KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Extending Protected Module Architectures with a Secure I/O Framework

Dennis Frett

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor:
Prof. dr. ir. F. Piessens

Assessoren:
Prof. dr. L. De Raedt
Prof. dr. B. Crispo

Begeleiders:
Dr. R. Strackx
Ir. F. Mennes, VASCO Data Security

Academiejaar 2014 – 2015

Preface

This thesis is presented in the context of the KU Leuven program 'Master of Engineering: Computer Science, Specialisation: Secure Software'. It is the culmination of my studies to date in computer science, with a particular focus on security.

The completion of this thesis and my studies in general would not have been possible without the support and assistance of many people. First, thanks to my supervisor Professor Frank Piessens for assisting me in the choice of the topic and ongoing support in developing it. Thanks to my instructor Frederik Mennes from VASCO for his valuable and detailed feedback. Thanks also to Job Noorman who provided me with a working prototype of Sancus, which I used to implement secure hardware, and lots of information on working with it. The person who helped me the most during the last year was Raoul Strackx. Our weekly meetings and his continuous assistance contributed significantly to the quality of the end result.

Also, thanks to many of my classmates for helping me with specific details of the C programming language, LaTeX-related questions and general discussions on the security of my implementation. And Eva, it was a pleasure to work on our respective theses together. I really enjoyed sharing this experience with you.

Most of all, I would like to thank my parents for helping me by proofreading my text and providing me with advice. But most importantly: thank you for supporting me and giving me the opportunity to pursue a university education.

Dennis Frett

Contents

Abstract

A computer system that is compromised by a software bug or an attacker could cause applications running on it to behave in unintended ways. For this reason, protected module architectures were developed. They allow software modules, called *protected modules*, to be run isolated from the rest of the system, and offer a third party the possibility to remotely verify the state of a system, a procedure called *remote attestation*. These protected module architectures, however, still have no way to securely interact with system hardware, thus limiting their practical use. This thesis designs and implements a framework allowing protected modules to access hardware in such a way that confidentiality and integrity of the data are ensured. Our results show that it is possible to implement this framework providing strong security guarantees and acceptable performance overhead.

Samenvatting

Een computersysteem dat gecompromitteerd is door een softwarefout of een aanvaller kan zorgen dat de programma's zich op een onverwachte manier gedragen. Beveiligde module-architecturen zijn ontwikkeld om dit tegen te gaan. Ze laten toe dat softwaremodules, *beveiligde modules* genaamd, geïsoleerd van de rest van het systeem uitgevoerd kunnen worden. Ook voorzien ze een manier voor een derde partij om van op afstand de toestand van een systeem te verifiëren. Deze beveiligde module-architecturen voorzien momenteel nog geen methode om op een veilige manier hardware apparaten te gebruiken, wat hun bruikbaarheid beperkt. In deze thesis ontwerpen en implementeren we een systeem dat beveiligde modules toelaat apparaten te gebruiken op een manier die de vertrouwelijkheid en de integriteit van de data verzekert. Onze resultaten tonen dat het mogelijk is om dit systeem te implementeren en sterke veiligheidsgaranties te bieden in combinatie met een aanvaardbare performantiekost.

List of Figures

Chapter 1

Introduction

Today, billions of devices are connected to the internet and more are joining every second [41]. These devices include personal computers, smartphones, servers and embedded devices. The tasks they perform can often be deemed sensitive or private: matters such as handling bank transactions, storing medical records and processing personal information. To guarantee the safety of this data, it is very important that these devices are secure and are not vulnerable to attacks. Most of them run some sort of operating system, developed by teams of a few hundred to a few thousand of people, consisting of millions of lines of code [18][17][19]. Applications and services, such as web browsers (Google Chrome, Mozilla Firefox, Microsoft Internet Explorer), HTTP servers (Apache HTTP Server, Microsoft IIS) and the like — consisting of thousands to millions of lines of code [4][3] — run on top of the operating system.

All the components of a system that need to be trusted in order to guarantee security are called the *trusted computing base* (TCB) [32]. For these systems to be secure, they must be *confidential*, *integer* and *available*: the CIA triad [30]. Online bank transactions, for example, require the following:

- **Confidentiality:** The information on your screen (bank account balance, transaction history) can only be seen by you. Eavesdropping by third parties should not be possible.
- **Integrity:** Information presented to you should be correct. A transaction that is submitted to the bank must not be modifiable by an attacker.
- **Availability:** The online bank system should always be available to reliably handle requests from a bank customer.

In this case, the total TCB consists among others of the TCB of the user's system and the TCB of the bank. This includes the user's operating system, web browser, system drivers, keyboard, screen and processor. In order to meet the security requirements, the entire TCB must be trusted (by definition). In the case of an online bank transaction, this means trusting millions upon millions of lines of code, made by

dozens of different companies. A larger TCB means that the chances of a bug or the introduction of a malicious component becomes much greater.

Software bugs can cause a system to behave in unintended ways. An attacker can try to exploit bugs by, for example, writing (*buffer overflow*) or reading (*buffer overread*) past the end of an allocated memory buffer. This way he could skip certain code from being executed or extract sensitive memory contents. Malicious components can contain so called *backdoors*. A person with knowledge of these backdoors could bypass normal authentication and gain unauthorized access to a system. It is clear that the presence of bugs or backdoors somewhere in the TCB can compromise the security of the whole system. Examples of software bugs with significant consequences in the last years include Heartbleed [24], Shellshock [26] and POODLE [25]. If we want to offer reliable security, we must make sure that as little code and hardware as possible needs to be trusted.

The concept of a protected module architecture (PMA) is a more recent development. PMAs, like Flicker [20], TrustVisor [21] and Fides [38], are able to protect memory segments from the rest of the system. This allows certain processes to run isolated from other processes and even from the operating system. As a result, far less code needs to be trusted, and the TCB is reduced significantly. A current limitation of PMAs is that they do not offer a secure way for these applications to gain access to hardware. Lots of applications rely on hardware to provide them with input and output. Results of a computation, for example, need to be printed to a screen and input needs to be entered on a keyboard.

This thesis will develop a framework that provides any protected module (PM) with a generic way to obtain secure input and output (I/O) from hardware. The current state of the art and its shortcomings are discussed in chapter 2. The problem we want to solve along with the attacker model is outlined in chapter 3. In chapter 4 we create the design for our framework which is implemented in chapter 5. Its security and performance is evaluated in chapter 6. In chapter 7 the implementation of a one-time password generator using this framework is described. We end with a conclusion of the project in chapter 8.

Chapter 2

Background

This chapter describes the current state of the art regarding protected module architectures (section 2.1), state continuity (section 2.2), formal program verification (section 2.3) and an overview of the cryptographic building blocks (section 2.4) used in this project.

Since the operating system has almost complete control over everything that happens on a PC, an attacker that is able compromise the operating system has extensive capabilities: reading, manipulating and executing arbitrary memory locations, detecting all keystrokes and reading the screen buffer. This means that without any additional protection in place, in the presence of kernel level malware, security cannot be guaranteed for any running processes, and all information handled on the operating system must be considered as compromised. Different methods have been proposed to guarantee the security of software on a system: anti-virus software, hardware support for virtual memory, memory-safe virtual machines,... All of these solutions require a large software layer in order to work. The operating system and virtual machine code is added to the trusted computing base. Also, some probabilistic measures that try to limit the damage that exploiting a bug in a system have been proposed. Examples of this include ASLR and the use of stack canaries. It was shown that, although they make it harder for an attacker, exploitation of these bugs is still feasible [40].

2.1 Protected module architectures

PMAs provide secure execution of small protected modules. This means that strong security guarantees can be provided even in the presence of kernel level malware. They also make it possible for a remote third party to collect measurements from the system. This allows the remote party to do a remote *attestation* of the system and verify that it is in a certain state [29][37][44]. Current PMAs isolate a protected module from the rest of the system using a small hypervisor that runs at a higher privilege level than the operating system. Intel is working on Intel Secure

Guard Extensions (SGX) that will support these isolation properties in the system hardware [8][2][11][22].

Different PMAs use different names for the software modules they isolate. Flicker calls them *Pieces of Application Logic* (PALs), in Fides they are called *Self-Protecting Modules* (SPMs), in Microsoft Hyper-V they use *Containers* and in Intel SGX the term *Enclaves* is used. In a very broad sense, these architectures offer similar security guarantees. In this thesis, the more general term *Protected Modules* will be used when referring to a general software module protected by a PMA. The specific terms will be used when explaining a given architecture in more detail.

2.1.1 Existing architectures

Flicker The first implementation of a PMA is Flicker [20], which utilizes hardware support for late launch and attestation in Intel and AMD processors. In a late launch environment, a hypervisor, a virtual machine monitor or any application can be started, at an arbitrary time after the system has booted, in a fresh environment. It is similar to running this process right after a clean system has started. The advantage of a late launch environment is that the system does not need to be rebooted [31]. It also greatly simplifies remote attestation, which will be explained later. Flicker provides isolation of security sensitive code from other software running on the same system, including the operating system. It is able to provide a means to prove to a remote party that code was executed with certain protection properties in place. It also allows for the creation of attestations of exactly what code was run on the system and what its input and output was. These attestations enable a third party to remotely verify the state of a system. It provides all of this with a minimal TCB consisting of less than 250 lines of code.

Late launch on these commodity processors is achieved through the *SKINIT*-instruction on AMD processors. This instruction, however, does not save the state of the operating system. Every time security-sensitive code needs to run, the state of the operating system must be saved. After the execution of the protected code this state needs to be restored.

To provide attestation that the code was executed correctly, Flicker makes use of the *Trusted Platform Module* (TPM) chip. After a Flicker session was executed, the PAL is removed from memory and the operating system state is restored. If state needs to be remembered between different Flicker sessions, TPM-based *Sealed Storage* can be used. This way, it can be ensured that a ciphertext can only be decrypted when the system is in a certain state. This way a state can be stored in unprotected memory and restored only when the same PAL is running. It is important to note that this system does not protect against providing a PAL with a stale stale. Flicker proposes a solution by making use of a secure TPM monotonic counter. Making sure a process cannot be served a stale state is called *state continuity*, and will be explained in more detail in section 2.2.

Access to the TPM chip is very slow, so a huge disadvantage of Flicker is that the heavy use of this chip and the need to save and restore the operating system

state between Flicker sessions incurs huge overhead. When using TPM-based Sealed Storage, the performance overhead increases. A large part of the overhead comes from setting up and tearing down a Flicker session, so making a session last longer will decrease the relative overhead but decreases the responsiveness of other running processes. Therefore Flicker is not yet a practical solution for providing isolation of protected modules on commodity systems.

Program-Counter Based Access Control In more recent research the concept of Program-Counter Based Access Control (PCBAC) was introduced [39][37]. It is a technique to provide isolation of protected modules based on the value of the *program counter* (commonly called the *instruction pointer* in Intel x86 processors). The instruction pointer is a processor register containing the memory address of the next instruction to be executed. These modules do not need to trust each other or the operating system.

A protected module is essentially a memory segment consisting of two parts. A *text section*, containing the protected program code and constants and a *data section*, containing the protected data. The text sections also contains a list of *entry points*. These entry points are memory addresses in the text sections from where control flow is allowed to enter the module. By restricting these entry points as the only possible entries to the module, it can be guaranteed that control flow in the module can be controlled completely by the module itself. A memory segment belonging to a protected module is called *protected* memory, all other memory on the system is called *unprotected*.

Enforcement happens by examining the value of the program counter whenever an address inside the memory segment of the protected module is accessed. The access rules of PCBAC can be summarized in the *access control matrix* (Figure 2.1). The details of this matrix can vary depending on the implementation, but the basic idea stays the same.

Whenever the program counter is pointing to unprotected memory, for example executing regular software or kernel code, and then jumps to another address in unprotected memory, full read, write and execute access is granted. This is the normal situation in systems where no memory protection is active: kernel level malware has full read, write and execute access to all memory.

When the program counter is pointing to unprotected memory and then a jump to an address in protected memory takes place, execution will only happen if the address is an entry point of a protected module. In all other cases, permission will be denied.

If the program counter is already pointing to an address in the protected memory section of a protected module PM_1, read and write access to the protected data section of PM_1 is given. Also, read and execute permission of entry points and the text section of PM_1 is granted.

If the program counter is pointing to the protected memory section of a protected module PM_1, jumps to the protected section of another protected module PM_2 are

handled as if coming from unprotected memory. So only if PM_1 tries to execute an entry point of PM_2, will it be granted.

This way, different protected modules do not need to trust one another.

from \ to	Protected			Unprotected
	Entry point	Text	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Figure 2.1: Access control matrix

Three implementations of PMAs making use of PCBAC were developed at KU Leuven: *Sancus*, *Fides* and *Salus*. In this thesis Fides and Sancus were used, and will be explained in more detail.

Fides This is an architecture that runs on top of commodity desktop PC processors [38]. PCBAC is implemented in a small hypervisor running at a higher privilege level than the operating system.

The implementation strategy can be seen in Figure 2.2. The Fides hypervisor runs two virtual machines with the same logical view of physical memory. One virtual machine (*Legacy VM*) runs the unprotected code. The other virtual machine (*Protected VM*) is responsible for running protected code. When code running in the legacy VM attempts to access protected code, it will be trapped to the Fides hypervisor. If a protected entry point was called, the hypervisor switches to the protected VM and executes from the entry point. All other access from the legacy VM to protected code will be blocked. The protected VM runs a small secure kernel which is responsible for handling access rules between protected modules. When the hypervisor first loads, it creates a new dynamic root of trust, allowing a third party to do an attestation of the correct launch of the architecture.

Fides implements secure and authenticated communication between SPMs as *authenticated service calls*. When SPM_1 wants to call a function on SPM_2, it first locates it's *security report*, which verifies the identity of an SPM. SPM_1 will then validate the security report and verify if it indeed belongs to SPM_2. It then places it's parameters and the return address into registers. Afterwards it calls function on SPM_2. Modules can thus be authenticated by validating their security report. Security of the parameters is guaranteed since no other process can intervene and read the registers.

Sancus This is an architecture that implements PCBAC in hardware on a microprocessor [27]. It allows for an infrastructure provider, who owns a set of microprocessor-based systems, to deploy third-party software modules. These software modules do not need to trust each other. Sancus also provides a strong remote attestation mechanism, allowing to remotely verify certain code has run in a correct way.

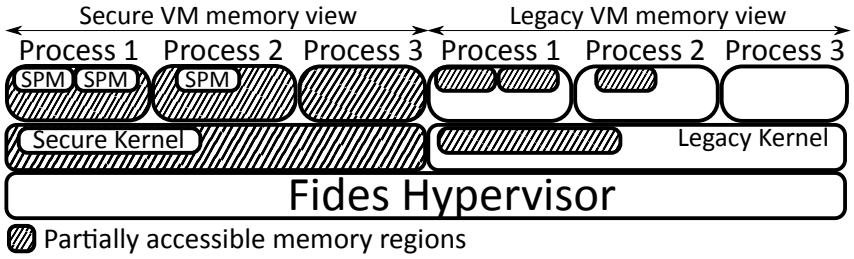


Figure 2.2: The implementation of Fides

The implementation runs on a Texas Instruments MSP430 FPGA. The processor is extended with a protected storage area. It maintains a list of all software modules that are loaded in memory along with their memory bounds. This way the isolation properties are enforced.

Encryption keys are based on the hash of the text section. When a software module requests encryption or decryption of a message, this key will be used. Tampering with module code will thus invalidate the key.

2.1.2 Limitations

It is important to note that PMAs do not defend against bugs in the modules themselves. If, for example, a buffer overread vulnerability exists, it might be possible that the module leaks its secrets to an attacker. It is advisable that protected software modules remain relatively small (small TCB). In this case, formal verification of a module is possible and strong guarantees against the existence of bugs in the modules can be given.

The PMAs outlined here still rely on the operating system to provide access to hardware. In other words, we need to trust the operating system for input and output. In the example of a remote banking application, the application itself can be implemented in a protected module, but it still needs to read the PIN through the system keyboard, and output (potentially sensitive) data to the screen. An attacker with kernel level access will be blocked from interfering with the inner workings of the module, but is still able to sniff the PIN code or read the screen buffer and intercept or change what is written on the screen.

2.2 State continuity

To minimize the TCB of PMAs, persistent storage of protected module state is delegated to the untrusted operating system. Whenever a stateful protected module restarts, it will attempt to load it's encrypted state from the untrusted file system, for example after a reboot. PMAs provide ways to easily cryptographically verify that a state was not tampered with, before loading it. They do not, however, provide a way to verify the *freshness* of a state. Without extra protection, an attacker could

easily always provide the same *stale* state to the protected module (*rollback attack*, thus never allowing it to advance. We should ensure state continuity.

One of the first proposed solutions to this problem is called *Memoir* [28]. Memoir is built on the Flicker PMA and provides state continuous execution of protected modules. It does this by using the TPM NVRAM. Freshness information of a state contains the history of requests to the protected module as a hash chain. Memoir has two implementations: *Memoir-Basic* and *Memoir-Opt*. Memoir-Basic requires a write to the TPM NVRAM on every request to the protected module that modifies its state. Current TPM NVRAM has very slow access times and should only support 100,000 write cycles. So constantly making requests to a stateful protected module would quickly wear it the TPM NVRAM memory. Their Memoir-Opt implementation tries to solve this by only writing to TPM NVRAM twice per boot. To do this, it requires an *uninterruptible power supply* (UPS) because, when it detects a system crash, it writes its state to TPM NVRAM. Since a UPS can easily be disconnected from a system and current TPM NVRAM specifications are not sufficient to provide performance, Memoir still has serious limitations.

Because of this, ICE [36][35] was introduced. ICE only needs to access TPM NVRAM during boot time, this significantly increases performance. State updates are done to *guarded memory* which are dedicated registers on the chip backed off-chip by a capacitor and non-volatile memory. Freshness information in ICE is stored as *guards*. Guards are tuples containing a base value that is hashed i times and the index value i. When power loss is detected, guarded memory is backed up to non-volatile memory. When power is reapplied, the register values are restored. In ICE, only the most recent, fresh state is backed up. The only information that can leak to an attacker is the backed up state. Since this is never a stale state, the attacker cannot use it to roll the module back to a previous state.

2.3 Formal verification

While PMAs give strong isolation guarantees, they do not protect against programming bugs in the protected module code. To be able to provide guarantees that certain bugs cannot exist in a protected module, memory-safe programming languages can be used. Memory-safe programming languages, like Java, protect against, amongst others, buffer overflows, buffer overread and use after free errors. Programming languages like C and C++, that allow arbitrary pointer arithmetic and that make the programmer responsible for allocating and freeing memory are typically not memory-safe. So, using a memory-safe programming language gives us some guarantees that so called memory access errors are very unlikely to occur.

By formally verifying our protected modules with program verifiers, like VeriFast [14], it can be proven that, even with the use of memory-unsafe programming languages, no memory access errors as well as no data races exist. To formally verify a program with VeriFast, each function must be annotated with a pre- and a postcondition. VeriFast will symbolically execute these functions and verify that each precondition is satisfied and check that, after the function was executed, the

postcondition holds. When VeriFast successfully verifies a program, it also proves that each function complies with the given pre- and postconditions. In practice, the Microsoft Hyper-V hypervisor, consisting of 100,000,000 lines of C source code and 5,000 lines of assembly was formally verified using the VCC verifier for Concurrent C [16]. Also, the XMHF hypervisor, consisting of 5630 lines of C code and 388 lines of assembly was formally verified using the CBMC model checker [42].

This shows that more and more companies are noticing the need to formally verify their security sensitive code and that in real-world situations, formal verification is feasible.

2.4 Cryptographic building blocks

This thesis builds on a few cryptographic building blocks to offer strong security guarantees. More specific, the Diffie-Hellman key exchange protocol, a solution to the socialist millionaire problem and encryption based on the SPONGENT hashing function will be used. We will give a very high level overview of these building blocks along with the security properties they provide. The details of their inner workings are beyond the scope of this thesis.

Diffie-Hellman key exchange protocol This protocol offers a way for two parties to agree on a shared key over an insecure channel [9]. This key can then be used for encrypting data being sent between them. A passive attacker, eavesdropping on the messages sent over the insecure channel, should be unable to learn anything about the key. The security of the protocol is based on the fact that no efficient solution to the *discrete logarithm problem* is known. It does not, however, protect against an active man-in-the-middle (MITM) attack. In chapter 6, this attack and its implications will be explained in more detail. This protocol does not authenticate the parties, but only provides them with a way to agree on a key. We say that the Diffie-Hellman key exchange protocol is an *unauthenticated key exchange protocol*.

Solution to the socialist millionaire problem The socialist millionaire problem [43][15] is a cryptographic problem in which two parties want to determine if both of them have access to the same shared secret over an insecure channel, without any of the two parties learning anything more than the equality of the secret. We will refer to a solution to this problem as the *socialist millionaire protocol*. By using this protocol on top of the Diffie-Hellman key exchange to verify if both parties have access to the same secret, we create a *password-authenticated key exchange protocol*. This way we can verify that no MITM attack took place during the initial key exchange. Many protocols exist that offer *authenticated key exchange*, but most of them rely on the fact that both parties must already have access to each others public keys prior to executing the protocol. This has a huge practical impact, since some sort of public key infrastructure (PKI) must be set up. By using the socialist millionaire protocol, both parties only need to have access to some sort of shared secret. As is explained in chapter 4, this is a realistic assumption. An attacker eavesdropping on the protocol

will learn nothing, not even whether the shared secrets of both parties match. An active attacker launching a MITM attack is unable to influence the outcome of the protocol, other than making it fail. This protocol is also being used in practice by the Off-the-Record Messaging protocol (OTR) to authenticate the identity of both communicating parties through a shared secret [7][1].

SPONGENT-based encryption SPONGENT [6] is a lightweight hash function, designed for use in embedded environments. These functions are then used to offer authenticated encryption [5]. Each message is encrypted along with a message authentication code (MAC), called a *tag* in the algorithm used. We will use these tags to ensure tampering with the ciphertext is detected.

This encryption algorithm offers confidentiality, integrity and fast performance, even when executed on an embedded device.

Chapter 3

Problem statement

In this chapter we will situate the problem, describe the attacker we want to defend against (section 3.1) and the security guarantees we expect from a solution (section 3.2).

This project aims to provide protected modules, running in an isolated way, with access to secure input and output. The user of a system should be able to connect hardware devices that offer encryption and authentication possibilities, we call this *secure hardware*. Protected modules should be able to request access to an arbitrary subset of these devices and be ensured that information exchanged between them is secure. We require it to hold up *confidentiality* and *integrity* of this information in the presence of a powerful attacker.

Software deployed on the system by an attacker should not be able to compromise the security of data exchanged with secure hardware, not even if it has kernel-level access. Hardware added to the system by the attacker, secure or not, should not be able to compromise the security of other secure hardware. Also, adding software or hardware to the system should also not allow the attacker to interfere with the isolation and security properties that our protected module architecture offers. An overview of the trust model can be seen in Figure 3.1.

3.1 Attacker model

The goal is to defend against a powerful attacker. First, he has kernel level access on the system. This means he is able to execute arbitrary code at the kernel privilege level, exercise complete control over all applications running on the operating system, to watch and modify communication between processes, and between processes and hardware. Second, the attacker is can deploy his own protected modules on the system that can request access to secure input and output. Lastly, the attacker is able to add his own virtual or physical devices to the system. An attacker is not able to break cryptographic primitives and does not have physical access to the hardware.

We use a PMA to isolate our protected modules from the rest of the system. This means that information inside the protected memory cannot leak to an attacker. Greater security guarantees can be given depending on the specific PMA used. Intel SGX will defend against hardware attacks against Enclaves, whereas Fides cannot protect the SPMs in the presence of a hardware attack. At the time this thesis was being written, Intel SGX has not yet been released.

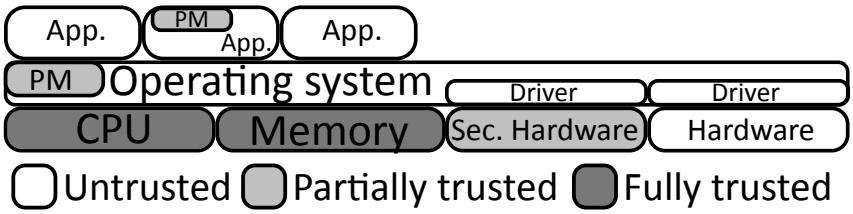


Figure 3.1: Trust model of the system

3.2 Security guarantees

We want a protected module to be able to communicate with secure hardware while providing two of the three CIA triad guarantees, as explained in chapter 1 [13].

- **Confidentiality**
  The information leaving or entering a protected module through a secure path should not be available or disclosed to unauthorized entities. Only the protected module and the secure hardware interacting with each other should have access to it.

- **Integrity**
  The data received by either a protected module or secure hardware should always be accurate and complete. An unauthorized party should be unable to modify the information in the system.

- **Availability**
  This means that the framework should provide protected modules with access to hardware at all times. Availability **cannot** be guaranteed in the presence of kernel level malware, since interaction with the hardware requires help from the operating system. The operating system can thus deny a protected module access to hardware or is able to simply power off the system. More advanced PMAs can be used to guarantee availability.

It must be noted that the secure hardware being used must be trusted in order to guarantee confidentiality and integrity of data passing through it. Compromised hardware will obviously be able to leak this information. Protected modules in the hands of an attacker requesting secure input are also able to leak this data. Components, such as secure hardware and protected modules, that are able to

leak some information, but only if it is directly available to them are *partially trusted*. Partially trusted components are unable to compromise security of other components. Components that are untrusted, like the operating system and regular (non protected) applications, are unable to interfere with the correct execution of other trusted or partially trusted components. Since the CPU has complete control over the system and its memory, it is a *fully trusted* component. Fully trusted components can compromise the whole system when in hands of an attacker. This can be seen in Figure 3.1. Depending on the specific implementation, some details of the trust model may differ. In hypervisor-based implementations, like Fides, the main memory is trusted since it contains the secrets belonging to protected modules in an unencrypted form. In these implementations, also a hypervisor must be trusted. In hardware-based implementations, such as Intel SGX, the information that leaves the CPU can be encrypted. In that case, the memory does not need to be trusted.

Chapter 4

Design

In chapter 3 we described the problem we want to solve, along with the properties we expect from a solution in the presence of a specific attacker. Now, we will propose such a solution. A framework for providing a secure path from protected modules to hardware (section 4.1) and a set of protocols that can be used to deploy the framework on a system (section 4.2) will be explained in detail.

To provide a meaningful service, applications need to interface with the outside work. We distinguish between two types of applications: services and client applications. Services only offer a software interface providing, for example, a pseudo random number generator or a HTTP server. These services can be run inside a protected module to isolate them from the rest of the system and provide good security guarantees. Client applications, however, usually rely heavily on access to hardware to obtain user input and provide output. A banking application, for example, would require a user to provide his PIN and transaction details through a keyboard and account information needs to be shown on a screen. All of this information is potentially sensitive. PMAs by themselves provide the means for a protected module to be securely executed in isolation from the rest of the system. For now, however, they do not provide a way to securely interact with hardware, which limits their usability in real world scenarios.

At the time of writing this thesis, Intel SGX was not yet available. Fides and Sancus were chosen as protected module architectures since they are readily available and easy to use. It is important to note that the work done here is easily portable to any other PMA that works in a similar way, like Intel SGX and Microsoft Hyper-V.

In this project, a framework was implemented that provides protected modules with a means of secure input and output. We call this a *secure path*.

4.1 Secure path

A secure path, as the name suggests, provides protected modules with a way to obtain secure input from, and send secure output to, special hardware devices. The

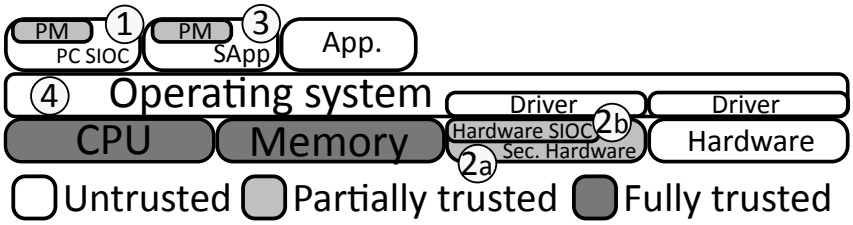


Figure 4.1: Secure path

framework implemented here should provide us with the security guarantees outlined in section 3.2.

This framework consists of a software Secure I/O Controller (*PC SIOC*) (part 1 of Figure 4.1). This component is responsible for mediating all access to secure hardware (part 2) from Secure Applications (*SApps*) (part 3) on the PC. It maintains a list of all trusted secure hardware devices along with their symmetric encryption keys in its protected memory, as well as a list of trusted SApps that are allowed access to trusted secure hardware. The PC SIOC interacts with the operating system (part 4) to connect to trusted secure hardware. A clean system starts out with a PC SIOC that trusts no hardware device and no SApp. The PC SIOC accepts data from a trusted SApp, encrypts it, and sends it to a secure hardware device as secure output. Conversely, it accepts encrypted data from secure hardware, decrypts it, and sends it to a trusted SApp as secure input. The PC SIOC is the heart of the framework.

Each secure hardware device (part 2a) is equipped with a hardware Secure I/O Controller (*Hardware SIOC*) (part 2b). The Hardware SIOC of secure output devices is responsible for receiving encrypted data from the PC SIOC, decrypting it and displaying it on the hardware. For secure input devices, it should encrypt the data and send it to the PC SIOC. Secure hardware can also be used as regular, unsecured hardware. In this case, the Hardware SIOC is bypassed and no encryption or decryption is done. Regular, non-protected applications can then interact with the hardware in an unsecured way. An LED on the hardware itself should indicate whether it is running in secured or unsecured mode. This allows the user to easily verify if a secure path is in place, much like the lock symbol placed next to a URL in a web browser whenever a connection is secured by SSL/TLS. When the LED is on, we say that the hardware is running in *secured mode*, otherwise it is running in *unsecured mode*.

Part 3 on the figure are the SApps that requires secure I/O from a hardware device. A trusted SApp can request access to one or more trusted secure hardware devices. The requested devices will then run in secured mode, all other hardware devices connected to the system will run in unsecured mode. The SApp gains complete control over the hardware devices in secured mode. We now call the SApp *active*. This means that two SApps can never control secure hardware devices simultaneously. In order to prevent phishing attacks, the active SApp must be clearly distinguishable from other processes or non-active SApps running on the

system. This can be achieved by having the user's secure screen draw a special border around all the windows of the SApps graphical user interface and requiring the SApps' windows to be on top of every other window on the system. Also, a unique SApp-name must be shown in the title bar of each window. This allows the user to easily verify which SApp is currently active and which secure hardware devices it has access to. An active SApp without a GUI can be listed in the GUI of the PC SIOC. Because it is clear which SApp has exclusive access to the hardware, a trusted SApp controlled by an attacker is unable to trick the user into thinking the attacker SApp is a trusted bank module and entering his PIN. When a SApp is first trusted by the PC SIOC, the user can choose this unique SApp-name that will identify it once it's active. These requirements might be overly restrictive, and future implementations of secure paths could relax these. Communication between the SApps and the PC SIOC should also happen in a secure way.

We say that a secure path is completely set up once the PC SIOC trusts at least a secure keyboard, a secure screen and a SApp.

4.2 Deployment

We will now explain how a secure path can be set up on a clean system. The system starts with a PC SIOC installed. Each system comes pre-programmed with a unique *System ID*. This System ID can be read by the PC SIOC and is also physically printed on the case of the computer. In addition, each secure hardware device comes with a *Device ID* that is also printed on it. The system is also equipped with a hardware button, called the *Hardware Secure Button*, that a user must press to indicate he wants to add a new hardware device. At the start, the PC SIOC trusts no hardware device and no SApp.

To obtain a working secure path, a trusted secure keyboard, a trusted secure screen and a trusted protected module must be added in a secure fashion. Setting up a secure path must happen in such a way that an attacker cannot read or modify the data sent between the SApp and the secure hardware. So, confidentiality and integrity of I/O data needs to be preserved, as explained in section 4.1. Therefore we must use a protocol that guarantees these security requirements. The protocols used to add the hardware device and the SApp will be outlined below, the security of these methods will be analyzed in section 6.1.

Adding a trusted hardware device We start by adding a trusted keyboard using the following protocol. The PC SIOC or the Hardware SIOC of the secure keyboard do not need to have access to any keys or certificates at this point. The user plugs in the keyboard equipped with a Hardware SIOC, which is untrusted for now (step 1 in Figure 4.2) and presses the Hardware Secure Button on the system to indicate he wants to add a new trusted hardware device (step 2). Using the Diffie-Hellman key exchange protocol, an encryption key will be agreed upon between the PC SIOC and the Hardware SIOC in the device (steps 3 and 4). The SIOC

challenges the new hardware to enter the System ID using the socialist millionaire protocol (step 5). When the keyboard receives this request, it enters authentication mode (step 6). In this mode, entered keystrokes will not be sent to the PC but buffered until the enter key is pressed. When the enter key is pressed, a response is generated using the socialist millionaire protocol (step 7). In step 8, the SIOC verifies the response. If the response shows that the System ID was entered correctly, the new keyboard is now trusted. The PC SIOC responds with an *OK* message (step 9). Now the keyboard's Hardware SIOC still needs to verify the PC SIOC and challenges it for the System ID (step 10). The PC SIOC receives this challenge, calculates the appropriate response (step 11) and sends this to the keyboard (step 12). In step 13 the Hardware SIOC verifies the response. If it is correct, the keyboard now trusts the PC SIOC and responds with an *OK* message (step 14).

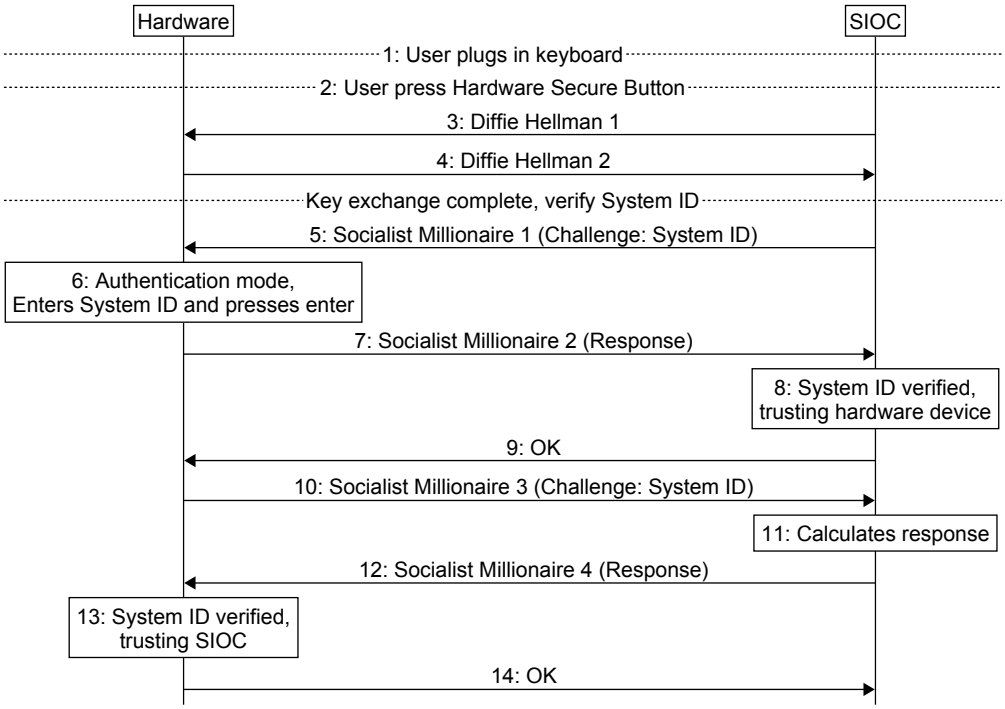


Figure 4.2: Adding new trusted input device

We end up with a symmetric encryption key shared between the PC SIOC and the Hardware SIOC of the trusted keyboard, both of which are stored in their respective protected memory sections.

Once a secure keyboard is in place, we can easily add any other hardware device to the system as follows. The PC SIOC now has an encryption key shared with the Hardware SIOC of the trusted keyboard. The user plugs in the new device (step 1 of Figure 4.3) and presses the Hardware Secure Button on the system to indicate

he wants to add new secure hardware (step 2). An encryption key will be agreed upon between the Hardware SIOC and the PC SIOC (steps 3 and 4). The user is now prompted to enter the Device ID of the new hardware on the trusted keyboard (step 5). This way the Device ID is securely transmitted to the PC SIOC, which will now challenge the new hardware (step 6). The Hardware SIOC receives this challenge, calculates a response (step 7) and sends it back to the PC SIOC (step 8). Once the PC SIOC verifies the Device ID (step 9), an *OK* message will be sent (step 10). The hardware will now verify the PC SIOC by challenging for the Device ID (step 11). A response will be calculated (step 12) and returned (step 13). Once the response is verified, the PC SIOC is now trusted by the Hardware SIOC (step 14) and an *OK* message is sent.

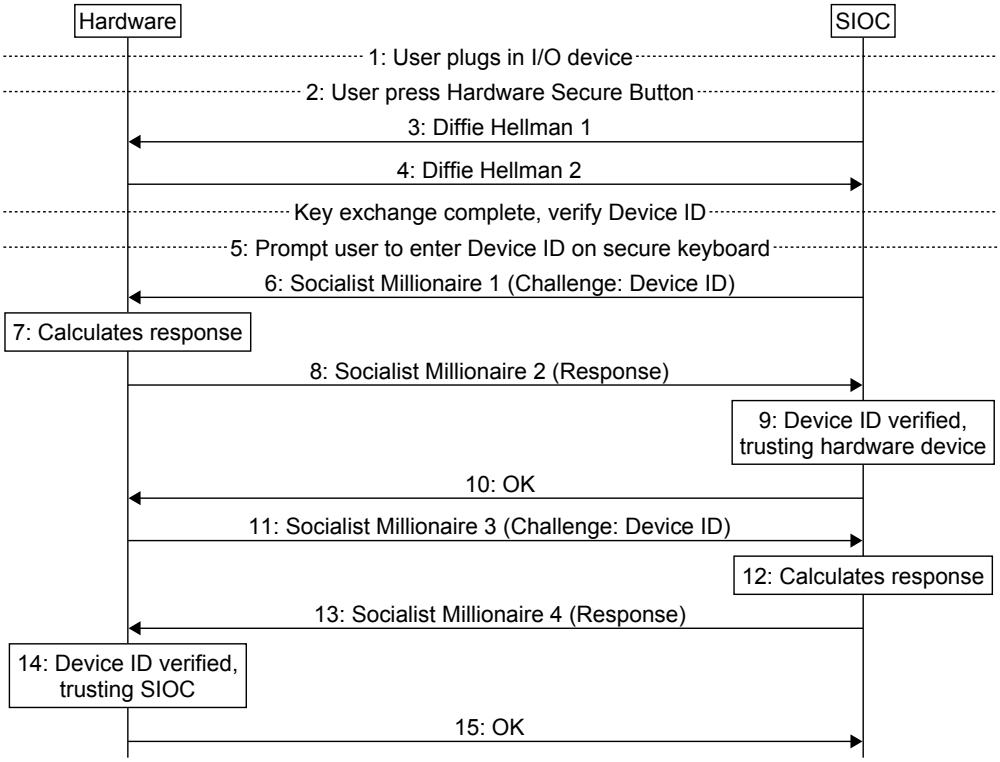


Figure 4.3: Adding a new trusted hardware device with trusted keyboard in place

After the protocol is executed, the PC SIOC now also shares a symmetric encryption key with the Hardware SIOC of the newly added device. These keys, again, are stored in their respective protected memory sections.

The System ID is only needed when adding a secure input device for the first time to a clean system. Afterwards only the Device ID of a new I/O devices needs to be entered.

It must be noted that the secure hardware device we want to add is trusted. If this device itself is in the hands of an attacker, it could easily leak all I/O data available to it, the Device ID or its encryption key and thus compromise the confidentiality or integrity of the data. It can however not compromise the data of *other* secure hardware devices. If the first secure keyboard is compromised, it could leak the System ID, which would compromise all future secure I/O. A way to manually change the System ID could be implemented, but this is left as future work.

The protocols outlined here enable both the PC SIOC and the Hardware SIOC to mutually authenticate each other. An encryption key is exchanged that allows confidentiality and integrity of communications between them. These properties will be evaluated in section 6.1.

Adding a trusted SApp Once a trusted input and a trusted output device are in place, SApps needing access to secure hardware can be added as follows. If a certificate authority (CA) is used to sign SApps, the PC SIOC should have access to one or more root certificates. At this point the PC SIOC should share symmetric encryption keys with both a trusted secure keyboard and a trusted secure screen. An untrusted SApp will request secure I/O (step 1 of Figure 4.4) from the PC SIOC. The PC SIOC will verify that the SApp is signed by a trusted CA (step 2) and prints a message to the trusted screen containing a notification that a new SApp is requesting access, whether it is signed by a trusted CA, and a random string (step 3 and 4). Regardless of whether the SApp was signed, the user can choose to allow it by entering the shown random string on the trusted keyboard (step 5 and 6). If the string was entered correctly (step 7), the SApp is sent an *OK* message (step 8) and allowed access to trusted secure hardware devices.

After the protocol finishes, the PC SIOC now trusts the SApp. This means it stores a unique SApp identifier in its protected memory.

The protocol outlined here enables the PC SIOC to authenticate the SApp unilaterally. These properties will be evaluated in section 6.1.

Using the secure path Once a secure output device, a secure input device and SApp are trusted by the PC SIOC, the trusted path is ready to be used. We will give an overview of how the secure path works by giving an example of a trusted SApp outputting 'Please enter PIN' and receiving back '1234' in a secure way.

In step 1 of Figure 4.5 the trusted SApp requests access to the secure screen D_1 and the secure keyboard D_2. The PC SIOC receives this requests, activates these two secure devices and makes sure all other secure devices become inactive (step 2). These inactive devices can still be used, but encryption is turned off. With an *OK*-message (step 3), the trusted SApp is notified he now has exclusive access to the requested devices (step 4).

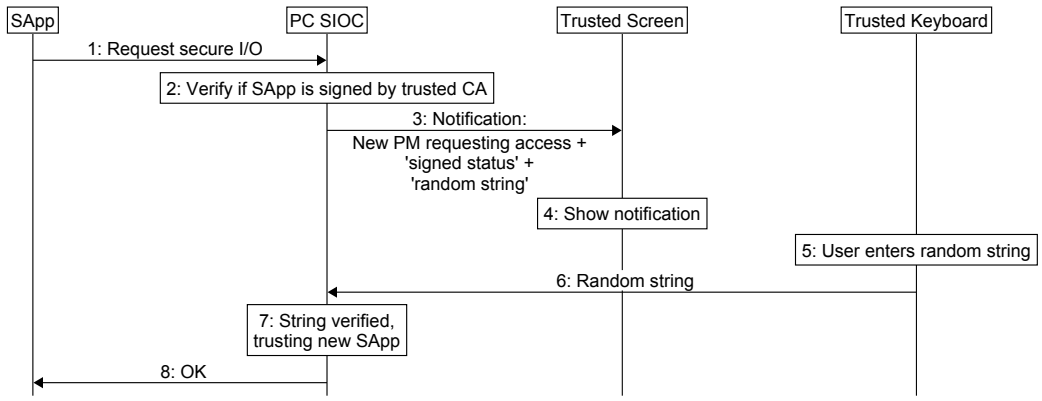


Figure 4.4: New SApp requesting secure I/O

In step 5 the trusted SApp sends a request to the PC SIOC, asking it to print the string 'Please enter PIN' to the secure screen (D_1). The PC SIOC encrypts this message with the shared encryption key K_{D_1} for D_1 (step 6) and sends it to the secure screen (step 7). The secure screen will decrypt the string with K_{D_1} and display 'Please enter PIN' on the screen (step 8). An acknowledgment is sent from D_1 to the PC SIOC (step 9). Before forwarding anything to the SApp, its SApp ID is verified (step 10). Only if the trusted SApp is still loaded correctly, the acknowledgment is forwarded (step 11). The trusted SApp now knows that his string was printed on D_1 in secure fashion.

The user enters his pin, '1234', on the trusted keyboard (D_2) which gets encrypted character-by-character with it's shared key K_{D_2} (step 12). Note that this is not the same shared key as for D_1, each trusted secure hardware devices has a different shared key that only he and the PC SIOC know. This encrypted PIN gets sent (character-by-character) to the PC SIOC (step 13) and decrypted with K_{D_2} in step 14. Also, it is again verified that the SApp is still correctly loaded. If so, the entered characters, '1234', are now forwarded to the trusted SApp (step 15) and can be processed (step 16).

This protocol ensures that communication that goes through the secure path is both integrity and confidentiality protected and that SApps are able to unilaterally authenticate secure hardware devices. We also want the data received by a SApp or a Hardware SIOC to be fresh. This means an attacker must be unable to record communication and send it again later, a so called *replay attack*. These properties will be evaluated in section 6.1.

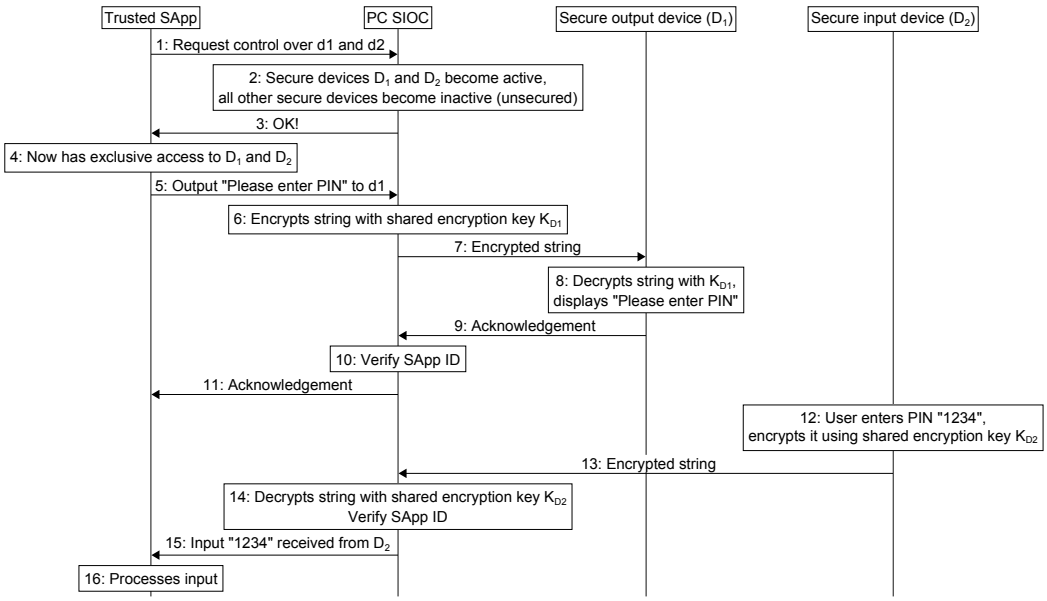


Figure 4.5: Schematic overview of example usage of a secure path

Chapter 5

Implementation

As a proof of concept, the framework outlined in chapter 4 was implemented. It was explained earlier that our system consists of Secure Applications requesting secure I/O, a Software Secure I/O Controller, a Hardware Secure I/O Controller per secure hardware device and a way for them to interact. The security properties outlined in chapter 3 are achieved using a protected module architecture and by encrypting all sensitive data that leaves protected memory with an authenticated encryption protocol.

Our secure path relies on encryption to protect information passing through it. Encryption and decryption only takes place in the PC and Hardware SIOCs. Unencrypted information can never leave protected memory. In our implementation, an authenticated symmetric encryption algorithm using SPONGENT [6][5] as an underlying hash function was used. SPONGENT is a lightweight hash function, designed for use in embedded environments. Each message is encrypted along with a message authentication code, called a *tag*. MACs make sure that tampering with ciphertext will be detected and are thus used to ensure integrity of our data.

The architecture of the system is shown in Figure 5.1 (extended from Figure 3.1). The meanings of *fully trusted* and *partially trusted* are explained in section 3.2. Details about the implementation are explained in the following paragraphs.

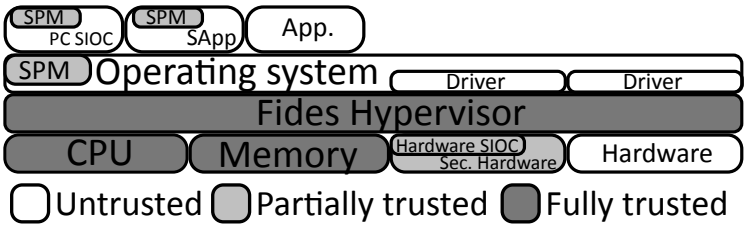


Figure 5.1: Architecture of the implementation

Protected module architecture Our framework relies on a PMA for isolating memory segments from the rest of the system. Thus, strong security guarantees

are achieved without having to trust a very large trusted computing base. We chose to implement the PC SIOC with Fides [38], and the Hardware SIOC with Sancus [27]. This means that on the PC, protected modules are implemented as a Fides Self-Protecting Module. In the secure hardware, the Hardware SIOC is implemented in a Sancus Software Module.

Secure Applications SApps consist of two parts: an unprotected and a protected part. The unprotected part contains code that is not security sensitive and does not need to be isolated from the system. The unprotected code runs in an untrusted environment, so its data and code cannot be trusted. The protected part is implemented as a protected module, in our case a Fides SPM. The unprotected code can interact with the protected module through a well-defined interface. Since an attacker could have complete control over the unprotected code, the protected module needs to make sure that all input from it is validated. If we try to keep the protected modules small, they can be formally verified with, for example, the VeriFast [14] program verifier. This can significantly increase our trust in a protected module.

Secure I/O Controllers SIOCs are the basis of this project. The PC SIOC is implemented as a SApp. This means that it is also split into protected and unprotected code. The unprotected code is responsible for interacting with the operating system. When, for example, the PC SIOC receives input, the unprotected code receives encrypted data from the hardware through the operating system. It then does some processing on this data and presents it as input to the protected module. The protected module will then decrypt the data and pass it to the currently active SApp (secure input). Conversely, the unprotected code receives encrypted data from the protected module and passes this on to the hardware through the operating system (secure output). Decrypted data never resides in the unprotected memory; the code in unprotected memory is only responsible for processing encrypted data and interacting with the hardware and the operating system.

The Hardware SIOC will be explained in more detail in the next paragraph.

Secure hardware The Hardware SIOC protected module was implemented as a Sancus Software Module on a Texas Instruments MSP430 FPGA (clocked at 20Mhz). As a secure keypad the Digilent PmodKYPD was used. As a secure screen we used the Digilent PmodCLS. Both hardware devices were used as an add-on to the FPGA. To offer additional protection against malware that might be running on the FPGA itself, a memory-mapped I/O (MMIO) module for each of these modules was re-written in assembly. These modules are protected by Sancus and are mapped over the memory addresses that are used to interact with the keypad and screen. When accessing a hardware device, the MMIO module checks if the requesting application ID corresponds to that of the Hardware SIOC. If this is not the case, access to the hardware is denied. This means that only the Hardware SIOC has access to the keypad and screen. Since the these MMIO modules are implemented as protected modules, their memory is isolated from the rest of the system. Conceptual, the

rest of the Hardware SIOC works analogously to the PC SIOC; it is responsible for receiving encrypted data from the PC SIOC and decrypting it as well as receiving unencrypted input from hardware, decrypting it and sending it to the PC SIOC.

In this implementation we chose that, whenever a key was pressed on the keypad, this single character was instantaneously encrypted and sent to the PC SIOC. We shall explain the implications of this in section 6.1.

Communication Communication between SApps and the PC SIOC is done using the built-in SPM communication channel of Fides. Data sent over this channel does not need to be encrypted by the SPMs, security is handled by Fides itself, as explained in section 2.1. Communication between the PC SIOC and the Hardware SIOC is secured using the encryption algorithm mentioned earlier. The format of an example message of 184 bits in length sent between SIOCs is shown in Figure 5.2. In this message, the length of the ciphertext is 8 bit, the length of the tag is 64 bit. Also, as can be seen, each message is encrypted along with a 32 bit nonce, implemented as a counter. This means that no two messages containing the same payload will be encrypted to the same ciphertext, given that the nonce doesn't wrap. For messages coming from the PC SIOC, the first bit of the nonce is set to 0. Conversely, for messages coming from the Hardware SIOC, the first bit is 1. Since the nonce is a 32 bit unsigned integer and the first bit identifies the SIOC, more than 2 billion (2,147,483,648 to be exact) messages can be encrypted by a SIOC before the nonce wraps. In some real world scenarios this might not be enough. In that case, larger nonce sizes can be used without breaking the current implementation. An example message of 23 bytes (184 bits) in hexadecimal format is `1711072201d92204000007182208e17c545dc8f2504433`. Here, the nonce counter value is 0x00000718 (1816 in decimal). These messages are sent over an UART connection.

Bits

0	8	16	24 32
Length of total string	Start symbol	Mode	Delimiter symbol
Length of ciphertext	Ciphertext	Delimiter sbymtol	Length of nonce
Nonce			
Delimiter symbol	Length of tag	Tag	
Tag cont.			
Tag cont.		End symbol	

Figure 5.2: Format of an example message. Total length of 184 bits, ciphertext length of 8 bit and nonce length 32 of bits

Fides, the PC SIOC and SApps and run on a Dell Inspiron 7520 SE laptop with an Intel i7-3632QM processor All code running on the PC was written using the C programming language. Implementations on the FPGA were done in C++ with exception of the MMIO modules, which were written in assembly.

Chapter 6

Evaluation

In this chapter we evaluate whether our implementation from chapter 5 meets the security properties requested in chapter 3 (section 6.1). Also, the performance of our implemented framework will be measured (section 6.2).

For our implementation it is necessary that it is secure and that its performance is acceptable. In the following sections we will show that our framework guarantees integrity and confidentiality of data sent between the protected modules and the secure hardware. Also we will show that it is done with acceptable overhead.

6.1 Security evaluation

We repeat that, for our secure path to be considered secure, it needs to guarantee *confidentiality* and *integrity* of our I/O data. We will show that an existing secure path in our framework satisfies our security requirements (subsection 6.1.1), and also that setting up a secure path can be done in a secure fashion (subsection 6.1.2).

6.1.1 Existing secure path

Given that we have set up a secure path, the following is true by definition: a secure input device is trusted by the PC SIOC, a secure output device is trusted by the PC SIOC and a SApp is trusted by the PC SIOC (see Figure 4.1). We want to show that confidentiality and integrity are preserved, even in the presence of kernel-level malware or malware running on the FPGA. Also the data sent over the communication channel must be guaranteed fresh. We briefly mention two potential attacks against our implementation, along with a defense.

Confidentiality Sensitive data can reside in protected or in unprotected memory. In its unencrypted form it is always inside protected memory. Sensitive data leaving protected memory is always encrypted. Our protected module architecture ensures that all data in protected memory remains isolated from other processes on the system. This means that sensitive data, in its unencrypted form, can't leak to an

attacker. An attacker is also unable to interfere with encryption, decryption or processing of the sensitive data inside a protected module. In a secure path the PC SIOC and the Hardware SIOC both have a copy of their symmetric encryption keys in protected memory. This means this key also cannot leak to an attacker. All sensitive data that leaves protected memory and is sent over untrusted channels is encrypted using this encryption key. Each Hardware SIOC shares a different encryption key with the PC SIOC. A malicious Hardware SIOC will be unable to compromise confidentiality of data coming from, or going to another Hardware SIOC. Once input from secure hardware is decrypted in the PC SIOC, it is passed to a SApp securely using the Fides SPM communication channel. This gives a strong guarantee that sensitive data remains confidential in our framework.

Integrity Encrypted messages are sent along with a message authentication code. A MAC can be seen as a hash based on cleartext that is being encrypted and the encryption key. After decrypting data, the receiving party will calculate the expected MAC for the cleartext. If it matches the received MAC, the receiver can be sure that the ciphertext was not tampered with. An attacker that tampers with the ciphertext will be unable to compute a valid MAC for the corresponding cleartext since he doesn't have access to the encryption key. This way the integrity of our data is ensured.

Freshness To ensure that the data received by the PC SIOC or the Hardware SIOC is not stale or replayed data, the counter value of each received message is inspected. Every message sent between both SIOCs uses a monotonically increasing counter as nonce. The PC SIOC keeps a list of each of the Hardware SIOCs it trusts along with the counter value of their last message. Each Hardware SIOC also keeps the last seen counter value of the PC SIOC. Each time a SIOC receives a message, the counter is compared to the stored value. If it is strictly greater, the newly received message is guaranteed fresh. If it is equal to, or smaller than the stored value, it is stale data and the message should be discarded. For this reason it is important that the counter value must not wrap.

FPGA Malware The memory-mapped I/O code that is used to interact with the hardware on the FPGA were implemented as protected modules. This means that no malware running on the FPGA can directly communicate with the hardware, and all hardware access is mediated through the MMIO module. The MMIO module only allows the Hardware SIOC direct access to the hardware.

Side-channel attacks It must be noted that by encrypting and sending a character immediately when a key was pressed on the keypad, this implementation is likely susceptible to side-channel attacks. Since malware might not be able to see *which* key was pressed, it will still be able to see *when* this happened. By carefully measuring the time between each key press, a so called *timing attack*, an attacker might try to infer what keys were pressed. On a QWERTY keyboard, for example, the time

between an `f` and a `j` being pressed is going to be significantly shorter than between a `b` and an `8`. By measuring and analyzing input, confidentiality could be partially or completely defeated [10]. Side-channel attacks are out of scope of this thesis, but we will point out that a few methods exist to protect against these attacks. We could make sure that characters are only sent every predefined time quantum, for example every 100 ms. This way, statistical analysis of timing differences between pressed characters becomes harder. This type of solution could lead to performance problems, since an input lag of 100 ms might not be acceptable in every case. Another solution might be to only send characters every time quantum of 10 ms or even 1 ms, and fill in unused time quanta with dummy characters. The SApp that is receiving input just discards dummy characters and only processes 'real' input. The attacker will then just see a stream of characters coming in, one every 10 ms or 100 ms but is unable to distinguish dummy characters from real characters. This way he is unable to infer any information from inter-character timing. This solution does incur high overhead.

Phishing attack An attacker that was able to deploy a SApp on the system and have the PC SIOC trust it could attempt a phishing attack. He could, for example, present his SApp as being the SApp for processing bank transactions. Since an LED will show the user of the system that his input is now handled securely, he could be tricked into entering his PIN. To defend against this type of attack, it must be clear to the user exactly which SApp is active and receiving his data. As explained in chapter 4, each SApp has a unique SApp-name, chosen by the user, that is shown at all times when the SApp is active. Also the currently active SApp should be on top of every other window on the screen and it should clearly be shown what windows belong to it. Before entering sensitive data, a user must check whether the currently active SApp matches the SApp it claims to be.

This shows that, given a secure path was already set up securely, an attacker with kernel level access is unable to intercept or modify the cleartext in an undetectable way, thus demonstrating that our framework provides confidentiality and integrity. Even an attacker who is able to deploy it's down software on the FPGA is is not able to compromise security.

6.1.2 Setting up a secure path

We will now show that setting up a secure path can also be done securely. Setting up a secure path correctly means the addition of a new trusted input device, a new trusted output device and the addition of a new SApp that requires secure I/O.

Adding a new trusted hardware device To interact with hardware, the SApp relies on facilities provided by the operating system. Since, because of our threat model, we do not trust the operating system, we need a way to verify that we are adding the correct secure hardware device. This means we need to check if the user explicitly requested the addition of a new hardware device and that the device

that is added to the PC SIOC is the same device the user wanted to add. If this is not verified, an attacker that is able to compromise the operating system could add his own hardware devices to the PC SIOC, as shown in Figure 6.1. If this were possible, an attacker could add his own virtual secure output device instead of the user-intended one. He could then add the user' screen to a PC SIOC controlled by him, which would allow him to read all secure output from the user, and forward it transparently to the users' screen. To mitigate this threat, we authenticate that the correct hardware is being added using the socialist millionaire protocol on top of the Diffie-Hellman key exchange protocol. The security properties of these protocols is given in section 2.4.

When we add a new trusted I/O device to the PC SIOC, a key exchange takes place with the Diffie-Hellman key exchange protocol. This protocol allows two parties to agree on an encryption key over a public channel. It prevents an eavesdropper from learning the secret key. It does not, however, protect against an active man-in-the-middle attack during the exchange itself, it is a *non-authenticated key exchange protocol*. If the attacker is able to intercept and modify messages on the channel during the exchange, he is able to manipulate the protocol such that he can agree on one key with the PC SIOC, K_1, and on a different key with the Hardware SIOC, K_2. This means that if the Diffie-Hellman key exchange was attacked, the PC SIOC and the Hardware SIOC will have two different keys after the key exchange, both of which are known by the attacker (see Figure 6.2). He can then intercept any message from the PC SIOC, decrypt it with K_1, re-encrypt it with K_2 and send it to the Hardware SIOC. Traffic in the other direction can be compromised in an analogous fashion. The attacker thus ends up able to intercept and modify any message between both SIOCs.

To mitigate this type of attack, we must verify that the key exchange was completed between the two intended SIOCs. This means we must add another layer to Diffie-Hellman to authenticate the keys after the exchange. We need to verify that $K_1 == K_2$ using the socialist millionaire protocol. After the key exchange the PC SIOC challenges the new I/O hardware for the System ID and it enters authentication mode. This means that keystrokes will not be sent, but buffered until the enter key is pressed. When the enter key is pressed, a response based on the hash of the entered System ID and K_2 will be generated and sent to the PC SIOC. The PC SIOC also computes a string based on the hash of the System ID and K_1. If the strings are the same, it means that both keys as well as the System ID are equal and thus no MITM attack took place. After the PC SIOC has verified the new Hardware SIOC, the Hardware SIOC will also challenge the PC SIOC for the System ID and its key in the same way.

If an attacker was able to compromise the Diffie-Hellman key exchange, both SIOCs will have a different encryption key. This will be detected by the socialist millionaire protocol since the attacker does not know the System ID, and is unable

to forge a response with the correct key and correct System ID. If he simply relays the challenge and response messages, the other party will detect that a different key was used and the verification fails. Because the protocol we use is able to agree upon a shared key and authenticate that the key exchange was not compromised by an attacker, our protocol is a *password-authenticated key exchange protocol*.

The security of the framework relies on the secrecy of the System ID. If an attacker has access to it, he is able to add his own trusted secure I/O devices. This System ID, however, is only requested whenever a new untrusted secure input device is added to a clean system for the first time, which only happens seldom in practice. This is also the only time an attacker can try to guess it. This leaves very little chance for a successful brute-force attack of the System ID if we ensure it is long enough. Since the attacker does not have physical access to the users' system, he is also unable to just read the System ID from the case.

When the trusted keyboard is in place, new secure hardware devices are added by entering their Device ID on this trusted keyboard. As stated in subsection 6.1.1, data coming from a trusted input device and going to the PC SIOC is confidentiality and integrity protected. This means that the attacker cannot interfere with the Device ID being entered. The user is sure the device he added is the correct device and an attacker is unable to add his own devices without explicit manual user interaction.

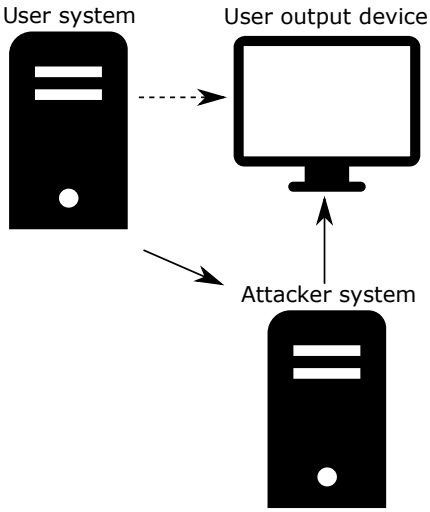


Figure 6.1: Man-in-the-middle attack. User thinks his system is directly connected to the screen (dashed arrow), in reality an attacker can intercept, see and modify all data (solid arrows)

Adding a new trusted SApp It is important that we are able to verify what SApps are trusted, since a compromised SApp that is allowed access to secure I/O is able to leak this data easily. SApp providers, like banks, could have their modules

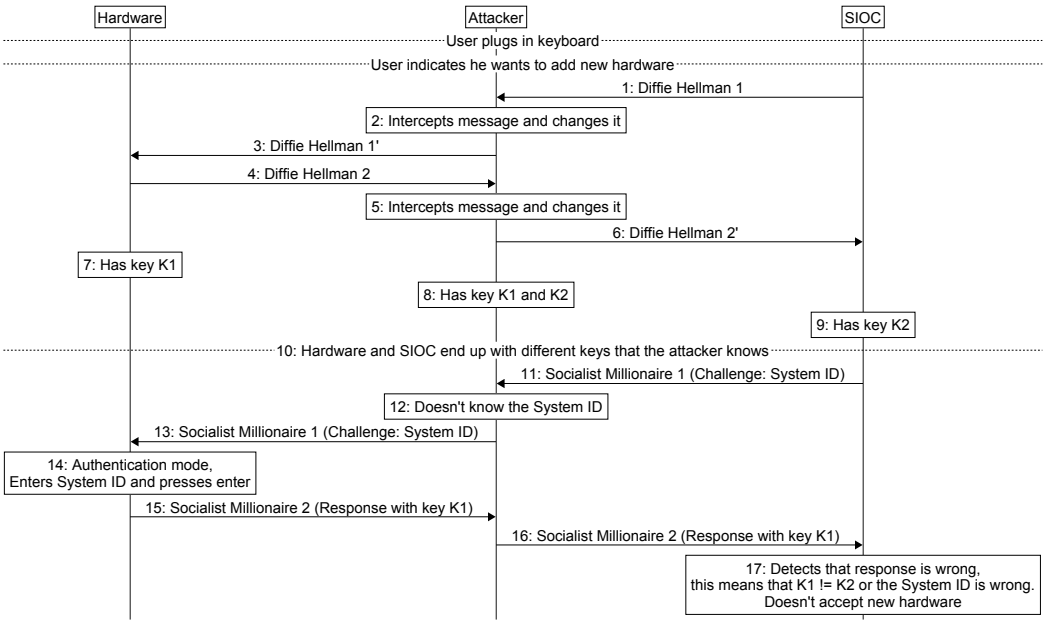


Figure 6.2: Man-in-the-middle attack on key exchange

signed by trusted certificate authorities. If we trust the CA not to sign malicious modules and thoroughly verify it, this signature adds an additional layer of trust. Whenever a new SApp requests access to secure I/O, a message is shown to the user on his secure screen. This message also contains a random string. Since an attacker is unable to read or modify this information, a new SApp can never be trusted without manual user interaction: typing the random string on this secure keyboard. This random string is sent over an already established secure path, so an attacker is unable to read or modify it.

Additional security can be gained by having new SApps that need secure I/O request permission to access certain devices. A SApp that is used to authenticate a user to the bank will probably never need access to a secure camera or microphone. A SApp is only able to request secure I/O from devices it has permission to access.

6.2 Performance Evaluation

To test the performance of our implementation, the overhead caused by protecting the SIOCs, the protected modules and encrypting the I/O data was measured. All tests were done on a Dell Inspiron 7520 SE laptop with an Intel i7-3632QM processor clocked at 2.2 Ghz. Fides does not currently support multiple cores, so only one of the four cores was enabled.

Encryption/decryption overhead To measure the overhead caused by encrypting and decrypting our messages in the PC SIOC, 10,000 characters were sent from unprotected memory to the PC SIOC. There they were encrypted, decrypted and

returned. Once the character was returned correctly, the next character was sent. Time was measured by counting the number of CPU cycles the tests took. Because of this, CPU frequency scaling was disabled on the system. This measurement gives us a good idea of how much overhead Fides causes by switching the stack each time control is handed over to an SPM and back, and how much time encryption and decryption takes. We find that encrypting and decrypting 10,000 characters takes 45538665095 CPU cycles. On the test system this translated to a total time of 20.699 seconds or 483.11 characters per second.

To measure the overhead caused by encrypting and decrypting our messages on the PC SIOC as well as the Hardware SIOC, 10,000 characters were encrypted on the PC SIOC and sent to the Hardware SIOC. The Hardware SIOC then decrypted these characters and displayed them on the secure screen. Afterwards the character was re-encrypted and sent back to the PC SIOC. When the PC SIOC received the character, it proceeded to encrypt and send the next one. The number of CPU cycles this took on the PC was measured. This time gives us an idea of the round-trip time (RTT) of a character sent from the PC SIOC to the Hardware SIOC and back. Then the RTT was again measured, this time with no encryption in place. By comparing these two RTT values, we can see the overhead caused by encryption and decryption in the PC and the Hardware SIOCs.

Sending 10,000 characters one by one from the PC SIOC to the Hardware SIOC and back with encryption, decryption and both PMAs enabled took 472.773 seconds. This means that, on average, 21.15 characters were sent back and forth per second. The RTT per character then was 47.28 ms. With encryption disabled but the PMA security in place, this test took 444.817 seconds. Thus the average was 22.48 characters per second, or an RTT of 44.48 ms. A difference of 1.76 ms in RTT was measured, caused by Fides.

PMA Overhead We also measured the overhead caused by the PMAs themselves. For this, all protection and isolation measures were disabled. The SIOCs were implemented as legacy applications and no PMA was active on the system. Again, 10,000 characters were sent back and forth between the PC and Hardware SIOC (which are not secure in this implementation) while measuring the CPU cycles. These measurements were compared to those of the framework with no encryption enabled. We found that this took a total of 240.128 seconds. On average, 41.64 characters per second and an RTT of 24.01 ms was obtained. Most of this overhead is caused by isolating the SIOCs with a PMA. To measure the share of each of the two PMAs, the test was repeated with Fides disabled, but Sancus enabled. We see that we get virtually the same results that we obtained with Fides on. This means that almost all of the overhead is caused by Sancus isolating the Hardware SIOC.

The results of this tests can be found in Figure A.1 and Figure A.2. Repeating the tests showed that the variance between tests was negligible and that the measured RTTs were consistently the same.

We conclude that encryption and decryption in our PC SIOC isolated by Fides is responsible for very little overhead. Also, a large part of the time increase caused by a PMA comes from the need for switching the stack every time control is handed over to a protected module, which takes some time. Since for every received character, multiple stack switches are needed, this amount becomes significant when many characters are sent. Fides was run on a fast processor, so the overhead it causes was negligible. Isolating the Hardware SIOC with Sancus on the MSP430 microcontroller slows down the framework, but this can be expected from a device with a clock speed of 20Mhz that is designed specifically for low power consumption. Figure 6.3 shows the relative time it takes the each part of the framework to process a message. It can be seen that encryption and decryption takes longer on the PC than on the FPGA. This is likely caused by a more efficient implementation of the encryption algorithm on the FPGA. Transferring the message between both SIOCs over the UART connection takes up most of the time. Using USB for connecting secure hardware to the PC can significantly reduce this time.

The time it takes for receiving input and sending output to the secure screen is acceptable since they appear to happen instantly. By providing a hardware instruction to switch the stack in hardware-based PMAs, like Intel SGX, these times could be reduced significantly. It is important to remember that this implementation serves as a proof of concept prototype. In a real-world scenario, Hardware SIOCs will probably be implemented in dedicated hardware chips with a cryptographic coprocessor.

	Time (seconds)	Time (relative to total)
Transfer overhead	240.128	50.79%
Sancus overhead	204.656	43.29%
PC encryption/decryption	20.667	4.37%
Hardware encryption/decryption	7.289	1.54%
Fides overhead	0.0326	0.01%
Total time	**472.7729**	**100%**

Figure 6.3: Relative time each part of the framework takes to send a 10,000 messages back and forth between the PC SIOC and the Hardware SIOC

Chapter 7

Use case: One-Time Password Generator

We implemented our framework providing a secure path from protected modules to hardware in chapter 5. The security and performance of this implementation was evaluated in chapter 6. Now we will implement a Secure Application that makes use of our framework, more specifically, a one-time password generator. We will start out by explaining the need for one-time passwords. In section 7.1, three algorithms used for generating these passwords are outlined. After introducing one-time passwords, we will briefly outline the implementation (section 7.2) along with a discussion (section 7.3).

Many online services make use of static passwords to authenticate a user. This kind of authentication holds significant risk if it is used to protect sensitive data. Once a static password has been compromised, it can be used over and over by the attacker to authenticate as the legitimate user. If, for example, a bank would protect a client's account in this way and the password would be compromised by a phishing attack, an attacker would have complete control over the client's money. A one-time password is a string that can be used to authenticate a user for a specific service only once. After the one-time password has been used, it cannot be used again. This prevents an attacker from re-using a password that has been recorded earlier. One-time passwords can be generated by a small electronic device, by software running on a PC or smartphone, or is sometimes distributed as a list on paper. If a one-time password somehow gets compromised, the attacker can use this password only once and, depending on the specific implementation, only for a very short time-period. The one-time password schemes outlined here can also be used for mutual verification. This allows the client to be sure he is talking to a genuine server, and the server is able to verify the client's token. Using a password together with a one-time password is called *Two-Step Authentication*. When combined with, among others, a fingerprint scan or a specific bank card, it is called *Two-Factor Authentication*.

For this reason, most banks as well as some companies use one-time passwords to authenticate their clients or employees. Depending on the use, one-time password token generators with or without built-in card readers can be used. VASCO is currently the world leader in providing Two-Factor Authentication services.

7.1 Algorithms

We will now go into more detail concerning three RFCs that have been proposed as standards for one-time password token generation and validation:

- HMAC-Based One-Time Password Algorithm (HOTP - RFC4226) [12]
- Time-Based One-Time Password Algorithm (TOTP - RFC6238) [34]
- OATH Challenge-Response Algorithm (OCRA - RFC6287) [33]

HMAC-Based One-Time Password Algorithm With the HMAC-Based One-Time Password Algorithm (HOTP), the generated value is based on a counter that must be synchronized between the generator (client) and the verifier (server). When the client generates a new HOTP value, it first increases its counter and calculates the token based on the new counter value. When the server receives a token, it validates it by calculating the correct HOTP value for its counter. If the received token is the same, the token is validated and the server increases its counter. Since the counter increases monotonically, each one-time password is based on a different counter value. This ensures that an intercepted token cannot be used twice.

Because the client increases its counter every time it generates a HOTP token and the server only increases its counter when a token is validated, the counter of the client sometimes can be higher than the counter of the server. This can cause the validator to reject a valid one-time password since its counter is lower and thus expects a different token. To mitigate this problem, the server has a specific look-ahead window (s). When the server receives a different value from the one it calculated and has counter value c, it will validate the token for $c+1$, $c+2$, ... until the token is validated or until value $c+s$ has failed. If for a certain value $c+n$ the token is validated, the counter on the server is set to $c+n+1$. The HOTP RFC does not specify what happens if the difference between the client counter and the server counter becomes larger than s, a new HOTP generator can be deployed to the user, or the current generator can be manually resynchronized by the verifying party.

Time-Based One-Time Password Algorithm The Time-Based One-Time Password Algorithm (TOTP) uses a timestamp as the shared value between the client and the server. This means that the token generator needs an internal clock or access to an external clock (e.g. online) that is synchronized with the verifier. The default size of a time-step is 30 seconds. This means that time is used as a counter that is increased automatically every half minute. Due to user or network delay, a token that is used at the end of a time-step might not be valid by the time it reaches

the verifier. Usually the verifier will also accept a token from the previous time-step. Since a user can only authenticate himself once per time-step, the time-step should be small enough. TOTP is internally very similar to HOTP, since the time-step can be viewed as a counter. However, we will see that it results in very different security properties.

OATH Challenge-Response Algorithm OATH Challenge-Response Algorithm (OCRA) is a generalization of the HOTP algorithm where the generated token is not solely based on a counter value and the shared key. In this scheme, the verifier sends a randomly generated challenge. The client will compute a response based on a shared key, the challenge and optionally among others a password, a counter and a time-step.

Differences between HOTP, TOTP and OCRA An HOTP token remains valid until a user uses it to log in or until a user logs in with a more recent token. If a user rarely uses a token, it can potentially stay valid for a long time. The security benefit of using TOTP instead of HOTP is that a compromised TOTP token will automatically become invalid after a certain time-step, even if it was never used. Because HOTP requires a synchronized counter between the client and the server, it may not be practical to use it in situations where many tokens may be generated, but not used to log in. In this case, the client counter is likely to drift too far apart from the server counter. When this happens, the client's generator needs to be replaced or some sort of manual counter resynchronization needs to be done.

TOTP on the other hand requires a synchronized clock between the client and the server. If the client is connected to the Internet, it can retrieve time securely from a remote server, if the client is offline, it needs an internal clock. Since the two clocks can drift apart, the verifier needs to check a given number of time-steps backward and forward.

And finally, to use OCRA, a synchronized value between the client and server is not strictly necessary but the RFC also includes the option of using a shared counter or time-step. So in situations where having a synchronized value is not practical, OCRA can be used.

7.2 My Work

As a proof of concept, a one-time password generator was implemented as a SApp in a Fides Self-Protecting Module. This token generator supports all three RFCs outlined in section 7.1. All input and output is done in a secure fashion using the implemented framework. It requests the users' PIN by displaying a message on the secure screen. The user then enters his PIN on the secure keypad and it is sent to the PC SIOC which passes it on to the one-time password generator protected module. The protected module calculates a response based on the PIN and displays it on the secure screen. Since the one-time password generator is implemented as a SApp and input/output is provided through our secure framework, strong guarantees can be

given that no other process running on the system, including the operating system, can interfere during the computation of a token. Also, since in some cases the token is based on a PIN code, it can be entered in such a way that it cannot be intercepted by malware running on the system.

7.3 Discussion

We showed here that it is practical to implement a one-time password generator in a protected module using our secure I/O framework. Implementing other security sensitive processes inside a protected module architecture is feasible as long as the process itself is relatively small or as long as the process can be split up into smaller components. Careful thought needs to be given as to what parts of a process are required to run inside an isolated environment and what parts do not need to be trusted.

In our use case, we decided to implement everything into an SPM. This makes sense, since a one-time password generator is relatively small in size and all input and output is delegated to the secure I/O framework.

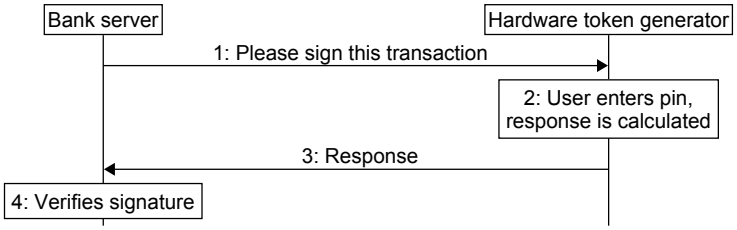
It is important to note that we don't aim to replace the conventional hardware one-time password generators, like the Vasco DIGIPASS, with our implementation. One-time passwords are a means to provide a user with a password to authenticate himself once. A better replacement would be a *bank account authenticator module*. It would provide an SSL/TLS connection from the bank to the authenticator module. A challenge can then be sent to this module by the bank. By entering his PIN, the user signs this challenge and sends this as a response to the bank. By utilizing the remote attestation possibilities that protected modules offer, the bank can also verify that the system is in a valid state before sending a challenge. One-time passwords are not needed anymore in this scenario.

Using hardware token generators, a user can still be tricked into entering the one-time password or his digital signature into a phishing site. Since not all users of hardware token generators necessarily know what the information they are entering means, they can be convinced to sign a transaction set up by an attacker. Since a protected module usually has far greater computational power and is able to make a secure connection to the bank, it is possible to automate some security checks and give the user more feedback into what he is signing exactly. Also this module could perform so-called *mutual authentication*. To explain this in more detail, the different protocols for one-way authentication with a regular hardware token generator and mutual authentication with a bank account authenticator module will be compared.

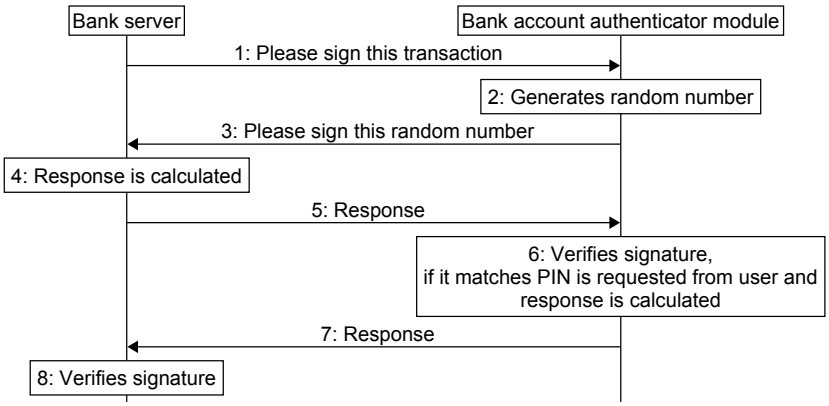
One-way authentication In step 1 of Figure 7.1a, the bank requests a user to sign a transaction. The PIN is entered on the one-time password generator (step 2) and the token is sent to the server (step 3). If the signature is valid, the transaction is authorized (step 4).

Mutual authentication In step 1 of Figure 7.1b, the bank requests a user to sign a transaction. The module will now authenticate the bank server by generating a random number and requesting the server to sign it (steps 2 and 3). The server signs this number (step 4) and responds to the module (step 5). The authenticator module verifies the signature. If it is valid, the PIN will be requested (step 6) and the transaction will be signed (step 7).

Mutual authentication ensures that a malicious server would be unable to trick a user into signing a transaction.



(a) One-way authentication



(b) Mutual-authentication

Figure 7.1: Authentication difference between one-time password generators and bank account authenticator modules

Another important note is that neither hardware token generators nor bank authenticator modules implemented in a PMA provide complete security against every type of attacker. We want to stress that, while PMAs can increase security and usability significantly, it is always important to be aware of the exact security guarantees an implementation gives and be aware of possible attacks against it. An attacker that is able to compromise the kernel might gain access to a camera or microphone connected to the system. This attacker could then try to watch the user entering his PIN or try to listen to the keystrokes through the microphone. This type of attack is beyond the scope of this thesis but might leak the PIN whether the user is using a hardware token generator or an authenticator module.

Chapter 8

Conclusion

In this chapter we will reflect on the work that was done in this thesis and place it in the context of the present day. Also, we will look at what protected module architectures might mean for the future.

We showed that applications can be run on a largely compromised system in an isolated fashion by using PMAs. By creating a secure path, we showed that these applications can interact with special hardware in such a way that the confidentiality and integrity of the data exchanged can be guaranteed. A framework, providing applications with the means of setting up a secure path with secure hardware was made. As a proof of concept, a one-time password generator using Fides and Sancus as PMA, was implemented using the framework. We evaluated the security of these implementations and showed that it is indeed possible to provide strong safety guarantees in the presence of a powerful attacker. Also, the performance of the implementation was evaluated. We concluded that the overhead, caused by isolating software modules from the rest of the system and using encryption to secure I/O data, is acceptable.

Hardware protection The PMAs that are currently available usually rely on a small hypervisor to provide memory isolation. An advanced attacker with physical access to the system might still be able to compromise the security of a protected module by launching a hardware attack against it. He can, for example, simply disconnect the memory and read out all data in it. He may also probe the memory bus and sniff all information coming through. The upcoming Microsoft Windows 10 operating system is said to include a PMA based on the Microsoft Hyper-V hypervisor, and thus will likely be vulnerable to hardware attacks. PMAs that provide isolation support through the system chipset, like Intel SGX, also protect against these advanced hardware attacks. Since, with these systems, all data that leaves the chipset is automatically encrypted and the key never leaves the CPU, probing the system memory will only show encrypted data.

Digital Rights Management In addition to banks, movie studios and record labels are very interested in upcoming PMAs that might provide applications with secure input and output. A huge problem for these companies in the digital age is handling piracy. Content that is shown on a screen or played through the sound card of a commodity PC can easily be intercepted and recored without any loss of quality. These copies can than be distributed without any control from the content creators. The set of technologies that are being created to mitigate this problem is called Digital Rights Management (DRM). Intel Insider, and previously Intel Protected Audio Video Path (PAVP) are technologies that allow for a media file to be played on a commodity PC in a controlled and secure way. More recently, Microsoft announced their PlayReady 3.0 DRM system. This system should allow content to be played on chips from Intel, AMD, Qualcomm and Nvidia [23]. The details of the inner workings of Intel Insider and Microsoft PlayReady 3.0 are not known as for now, but it is very likely that the media will be distributed as an encrypted file. The decryption of the data happens inside of the graphics chipset. This way, no software running on the PC should be able to record the media.

Future It is obvious that more and more companies and individuals recognize the need for the protection of processes running in an untrusted environment. Different technologies for reaching strong security guarantees, be it for banking applications, DRM or others, are being released regularly. For now, these technologies might still lack widespread support, but they are becoming more advanced each year.

It can be expected that within a few years technologies like Intel SGX will be available in commodity PCs. The isolation guarantees, remote attestation and secure input/output will allow remote parties to place a significant trust in home PC systems or servers. This way significantly greater amounts of important data, whether privacy, piracy or economically sensitive, could be processed on systems that are completely untrusted as for now.

Appendix A

Performance testing results

	Encryption/Decryption, No I/O		
Nr. chars	Nr. Cycles	Seconds	Chars/sec
25	123282030	**0.056**	446.13
50	286846735	**0.130**	383.48
100	518229224	**0.236**	424.52
200	1067881136	**0.485**	412.03
250	1321301362	**0.601**	416.26
300	1330482440	**0.605**	496.06
400	1838535395	**0.836**	478.64
700	3290236490	**1.496**	468.05
1000	4742337586	**2.156**	463.91
2000	9351812321	**4.251**	470.50
4000	18847213515	**8.567**	466.91
10000	45538665095	**20.699**	483.11

Figure A.1: Encryption/decryption performance in Fides

Nr. chars	Encryption/decryption enabled				Encryption/decryption disabled			
	Nr. Cycles	Seconds	Chars/sec	RTT (ms)	Nr. Cycles	Seconds	Chars/sec	RTT (ms)
25	25241205794	**11.473**	2.18	458.93	2403121526	**1.092**	22.89	43.69
50	27814994799	**12.643**	3.95	252.86	4878752853	**2.218**	22.55	44.35
100	33020040526	**15.009**	6.66	150.09	9787562144	**4.449**	22.48	44.49
200	43040173033	**19.564**	10.22	97.82	19558151975	**8.890**	22.50	44.45
250	48170850368	**21.896**	11.42	87.58	24421086864	**11.100**	22.52	44.40
300	53247606602	**24.203**	12.39	80.68	29325061010	**13.330**	22.51	44.43
400	63419467519	**28.827**	13.88	72.07	39120779451	**17.782**	22.49	44.46
700	93933925280	**42.697**	16.39	61.00	68468214708	**31.122**	22.49	44.46
1000	124479690402	**56.582**	17.67	56.58	97840447077	**44.473**	22.49	44.47
2000	226230735045	**102.832**	19.45	51.42	195703015363	**88.956**	22.48	44.48
4000	429654823125	**195.298**	20.48	48.82	391422009811	**177.919**	22.48	44.48
10000	1040100372344	**472.773**	21.15	47.28	978597492364	**444.817**	22.48	44.48

Nr. chars	Sancus on, Fides off, enc/dec disabled				All protection disabled			
	Nr. Cycles	Seconds	Chars/sec	RTT (ms)	Nr. Cycles	Seconds	Chars/sec	RTT (ms)
25	2435598204	**1.107**	22.58	44.28	1310993996	**0.596**	41.95	23.84
50	4899095478	**2.227**	22.45	44.54	2644170190	**1.202**	41.60	24.04
100	9782358214	**4.447**	22.49	44.47	5256136710	**2.389**	41.86	23.89
200	19577855530	**8.899**	22.47	44.50	10631502540	**4.833**	41.39	24.16
250	24449408784	**11.113**	22.50	44.45	13217282168	**6.008**	41.61	24.03
300	29327893497	**13.331**	22.50	44.44	15933340312	**7.242**	41.42	24.14
400	39104107206	**17.775**	22.50	44.44	21102710731	**9.592**	41.70	23.98
700	68471373708	**31.123**	22.49	44.46	37063822244	**16.847**	41.55	24.07
1000	97837913569	**44.472**	22.49	44.47	52806234860	**24.003**	41.66	24.00
2000	195701809629	**88.955**	22.48	44.48	105643199104	**48.020**	41.65	24.01
4000	391406340646	**177.912**	22.48	44.48	211288054029	**96.040**	41.65	24.01
10000	978525756049	**444.784**	22.48	44.48	528281719225	**240.128**	41.64	24.01

Figure A.2: Performance testing with secure I/O

Appendix B

Scientific article

See next page.

Extending Protected Module Architectures with a Secure I/O Framework

Dennis Frett
KU Leuven

Prof. dr. ir. F. Piessens
KU Leuven

Dr. R. Strackx
KU Leuven

Abstract

A computer system that is compromised by a software bug or an attacker could cause applications running on it to behave in unintended ways. For this reason, protected module architectures were developed. They allow software modules, called protected modules, to be run isolated from the rest of the system, and offer a third party the possibility to remotely verify the state of a system, a procedure called remote attestation. These protected module architectures, however, still have no way to securely interact with system hardware, thus limiting their practical use. This project designs and implements a framework allowing protected modules to access hardware in such a way that confidentiality and integrity of the data are ensured. Our results show that it is possible to implement this framework providing strong security guarantees and acceptable performance overhead.

1. Introduction

In our modern world we rely on electronic devices constantly to make our lives easier. Many of these devices perform tasks that can be deemed sensitive, private or even dangerous. A malfunctioning medical device or vehicle board computer can cause health damage or even death. Also, they might process personal information like passwords and medical data.

The collection of components that we need to trust in a system in order for it to hold up its security guarantees is called the *trusted computing base* (TCB) [12]. If any part of the TCB is exploitable or compromised it can mean that a part or all of the sensitive information in a system is compromised. Also, the device can behave in unintended ways, causing great harm. In general it is true that a larger TCB increases the chances of introducing an untrustworthy component. A smaller TCB makes it easier to offer reliable security.

As a way to drastically decrease the TCB of a system, protected module architectures (PMAs) were developed. Examples include Flicker [8], TrustVisor [9],Fides [13] and Sancus [11]. A PMA allows software modules, called protected modules, to be executed in isolation from the rest of the system. On a typical commodity PC the TCB for running an application in a secure way would include the application code itself, among others, the operating system, hardware device drivers, main memory and the processor. Modern operating systems are developed by teams of hundreds to thousands of people and consist of millions of lines of code [6][7]. When a PMA is in place, the TCB can be reduced to the protected module itself, the PMA code and some hardware. The operating system, among others, can thus be excluded from the TCB. This means that malicious code, even if it has kernel-level access, should not be able to interfere with the execution of a protected module. The exact trust model of a PMA depends on the implementation.

Lots of software needs access to hardware devices to perform their tasks. User input needs to be entered through a keyboard and results need to be shown on a screen, for example. Currently PMAs use the operating system to connect to hardware in an unprotected way. To increase the security and usefulness of protected modules we will design a framework that provides protected modules with a generic way to obtain secure input and output (I/O) from secure hardware devices. As a proof of concept we will develop this framework and implement a one-time password generator. We will also evaluate the performance and security of our solution.

2. Security Properties and Attacker Model

We want to develop a framework that allows protected modules to communicate with secure hardware providing strong security guarantees. First, *confidentiality* of the transfered data must be ensured. This means that an unauthorized party must not be able to intercept and access to the information being communicated. Second, *integrity* of this data must be protected. An unauthorized party must not be able to alter the com-

municated data in any way.

The attacker we want to defend against has a few strong capabilities: he has kernel level access, he is able to deploy his own protected modules and he is able to add his own virtual or physical devices to the system. This means he can execute tasks at the kernel privilege level. Also he can inspect, modify and execute all the memory the operating system has access to. An attacker is not able to break cryptographic primitives and does not have physical access to the hardware.

We distinguish between two types of trust. Components, such as secure hardware and protected modules, that are able to leak some information, but only if it is directly available to them are *partially trusted*. Partially trusted components are unable to compromise security of other components. Components that, when they cannot be trusted, compromise the security of the whole system are called *fully trusted*. Examples of fully trusted components include the CPU and the PMA code. This can be seen schematically in Figure 1. Depending on the specific implementation, some details of the trust model may differ. In hypervisor-based implementations, like Fides, the main memory is trusted since it contains the secrets belonging to protected modules in an unencrypted form. In these implementations, also a hypervisor must be trusted. In hardware-based implementations, like Intel SGX [3][10], the information that leaves the CPU can be encrypted. In that case, the memory does not need to be trusted.

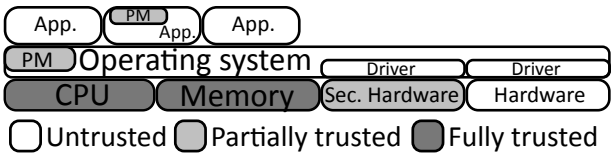


Figure 1. Trust model of the system

3. Model

In this project, a framework that provides protected modules with a means of secure input and output was designed and implemented. The connection between a protected module and secure hardware is called a *secure path*. This secure path should provide the security guarantees outlined earlier. It consists of a software Secure I/O Controller (*PC SIOC*) (part 1 on Figure 2), secure hardware devices (part 2) and Secure Applications (*SApps*) (part 3).

The PC SIOC is responsible for mediating all access to secure hardware from SApps. It maintains a list of all trusted secure hardware devices along with their symmetric encryption keys in its protected memory, as well as a list of trusted SApps that are allowed access to trusted secure hardware. The PC SIOC interacts with the operating system (part 4) to connect to trusted secure hardware. A clean system starts out with a PC SIOC that trusts no hardware device and no SApp. The PC SIOC accepts data from a trusted SApp, encrypts it, and sends it to a secure hardware device as secure output. Conversely, it accepts encrypted data from secure hardware, decrypts it, and sends it to a trusted SApp as secure input. The PC SIOC is the heart of the framework.

Each secure hardware device (part 2a) is equipped with a hardware Secure I/O Controller (*Hardware SIOC*) (part 2b). The Hardware SIOC of secure output devices is responsible for receiving encrypted data from the PC SIOC, decrypting it and displaying it on the hardware. For secure input devices, it should encrypt the data and send it to the PC SIOC. Secure hardware can also be used as regular, unsecured hardware. In this case, the Hardware SIOC is bypassed and no encryption or decryption is done. Regular, non-protected applications can then interact with the hardware in an unsecured way. An LED on the hardware itself should indicate whether it is running in secured or unsecured mode. This allows the user to easily verify if a secure path is in place, much like the lock symbol placed next to a URL in a web browser whenever a connection is secured by SSL/TLS. When the LED is on, we say that the hardware is running in *secured mode*, otherwise it is running in *unsecured mode*.

Part 3 on the figure are the SApps that requires secure I/O from a hardware device. A trusted SApp can request access to one or more trusted secure hardware devices. The requested devices will then run in secured mode, all other hardware devices connected to the system will run in unsecured mode. The SApp gains complete control over the hardware devices in secured mode. We now call the SApp *active*. This means that two SApps can never control secure hardware devices simultaneously. In order to prevent phishing attacks, the active SApp must be clearly distinguishable from other processes or non-active SApps running on the system. This can be achieved by having the user's secure screen draw a special border around all the windows of the SApps graphical user interface and requiring the SApps' windows to be on top of every other window on the system. Also, a unique SApp-name must be shown in the title bar of each window. This allows the user to easily verify which SApp is currently active and which secure hardware devices it has access to. An active SApp without a GUI can be listed in the GUI of the PC SIOC. Because it is clear which SApp has exclusive access to the hardware, a trusted SApp controlled

by an attacker is unable to trick the user into thinking the attacker SApp is a trusted bank module and entering his PIN. These requirements might be overly restrictive, and future implementations of secure paths could relax these.

We say that a secure path is completely set up once the PC SIOC trusts at least a secure keyboard, a secure screen and a SApp.

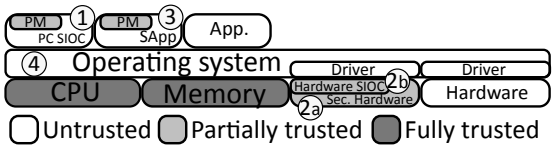


Figure 2. Secure path

4. Implementation

The framework outlined in the previous section along with a one-time password generator SApp was implemented as a proof of concept. The security guarantees are achieved using a protected module architecture to ensure that SApps can run isolated from the rest of the system. To protect the integrity and confidentiality, all sensitive data leaving the memory protected by the PMA is encrypted and sent along with a message authentication code (MAC). Encryption and decryption only takes place in the PC and Hardware Secure I/O Controllers. In our implementation, an authenticated symmetric encryption algorithm using SPONGENT [2] as an underlying hash function was used [1]. SPONGENT is a lightweight hash function, designed for use in embedded environments. The framework was implemented on Dell Inspiron 7520 SE laptop with an Intel i7-3632QM processor. All code running on the PC was written using the C programming language. Implementations on the FPGA used as Hardware SIOC were done in C++ with exception of the memory-mapped I/O (MMIO) code, which was written in assembly.

The architecture of the system is shown in Figure 3. The meanings of *fully trusted* and *partially trusted* are explained in the previous section. The implementation details of the different components of our framework are explained in the following paragraphs.

Protected module architecture A PMA is used to isolate memory segments from the rest of the system. This way our TCB is reduced significantly. Software modules that need to be protected on the PC are isolated with the Fides PMA. Software running on the secure hardware is protected with Sancus. This means that on the PC, protected modules are implemented as a Fides

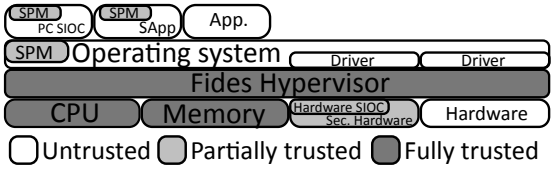


Figure 3. Architecture of the implementation

Self-Protecting Module (SPM). In the secure hardware, the Hardware SIOC is implemented in a Sancus Software Module (SM).

Secure Applications SApps consist of two parts: an unprotected and a protected part. The unprotected part is responsible for tasks that are not security sensitive. The unprotected code can be used, among others, to interact with hardware and process encrypted data. The protected part can handle security sensitive tasks, therefore it is implemented as a protected module, in our case a Fides SPM. Communication from the unprotected code to the protected module happens through well-defined interfaces. Since malware is possibly able to compromise code running in the unprotected part, all input coming from unprotected code should be deemed potentially compromised and should be validated.

Secure I/O Controllers The PC SIOC was implemented as a SApp. This means it consists of unprotected and protected code. Unprotected code is responsible for receiving and processing encrypted data. It can receive encrypted data from the operating system, coming from a Hardware SIOC. The unprotected code then processes this data and passes it to the protected code. Conversely, it can receive encrypted data form the protected code, and pass it on the the Hardware SIOC. The PC SIOCs protected code then is responsible for decrypting and verifying incoming data, and for encrypting outgoing data. The Hardware SIOC will be explained in more detail in the next paragraph.

Secure hardware The Hardware SIOC works analogously to the PC SIOC. It was implemented as a Sancus Software Module on a Texas Instruments MSP430 FPGA (clocked at 20Mhz). It was extended with a Digilent PmodKYPD keypad and a Digilent PmodCLS screen. We also protect against malware that might be running on the FPGA itself by implementing MMIO protected modules to interact with the keypad and screen. These MMIO modules verify they can only be called by the Hardware SIOC, thus ensuring that malware is unable to interact with the keypad or screen. Since the MMIO modules are implemented as

protected modules, their memory is isolated from the rest of the system. Conceptual, the rest of the Hardware SIOC works analogously to the PC SIOC; it is responsible for receiving encrypted data from the PC SIOC and decrypting it as well as receiving unencrypted input from hardware, decrypting it and sending it to the PC SIOC.

Communication Communication between SApps and PC SIOC happens using the Fides secure communication channel. Encryption and authentication of this data is handled by Fides itself. Communication between the PC SIOC and the Hardware SIOC is done using the SPONGENT-based encryption algorithm. Each message is encrypted along with a 32 bit nonce, implemented as a counter. This means that no two messages containing the same payload will be encrypted to the same ciphertext, given that the nonce doesn't wrap. For messages coming from the PC SIOC, the first bit of the nonce is set to 0. Conversely, for messages coming from the Hardware SIOC, the first bit is 1.

5. Evaluation

We will verify that our framework is secure and offers our required security guarantees, and that its performance is acceptable.

5.1. Security evaluation

Security of an existing secure path will be evaluated along with the security of setting up this secure path. For the framework to be secure, it needs to guarantee *confidentiality* and *integrity* of our I/O data.

Existing secure path Given that a secure path was set up, the PC SIOC trusts a secure input device, a secure output device and a SApp (see Figure 2).

Sensitive data can reside in protected or in unprotected memory. Also the PC SIOC and the Hardware SIOC keep a copy of their symmetric encryption keys in protected memory. Our framework makes sure that sensitive data in unencrypted form always resides in protected memory. Sensitive data in unprotected memory is always encrypted with the symmetric encryption key. Our protected module architecture protects all data and code in protected memory. This means our key and unencrypted sensitive data cannot leak to an attacker. Without the encryption key, an attacker is unable to gain access to the cleartext. This gives a strong guarantee that our framework ensures the confidentiality of sensitive data.

By sending a MAC along with the encrypted data, tampering with the ciphertext can be detected. A MAC can be seen as a hash based on cleartext that is being encrypted and the encryption key. The receiving party can decrypt the ciphertext and calculate the MAC for the cleartext. If this calculated MAC matches the MAC that was received, the receiver can be sure that the ciphertext was not tampered with. An attacker that tampers with the ciphertext will be unable to compute a valid MAC for the corresponding cleartext since he doesn't have access to the encryption key. This way the integrity of our data is ensured.

If we implement our secure keypad such that it sends a character immediately when a key is pressed on the keypad, it would likely be susceptible to side-channel attacks. Malware can not see *what* key was pressed but detect *when* a key was pressed. By carefully measuring and analyzing the timing between key presses, an attacker might infer what keys were pressed (timing attack) [4]. Defenses against these attacks exist: we can flood the channel with dummy key presses or delay the sending of a character. These solutions make it significantly harder or impossible for an attacker to learn anything about the entered characters. This type of attack, however, is out of scope of this article.

An attacker might also try to trick a user into entering sensitive data into a SApp he controls. He can, for example, present his malicious SApp as the SApp used for processing bank transactions and request the users' PIN. By making sure every SApp has a unique name chosen by the user, as explained in section 3, and showing this name at all times, we can validate whether we are entering our sensitive information into the right SApp.

Setting up a secure path We will now show that setting up a secure path can also be done securely. Setting up a secure path correctly means adding a secure input device, a secure output device and a SApp securely.

Adding a new secure I/O device is always initiated by the user by pressing a hardware button. This means an attacker is unable to initiated the protocol, since he doesn't have physical access to the system. A user plugs in the new hardware devices, which is untrusted for now and presses the button. This will initiated the Diffie-Hellman key exchange protocol between the PC SIOC and the Hardware SIOC of the new hardware device. If no man-in-the-middle attack took place at this point, both SIOCs end up with the same symmetric encryption key. If an attacker was present at this step and was able to modify the messages of the protocol, he might force the situation that both SIOCs end up with different keys. The attacker knows both of these keys and is then able

to read and modify all messages sent between the PC SIOC and the Hardware SIOC by decrypting the messages with the correct key, maybe modifying the message, and re-encrypting it with the other key. To mitigate this attack, we check that the encryption keys of the PC SIOC and the Hardware SIOC match after the key exchange. If they match, no attack took place, and the confidentiality of the messages can be ensured. If they do not match, the protocol was potentially attacked and the SIOCs should not trust each other.

To check whether both encryption keys match, a solution to the socialist millionaire problem [5] is used, we call this the socialist millionaire protocol. This protocol can be used by two or more parties to verify whether they have access to the same shared secret over an insecure channel. This protocol does not reveal any other information to the other party except the equality of the shared secret. An eavesdropper or an active attacker learns nothing, not even if the two secrets match. The active attacker can also not influence the protocol, except make it fail. As a shared secret we use a *System ID* along with the symmetric encryption key we got after the key exchange. This System ID a unique string that is accessible to the PC SIOC in a secure way and is also physically printed on the users' system. Since our attacker does not have physical access to the system, he cannot see the System ID. When the user adds a secure input device for the first time, he will be requested to enter the System ID on it after the key exchange. Both the PC SIOC and the Hardware SIOC will verify whether the other party has the same System ID and the same encryption key. If both of these match, we know that no man-in-the-middle attack took place and the new input device can be trusted. If they do not match we know that the user entered the wrong System ID or the key exchange was compromised. Either way, the protocol fails. Since the attacker does not know the System ID, he cannot successfully complete the protocol after he compromised the key exchange.

Once a secure keyboard is trusted by the PC SIOC, new hardware devices are added as follows: the user plugs in the new device and presses the hardware button. Keys will be exchanged between the new Hardware SIOC and the PC SIOC. The user will then be prompted to enter the *Device ID* on the previously trusted secure keyboard. This Device ID is a unique string printed on each secure hardware device. By entering this Device ID on the trusted keyboard, the user makes explicit the device he wants to add. Since data coming from the keyboard is confidentiality and integrity protected, an attacker cannot interfere with the Device ID being sent. The user can be sure the correct device was added, and an attacker is unable to add his own devices.

Adding a new trusted SApp An attacker must not be able to deploy his own SApp on a system and have the PC SIOC trust it without the user explicitly permitting this. Whenever a new, still untrusted, SApp requests access to secure I/O, the user will be presented with a random string on his secure screen. Entering this random string on the secure keyboard makes the PC SIOC trust the new SApp and grants it access to secure I/O. An attacker is unable to intercept or modify this random string. This way a user explicitly permits access to a new SApp and a remote attacker will be unable to do so.

5.2. Performance evaluation

The performance of our implementation, the overhead caused by protecting the SIOCs, the protected modules and encrypting the I/O data was measured. All tests were done on a Dell Inspiron 7520 SE laptop with an Intel i7-3632QM processor clocked at 2.2 Ghz. Fides does not currently support multiple cores, so only one of the four cores was enabled.

The system was tested by sending 10,000 characters, one by one, to the PC SIOC. It then proceeds to encrypt these, send them to the Hardware SIOC. When the Hardware SIOC receives an encrypted message, it decrypts it and displays the character on the secure screen. The character is then re-encrypted and sent back to the PC SIOC. Once the PC SIOC receives it, it decrypts it. Once a character has made a complete round-trip, the next character is sent. The total time it takes for this test is divided into five parts: the time it takes to transfer messages between the SIOCs, the overhead caused by the Sancus PMA, the overhead caused by the Fides PMA, the time it takes to encrypt/decrypt a message on the PC and the time it takes to encrypt/decrypt a message on the FPGA. The results of this test can be seen in Figure 4. The largest part of the time it takes goes to transferring messages between both SIOCs. Communication with secure hardware is done over an UART connection. Communication over an USB connection might significantly reduce the time this takes. Also we note that encryption on the PC takes longer than on the FPGA. This is likely caused by a more efficient implementation on the FPGA.

We conclude that our framework incurs acceptable overhead and that performance improvements can be made relatively easy.

6. Related Work

One of the building blocks that offer significant security guarantees to this framework are protected module

	Time (seconds)	Time (relative to total)
Transfer overhead	240.128	50.79%
Sancus overhead	204.656	43.29%
PC encryption/decryption	20.667	4.37%
Hardware encryption/decryption	7.289	1.54%
Fides overhead	0.0326	0.01%
Total time	**472.7729**	**100%**

Figure 4. Relative time each part of the framework takes in the test

architectures. One of the first PMAs was Flicker [8]. To provide strong isolation guarantees it made heavy usage of the slow TPM chip. This caused significant overhead to the system. This project makes use of Fides [13] to implement protected modules in a commodity PC and Sancus [11] to implement secure hardware, which make use of Program-Counter Based Access Control (PCBAC). Protected modules in PCBAC can be seen as continuous memory segments. A hypervisor (with Fides) or hardware (with Sancus) inspects the program-counter (or *instruction pointer* in Intel x86 processors) every time a jump is made into the memory of a protected module from outside of it. Its memory is protected by disallowing other processes to read most memory belonging to a protected module. The PCBAC system also ensures that a protected module can only be executed from a specific well-defined entry point. This gives a protected module in this system complete control over its own secrets and control flow, which provides protected modules with significant security guarantees.

7. Conclusion

A framework was designed and implemented that provides protected modules, being isolated from the rest of the system, with the means to a secure path to hardware devices. Using this framework gives strong security properties. Kernel-level malware is unable to interfere with the inner workings and memory of protected modules. These modules are able to get input from, and send input to secure hardware in such a way that confidentiality and integrity of this data is provided. Malware should be unable to inject, modify or access data being transferred between the hardware and the protected module. By evaluating the security properties and performance of this framework, it was concluded that confidentiality and integrity of the data was indeed preserved and that the performance was acceptable.

References

[1] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7118 LNCS:320–337, 2012.

[2] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62(10):2041–2053, 2013.

[3] Intel Corporation. Software Guard Extensions Programming Reference. (September), 2013.

[4] San Francisco, David Martin, and Andrew Schulman. Proceedings of the 11 th USENIX Security Symposium. 2002.

[5] Markus Jakobsson and M Yung. Proving Without Knowing: On Oblivious, Agnostic and Blindfolded Provers. *Advances in Cryptology - CRYPTO'96*, pages 186–200, 1996.

[6] V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. 2005.

[7] David McCandless. *Knowledge is Beautiful*. William Collins.

[8] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. *EuroSys*, 42:315, 2008.

[9] Jonathan M Mccune, Ning Qu, Yanlin Li, Anupam Datta, Virgil D Gligor, Adrian Perrig, and Zongwei Zhou. TrustVisor : Efficient TCB Reduction and Attestation TrustVisor : Efficient TCB Reduction and Attestation. *Oakland*, 2010.

[10] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*, pages 1–1, 2013.

[11] Job Noorman, Bart Preneel, K U Leuven, Pieter Agten, Wilfried Daniels, Raoul Strackx, Christophe Huygens, Frank Piessens, Anthony Vanherrewege, and Ingrid Verbauwhede. Sancus : Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. *22nd USENIX Security*, 2013.

[12] J. M. Rushby. Design and verification of secure systems. *Proceedings of the eighth symposium on Operating systems principles - SOSP '81*, pages 12–21, 1981.

[13] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. *Proceedings of the 2012 ACM conference on ...*, pages 2–13, 2012.

Appendix C

Populariserend artikel

Zie volgende pagina.

Zijkanaal aanvallen

Dennis Frett

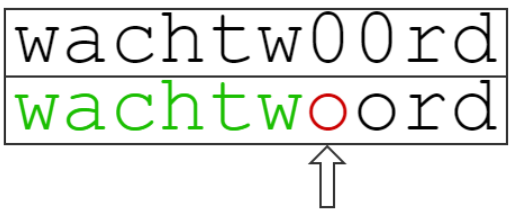
Zijkanaal aanvallen zijn cryptografische aanvallen op zowel elektronische als (elektro)mechanische systemen. Ze worden in het beveiligen van deze systemen vaak over het hoofd gezien. In dit artikel wordt daarom duidelijk gemaakt wat een zijkanaal aanval inhoudt en wat er aan gedaan kan worden.

Aan de hand van voorbeelden moet de werking van verschillende aanvallen duidelijk worden. Het is belangrijk te vermelden dat hiermee enkel een inleiding gegeven wordt en dat er niet getracht wordt om een exhaustieve lijst van mogelijk zijkanaal aanvallen te geven.

1 Wat is een zijkanaal aanval?

1.1 Eenvoudig voorbeeld

Stel - bij wijze van voorbeeld - dat we met een computer werken die een wachtwoord op de volgende manier nagaat op correctheid: het ingegeven wachtwoord wordt teken per teken vergeleken met het opgeslagen juiste wachtwoord. Als een karakter niet overeenkomt zal de computer direct reageren met *Fout!*. Indien alle karakters overeenkomen, zal hij *Correct!* weergeven (zie figuur 1). Als het juiste wachtwoord 10 karakters lang is en kan bestaan uit de 62 verschillende alfanumerieke tekens (26 hoofdletters, 26 kleiner letters en 10 cijfers), dan hebben we 10^{62} mogelijke wachtwoorden. Aan het antwoord van de computer (*Fout!* of *Correct!*) is enkel af te leiden of



Figuur 1: Wachtwoord wordt karakter per karakter vergeleken

het ingevoerde wachtwoord al dan niet correct is. Bestaat er echter een manier om meer informatie af te leiden, namelijk in welke mate het opgegeven wachtwoord fout is?

Aangezien we uit het antwoord van de computer zelf geen extra informatie kunnen afleiden gebruiken we een andere bron: *hoe lang duurt het vooraleer de computer dit antwoord geeft?*. Stel dat deze computer één teken per milliseconde kan testen op correctheid en het antwoord van de computer is *Fout!* na 7 ms, dan kunnen we afleiden dat het zevende karakter fout is. Aangezien het systeem direct stopt bij het eerste foute teken, weten we dus dat de zes eerste tekens juist zijn. Om dit systeem te kraken kunnen we dus de wachtwoorden `aaaaa`, `baaaa`, `caaaa`... invoeren tot het systeem er 2 ms over doet om *Fout!* te melden, bijvoorbeeld bij `waaaa`. Dan weten we dat het gezochte wachtwoord begint met de letter `w`. We herhalen dit nu met `waaaa`, `wbaaa`, `wcaaa`... om de tweede letter te vinden. Dit wordt herhaald tot alle letters van het wachtwoord gevonden zijn, en het systeem dus reageert met *Correct!*. Op deze manier kunnen we het wachtwoord karakter per karakter raden en moeten we maximaal $10 * 62$ mogelijkheden proberen, wat significant minder is dan 10^{62}.

Dit voorbeeld geeft een idee wat een zijkanaal aanval is: *we maken gebruik van informatie dat een systeem lekt, via een zogeheten zijkanaal, door de manier waarop het gebouwd (geïmplementeerd) is.* In het voorgaande voorbeeld kijken we naar de tijd die verloopt vooraleer we een antwoord krijgen. De zijkanaal aanval in dit voorbeeld noemen we een tijdsaanval.

Een aanval op dit systeem waarbij we gewoon elk mogelijk wachtwoord testen (10^{62} mogelijkheden) is geen zijkanaal aanval, want er wordt nergens gebruik gemaakt van informatie die lekt door de implementatie.

1.2 Akoestische aanval

Een akoestische aanval is een aanval waarbij informatie verkregen wordt door te luisteren naar geluiden die een systeem voortbrengt. Het eerste gedocumenteerde voorbeeld van een zijkanaal aanval is een akoestische aanval tegen een rotormachine [3]. Rotormachines werden gebruikt om gevoelige berichten te versleutelen en te ontsleutelen met als doel deze geheim te houden. Het bekendste voorbeeld hiervan is de Enigma machine (zie figuur 2), die tijdens de Tweede Wereldoorlog gebruikt werd.

Werking Een rotormachine bestond over het algemeen uit de volgende onderdelen:

- **Toetsenbord**
  Het toetsenbord bestond meestal uit 26 of 25 (‘W’ werd dan ‘VV’) letters en had veel weg van het toetsenbord van een typemachine.

- **Rotoren**
  Dit waren schijven met aan beide kanten voor elke letter op het toetsenbord een elektrisch contactpunt (zie figuur 3). In de rotor waren de contactpunten aan beide zijden willekeurig met elkaar verbonden. De letter ‘A’ was dan bijvoorbeeld aangesloten op ‘D’ (zie figuur 4).

- **Manier om gecodeerde letter weer te geven**
  Letters werden bijvoorbeeld weergegeven op een paneel met evenveel lampjes als letters. Elk lampje stelde dan een letter voor.

- **Een stroombron**
  Meestal werd een batterij gebruikt als stroombron.

Het apparaat werd dan gebruikt om letter per letter een tekst te versleutelen. De gebruiker toetste de eerste letter van de tekst in op het toetsenbord, waardoor er een lampje oplichtte. Dat lampje stelde de eerste letter van de versleutelde tekst voor. Dit werd herhaald voor elke letter.

De toetsen waren schakelaars die de stroombron koppelde aan de lampjes via een elektrisch circuit dat door de rotoren heen ging (zie figuur 4). De rotoren hadden door hun interne bedrading als functie een letter om te zetten in een andere.

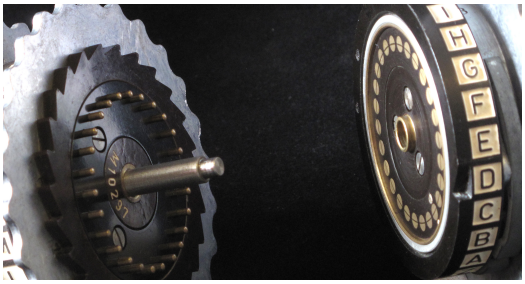
Om er voor te zorgen dat eenzelfde letter telkens in een andere letter omgezet zou worden, roteerde de eerste rotor één stap telkens een letter ingetoetst werd. Als de eerste rotor helemaal rond is, zal de tweede rotor een stap opschuiven. Op deze manier werd eenzelfde ingetoetste letter telkens vervangen door een andere letter. Om bijvoorbeeld de tekst ‘HALLO’ te versleutelen moest eerst de ‘H’ toets ingeduwd worden en de letter opgeschreven worden die oplichtte. Dit werd ook gedaan met de letters ‘ALLO’. De versleutelde tekst is dan bijvoorbeeld ‘YDMJL’. Merk op dat de letter ‘L’ twee keer naar een verschillende letter versleuteld werd, namelijk ‘M’ en ‘J’. Dit komt omdat de eerste rotor na elke toets één stap draaide. De begintoestand van de rotoren bepaalde dus wat de versleutelde tekst werd bij een bepaalde invoer. Om de tekst te kunnen ontcijferen moest de ontvanger ook een rotormachine hebben waarbij de rotoren in dezelfde beginpositie stonden. Deze begintoestand noemen we de encryptiesleutel. Om een onderschept bericht te ontcijferen moest men dus deze sleutel achterhalen.

Figuur 2: Een Enigma Machine met 3 rotoren in het Imperial War Museum, Londen

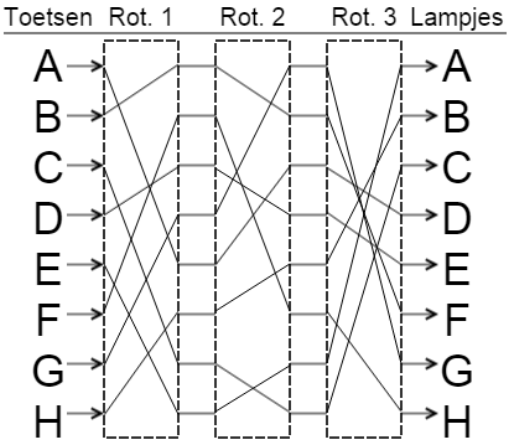
Aanval De Britse geheime dienst (MI5) wilde in 1965 de Egyptische Ambassade in Londen afluisteren, die gebruik maakte van een Hagelin rotormachine. De computerkracht die de Britten toen tot hun beschikking hadden was niet voldoende om de sleutel van de Egyptenaren te kraken. Daarom maakten ze gebruik van een akoestische zijkanaal aanval.

De MI5 plaatste in het geheim een microfoon in dezelfde ruimte als de rotormachine. Hierdoor konden ze luisteren naar het klikken van de rotoren van de Hagelin machine terwijl een boodschap versleuteld werd. Door dit geluid te analyseren konden ze afleiden wat de beginpositie van de rotoren was en hadden ze dus de encryptiesleutel in handen. Hiermee waren ze in staat jarenlang de communicatie van de Egyptische Ambassade af te luisteren [3]. Dus als zijkanaal werd hier het geluid van de elektromechanische machine gebruikt.

Rotormachines worden tegenwoordig niet meer gebruikt, maar akoestische aanvallen blijven zeer actueel. Er zijn aanvallen bekend waarbij de aanvaller met een microfoon luistert naar de geluiden die de elektronische componenten van een computer voortbrengt. Op deze manier kunnen onder andere encryptiesleutels achterhaald worden [1].



Figuur 3: Twee Enigma rotoren



Figuur 4: Schematisch overzicht van een Enigma machine

1.3 Warmteaanval

Ook de temperatuur van bepaalde onderdelen van een systeem kan gevoelige informatie lekken naar een aanvaller. Men kan bijvoorbeeld iemand zijn PIN code afleiden na het intoetsen aan een bank- of betaalautomaat.

Werking Wanneer een klant geld wilt afhalen of een betaling wilt goedkeuren doet hij dit meestal door het invoeren van zijn PIN code. Deze code wordt meestal met de vingers op een metalen of plastieken numeriek toetsenbord ingevoerd. Als de PIN code correct is, wordt de transactie door de bank goedgekeurd.

Aanval Enkele jaren geleden werd een aanval besproken om de PIN code van een gebruiker na het intoetsen aan een bankautomaat te achterhalen [2]. Dit deed men door te kijken naar de temperatuur van de toetsen (zie figuur 5).

Figuur 5: Warmtefoto van een numeriek toetsenbord net na het invoeren van de PIN code '`12345`'

Als een toets ingedrukt, zal deze opwarmen door de lichaamswarmte van de vinger. Met een infrarood warmtecamera is het mogelijk te detecteren welke toetsen nog warm zijn en dus net ingedrukt werden. Deze warmte verdwijnt langzaam, hierdoor is het ook mogelijk om af te leiden in welke volgorde dit gebeurde: de warmste toets werd het meest recent ingedrukt, de minst warme het eerst.

Doctoraatstudent Keaton Mowery en zijn team aan de University of California hebben onderzocht hoe praktisch zo'n aanval is. Ze ondervonden dat als men direct na het gebruik van de automaat een warmtefoto nam, de PIN code in 80% van de gevallen achterhaald kon worden. Na één minuut was dit nog 50% [2]. Warmtecamera's worden steeds betaalbaarder en zijn zelfs beschikbaar als toevoeging voor smartphones (bijvoorbeeld de FLIR ONE™ warmtecamera voor de Apple iPhone). Hierdoor wordt het uitvoeren van een warmteaanval voor iedereen betaalbaar.

2 Wat kunnen we ertegen beginnen?

Zoals reeds gezegd werken zijkanaal aanvallen met informatie die een systeem onbedoeld door zijn implementatie lekt. Daarom zal elke bescherming hiertegen er voor zorgen dat er minder informatie lekt of dat de gelekte informatie onbruikbaar is voor de aanvaller.

Tijdsaanval De tijd die het duurt voor een systeem een antwoord geeft zou geen informatie mogen lekken. Men kan het systeem zo ontwerpen dat het altijd reageert na een constante tijd. Deze tijd is dan onafhankelijk van de opgegeven invoer. Dit is geen gemakkelijke taak. Deze constante tijd moet lang genoeg zijn zodat het systeem hierbinnen al zijn berekeningen kan doen. Als gevolg hiervan moet de langst mogelijke tijd genomen worden die het systeem nodig kan hebben, de *worst-case* tijd. In ons eerste voorbeeld zou dit betekenen dat we een maximumlengte van wachtwoorden instellen, bijvoorbeeld 20 karakters, en dat het systeem dan altijd pas na 20 ms met *Fout!* of *Correct!* reageert. Op deze manier zal de tijdsduur geen informatie lekken aangezien het systeem nu altijd even lang nodig heeft om te reageren.

Akoestische aanval Beschermen tegen een akoestische aanval, zoals het afluisteren van een rotormachine of PC, is niet evident. Hiervoor zou men bijvoorbeeld op willekeurige momenten een klik kunnen laten horen die niet te onderscheiden is van een echte klik. Op deze manier kan een afluisterende aanvaller moeilijker afleiden wat de gebruikte sleutel is. Ook is het mogelijk om te zorgen voor voldoende achtergrondgeluiden terwijl men de rotormachine gebruikt. In dat geval wordt het moeilijker voor een afluisterende partij om de klikken te onderscheiden. Een derde manier is het toestel van betere geluidsisolatie te voorzien zodat de klikken minder duidelijk hoorbaar zijn.

Warmteaanval De warmteaanval op de bankautomaten kan vermeden worden door bijvoorbeeld de toetsen uit een materiaal te maken dat zeer snel opwarmt, maar ook zeer snel afkoelt.

3 Conclusie

Naast de hier vermelde voorbeelden bestaan zijkanaal aanvallen in zeer veel uiteenlopende vormen. Er kan gekeken worden naar de hoeveelheid stroom dat een apparaat gebruikt, naar de elektromagnetische golven die het systeem lekt, gegevens die in geheugen achterblijven nadat ze verwijderd werden,...

Zijkanaal aanvallen zijn zeer divers en maken gebruik van fouten in de implementatie in plaats van zwakheden in het algoritme. Ook is het vinden van tegenmaatregelen geen gemakkelijke taak. Daardoor worden zijkanaal aanvallen zeer vaak over het hoofd gezien en vormen ze een realistische en niet te onderschatten bedreiging.

Referenties

[1] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in CryptologyCRYPTO96*, pages 104–113. Springer, 1996.

[2] Keaton Mowery, Sarah Meiklejohn, and Stefan Savage. Heat of the moment: characterizing the efficacy of thermal camera-based attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 6–6. USENIX Association, 2011.

[3] Peter Wright and Paul Greengrass. *Spycatcher: The candid autobiography of a senior intelligence officer*. Dell, 1988.

Bibliography

[1] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. *Proceedings of the 2007 ACM workshop on Privacy in electronic society - WPES '07*, pages 41–47, 2007.

[2] Ittai Anati and Shay Gueron. Innovative technology for cpu based attestation and sealing. *Proceedings of the 2nd . . .*, pages 1–7, 2013.

[3] Apache. The Apache HTTP Server Open Source Project on Open Hub. https://www.openhub.net/p/apache.

[4] Adam Barth, Collin Jackson, and Charles Reis. The Security Architecture of the Chromium Browser. *Proceedings of WWW 2009*, 2008.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7118 LNCS:320–337, 2012.

[6] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62(10):2041–2053, 2013.

[7] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-Record Communication, or, Why Not To Use PGP. *Proceedings of the 2004 ACM workshop on Privacy in the electronic society (WPES '04)*, pages 77–84, 2004.

[8] Intel Corporation. Software Guard Extensions Programming Reference. (September), 2013.

[9] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.

[10] San Francisco, David Martin, and Andrew Schulman. Proceedings of the 11 th USENIX Security Symposium. 2002.

[11] Matthew Hoekstra, Reshma Lal, and Pradeep Pappachan. Using innovative instructions to create trustworthy software solutions. *Proceedings of the 2nd*

International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13, pages 1–8, 2013.

[12] Frank Hoornaert, David Naccache, Mihir Bellare, and Ohad Ranen. HOTP: An HMAC-Based One-Time Password Algorithm.

[13] ISO. Information technology – Security techniques – Information security management systems – Requirements. ISO 27001:2013, International Organization for Standardization, Geneva, Switzerland, 2013.

[14] Bart Jacobs and Frank Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[15] Markus Jakobsson and M Yung. Proving Without Knowing: On Oblivious, Agnostic and Blindfolded Provers. *Advances in Cryptology - CRYPTO'96*, pages 186–200, 1996.

[16] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM 2009: Formal Methods*, pages 806–809. Springer, 2009.

[17] Angel Leon. How many lines of code does it take to create the Android OS? - Gubatron.com. http://www.gubatron.com/blog/2010/05/23/how-many-lines-of-code-does-it-take-to-create-the-android-os/.

[18] V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. 2005.

[19] David McCandless. *Knowledge is Beautiful*. William Collins.

[20] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. *EuroSys*, 42:315, 2008.

[21] Jonathan M Mccune, Ning Qu, Yanlin Li, Anupam Datta, Virgil D Gligor, Adrian Perrig, and Zongwei Zhou. TrustVisor : Efficient TCB Reduction and Attestation TrustVisor : Efficient TCB Reduction and Attestation. *Oakland*, 2010.

[22] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*, pages 1–1, 2013.

[23] Microsoft. WHT207 - Ultra High Definition (UHD) and Hardware DRM Investments in Windows 10, 2015.

[24] NIST. CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160.

[25] NIST. CVE-2014-3566. Available from MITRE, CVE-ID CVE-2014-3566.

[26] NIST. CVE-2014-6271. Available from MITRE, CVE-ID CVE-2014-6271.

[27] Job Noorman, Bart Preneel, K U Leuven, Pieter Agten, Wilfried Daniels, Raoul Strackx, Christophe Huygens, Frank Piessens, Anthony Vanherrewege, and Ingrid Verbauwhede. Sancus : Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. *22nd USENIX Security*, 2013.

[28] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. *Proceedings - IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

[29] Daniele Perito, Gene Tsudik, and Karim El Defrawy. SMART : Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. *Security*, 2012.

[30] Chad Perrin. The CIA Triad - TechRepublic. http://www.techrepublic.com/blog/security/the-cia-triad/488, 2008.

[31] Graeme Proudler, Liqun Chen, and Chris Dalton. *Trusted Computing Platforms: TPM2. 0 in Context*. Springer, 2015.

[32] J. M. Rushby. Design and verification of secure systems. *Proceedings of the eighth symposium on Operating systems principles - SOSP '81*, pages 12–21, 1981.

[33] Johan Rydell, Siddharth Bajaj, David Naccache, and Salah Machani. OCRA: OATH Challenge-Response Algorithm.

[34] Johan Rydell, Mingliang Pei, and Salah Machani. TOTP: Time-Based One-Time Password Algorithm.

[35] Raoul Strackx, K U Leuven, Bart Jacobs, K U Leuven, Frank Piessens, and K U Leuven. ICE : A Passive , High-Speed , State-Continuity Scheme.

[36] Raoul Strackx, K U Leuven, Bart Jacobs, K U Leuven, Frank Piessens, and K U Leuven. ICE : A Passive , High-Speed , State-Continuity Scheme (Extended Version).

[37] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, Frank Piessens, and K U Leuven. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[38] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. *Proceedings of the 2012 ACM conference on ...*, pages 2–13, 2012.

[39] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, 50 LNICST:344–361, 2010.

[40] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. *Proceedings of the 2nd European Workshop on System Security EUROSEC09*, pages 1–8, 2009.

[41] Karen (Cisco) Tillman. How Many Internet Connections are in the World? Right. Now. http://blogs.cisco.com/news/cisco-connections-counter/, 2013.

[42] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. *Proceedings - IEEE Symposium on Security and Privacy*, pages 430–444, 2013.

[43] Andrew C Yao. Protocols for secure computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE, 1982.

[44] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. *Proceedings - IEEE Symposium on Security and Privacy*, pages 616–630, 2012.

Fiche masterproef

Student: Dennis Frett

Titel: Extending Protected Module Architectures with a Secure I/O Framework

Nederlandse titel: Beveiligde Module-Architecturen Uitbreiden met Veilige Invoer en Uitvoer

UDC: 681.3

Korte inhoud:

Een computersysteem dat gecompromitteerd is door een softwarefout of een aanvaller kan zorgen dat de programma's zich op een onverwachte manier gedragen. Beveiligde module-architecturen zijn ontwikkeld om dit tegen te gaan. Ze laten toe dat softwaremodules, *beveiligde modules* genaamd, geïsoleerd van de rest van het systeem uitgevoerd kunnen worden. Ook voorzien ze een manier voor een derde partij om van op afstand de toestand van een systeem te verifiëren. Deze beveiligde module-architecturen voorzien momenteel nog geen methode om op een veilige manier hardware apparaten te gebruiken, wat hun bruikbaarheid beperkt. In deze thesis ontwerpen en implementeren we een systeem dat beveiligde modules toelaat apparaten te gebruiken op een manier die de vertrouwelijkheid en de integriteit van de data verzekert. Onze resultaten tonen dat het mogelijk is om dit systeem te implementeren en sterke veiligheidsgaranties te bieden in combinatie met een aanvaardbare performantiekost.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotor: Prof. dr. ir. F. Piessens

Assessoren: Prof. dr. L. De Raedt
Prof. dr. B. Crispo

Begeleiders: Dr. R. Strackx
Ir. F. Mennes, VASCO Data Security