For hardware attestation we multiplied the amount of clients with the number of cycles required for computing the hash and calling the server SM. For software attestation, we multiplied the number of cycles required for calling the server SM with the number of clients and added the number of cycles required for registering the server SM.

Even though scenarios with over 16 clients are uncommon for embedded systems, similar results can be obtained with fewer clients that would be loaded and unloaded regularly due to limited available memory.

The results of the benchmark can be seen in Table 5.5 and in Figure 5.3. The software attestation mechanism is significantly slower than the hardware attestation, as expected from the micro benchmark evaluation. However, if the number of clients increases, the difference between the two implementations diminishes, the software implementation even having better results when many clients are used. This is due to the fact that the software attestation implementation computes just one hash on the server independent of the number of clients, while the hardware implementation computes one hash on the server for each client. The number of clients required for the software attestation implementation to be more efficient than the hardware attestation decreases if the size of the server SM increases. In the case of the small server of 536 Bytes, 27 clients are required for the software attestation mechanism to be more efficient, while for the large server of 8624 Bytes only 16 clients are required. As it can be seen in Table 5.5 the number of cycles increases slightly with the increase of clients. This increase is due to the additional call to the kernel that the client has to make in order to check whether the server is registered or not.

5.3.3 Code and binary size

Next, the source LOC and binary size are discussed. The results can be seen in Table 5.6. The Source LOC of the software Local Attestation mechanism depends greatly on the LOC of the hashing implementation. In the version that uses the SPONGENT algorithm the implementation accounts for 58% of the total LOC of the component and in the version that uses the SPONGENT algorithm the implementation accounts over 35% of the total LOC of the component. The component has more LOC than an average SM. If this component is used with the ISMC, the LOC is more than twice of an average SM. However, it is still small enough to be verified thoroughly for bugs and vulnerabilities.

It is important to note that the binaries have been compiled without any size optimizations due to bugs in the implementation of the compiler. Compiling them with size optimizations should create more compact modules.

5.3.4 Memory consumption

The memory size required by the software local attestation component is analyzed. The memory size this component is dependent on one parameter: *TOTAL_SM*, which represents the maximum number of SMs that can be deployed. A configu-

Table 5.6: Source LOC and binary size comparison for ISMC, Local Attestation mechanism, hashing algorithms and average SM

Software Module	Source LOC	Binary size (B)
ISMC	177	4966
Sw. Local attestation without cryptographic algorithm	308	3072
SPONGENT	434	6538
SHA-2	171	3494
Sw. Local attestation with SPONGENT and without ISMC	742	9322
Sw. Local attestation with SHA-2 and without ISMC	479	4740
Sw. Local attestation with SHA-2 and with ISMC	656	9706
Average SM	300	2156

ration with *TOTAL_SM* equal to 2 would require a stack size of 220 Bytes. Each additional SM would require an extra 84 Bytes. This is mainly because the local attestation mechanism stores in software the expected hash for each SM. The software local attestation component can also be used together with the ISMC adding *4 * MAX_STACK_SIZE + 4 * TOTAL_SM + 2* Bytes.

The security kernel used for the macro benchmarks contained both the ISMC and the software local attestation component, having *TOTAL_SM* equal to 4 and the *MAX_STACK_SIZE* equal to 10. This version requires a memory size of 434 Bytes.

5.4 Discussion

5.4.1 Applications

The software attestation mechanism is a step towards real-time compatibility. In order to support interruptibility, Sancus would need to enforce a mechanism that would securely save the state of the SM that is running at the moment when an interrupt occurs before jumping to the corresponding interrupt routine. A solution that provides secure interrupts for the original version of Sancus is described in [6]. Another source of inspiration could be TyTAN [5], which also support secure interrupts. Even with the proper mechanism, the current implementation does not support interruptibility while still being trustworthy. This is because the kernel cannot guarantee that the measurement is correct if interrupts are possible. More specifically, an attacker can interrupt the kernel while it is computing the measurement, disable the protection of the SM that is being measured, modify it by extending the code region, protect it again and return to the kernel. The kernel would obtain the

expected hash on an modified SM without detecting that it has been tampered. A possible solution would be to make the measurement atomic. However, this would limit the applications for which the software attestation can be used given that the software implementation is significantly slower than the hardware implementation. A better solution is to add an extra verification after the measurement is complete which would check that the SM that is being registered still has the same id. If the SM has been tampered with the ID cannot be the same during the same boot cycle [21].

Depending on the requirements and the hashing algorithm used, the software local attestation implementation described in this chapter can be used for various applications where the deadlines are in the order of milliseconds, such as non-critical automotive (e.g multi-media applications) and medical applications (e.g. patient data gathering applications). However, for critical applications such as the controller for brakes of a car or the controller of an implant would require faster response times than even one millisecond, making the current implementation inadequate, even when using an efficient hashing algorithm such as SHA-2.

5.4.2 Area

The area in term of registers and LookUp Tables (LUTs) was measured based on the data generated in the Xilinx Design Summary and Module Level Utilization reports obtained from the Xilinx tools. A Sancus prototype with 128 bits security and 4 SPMs requires a total of 3031 registers and 5211 LUTs. The cryptographic module and the isolation mechanism requires 650 registers and 1060 LUTs. The self-protecting module mechanism requires 884 registers and 1336 LUTs, the majority of which is used for storing the keys and layout of each SM. Using the software attestation mechanism described in Section 5.2.2 would remove the need for the hardware cryptographic module, thus reducing the hardware costs significantly.

The benchmarks from Section 5.3 were run on an Sancus platform with 41 KB ROM, 16 KB of RAM and 512 B allocated for each SM stack.

5.5 Conclusion

In this Chapter, a software solution for secure linking and local attestation is described in detail. The change from a hardware implementation to a software implementation offers several trade-offs. A software implementation has reduced hardware costs, can be changed after deployment and can be interrupted. Interruptibility is essential for compatibility with real-time applications. However these advantages come at a price, the software implementation is significantly slower than the hardware implementation. Regarding interruptibility, special care must be taken such that the state of the cryptographic operation is saved before jumping to the interrupt routine, the routine cannot affect the result of the cryptographic operation and that after finishing the routine, control is immediately given back to the kernel and the cryptographic operation.

Chapter 6

Conclusion

The goal of this thesis was to design a security kernel for a Protected Module Architecture (PMA) and study the properties obtained. More specifically, we design a security kernel for Sancus. Sancus is a PMA designed for resource-constrained networked embedded systems. A few potential disadvantages for Sancus were identified. More specifically, the original version of Sancus has increased hardware costs, non-interruptible cryptographic primitives and special care is required to protect against call-stack shortcutting attacks. In order to try to tackle these disadvantages, a security kernel with two major components was designed. The components can either be used together or independently.

The fist component developed is the Inter-Software Module Communication (ISMC) component which provides a proxy-like inter-SM communication mechanism which protects against potential call-stack shortcutting attacks. The other component is a mechanism for local attestation which uses a software implementation for the cryptographic primitives. By moving the cryptographic primitives into software, the increased hardware cost and interruptibility is addressed.

The rest of this chapter is organized as follows: in Section 6.1 the main limitations and challenges encountered during the development of the security kernel prototype are discussed. In Section 6.2 potential applications for the security kernel are discussed and in Section 6.3 directions for future work are discussed.

6.1 Limitations and Challenges

The security kernel has two components: the ISMC and the software local attestation. The ISMC is a proxy for communication between SMs which provides protection against call-stack shortcutting attacks through the use of a shadow call-stack. The second component, the software local attestation, provides a local attestation without the use of the hardware cryptographic primitives. In this Section, the limitations and challenges of both components are discussed.

6.1.1 ISMC

The ISMC provides protection against call-stack shortcutting attacks, meaning that inter-SM calls happen in a predictable way. This minimizes the ability of malicious Software Modules to affect the behavior of other Software Modules. The ISMC also allows developers to focus on creating applications without having to implement themselves methods to protect against this kind of attacks.

The downside of the ISMC is that it adds a significant overhead during the first calls between the SMs and the ISMC due to cryptographic operations required for local attestations. Because of this, some applications with strict deadlines which would be deployed on the original version of Sancus would be able to comply with the deadlines. However, if the same applications would use ISMC for communication between SMs, then the same applications might not be able to comply with the deadlines. Besides the computational overhead introduced, the ISMC, being a software, component uses part of the memory available, reducing the amount of memory available for other SMs. Another limitation is given by the use of the shadow call-stack in the implementation of ISMC; the number of recursive calls are limited by the size of this shadow call-stack.

6.1.2 Software Local Attestation

The software local attestation prototype provides several advantages over the hardware implementation. First, by implementing the cryptographic primitives in software, the software local attestation prototype can reduce the hardware costs of Sancus, by removing the hardware cryptographic module. The original version of Sancus, has non-interruptible cryptographic primitives, making it incompatible with at least some real time applications. The prototype, being a software implementation, can be interrupted and is a step towards full real-time compatibility. Another advantage of the software local attestation implementation is the flexibility that it adds by allowing the possibility of changing the measurement implementation based on the application requirements such as security, binary size and speed without hardware changes.

The software implementation also has several limitations, such as significant computational overhead. The size of the overhead depends on the hashing algorithm chosen and its corresponding security. The prototype was tested with two hashing algorithms: SPONGENT and SHA-2. SPONGENT is a hashing algorithm optimized for hardware implementation and it is used in the hardware implementation of Sancus. The software local attestation component was deployed with a software version of SPONGENT in order to compare the performance against the hardware implementation. The computational overhead added by the software implementation of SPONGENT was massive. However, using SHA-2 instead of SPONGENT, led to acceptable overheads compared to the hardware implementation of SPONGENT.

Similar to ISMC, being a software component, it limits the available memory for other SMs. Also, even though this implementation is a step towards real-time compatibility, it still requires extra steps towards that goal.

6.2 Applications

6.2.1 ISMC

Even though, the addition of ISMC creates a significant overhead, it might be an acceptable trade-off for protection against potential call-stack shortcutting attacks. Depending on the application requirements, the ISMC can be used for applications that have deadlines in the order of milliseconds.

6.2.2 Software Local Attestation

Besides lower hardware costs, the software local attestation component also provides more flexibility, by allowing the possibility of updating the local attestation mechanism and changing the measurement implementation after deployment. This flexibility is useful especially for applications that are deployed for long periods of time. In this time, the measurement implementation might become vulnerable and the software implementation would allow for an easy replacement with an implementation that provides better security. In the case of a hardware implementation if the measurement implementation would become vulnerable, a replacement of the actual platforms would be needed, which could be very expensive.

The software local attestation component of the kernel adds significant overhead compared to the hardware implementation. However, this overhead is strongly correlated with the implementation used for computing the measurement. If SPONGENT, the same hashing algorithm as the hardware implementation, is used, then the computational overhead added is massive making it impractical for most applications. If SHA-2 is used, the computational overhead is much smaller and the software local attestation can be used for applications that have deadlines for attestations in the order of milliseconds.

6.3 Future work

The Secure Kernel prototype described in this thesis provides remote attestation in conditions more restrictive than the original version of Sancus. It has been left for future work to extend the prototype with an mechanism for remote attestation which would work in similar conditions as the original version of Sancus.

The software local attestation mechanism was only tested with two hash implementations. A more thorough evaluation of the software attestation using more hashing algorithms is left for future work. This evaluation would be helpful in finding the optimum hash implementation depending on the application requirements.

Finally, the local attestation prototype described here is only a step towards compatibility with real-time applications. The design and implementation of a real-time compliant solution is left to future work.

Appendices

Appendix A

Paper First Semester

A Security Kernel for Protected Module Architectures

Alexandru – Madalin Ghenea

1. Introduction

In recent years there has been a growing trend towards increasing the network connectivity of embedded computing devices and towards an Internet of Things (IoT) [1], [2].

A significant part of these devices are used in safety-critical and security-sensitive applications, such as automotive applications, industrial control systems, critical infrastructure, medical applications and e-payment systems [3], [4]. By adding network connectivity, the platforms are more susceptible to remote security threats. Studies have shown varied threats and vulnerabilities, such as using code injection to build self-propagating worms [5], [6], remote car hijacking [7], infecting network connected appliances with botnets [8].

While for high-end systems, connectivity and software extensibility problems are well studied, for resource-constrained embedded systems, finding efficient security solutions is still a very active area of research [3], [4], [9], [10], [11]. For high-end systems, there are many security solutions, depending on the requirements. For example, privilege levels [12], [13], support for virtual memory, secure co-processors [14], firmware security services, verifiable execution of measured code [15]. Due to economic reasons those solutions are not available for low cost embedded systems. In recent years a new concept called **Protected Module Architectures (PMAs)** has emerged, that provide strong isolation. PMAs are suitable for both high-end devices, providing extra security in combination with the solutions presented earlier and for low-end devices, providing strong isolation without the need of more expensive mechanisms. In PMAs the security-critical components are separated into smaller protected modules, which are isolated by hardware or a small software layer from all other software on the system such that they cannot be tampered with. The separation allows for the security critical modules to be rigorously verified for bugs and vulnerabilities.

There is a wide array of implementations for PMAs, on different levels of architecture going from complete hardware implementation (SMART [11], Sancus [9]) to a mix between hardware and software (TrustLite [3], TyTAN [4], Intel SGX [17]) to OS kernel implementations (Salus [18]) or hypervisor-based solutions (Fides [19]), providing different security guarantees based on the implementation.

2. Background

This section provides background information about Trusted Computing Base, the concept of Protected Modules Architecture, introduces an attacker model used against PMAs, several security properties that can be offered by PMAs and presents three different implementations of PMAs designed for low-end embedded platforms.

Most devices have multiple software programs deployed from mutually untrusting software providers. Because the software programs deployed are mutually untrusting, there is a need for an isolation mechanism, such that software programs cannot influence each other in undesired ways. The conventional software isolation mechanism relies on the Operating System to allocate virtual address spaces for each software program. The main drawback of this approach is that the security and integrity of the programs rely on the kernel of the Operating System, which can be very large and hard to check for vulnerabilities. That is why instead of trying to guarantee the security of large components, another approach is to minimize the security critical components.

Trusted Computing Base (TCB) refers to the hardware and software components of the security architecture that provide the security guarantees. Exploiting components outside the TCB cannot affect the expected behavior of the TCB. Because the TCB is so critical to the security and integrity of the whole platform it is important for it to be as small as possible to allow exhaustive checks for bugs and other vulnerabilities.

PMAs are security architectures that provide strong isolation for software programs without the need of an operating system. PMAs usually also provide other security features such as local and/or remote attestation, sealing etc.[20] While PMAs can be used for both high end systems in order to strengthen security even more and low end systems where PMAs can provide strong security guarantees without the need for more expensive mechanisms, the focus in this paper will be on PMAs that are targeted for low end embedded systems. Many of the PMAs designed for embedded systems use Program Counter Based Memory Access Control (PCBMAC) to enforce software isolation [21], [22]. The PCBMAC is a mechanism that enforces memory access control rules that can take the program counter into consideration. While PCBMAC has a lower hardware cost than virtual memory and does not influence the critical path [9], it can guarantee even stronger isolation between software, thus making it suitable for low-end embedded systems [9], [11].

As previously mentioned there is a large spectrum of PMA implementations, from complete hardware implementation on the one end to complete software implementations on the other. Hardware implementations can withstand attackers that can manipulate any deployed software, but cannot be upgraded or modified afterwards, thus being less flexible than software implementations which can provide upgradability, but require mechanisms to protect TCB code from attackers. The remainder of this section will revolve around three implementations: Sancus, TrustLite and TyTAN. Sancus relies only on a hardware TCB and is on the hardware end of the PMA implementation spectrum. TrustLite has both hardware and software in its TCB, positioning it somewhere in the middle of the PMA implementation spectrum. TyTAN is an extension of TrustLite and has a significantly larger software TCB than TrustLite, situating it more towards the software end of the PMA implementation spectrum. First an attacker model that is common for these PMAs will be described. Afterwards, several security properties that can be offered by these PMAs are presented. Finally, each of the three architectures are discussed in terms of design choices, architecture and properties offered.

2.1. Attacker model

These three implementations can protect against a powerful attacker/adversary model. The attacker is assumed to be able manipulate all the software that is deployed and that is not part of the TCB. Furthermore, the adversary can also deploy new software on the platform. Regarding cryptography, the Dolev-Yao [23] attacker model is used and regarding network communication, the attacker controls all communication with the platform. Finally, hardware attacks are out of scope [3], [4], [9].

2.2. Security properties

Next, the security guarantees desired are presented. The three PMAs provide at least some of these guarantees.

- **Software module isolation.** This means that the software modules are isolated from each other. More specifically, no other software module (SM) can modify the code, read or modify the state of another SM.
- **Local attestation.** More specifically, an SM can detect whether a specific SM has been loaded correctly on the same device. Note that the loading process is not trusted.
- **Remote attestation.** This specifies that an outside party (not on the same device), for example the software provider, can detect with high assurance that a specific SM is loaded correctly on the device. Note that the loading process is not trusted.
- **Secure linking.** An SM can detect with high assurance that it is calling a specific SM. Also, an SM can detect that it has been called by a specific SM.

Next the three architectures are presented. For each architecture, the specific terms are used as they are presented in their papers.

2.3. Sancus

Sancus [9] is a Protected Module Architecture tailored for low-end networked embedded systems that provides software isolation, local and remote attestation with only a hardware TCB [9]. It is implemented around an MSP430 microprocessor, which is suitable for low-end embedded applications. Also a special C compiler has been developed that compiles C modules to Sancus SMs [9].

Sancus is based around a setting in which multiple programs from different, mutually untrusting, software providers are deployed on the system [9]. Each SM has an identity, which consists of the content of the text section and the start and end of the addresses of the protected text and data sections. These addresses are known as layout.

Next, the main guarantees offered are discussed in terms of how they are implemented.

Software Module Isolation. The SM isolation is enforced by a hardware PCBMAC. More specifically, the PCBMAC ensures that the protected data section is only accessible when the program counter is in the corresponding code section of the same module [9]. Also, each SM can only be accessed through a single entry point, making it impossible for attackers to misuse specific code chunks. After an SM is loaded by an untrusted loader, its protection can be activated using a hardware instruction, called *protect*. This instruction verifies that the SM does not overlap with other SMs, generates a key used for authentications and attestations and assigns a sequential ID to the protected SM. This ID is guaranteed to be unique within a boot cycle and it's not reused even after the module is unloaded. Note that the ID is different from the identity of the SM. Once the *protect* instruction has finished, the code of the protected SM cannot be modified anymore.

To disable protection, the *unprotect* instruction is used. This instruction can be called only by an SM on itself.

Furthermore, for ensuring strong isolation, each module has its own call-stack to prevent leaking of stack allocated variables.

Remote attestation. As stated previously, the key generated can be used for remote attestation. This is done with the help of the instruction *MAC-seal* which is used to compute the MAC of a message based on key of the SM. This MAC can be used to protect the integrity of the message sent to the software provider [9].

Local attestation. Sancus provides both callee and caller authentication. The solution for callee authentication is explained first. In order for an SM_1 to verify that an SM_2 has not been tampered with, the following instruction is used:

MAC-verify address, expected MAC

This instruction checks that a module is loaded and protected at the provided address, computes the MAC of the identity of that module using the key of SM_2, compares it with the expected MAC and if the MACs match it returns the ID of SM_2, otherwise it returns 0. In order to use this instruction for local

attestation, SM_1 must be deployed with the expected MAC of SM_2. Because computing the MAC is an expensive operation, after identifying SM_2, the instruction *get-id address* can be used for further attestations, which returns the id of the module located at the address given as parameter. It is important to note that *get-id* is significantly faster than MAC-verify and does not use any cryptographic primitives.

Caller authentication can be resolved with the help of the instructions *get-caller-id*, which returns the ID of the previous running SM and *MAC-verify* which was explained earlier.

2.4. TrustLite

TrustLite [3] is security architecture that enforces isolation of software modules and provides local attestation with a small hardware and software TCB. Because TrustLite has a smaller hardware TCB it also has smaller hardware extension cost compared to Sancus.

In TrustLite, the code, associated data and meta-data is known as a *program*. A *task* describes the runtime state of a program, including its CPU state, call stack and other volatile data. Tasks that are designed to implement a particular security mechanism are called *trusted tasks* or *trustlets* [3].

Software module Isolation. Isolation is ensured in hardware using an Execution-Aware Memory Protection Unit (EA-MPU). The EA-MPU organizes the physical memory into regions, each with associated access control rules. This is similar to the PCBMAC, in both cases data is accessible based on the address of the instruction that is being executed.

The initialization of the EA-MPU and the loading of programs, including trustlets, is done at startup by a Secure Loader which is part of the TCB [3]. The EA-MPU registers cannot be modified after completing the initialization. This implies, trustlets cannot be loaded or unloaded dynamically at runtime, which makes TrustLite less flexible than Sancus which can load/unload SMs at runtime. Finally, similar to Sancus to protect against code reuse, each trustlet can only be accessed through entry vectors.

Local attestation. TrustLite provides callee authentication. A trustlet T_1 can check that a trustlet T_2 has been correctly loaded by verifying that the MPU registers are configured correctly and can compute a hash of the code to verify if T_2 has been tampered with before loading. TrustLite also provides caller authentication through the use of a three-way handshake protocol between trustlets.

2.5. TyTAN

TyTAN [4] is an extension of TrustLite that provides strong isolation, local and remote attestation, dynamic software loading at runtime and real-time guarantees [4]. The hardware TCB of TyTAN is smaller than that of TrustLite, while the software TCB of TyTAN is significantly larger than that of TrustLite.

TyTAN is based around a similar setting to Sancus, in which multiple tasks from different providers are deployed. The tasks deployed on the device are mutually untrusted.

Software Module Isolation. Isolation is enforced again using the EA-MPU. The EA-MPU is initialized by a Secure Boot which loads all the other tasks that are part of the TCB, such as the EA-MPU driver. The EA-MPU driver controls the EA-MPU configuration at runtime, allowing for dynamic loading and unloading of tasks at runtime. This design choice makes TyTAN more flexible than TrustLite, which does not provide dynamic loading of programs.

Remote attestation. The Root of Trust for Measurement (RTM) task is responsible for local and remote attestation [4]. RTM computes a hash based on the code of for each task loaded and the hash is then used as the identity for the task and can only be modified by RTM.

Remote attestation is obtained using MACs, in a similar fashion to Sancus. The MAC is computed by the Remote attest task, using an attestation key derived from a platform key and the identity of the task. Because RTM and Remote attest are tasks, they can be interrupted, including during the computation of the hash making TyTAN compatible with real-time applications.

Local attestation. As mentioned previously, the RTM task is also responsible for local attestation. The identity can be used to attest the integrity of the task to other tasks. Communication between tasks is done via the Inter-Process Communication (IPC) Proxy task. TyTAN also provides caller and callee authentication using the IPC Proxy.

3. PROBLEM STATEMENT AND SOLUTION

While Sancus is a very powerful architecture, care has to be taken to guard inter-SM flow. This paper proposes a solution for Sancus to *Call Flow Integrity*. This means that the inter-SM calls happen in a predictable way. The called SM can only return to the caller SM. This minimizes the ability of malicious SMs to affect the behavior of other SMs. In the current version of Sancus there are scenarios in which the Call Flow Integrity of the SMs is not maintained. The remainder of this section is split into several subsections. First, a couple of attacks that change the control flow of Sancus are presented and afterwards the defense mechanisms that can solve protect against these potential attacks are discussed.

3.1. Call flow attack scenarios

A first real world scenario in which this can happen is a fire alarm system deployed on a Sancus platform. The fire alarm application contains 3 SMs: a Control Module SM which controls the sound alarm and contacts the fire department, a Smoke Sensor SM, which gathers data from the smoke sensors and a Sensor Data Logger SM that logs the data from the smoke sensors. The normal call flow would be as follows. At periodic amounts of time, the Control Module requests sensor data from the Smoke sensor. The Smoke sensor calls the Sensor Data Logger to log the data. The control returns to the Smoke Sensor which returns the sensor data to the Control Module. This flow can be seen in Figure *1*. However, if the Sensor Data Logger has a bug or is malicious, it can return directly to Control

Module with possibly incorrect data, starting false alarm or even worse not starting alarms when it should. The malicious behavior can be seen in Figure *2*.

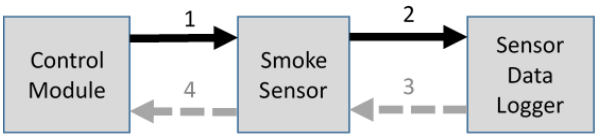

Figure 1 Normal Call Flow (Black arrows represent inter SM calls, gray arrows represent the returns from these calls and the number above these arrows represent the order in which the operations happen.)

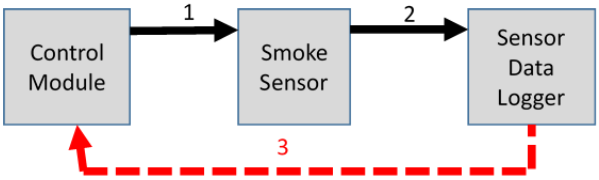

Figure 2 Fire alarm - Malicious Call Flow (The red arrow represents the malicious return call)

Another more complex scenario in which the inter SM call flow is affected is when SMs have to call each other several times. In this scenario, 3 SMs are deployed: A, B and C. The normal call flow is as follows: A calls B, which calls C. In order for C to give a result to B it needs extra information from B and it calls back to B. B returns the extra information to C, C finishes its computation and returns back to B and, finally, B returns to A. Again, if B has a bug or is malicious, it can change the call flow of the application. After C calls back to B for more data, B can return to A without waiting for the response from C. In this situation C becomes unavailable, because it is waiting for a return from B for a potential infinite amount of time. The normal, and the malicious call flow can be seen in Figure *3*.

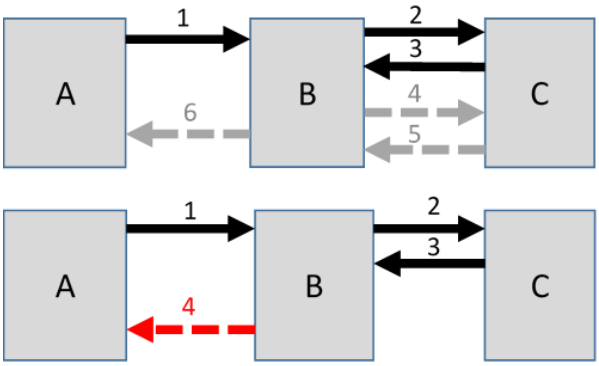

Figure 3 Call flow scenario - Normal call flow (Top modules) versus malicious call flow (Bottom modules)

3.2. Defense mechanisms

For the first scenario, the most straightforward solution would be for each SM to check that the return call is indeed from the same SM that was called. This can be done by using the *get_caller_id* primitive that returns the id of the previous SM, which in this case is the SM that used the return call. This can be enforced either at compiler level, or manually by the developers. The tradeoff between the two approaches is that the compiler approach would give a default behavior for this kind of violations (e.g. system restart), while the manual approach would allow for specific behavior based on the implementation of each SM. However, the manual approach adds extra burden on the developers, by making sure that all calls are checked.

For the second scenario, the method described in the previous paragraph is not sufficient. Because of this, another solution that also covers the first scenario is proposed. The solution consists of a small security kernel, which would basically be an SM responsible for all inter-SM communication. This SM is called Inter Software Module Communication (ISMC) and contains an internal shadow call stack in order to ensure call flow integrity.

More precisely, when an SM1 wants to call another SM2 it does so by calling the ISMC and sending as a parameter the entry address of SM2. ISMC pushes the address of the caller, SM1, in the shadow call stack and calls the intended SM2. When SM2 finishes, it returns to ISMC. Then ISMC checks that it is the same SM that it has called and then proceeds to pop the address of SM1 from the shadow stack and returns to SM1. The solution for scenario 1 using ISMC can be seen in Figure *4*.

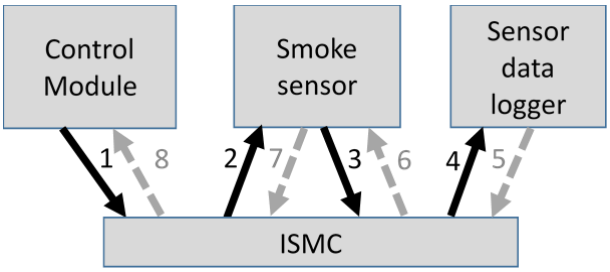

Figure 4 ISMC based solution for fire alarm scenario.

To ensure that all communication is done via ISMC all SMs should check whenever they are called that they are called by ISMC and that when they get a return that it is from ISMC. This can be done by using the primitive *get_caller_id*. More concretely, whenever a benign SM_1 is called by another SM_2, it checks with *get_caller_id* that ISMC is the caller. If another the caller is another SM then it simply ignores the call and returns back to the caller. Also, after each return the former caller SM_2 should first verify the identity of the SM from which it has returned and if the identity does not match the ISMC it should have a default behavior (e.g. system restart). In the current implementation, ISMC relies only on the *get_caller_id* and *get_id* instructions and does not use any cryptographic instructions.

The ISMC can be deployed on Sancus in at least two ways. First, a secure boot similar to that of TyTAN [4] can be used to load ISMC in a secure way with a predefined ID before the other SMs. Because ISMC is loaded securely, the other SMs don't need to verify if ISMC has loaded correctly. An SM only needs to verify the intended callee and then call it via ISMC. A second approach is that ISMC is loaded by untrusted software similar to the rest of the SMs. In this case the SMs need to verify ISMC before calling it.

While this design protects against hijacking the call flow between SMs at least in the scenarios presented, it also adds computational overhead due to the extra calls and checks

required by this design.

4. Results

Three benchmarks have been measured to study the impact on performance made by the addition of the ISMC. First benchmark is composed of two protected SMs, A and B, in which A makes a simple call to B. The Second benchmark consists of three protected SMs, A, B and C, in which A calls B and B then calls C. The last benchmark is again comprised of three protected SMs, which behave as presented in Scenario 2 from Section 3.2.

The baseline for these scenarios is measured using a Sancus without a security kernel and compared against two implementations of Sancus with the security kernel (ISMC). The first implementation of the security kernel, the ISMC is assumed to be loaded by a secure boot before any other SM and has a predefined ID. In the second implementation, it is assumed to be loaded by untrusted software.

Table 1 CPU cycles benchmark measurements comparing the Baseline (Sancus without ISMC) with versions of Sancus with ISMC with and without the assumption of being loaded by trusted software

Benchmark	Init. cycles	First run cycles	Second run cycles	Total cycles
Baseline	32553	11873	244	44670
Call between two SMs via ISMC with Secure Boot	83424	15047	949	99420
Call between two SMs via ISMC without Secure Boot	86400	92665	973	180038
Baseline	48456	27090	484	76030
Cascade call with 3 SMs via ISMC with Secure Boot	115509	47838	2160	165507
Cascade call with 3 SMs via ISMC without Secure Boot	116439	159585	2180	278204
Baseline	54594	50441	1048	106083
Scenario 2 via ISMC with Secure Boot	132249	98370	4746	235365
Scenario 2 via ISMC without Secure Boot	132807	245758	4786	383351

The results of these benchmarks measured in CPU cycles can be seen in Table 1 and the added overhead of the versions of Sancus with ISMC compared to the Baseline can be seen in Table 2. The measurements are split into 4 columns. The first column shows the number of cycles required for initialization, the computation of the keys for each SM and enabling the protection. The second column shows the number of cycles required for completing the benchmark for the first run, when the SMs use the expensive cryptographic instruction *MAC-verify*. The third column displays the number of cycles required for the benchmarks to complete when *MAC-verify* has already been used for attesting the callers and now the much less expensive instruction *get-id* is used. Because the number of cycles needed for the completion of a benchmark can be calculated deterministically, there is no need for more than two runs. Finally, the last column showcases the total number of cycles, the sum of all the other three columns.

As it can be seen, the addition of the ISMC adds a significant amount of cycles in all columns, compared to the version without ISMC. The initialization adds a large overhead due to the fact that ISMC is significantly larger than the other SMs which are just dummy SMs.

For the first run, the implementation in which ISMC is loaded securely takes longer than the baseline. This is because, as mentioned in Section 3.2, each SM has to check that is only called by ISMC and calls return only from ISMC. These extra checks increase the amount of code of each SM and thus amount of time required to compute the MACs of SMs. Another reason is that each inter-SM call is done through the ISMC, adding additional calls, therefore more cycles. Regarding the implementation that is assumed to be loaded by untrusted software, the amount of cycles is significantly larger than the other ISMC with secure loading. Because ISMC is not loaded by trusted software, each SM has to attest ISMC with the help of *MAC-verify*. Also, SMs need to attest their intended callee before the actual call via ISMC. These verifications require a significant amount of cycles and are the main reason why the overhead is much larger than the other two implementations.

The second run does not use any cryptographic operations, since all the attestations have been done in the first run. Because of this the amount of cycles in which each benchmark is completed is very small. The overhead seen in the second run is due to the usage of the shadow call stack. The overhead in this case appears to be large also because the other SMs, besides ISMC, are just dummy SMs without any other purpose other than calling a specific SM as explained earlier.

Table 2 Overhead (in CPU cycles) added by the versions of Sancus with ISMC compared to the Baseline

Benchmark	Init. overhead	First run overhead	Second run overhead	Total overhead
Call between two SMs via ISMC with Secure Boot	50871	3174	705	54750
Call between two SMs via ISMC without Secure Boot	53847	80792	729	135368
Cascade call with 3 SMs via ISMC with Secure Boot	67053	20748	1676	89477
Cascade call with 3 SMs via ISMC without Secure Boot	67983	132495	1696	202174
Scenario 2 via ISMC with Secure Boot	77655	47929	3698	129282
Scenario 2 via ISMC without Secure Boot	78213	195317	3738	277268

5. Discussion and future work

As stated previously, Sancus has powerful security guarantees with only a hardware TCB [9]. However, there are also drawbacks due to this design choice. First, the TCB cannot be upgraded. Upgradeability is desired if, for example, a bug is found in the TCB or the IP decides to change the cryptographic implementation. Another drawback of Sancus, as stated in [4] is that depending on the application requirements, it may not be suitable for real-time applications. This is because, in the current implementation of Sancus, the cryptographic operations

cannot be interrupted. If an application has stricter time requirements than the time needed to execute a cryptographic operation, then the application is not be able to meet the requirements.

A possible solution for these problems would be to extend the security kernel presented in Section 3 and add it to the TCB. This kernel would contain a software implementation for the cryptographic primitives and the ISMC that would protect against call flow attacks.

The design choice of moving the cryptographic module from hardware to software offers more flexibility. Because it is a software implementation, it is easier to change the cryptographic algorithm based on new implementations, or based on the purpose for which the platform is used. The software implementation combined with the modifications made in [24] allows for interruptability, making the platform compatible for true real-time application.

Because the attacker can manipulate any code on the platform, a secure boot is necessary to guarantee that the security kernel is loaded correctly. More precisely, it is assumed that the security kernel is loaded first at startup and it can be detected if it was tampered before being loaded. This can be done using a secure boot similar to the one presented in TyTAN.[4] After it has been correctly loaded, the hardware primitives combined with the security kernel should be sufficient to maintain the same security guarantees as the original version of Saancus.

The choice of having a software implementation, however, may add computational overhead, but this will be studied in the future, after adding the cryptographic module to the kernel.

6. Conclusions

This paper proposes an extension of the TCB of Sancus with a small security kernel that provides *call flow integrity*. However, this kernel adds significant overhead. In the future, the security kernel will be extended to contain a software cryptographic module that replaces the hardware implementation.

References

[1] P. Friess, *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

[2] S. Li, L. Da Xu, and S. Zhao, "The internet of things: a survey," *Inf. Syst. Front.*, vol. 17, no. 2, pp. 243–259, 2015.

[3] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite," *Proc. Ninth Eur. Conf. Comput. Syst. - EuroSys '14*, pp. 1–14, 2014.

[4] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," *Proc. 52nd Annu. Des. Autom. Conf. - DAC '15*, pp. 1–6, 2015.

[5] T. Giannetsos, "Self-Propagating Worms in Wireless Sensor Networks ACM CoNEXT - Student Workshop," pp. 31–32, 2009.

[6] A. Francillon and C. Castelluccia, "Code Injection Attacks on Harvard-Architecture Devices," *15th ACM Conf. Comput. Commun. Secur.*, pp. 15–26, 2008.

[7] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," *Blackhat USA*, vol. 2015, pp. 1–91, 2015.

[8] Proofpoint, "Proofpoint Uncovers Internet of Things (IoT) Cyberattack," 2014.

[9] J. Noorman *et al.*, "Sancus : Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base," *22nd USENIX Secur.*, 2013.

[10] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," *Proc. - IEEE Symp. Secur. Priv.*, vol. 2004, pp. 272–282, 2004.

[11] K. El Defrawy, D. Perito, and G. Tsudik, "SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust."

[12] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," *Proc. 1997 IEEE Symp. Secur. Priv.*, p. 65--, 1997.

[13] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," *Proc. - IEEE Symp. Secur. Priv.*, pp. 414–429, 2010.

[14] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture.," in *USENIX Security Symposium*, 2004, vol. 13, pp. 223–238.

[15] D. Grawrock, *Dynamics of a Trusted Platform: A Building Block Approach*, 1st ed. Intel Press, 2009.

[16] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation," *IEEE Trans. Comput.*, pp. 1–1, 2017.

[17] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," *Proc. 2nd Int. Work. Hardw. Archit. Support Secur. Priv. - HASP '13*, pp. 1–1, 2013.

[18] R. Strackx, P. Agten, N. Avonds, and F. Piessens, "Salus: Kernel support for secure process compartments," *EAI Endorsed Trans. Secur. Saf.*, vol. 15, no. 3, 2015.

[19] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," *Proceedings of the 2012 ACM conference on Computer and communications security. ACM*, pp. 2–13, 2012.

[20] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, F. Piessens, and K. U. Leuven, "Protected software module architectures," *ISSE 2013 Securing Electronic Business Processes. Springer Fachmedien Wiesbaden*, pp. 241–251, 2013.

[21] R. Strackx, F. Piessens, and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," pp. 344–361, 2010.

[22] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," C*omputer Security Foundations Symposium (CSF), 2012 IEEE 25th.*, pp. 171–185, 2012.

[23] D. Dolev, "On the Security of Public Key Protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[24] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, "Secure interrupts on low-end microcontrollers," *Proc. Int. Conf. Appl. Syst. Archit. Process.*, pp. 147–152, 2014.

Appendix B

Paper Second Semester

Protected Module Architectures

Alexandru - Madalin Ghenea

In recent years there has been a growing trend Internet of Things (IoT), where more and more devices around us are connected to the internet [23, 10]. A significant part of these devices are used in safety-critical and security-sensitive applications, such as automotive applications, industrial control systems, critical infrastructure, medical applications and e-payment systems [12, 4]. By adding network connectivity, the platforms are susceptible to more attacks from remote attackers. Studies have shown varied threats and vulnerabilities, such as using code injection to build self-propagating worms [8, 7], remote car hijacking [13], infecting network connected appliances with botnets [17].

While for high-end systems, connectivity and software extensibility problems are well studied, for resource-constrained embedded systems, finding efficient security solutions is still a very active area of research [12, 4, 14, 19, 5, 1]. For high-end systems, there are many security solutions, depending on the requirements. For example, privilege levels [3, 16], support for virtual memory, secure co-processors [18], firmware security services, verifiable execution of measured code [9]. These solutions require extra hardware, which in turn increases the costs of the system. Because of the increased costs, those solutions are not available for low cost embedded systems. In recent years a new concept called Protected Module Architectures (PMAs) has emerged, that provide strong isolation. PMAs are suitable for both high-end devices, providing extra security in combination with the solutions presented earlier and for low-end devices, providing strong isolation without the need of more expensive mechanisms. In PMAs the security-critical components such as applications are separated into smaller protected modules, which are isolated by hardware or a small software layer from all other software on the system such that they cannot be tampered with. The separation allows for the security critical modules to be rigorously verified for bugs and vulnerabilities.

Most devices have multiple software programs deployed from mutually distrusting software providers. Because the software programs deployed are mutually distrusting, there is a need for an isolation mechanism, such that software programs cannot influence each other in undesired ways. The conventional software isolation mechanism relies on the Operating System to allocate virtual address spaces for each software program. The main drawback of this approach is that the security and integrity of the programs rely on the kernel of the Operating System, which can be very large and hard to check for vulnerabilities. That is why instead of trying to guarantee the security of large components,

another approach is to minimize the security critical components.

Trusted Computing Base (TCB) refers to the hardware and software components of the security architecture that provide the security guarantees. Exploiting components outside the TCB cannot affect the expected behavior of the TCB. Because the TCB is so critical to the security and integrity of the whole platform it is important for it to be as small as possible to allow exhaustive checks for bugs and other vulnerabilities.

PMAs are security architectures that provide strong isolation for software programs without the need of an operating system. PMAs usually also provide other security features such as local and/or remote attestation, sealing and data and/or code confidentiality [20]. While PMAs can be used for both high end systems in order to strengthen security even more and low end systems where PMAs can provide strong security guarantees without the need for more expensive mechanisms, the focus in this paper will be on PMAs that are targeted for low end embedded systems. Many of the PMAs designed for embedded systems use Program Counter Based Memory Access Control (PCBMAC) to enforce software isolation [21, 2]. The PCBMAC is a mechanism that enforces memory access control rules that can take the program counter into consideration. While PCBMAC has a lower hardware cost than virtual memory and does not influence the critical path [14], it can guarantee even stronger isolation between software, thus making it suitable for low-end embedded systems [14, 5].

As previously mentioned there is a large spectrum of PMA implementations, from complete hardware implementation on the one end to complete software implementations on the other. Hardware implementations can withstand attackers that can manipulate any deployed software, but cannot be upgraded or modified afterwards, thus being less flexible than software implementations which can provide upgradability, but require mechanisms to protect TCB code from attackers. The remainder of this paper will revolve around three implementations: Sancus, TrustLite and TyTAN. Sancus relies only on a hardware TCB and is on the hardware end of the PMA implementation spectrum. TrustLite has both hardware and software in its TCB, positioning it somewhere in the middle of the PMA implementation spectrum. TyTAN is an extension of TrustLite and has a significantly larger software TCB than TrustLite, situating it more towards the software end of the PMA implementation spectrum. First an attacker model that is common for these PMAs will be described. Afterwards, several security properties that can be offered by these PMAs are presented. Finally, each of the three architectures are discussed in terms of design choices, architecture and properties offered.

1 Sancus

Sancus is a Protected Module Architecture tailored for low-end networked embedded systems that provides software isolation, local attestation, remote attestation and confidential deployment with only a hardware TCB [15]. Also, a special C compiler has been developed that compiles C modules to Sancus

SMs.[15]

Sancus is based around a setting in which a single infrastructure provider, IP, owns and administers a set of microprocessor-based systems that referred to as nodes N_i. These nodes are utilized by mutually distrusting third-party software providers SP_j, which develop and deploy software modules $SM_{j,k}$ on the nodes [15].

Each SM has an layout which consists of the start and end of the addresses of the text and data sections of that SM. Each SM also has an identity, which consists of the layout and a hash based on the content of the text section. Adding layout to the identity allows for multiple instances of the same SM on the same platform, as the layout and thus the identity would be different for each instance [15]. Also, the layout is used to easily verify that an SM that is being loaded does not overlap with other SMs.

Next, the main guarantees offered are discussed in terms of how they are implemented.

1.1 Software Module Isolation

The SM isolation is enforced by a hardware Program Counter Based Memory Access Control or PCBMAC. More specifically, the PCBMAC ensures that the protected data section is only accessible when the program counter is in the corresponding code section of the same module [15]. Also, each SM can only be accessed through a single entry point, making it impossible for attackers to misuse specific code chunks. These access control rules can be seen in Figure 1. After an SM is loaded by an untrusted loader, its protection can be activated using a hardware instruction, called *protect*. This instruction verifies that the layout of the SM being protected does not overlap with other SMs already deployed and then saves the layout into a protected storage that is not accessible from software. Afterwards, it enables the access rules as presented earlier and a key based on the Node, SP and SM is created and stored in protected storage. This key is used for attestations and authentications and can be generated by the Software provider as well. If the code has been tampered with, the key generated would not match and the attestations/authentications would fail. The instruction also assigns a sequential ID to the protected SM. This ID is guaranteed to be unique within a boot cycle and it's not reused even after the module is unloaded. Note that the ID is different from the identity of the SM. Once the protect instruction has finished, the code of the protected SM cannot be modified anymore [15]. The memory layout of a node with a software module loaded can be seen in Figure 2.

To disable protection, the *unprotect* instruction is used. This instruction can be called only by an SM on itself. Note that the instruction clears the code and data section in order to prevent leaking confidential data [15].

Furthermore, for ensuring strong isolation, each module has its own call-stack to prevent leaking of stack allocated variables. This is done with the help of the compiler, for each SM a stack is created in the protected memory. Whenever a

From/to	Entry	Text	Protected	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/ Other SM	r-x	r--	---	rwx

Figure 1: "Memory access control rules enforced by Sancus using the traditional Unix notation. Each entry indicates how code executing in the "from" section may access the "to" section." [15]

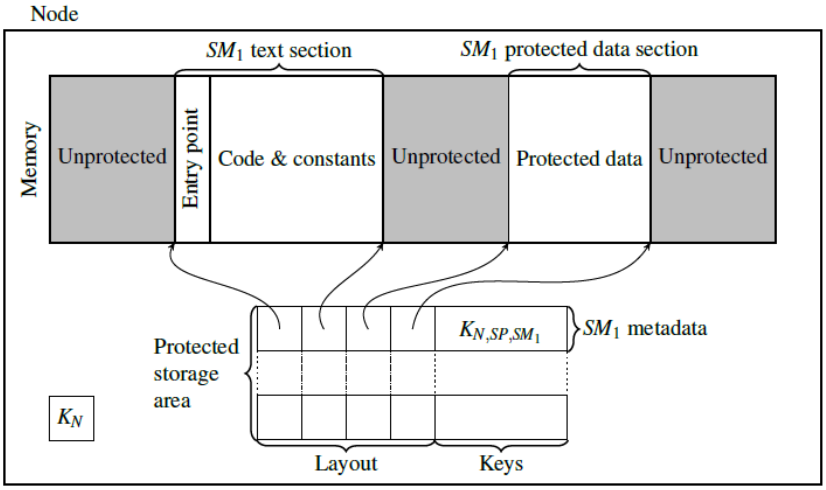

Figure 2: "A node with a software module loaded. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions." [15]

call outside the SM is made, the stack pointer is saved in the protected memory and after the return from the call the stack pointer is restored [15].

Sancus can also provide confidential loading. This is done by using the *protect* instruction with the following signature:

protect layout, SP, MAC

This variant of *protect* behaves similarly to the original *protect*. The only difference is that this protect, decrypts the text section of the module being loaded using a key derivated from the Node and SP. If the Message Authentication Code (MAC) obtained after decrypting is the same as the MAC given as parameter, the *protect* instruction continues with the computation of the SM key. Otherwise the instruction fails, clearing the text section and disabling the protection [15].

1.2 Remote Attestation and Secure Communication

As stated previously Sancus can provide remote attestation and secure communication. This is done using the following instructions:

encrypt plaintext, associated data, ciphertext (output), tag (output) [,key]

decrypt ciphertext, associated data, tag, plaintext (output),[,key]

This *encrypt* instruction takes as input plaintext and associated data, and based on the input and a key, computes a ciphertext and a tag. The tag is a Message Authentication Code (MAC) corresponding to both the plaintext and the ciphertext [15]. The parameter key used for the *encrypt* and *decrypt* instructions is optional. If no key is given, the SM key generated after enabling protection is used. Using these two instructions an SM can securely communicate with Software Provider (SP). Whenever an SM would want to send a message to the SP, it would first encrypt the data using the *encrypt* instruction. Then, the SM would send the ciphertext and the corresponding tag to SP. Using a ciphertext instead of plaintext provides confidentiality between the SM and SP. The SP, would then decrypt the ciphertext and verify the authenticity and integrity of the ciphertext using the associated tag. To satisfy remote attestation, a freshness guarantee must be added which can be in the form of a nonce [15].

1.3 Local attestation

Sancus provides both callee and caller authentication. The solution for callee authentication is explained first. In order for an SM1 to verify that an SM2 has not been tampered with, the following instruction is used:

attest address, expected hash

This instruction checks that a module is loaded and protected at the provided address, computes the hash based on the text section and the layout, compares it with the *expected hash* parameter and if the hashes match it returns the ID of SM2, otherwise it returns 0. In order to use this instruction for local attestation, SM1 must be deployed with the expected hash of SM2. Because computing the hash is an expensive operation, after identifying SM2, the instruction *get-id*

address can be used for further attestations, which returns the ID of the module located at the address given as parameter. Indeed, this is sufficient because as long as SM2 protected, it cannot be tampered with. It is important to note that *get-id* is significantly cheaper than *attest* and does not use any cryptographic primitives [15]. It is important to note that the ID of an SM is different from its identity.

Caller authentication can be obtained in different ways, depending on the application requirements with the help of the *attest-caller* and *get-caller-id* instructions. Both instructions have similar behaviors as *attest* and *get-id*, but using the caller SM as parameter implicitly. One solution would be by first using *attest-caller* to identify them. This is needed only once. Afterwards, the *get-caller-id* instruction is used which returns the ID of the previous running SM. This ID is compared to the IDs obtained from calling *attest-caller*, thus identifying the caller. Again, *get-caller-id* does not use the cryptographic module and is much faster than *attest-caller*. Another less expensive solution, similar to the client-server model would be for the callee to directly use *get-caller-id* to get the id of the caller, without checking its integrity. This can be useful in scenarios in which the callee is not interested in integrity of the caller, but the value of the data returned by the callee is affected by the previous calls of the caller to the callee.

2 Related work

2.1 TrustLite

TrustLite is security architecture that enforces isolation of software modules and provides local attestation with a small hardware and software TCB. Because TrustLite has a smaller hardware TCB it also has smaller hardware extension cost compared to Sancus.

In TrustLite, the code, associated data and meta-data is known as a *program*. A task describes the runtime state of a program, including its CPU state, call stack and other volatile data. Tasks that are designed to implement a particular security mechanism are called *trusted tasks* or *trustlets*. [12]

Software module isolation Isolation is ensured using an Execution-Aware Memory Protection Unit (EA-MPU). The EA-MPU organizes the physical memory into regions, each with associated access control rules. These rules are kept in local registers only accessible to the MPU. The regions can be of two types: code regions or data regions. EA-MPU offers the means to link code regions to data regions. This means that both the instruction address of the executing instruction and the address of the data are taken validated. This is similar to the PCBMAC, in both cases data is accessible based on the address of the instruction that is being executed. The EA-MPU scheme can be seen in Figure 3 and an example of the EA-MPU control rules can be seen in Figure 4 . The access rules are more fine-grained in TrustLite than in Sancus. More specifically,

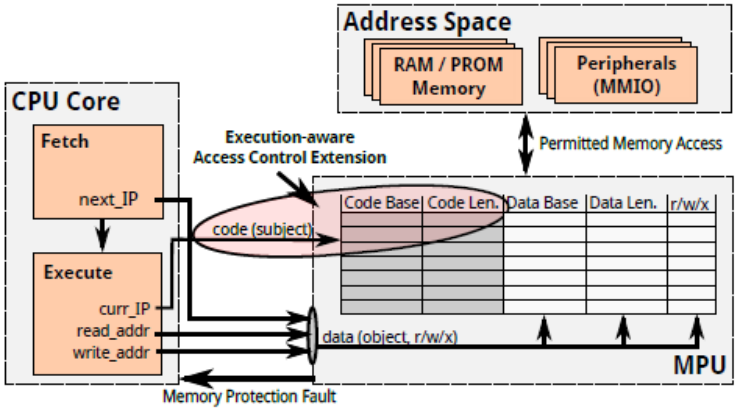

Figure 3: "Block diagram of EA-MPU" [12]

in TrustLite, the EA-MPU can allow for several code regions to have access to a protected data region, while in Sancus, without modifications, a protected data region can only be accessed by the corresponding protected code region (giving exclusive control to it). In [22] a solution for Sancus that grants shared access to data is presented.

The initialization of the EA-MPU and the loading of programs, including trustlets, is done at startup by a Secure Loader. The Secure Loader first clears out the MPU registers, then parses the meta-data of the trustlets present in PROM and allocates memory regions for each of them and initializes each trustlet. Afterwards, the Secure Loader configures the MPU according to the access control rules required by the loaded trustlets. Finally, the Secure Loader can optionally load an untrusted OS.[12] A schematic of the steps that the Secure Loader makes at startup can be seen in Figure 5.

The EA-MPU registers cannot be modified after completing the configuration step. This implies, trustlets cannot be loaded or unloaded dynamically at runtime, which makes TrustLite less flexible than Sancus which can load/unload SMs at runtime.

TrustLite, also provides an Exception Engine that preserves memory isolation even when software or hardware exceptions are used. Whenever an exception is triggered, the stack and instruction pointer and the CPU registers are saved and then the CPU registers are cleared before jumping to the exception handler. This provides protection against leaking data to the exception handler. When returning from an exception handler, the first thing done should be to restore the stack pointer, otherwise if another exception is triggered, the CPU may be stored to the wrong stack. Finally, similar to Sancus to protect against code reuse, each trustlet can only be accessed through entry vectors.

		Object \ Subject		MPU Access Control Rules		
				TL-A (0x00-0A)	TL-B (0x0A-0B)	OS (0x0B-0F)
PROM / Flash	0x00..	Trustlet A	entry	rx	rx	rx
			code	rx	r	r
	0x0A..	Trustlet B	entry	rx	rx	rx
			code	r	rx	r
	0x0B..	OS	entry	rx	rx	rx
			code	r	r	rx
SRAM / DRAM	0x10..	Trustlet A	data	rw	-	-
			stack	rw	-	-
	0x1A..	Trustlet B	data	-	rw	-
			stack	-	rw	-
	0x1B..	OS	data	-	-	rw
			stack	-	-	rw
Peripherals	0x20..	MPU	flags	r	r	r
			regions	r	r	r
	0x2A..	Timer	period	r	r	rw
			handler(ISR)	r	r	rw
	0x2B..	...				

Figure 4: "Example of MPU access control rules in a scenario with 2 trustles and an OS" [12]

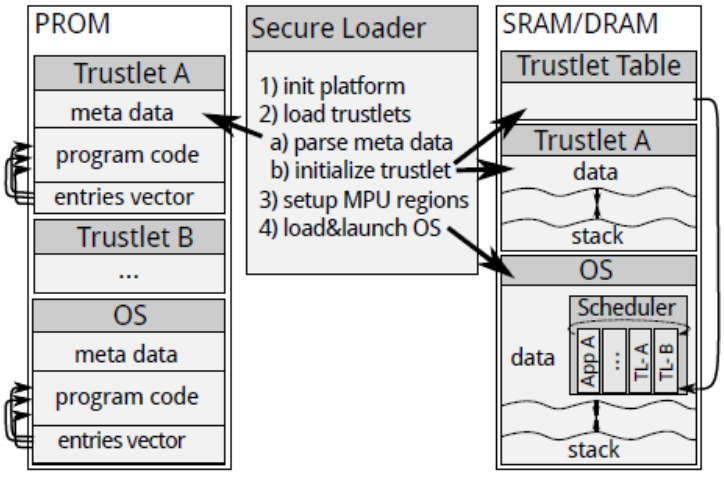

Figure 5: "Boostrapping Trustlets and OS from PROM" [12]

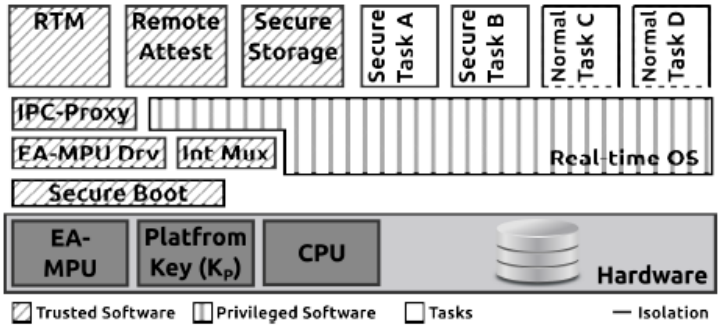

Figure 6: "TyTAN architecture" [4]

Local attestation TrustLite provides callee authentication. A trustlet T1 can check that a trustlet T2 has been correctly loaded by verifying that the MPU registers are configured correctly and can compute a hash of the code to verify if T2 has been tampered with before loading. The Secure Loader can also be modified to do these measurements for all the trustlets and provide this data to trustlets.

TrustLite does not provide caller authentication. T2 cannot reliably identify that it has been called by T1. If T2 could verify that it has been called by T1, then T2 can also check the integrity of T1 in a similar fashion to the callee authentication.

2.2 TyTAN

TyTAN is an extension of TrustLite that provides strong isolation, local and remote attestation, dynamic software loading at runtime and real-time guarantees. The hardware TCB of TyTAN is smaller than that of TrustLite, while the software TCB of TyTAN is significantly larger than that of TrustLite. The architecture of TyTAN can be seen in Figure 6

TyTAN is based around a similar setting to Sancus, in which there is a device manufacturer M, a device owner O, and multiple task providers Pi that provide programs called tasks. The tasks deployed on the device are mutually distrusted.

Isolation Isolation is enforced again using the EA-MPU. The EA-MPU is initialized by a Secure boot which loads all the other tasks that are part of the TCB, such as the EA-MPU driver. The EA-MPU driver controls the EA-MPU configuration at runtime, allowing for dynamic loading and unloading of tasks at runtime. This design choice makes TyTAN more flexible than TrustLite, which does not provide dynamic loading of programs.

Remote attestation The Root of Trust for Measurement (RTM) task is responsible for local and remote attestation. RTM computes a hash based on the code of for each task loaded and the hash is then used as the identity for the task and can only be modified by RTM.

Remote attestation is obtained using MACs, in a similar fashion to Sancus. The MAC is computed by the Remote attest task, using an attestation key derived from a platform key and the identity of the task. The obtained MAC along with the identity of the task can be sent to a remote party to attest the task. The platform key is accessible only to several tasks that are part of the TCB and the attestation key is accessible only to the Remote Attest task.

Because RTM and Remote attest are tasks, they can be interrupted, including during the computation of the hash making TyTAN compatible with real-time applications. During the computation of the id, the corresponding task cannot be unloaded.

Local attestation As mentioned previously, the RTM task is also responsible for local attestation. The identity can be used to attest the integrity of the task to other tasks.

TyTAN provides also caller and callee authentication. Communication between tasks is done via the Inter-Process Communication (IPC) Proxy task. Whenever a task T1 wants to call a task T2, it calls IPC Proxy and sends the message and the identity of T2 using the CPU registers. The IPC searches for the location T2, which can be seen as callee authentication and writes the message and the identity of T1 into T2's memory. EA-MPU ensures that only the IPC Proxy can write in T2's memory. Because of this the presence of the id of T1 in T2's memory can be seen as caller authentication.

2.3 SMART

SMART or **S**ecure and **M**inimal **A**rchitecture for (Establishing a Dynamic) **R**oot of **T**rust is the first embedded implementation of a Protected Module Architecture [5]. It is based on the Self Protecting Module (SPM) architecture described in [21]. SMART provides remote attestation, remote authentication and trusted code execution using a small TCB and minimal hardware changes [5]. It is developed on two low-end embedded microcontrollers, Texas Instruments MSP 430 and Atmel AVR.

Compared to newer PMAs, such as Sancus, TrustLite and TyTAN implementation SMART provides less security guarantees. More specifically, as it will be seen later in this subsection, SMART provides only partial isolation and does not provide local attestation.

The design of SMART is based on four main components [5]. First, the attestation code is deployed on read-only memory (ROM). Second, the key K used for attestation is stored in a memory region inside the CPU and can only be accessed from the attestation code present in ROM. Third, the microcontroller ensures that the key K can only be accessed from the attestation code.

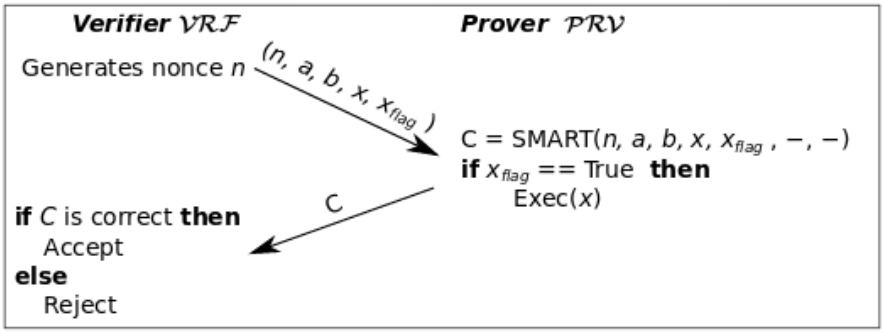

Figure 7: "SMART challenge-based protocol" [5]

SMART provides guaranteed execution of code on the SMART platform, which is called *Prover (PRV)*, to an external party called *Verifier (VRF)*, by using a challenge-based protocol. The protocol is initiated by the *Verifier*, which sends a message to the *Prover*. This message contains a nonce n, the address range that needs to be attested, which is described by the addresses a and b, an address x which specifies where the *Prover* could jump to after completing the attestation and a flag x_{flag} , that specifies whether to jump to the address described by x after attestation. After receiving the message, the *Prover*, first disables interrupts and then executes the *SMART* procedure, which computes a MAC on the memory region describe by addresses a and b using a pre-shared key between the *Prover* and the *Verifier*. It is important to note that the SMART procedure is stored in ROM memory and that the pre-shared key can only be accessed from within the SMART procedure. After computing the MAC, the interrupts are enabled and if the x_{flag} is true, the *Prover* executes the code starting from address x. Finally, after finishing the SMART procedure and the potential execution, the *Prover* sends the cryptographic checksum (the MAC) to the *Verifier*, which can check whether or not it is correct [5]. The protocol is described in Figure 7.

The interrupts are disabled for the duration of the checksum computation in order to guarantee atomic execution of the attestation code. In order to protect against code reuse attacks, the program counter can only jump at the SMART initial address and can only leave at the last SMART address. Any attempt to do otherwise would force an immediate hardware reset. The hardware access control logic that enforces the correct execution of SMART is also responsible for enforcing the access to the pre-shared key from inside the SMART procedure. Furthermore, in order to ensure that no data is leaked due to an incomplete SMART execution or even a power loss, memory cleanup is performed by the processor after each reset [5]. Also, the SMART procedure is not dependent on the pre-shared key, thus not leaking any information about it. Finally mentioned earlier, the SMART procedure is stored in ROM making it impossible for attackers to tamper with its content.

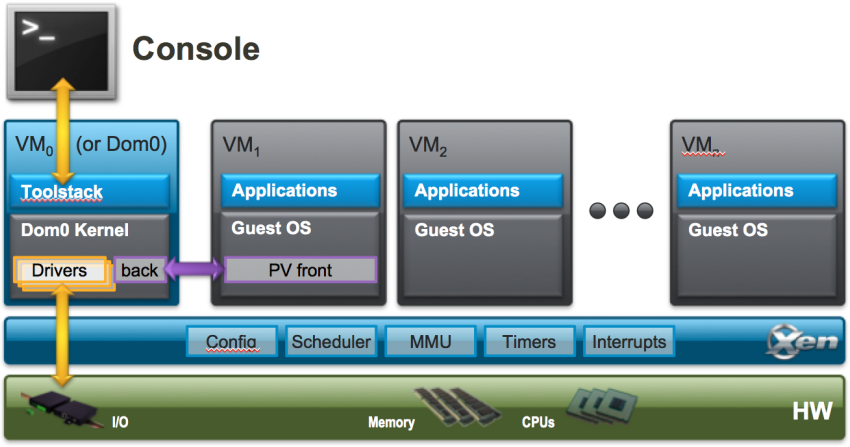

Figure 8: "XEN hypervisor Architecture" [6]

2.4 XEBRA

XEBRA is another PMA implementation based on SMART and the XEN hypervisor. XEBRA is a software solution that provides the same services as SMART, namely: remote attestation, remote authentication and trusted code execution [1]. Next, the XEN hypervisor is described briefly in order to understand XEBRA better and then XEBRA is explained in detail.

XEN hypervisor XEN is a light, bare-metal hypervisor that allows for multiple virtual machines (domains) to run on the same physical machine. XEN is small, compared to other hypervisors, containing less than 150,000 lines of code. XEN runs in a more privileged CPU state than any other software on the machine and is responsible for the CPU scheduling, memory management, interrupts and for launching the first virtual machine, which is called the Control Domain or Domain 0. However, XEN has no knowledge of any I/O such as storage and networking. The I/O is managed by the Control Domain, which has the capability to access the I/O functions. The Control Domain can launch and interact with other Virtual Machines and provides a control interface. Without the Control Domain, the hypervisor cannot be used [6]. The XEN architecture can be seen in Figure 8.

XEBRA implementation XEBRA uses the Xen hypervisor to enforce a similar protocol as SMART in order to provide remote attestation, remote authentication and trusted code execution. More specifically, it requires at least 2 Virtual Machines (domains): the Control Domain and an Application Domain. The Control Domain is used to store the attestation code. On top of this, the Xen hypervisor is used to enforce isolation between the Control Domain and the

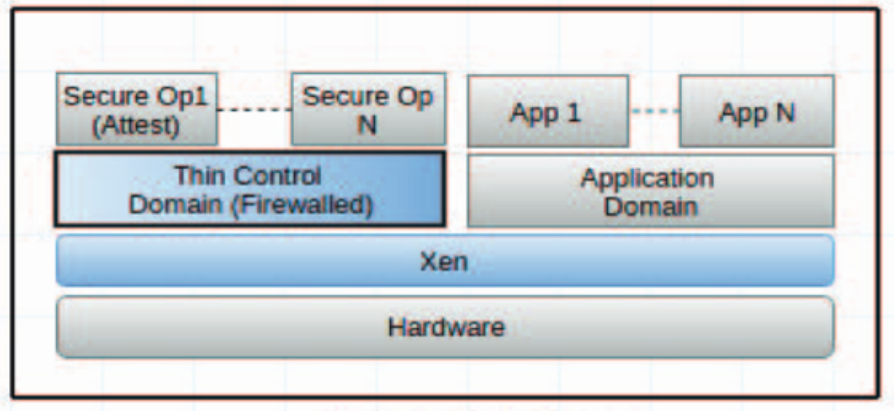

Figure 9: "XEBRA Device Structure" [1]

Application Domain enforcing, by extension, isolation between the attestation code which is stored in the Control Domain and the Application Domain. The isolation between the attestation code and applications is a key component of both SMART and XEBRA, even though the way it is enforced is very different. Xen also acts as a secure loader for the Control Domain, ensuring that it is loaded correctly. [1]

In order to minimize the attack surface, the Control Domain is firewalled and is can only communicate with the remote Verifier and the Application Domain. Because the Control Domain is the only domain that has direct access to the network, it shares network connectivity with the Application Domain through a bridged connection, allowing the Application Domain to communicate with the outside world. [1]

XEBRA device structure can be seen in Figure 9. As it can be seen, XEBRA could be extended with other secure operations besides the attestation code that are isolated from the Application Domain. The communication model is similar to SMART and can be observed in Figure 10. The communication model works as follows: Whenever the remote Verifier wants to attest a memory region it sends a message to the Application Domain. The message contains x and$y$, which represent the start and end address of the memory region which is requested to be attested, a *nonce* which is used to protect against replay attacks, and finally, the exe_{flag} , which if set to true executes the code starting from address x. The message is encrypted with a pre-shared symmetric key, shared between the Verifier and the Application Domain. Encrypting the message with this key ensures the integrity and origin of the message. After receiving the message, the Application Domain sends another message message to the Control Domain. This message forwards all the parameters received from the Verifier and also sends the actual contents of the memory region between addresses x and y. Next, the Control Domain computes a checksum based on the contents of

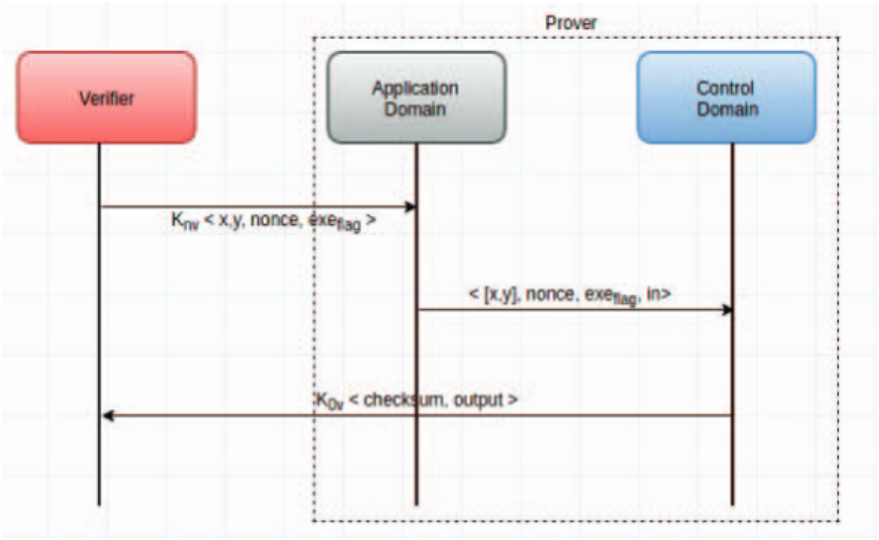

Figure 10: "XEBRA Communication Model" [1]

the desired memory region. After computing the checksum, the Control Domain executes the code if the exe_{flag} is set and finishing the potential execution, it sends the checksum and the output resulted from the code execution to the Verifier through a message. This final message is encrypted with another key, that is pre-shared between the Control Domain and the Verifier, in order to provide message integrity and proof of origin.

Even though XEBRA requires more advanced features such as CPU privilege levels, the implementation is light enough to run on higher-end embedded systems such as Intel Galileo. Another advantage of XEBRA is that being a complete software solution, it can be deployed on legacy systems. However, the TCB is very large, Xen having less than 150,000 LOC [6]. On top of that, the Control Domain implemented by a Linux distribution such as Alpine Linux has a TCB of over 20,000,000 LOC [11]. Because XEBRA has such a large TCB it is very difficult to thoroughly check it for bugs and vulnerabilities.

3 Discussion

Next, the trade-offs of the PMAs presented are discussed. The PMAs reviewed in this Chapter are examples of the large spectrum of possible implementations, ranging from complete hardware implementations such as Sancus [15], to implementations that rely on both hardware and software components such as SMART [5], TrustLite [12] and TyTAN [4] and finally to complete software implementations such as XEBRA [1].

A comparison between the different implementations presented in this chap-

	XEBRA	TyTAN	TrustLite	SMART	Sancus
Local Attestation	No	Yes	Yes	No	Yes
Remote Attestation	Yes	Yes	No	Yes	Yes
Modifiable Security Primitives	Yes	Yes	Yes	No	No
Software Module Isolation	Partial	Yes	Yes	Partial	Yes
Interruptible applications	No	Yes	Yes	No	No
Dynamic Software Loading	Yes	Yes	No	Yes	Yes

Figure 11: Comparison based on the features offered by different Protected Module Architectures

ter can be observed in Figure 11. Next, each of the features compared in Figure 11 are discussed.

3.1 Local attestation

Many embedded systems are deployed with multiple mutually distrusting applications. Because of this, local attestation solutions are needed in order for an application to detect whether or not other applications deployed on the same device have been tampered with. TrustLite provides only callee authentication, meaning that an application can attest another application. This is done by verifying that Memory Protection Unit registers that correspond to the application being attested are correctly configured and then computing a hash based on the content of the application that is being attested. TrustLite does not provide caller authentication, meaning that an when an application is called it cannot identify its caller. TyTAN on the other hand provides both caller and callee authentication through the use of the RTM task combined with the IPC proxy task.

Sancus provides both callee and caller authentication. callee authentication is provided through the use of the *attest* primitive, which computes a hash based on the layout and contents of an SM and compares the hash with an expected one. If the attestation is successful, the id of the attested SM is returned. For caller attestation, the *attest-caller* primitive is used which has a similar behavior with *attest*. For subsequent attestations it is sufficient to check that the id of the SM that is being attested has not changed. This is done with the use of the *get-id* and *get-caller-id* primitives.

3.2 Remote attestation

With the exception of TrustLite which does not guarantee remote attestation all other 4 PMAs presented guarantee remote attestation. All 4 implementations

use measurements to provide proof that the integrity of the software to remote parties. Sancus provides remote attestation through the use of *protect* primitive which guarantees that the content of the SM cannot be modified combined with *encrypt* primitive which computes a MAC on the data present in a memory region. TyTAN uses a similar approach as Sancus and uses a secure task called Root of Trust for Measurement (RTM) which computes the measurement for tasks which are protected by the EA-MPU. The main difference between Sancus and TyTAN is the way software protection and measurement computation is implemented. Sancus implements both primitives in hardware while TyTAN provides software isolation through a combination of hardware and software. Also, RTM is software component, while *attest* is implemented in hardware.

SMART and XEBRA rely on a challenge-based protocol which can be seen in Figure 7 for SMART and Figure 10 for XEBRA. These 2 PMAs do not provide software isolation for the application, that is why the additional steps are required. For SMART, the interrupts are disabled during the measurement, to guarantee that the content cannot be tampered with during the measurement. For XEBRA, the actual code that is to be measured is sent to the attestation code on a separate virtual machine, to guarantee that the code is not modified during the measurement.

3.3 Modifiable security primitives

Another interesting feature is the ability of being able to modify the security primitives available without any hardware changes. This feature allows for easily updating primitives if bugs or vulnerabilities are found. Also, it allows for the primitives to be tailored on the application requirements. For example, the measurement implementation used for attestations can be selected based on the requirements of an application. TrustLite, TyTAN and XEBRA are more flexible from this point of view and support this, having most of the security primitives implemented in software. On the other side SMART has the attestation code written in ROM. Depending on the technology used for ROM the attestation code can or cannot be modified. Sancus, being a PMA which has only hardware TCB does not support modifiable security primitives, making it less flexible than other implementations from this point of view.

3.4 Software Module Isolation

Complete software module isolation is guaranteed for Sancus, TrustLite and TyTAN. In Sancus it is enforced in hardware, through the use of the protect primitive, which explained in detail in Section 1. TrustLite and TyTAN both use an hardware memory protection unit which contains memory regions and grants read/write/execution rights based for each SM based on a policy that is configured by a secure initializer at startup. In TrustLite these rights are static, meaning that after they are set they can only be changed at reset by the secure initializer. In TyTAN this is not the case, these rights can be changed at runtime through the use of a memory protection unit driver.

SMART and XEBRA, provide only partial software module isolation. Both in SMART and XEBRA, the attestation code is isolated from the rest of the applications present on the device. However, the applications themselves are not isolated from each other. For SMART, the isolation between the attestation code and the applications is enforced in hardware, as described in Section 2.3. Even though XEBRA draws inspiration from SMART, the way it enforces isolation is completely different. XEBRA uses the Xen hypervisor to provide isolation between Control Domain, where the attestation code is deployed and the Application Domain, where the rest of the applications reside.

3.5 Interruptible applications

Interruptibility is essential for compatibility with real-time applications. Interruptibility refers to the capability of securely stopping an application to execute another application and after finishing the execution returning to the first application and continuing execution. Only TrustLite, TyTAN and XEBRA support interruptible applications. SMART and Sancus, on the other side, do not support interruptible applications. SMART requires interrupts to be disabled while computing the measurement or when executing the measured code. For Sancus, the main limitation in this regard is that the cryptographic operations cannot be interrupted.

3.6 Dynamic Software Loading

Dynamic software loading refers to the capability of loading and unloading software at runtime. With the exception of TrustLite, all other 4 PMAs support this. Sancus, SMART and XEBRA can have dynamic software loading with the help of an untrusted loader. TrustLite is less flexible from this point of view and does not allow dynamic software loading. That is because memory access right are written in the EA-MPU at startup and these rights cannot be changed after boot. TyTAN allows dynamic software loading through the use of an Memory Protection Unit driver which can change EA-MPU configuration at runtime.

4 Conclusion

The need for secure solutions for embedded systems has lead to the concept of Protected Module Architecture. This topic is still extensively researched, that is why there is a wide variety of implementations using different approaches to provide similar security guarantees. In this paper, 5 PMA implementations were discussed. The PMAs presented highlight the solution diversity, ranging from solutions that rely on software to solutions that rely strictly on hardware to provide the security guarantees.

References

[1] N. Agarwal and K. Paul. Xebra: Xen based remote attestation. In *Region 10 Conference (TENCON), 2016 IEEE*, pages 2383–2386. IEEE, 2016.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 171–185. IEEE, 2012.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.

[4] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[5] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.

[6] L. Foundation. Xen project software overview. https://wiki.xen.org/wiki/Xen_Project_Software_Overview.

[7] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[8] T. Giannetsos, T. Dimitriou, and N. R. Prasad. Self-propagating worms in wireless sensor networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies*, pages 31–32. ACM, 2009.

[9] D. Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[10] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[11] S. L. . N. I. A. M. T. L. F. Jonathan Corbet, LWN.net;Greg Kroah-Hartman. Linux kernel development. http://go.linuxfoundation.org/e/6342/el-Development-Report-2016-pdf/3g9jbh/778929651.

[12] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, page 10. ACM, 2014.

[13] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[14] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, pages 479–494, 2013.

[15] J. Noorman, J. VAN BULCK, J. T. MUHLBERG, and F. PIESSENS. Sancus 2.0: A low-cost security architecture for iot devices, 2017.

[16] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 414–429. IEEE, 2010.

[17] Proofpoint. Proofpoint uncovers internet of things (iot) cyberattack. 2014.

[18] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[19] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[20] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[21] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. *Security and Privacy in Communication Networks*, pages 344–361, 2010.

[22] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. Secure resource sharing for embedded protected module architectures. In *IFIP International Conference on Information Security Theory and Practice*, pages 71–87. Springer, 2015.

[23] O. Vermesan and P. Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

Bibliography

[1] N. Agarwal and K. Paul. Xebra: Xen based remote attestation. In *Region 10 Conference (TENCON), 2016 IEEE*, pages 2383–2386. IEEE, 2016.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 171–185. IEEE, 2012.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.

[4] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede. Spongent: A lightweight hash function. *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 312–325, 2011.

[5] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[6] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 147–152. IEEE, 2014.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[8] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.

[9] L. Foundation. Xen project software overview. https://wiki.xen.org/wiki/Xen_Project_Software_Overview.

[10] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[11] T. Giannetsos, T. Dimitriou, and N. R. Prasad. Self-propagating worms in wireless sensor networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies*, pages 31–32. ACM, 2009.

[12] D. Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[13] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[14] T. Instruments. Msp430x1xx family user's guide. http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[15] S. L. . N. I. A. M. T. L. F. Jonathan Corbet, LWN.net;Greg Kroah-Hartman. Linux kernel development. http://go.linuxfoundation.org/e/6342/el-Development-Report-2016-pdf/3g9jbh/778929651.

[16] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, page 10. ACM, 2014.

[17] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[18] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[19] J. T. Mühlberg, S. Cleemput, M. A. Mustafa, J. Van Bulck, B. Preneel, and F. Piessens. An implementation of a high assurance smart meter using protected module architectures. In *IFIP International Conference on Information Security Theory and Practice*, pages 53–69. Springer, 2016.

[20] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, pages 479–494, 2013.

[21] J. Noorman, J. VAN BULCK, J. T. MUHLBERG, and F. PIESSENS. Sancus 2.0: A low-cost security architecture for iot devices, 2017.

[22] N. I. of Standards and T. (NIST). Sha-2 standard. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[23] G. Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[24] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 414–429. IEEE, 2010.

[25] Proofpoint. Proofpoint uncovers internet of things (iot) cyberattack. 2014.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[27] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[28] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel support for secure process compartments. *EAI Endorsed Transactions on Security and Safety*, 15(3), 2015.

[29] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[30] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13. ACM, 2012.

[31] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. *Security and Privacy in Communication Networks*, pages 344–361, 2010.

[32] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. Secure resource sharing for embedded protected module architectures. In *IFIP International Conference on Information Security Theory and Practice*, pages 71–87. Springer, 2015.

[33] O. Vermesan and P. Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

Master thesis filing card

Student: Alexandru - Madalin Ghenea

Title: A Security Kernel for Protected Module Architectures

UDC: 621.3

Abstract:
More and more embedded devices are being connected to a network. Network connectivity adds more attack vectors, making them more susceptible to security threats. While for high-end systems there are many security solutions, most of these solutions rely on features such as virtual memory and CPU privilege levels, which are not available for low-end embedded systems. Because of this, finding security solutions for low-end embedded systems is still an active area of research. Therefor, security solutions are required to make embedded systems more secure. This need has lead to the creation of the concept of Protected Module Architecture (PMA), which can provide strong security guarantees. There are many implementations of PMAs, both for high-end devices and for low-end devices. One PMA implementation which focuses on low-end devices is Sancus [21]. Sancus provides strong security guarantees by relying on a small Trusted Computing Base made only of hardware.

This master's thesis studies the trade-offs and the security properties that can be obtained by adding a security kernel. The security kernel prototype developed for this thesis is composed of two components which try to tackle two main goals. One component is designed to protect against call-stack shortcutting attacks, through the use of a shadow call-stack. Based on the results obtained, the component that protects against call-stack shortcutting adds a computational overhead that should be acceptable for many applications.

The other component is designed to replace the hardware cryptographic module with a software implementation while trying to maintain the same security guarantees. A software implementation would reduce hardware costs, would allow for more flexibility, by being able to change the implementations after deployment and would be a step towards interruptibility since the hardware implementation cannot be interrupted. The prototype developed provides only local attestation under the same conditions as the hardware implementation. Remote attestation, however is restricted by more assumptions. It is left for future work, to provide a solution for remote attestation under the same assumptions as the original version of Sancus. The measurements that were done show that the software implementation adds significant computational overhead and that this overheard depends heavily on the algorithm used for implementing the cryptographic operations.

The developed components can be used either together or independently, depending on the requirements of the application for which the security kernel is used.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, specialisation Secure Software

Thesis supervisor: Prof. dr. ir. Frank Piessens

Assessors: Dr. A. Hovsepyan
Dr. J.T. Mühlberg

Mentors: Dr. J.T. Mühlberg
Ir. J. Van Bulck