KU LEUVEN
FACULTEIT
INGENIEURSWETENSCHAPPEN

Safe Interacting Enclaves for Heterogeneous Protected Module Architectures

Sten Verbois

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

Promotor:
Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessor:
Dr. R. Strackx
Dr. G. Sedrakyan

Begeleider:
Ir. J. Van Bulck

Academiejaar 2017 – 2018

Preface

I would like to start off by thanking the people that helped me throughout the last year. First of all, I would like to thank my mentors Jan Tobias Mühlberg, Jo Van Bulck, Raoul Strackx, and Neline van Ginkel for their continued support and guidance. Our numerous meetings helped shape this thesis from the initial topic proposal into the end result. I would also like to extend my gratitude to my promotor prof. Piessens and the entire DistriNet research group for making this interesting line of research possible at KU Leuven. And of course, thanks to all my friends and family for supporting me along the way.

Special thanks to Jago for our many cathartic discussions on the depths of Intel SGX.

Sten Verbois

Contents

Abstract

There has been a big increase in connected computing devices in recent years. Some of them are handling privacy-sensitive information in the cloud or performing safety-critical actions in modern automotive systems. Assuring all of those devices are secure and performing as expected is a big challenge. To that end, different software isolation techniques have been widely implemented in high-end systems. On embedded devices however, these techniques are often omitted because of resource constraints.

Recent research on Protected Module Architectures (PMAs) aims to provide efficient isolation of software modules from any compromised software running on the system. Concrete implementations of PMAs exist both for high-end systems and low-end embedded devices. In practice, many large applications will likely consist of a heterogeneous set of platforms and thus a heterogeneous set of PMAs.

This master's thesis looks into the secure interaction of different PMAs. One important observation is that there is no isolation mechanism that can protect against a modules' own source code when it contains memory safety vulnerabilities like the ones that are common in unsafe languages like C and C++ . These vulnerabilities can nullify any integrity or confidentiality guarantees a PMA provides for a module. Taking into account these memory safety issues, a first main contribution of this master's thesis is to propose the use of Rust as an alternative to C/C++ for writing code that executes inside a protected module. Rust is positively evaluated as a promising alternative in terms of provided security, performance, and programmer effort. Two other, more practical, contributions focus on the specific PMAs Intel SGX and Sancus in the context of automotive control networks. The first one being an SGX enclave containing a rust port of the LeiA message authentication protocol. This enclave is able to keep a secure log of all authenticated traffic it observed on the CAN bus. The second enclave is a realization of an attestation server for such networks, a central entity responsible for attesting participating Sancus modules and providing them with fresh connection keys. Both show the viability of interaction between PMAs via secure communication channels, implemented in a safe and fast programming language.

Samenvatting

Het aantal verbonden computersystemen is enorm gestegen in de laatste jaren. Sommigen zijn verantwoordelijk voor het behandelen van privacygevoelige informatie in de *cloud*, andere voeren veiligheidskritische handelingen uit in moderne voertuigen. Het is een grote uitdaging om ervoor te zorgen dat al deze apparaten veilig en naar verwachting functioneren. Verschillende beveiligingsmaatregelen zijn daarom al aanwezig op veel alledaagse computersystemen. Op kleine, energiezuinige apparaten ontbreken deze maatregelen echter vaak omwille van hun stroomverbruik.

Recent onderzoek naar beveiligingsarchitecturen heeft als doel efficiënte isolatie te voorzien voor softwaremodules ondanks de aanwezigheid van andere, gecompromitteerde software op het systeem. Concrete implementaties van deze architecturen bestaan voor zowel alledaagse computersystemen als kleine, energiezuinige apparaten. In de praktijk zullen grote toepassingen vaak een heterogene samenstelling zijn van verschillende computerplatformen en dus ook van verschillende beveiligingsarchitecturen.

Deze masterproef onderzoekt de veilige interactie tussen verschillende beveiligingsarchitecturen. Een belangrijk inzicht is dat geen enkele beveiligingsmaatregel kan beschermen tegen modules die fouten bevatten in hun geheugenmanagement zoals vaak het geval is in onveilige talen als C en C++ . De garanties die beveiligingsarchitecturen bieden voor de interne staat van de module worden door deze kwetsbaarheden volledig ongedaan gemaakt. Als eerste bijdrage stelt deze masterproef daarom Rust voor als alternatief voor C/C++ om code te schrijven voor beveiligde modules. Er is een positieve evaluatie gemaakt van Rust zowel wat betreft veiligheid, performantie, als gebruiksgemak. Twee andere bijdragen zijn meer praktisch van aard en focussen op de specificieke beveiligingsarchitecturen Intel SGX en Sancus in een context van automobielsystemen. De eerste hiervan is een SGX *enclave* die een Rust versie bevat van het LeiA protocol voor geauthenticeerde communicatie over CAN. Deze enclave is in staat een veilig logboek bij te houden van alle gemonitorde communicatie. De tweede enclave is een verwezenlijking van een attestatie server voor dit soort netwerken. Deze server is een centrale entiteit die verantwoordelijk is voor de attestatie en sleuteldistributie van deelnemende Sancus modules. Beiden tonen de haalbaarheid aan van de interactie tussen verschillende beveiligingsarchitecturen, geïmplementeerd in een veilige en snelle programmeertaal.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Glossary

Abbreviations

AS	Attestation Server
CAN	Controller Area Network
CSMA	Carrier-Sense Multiple Access
ECU	Electronic Control Unit
EDL	Enclave Definition Language
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
MAC	Message Authentication Code
MAL	Memory Access Logic
MEE	Memory Encryption Engine
OS	Operating System
PM	Protected Module
PMA	Protected Module Architecture
PRM	Processor Reserved Memory
ROP	Return-Oriented Programming
SLOC	Source Lines Of Code

Glossary

ID_{Send}	CAN identifier reserved for outgoing channel of attestation server.
ID_{Recv}	CAN identifier reserved for incoming channel of attestation server.
K_{PM}	A 128-bit module-specific key derived by the trusted Sancus hardware from the infrastructure provider identity, software provider identity and software module identity.
K_C	A 128-bit connection key used by VulCAN for its authenticated communication protocol.

Chapter 1

Introduction

In recent years there has been a big increase in connected computing devices. Having all those devices connected with each other and with the cloud brings with it a bunch of challenges. Not the least of which being the need to assure all of them are secure and performing as expected. This is especially the case for devices performing safety-critical actions or handling privacy-sensitive information.

For high-end systems, software isolation has been extensively studied. For example, most modern systems have support for virtual memory and privilege levels. However, they require software support from an Operating System (OS) kernel which, when written in an unsafe language like C or C++, may contain many memory safety vulnerabilities. Attackers that are able to exploit these vulnerabilities can bypass many of the isolation mechanisms. Resource-constrained embedded devices on the other hand often utilize a single address space with little to no support for software isolation available. Clearly, there is a need for a more fundamental approach to software isolation. Recent research has formulated an answer to these issues in the form of Protected Module Architectures (PMAs). PMAs allow for efficient isolation of software modules in the presence of malware or compromised privileged software like an OS. Concrete implementations of PMAs have been developed for both high-end systems (Intel SGX [18]) and low-end embedded devices (Sancus [47]). In practice, many large applications will likely consist of a heterogeneous set of platforms and thus a heterogeneous set of PMAs.

As an example context, most modern automotive vehicles are powered by embedded microprocessors. Potentially hundreds of Electronic Control Units (ECUs) are working together to keep the whole system operational. Many safety critical tasks are in that case performed by software modules, including steering, accelerating, and braking. Moreover, autonomous vehicles have increased the number of critical ECUs drastically by requiring a multitude of extra sensors and decision points for taking safety critical actions without any human intervention. To support these devices over a long period of time, they are often connected to the internet and extensible with software to enable new features and apply bug fixes. However, this also makes them vulnerable to remote attacks [21, 22, 44].

Additionally, communication between ECUs in an automotive system happens on a CAN bus, a multi-master bus without any built-in message authentication mechanisms. In response to recent standardization efforts, protocols have been proposed to add authentication to CAN messages. While these protocols protect against attackers that only have control over the CAN bus, attackers with arbitrary code execution capabilities are able to compromise unprotected confidential key material on the device. Protecting against those attackers is the aim of VulCAN [57], a design for efficient vehicle message authentication and software module attestation that builds upon lightweight trusted computing technology. Still, much work remains to be done to bring this design to consumer motor vehicles.

1.1 Contributions

The goal of this master's thesis is to explore what is needed to achieve safe interaction between PMAs. This includes both the development of safe protected modules, as well as securing the communication between them. The prototypes developed for this master's thesis are based on Intel SGX [1, 18] and Sancus [47]. Specifically, the thesis contains the following contributions:

- The requirements for setting up a secure communication channel between an SGX enclave and a Sancus module are analysed. As a demonstration, an implementation of an enclave logging authenticated traffic on a communication channel between Sancus modules over CAN was developed.

- A second practical contribution is presented in the form of an attestation server for VulCAN. It is able to perform remote attestation of Sancus modules on the network, making sure they have not been tampered with. If this is the case, they can be dynamically provided with keys for their authenticated communication protocol.

- Both previously mentioned prototype enclaves are developed using Rust, a safe systems programming language. Rust provides strong assurance that the resulting enclaves do not contain memory safety bugs. For a new systems programming language to be adopted over conventional ones like C or C++, it should ideally not exhibit any drawbacks compared to those languages. Therefore, the runtime performance of the Spongent cryptographic functions in Rust is demonstrated to be very similar to an implementation in C++. At the same time, it is shown that writing an enclave in Rust or even porting existing code to Rust does not require significant extra programmer effort.

All source code of both the Spongent cryptographic functions and SGX enclave implementations for VulCAN in Rust are publicly available at `https://github.com/stenverbois/spongent-rs` and `https://github.com/stenverbois/vulcan-rs` respectively.

1.2 Outline

The remainder of this thesis is structured as follows:

Chapter 2: Background. This chapter focusses on the relevant software security background. First, typical memory safety issues are discussed. Then, a brief introduction to the Rust programming language is given, together with how it can help mitigate these common memory safety errors.

Chapter 3: Safe interaction Between SGX and Sancus. The third chapter discusses in detail what is needed to make secure interaction between SGX enclaves and Sancus modules possible. More precisely, the chapter looks into extending Sancus' authentic execution to SGX. In addition, the chapter contains an evaluation of Rust's performance and ease of use.

Chapter 4: Attestation Server for VulCAN. This chapter details the design and implementation of an attestation server for VulCAN. It also presents a security discussion on the attestation and key distribution protocols. Special care is taken to mitigate key reinstallation attacks in the latter protocol.

Chapter 5: Conclusion. The final chapter sums up the contributions, their limitations, and gives direction to future work in this area.

Chapter 2

Background

Achieving software isolation in any computer system is a multifaceted problem. First of all, it is clear that even the most secure and advanced isolation mechanisms can not protect against a protected modules' own source code. Programmer errors can leak confidential information to the untrusted context via the specified input and output channels of the module itself. This becomes problematic in unsafe languages like C and C++ where not every possible memory interaction can be anticipated by the programmer or even static analysis tools.

As for the isolation mechanisms, many are already present on modern high-end systems. The shortcomings of these mechanisms and the need for strong software isolation on embedded systems have led to research into a more fundamental approach to software isolation.

The remainder of this chapter first discusses the problem of memory safety, and in particular language safety, in Sect. 2.1. Then, the software isolation provided by Protected Module Architectures (PMAs) is described in Sect. 2.2. Section 2.3 presents the background on VulCAN, a design for authenticated communication on vehicular control networks that makes use of PMAs. Finally, this chapter is concluded in Sect. 2.4.

2.1 Memory Safety

Unsafe memory accesses are a frequent source of security vulnerabilities [50, 20, 51, 59] that has been around for more than thirty years. This section starts off by introducing some common memory safety vulnerabilities. Then, it takes a look at some of the many research proposals targeting these errors. One particular branch of research argues the importance of language safety to help eliminate these issues. Thereafter, Rust's approach to memory safety is further explored.

Some of the most frequent programmer-introduced spatial and temporal memory safety vulnerabilities that appear in unsafe languages like C and C++ are outlined below.

Buffer Over-flow/-read. These violations occur when memory accesses take place out-of-bounds of the object belonging to the pointer. They are often triggered by malformed inputs to a program. Depending on whether data is being written or read, this can lead to data corruption or leaking of sensitive data respectively. Attackers exploit such vulnerabilities in OS kernels to overwrite nearby function pointers and perform privilege escalation attacks or compromise control flow integrity [24, 16].

Use After Free and Double Free. Both originate from a wrong use of `free` throughout the program. They occur when one part of the program considers a pointer to be valid, while it has already been freed at an earlier point during execution. They possibly result in modification of unexpected memory locations.

Data Race. Data races occur when multiple aliases exist to the same memory location, where at least one of them has write access, and without making use of any synchronization mechanism. As a result, shared memory can be corrupted and program correctness can be affected.

2.1.1 Countermeasures

A big step towards making programs more secure and reliable can be made by reducing the number of programmer introduced errors in the code. During development, static analysis can be used to find some of these errors or enforce good coding practices. When a more formal approach is required, tools like Verifast [35] are able to provide proof of the absence of illegal memory accesses and data races. This comes at the cost of significant additional programmer effort however, as each function requires multiple manual annotations. Some of them are automatic verifiers [23, 9] which have the downside of either not catching all errors, or rejecting some percentage of valid programs.

Other approaches exist to prevent some of the remaining errors from causing harm at runtime. Proposed techniques include: tagged pointers [7, 45, 6, 39] that store metadata about the actual pointer (like bounds information and in some cases temporal attributes), region-based memory management [29, 27, 53, 32] and adding conservative garbage collection [12]. Techniques that target legacy unsafe C code running on conventional hardware all have serious performance and/or memory overheads. Hardware-assisted approaches [40, 49] on the other hand achieve negligible runtime and memory overhead at the cost of requiring dedicated hardware support.

While all of the aforementioned approaches have their use cases, for a solution to be widely adopted it can not require a significant amount of extra programmer effort, it must be highly compatible with existing software, and it must not suffer from a serious performance or memory overhead. Examples of such solutions are address-space layout randomization (ASLR) [11] and Data Execution Prevention (DEP) in contemporary OSs like Windows and Linux, or stack canaries in compilers like GCC [62]. These solutions significantly raise the bar for attackers but none of them achieve the ultimate goal of completely eliminating memory safety vulnerabilities and the exploitation thereof.

2.1.2 Language Safety

While legacy C/C++ software suffers from the drawbacks of previously discussed retroactively applied security solutions, development of new software should not. In this case, one could consider to not use a memory unsafe programming language, but instead opt for a language that has memory safety as one of its core design goals.

2.1.3 Rust

Since the 1980s, the space of general purpose systems programming languages has been dominated by languages like C and C++ . Their focus on providing programmers with just enough useful low-level features made them a popular choice for writing software for time or resource constrained systems.

At the same time, the amount of manual memory management C requires has proven to be a real and well understood issue (see Sect. 2.1). Many new languages have since taken inspiration from C, while trying to be more memory safe by not making the programmer exclusively responsible for memory management. This is often achieved by introducing high-level abstractions together with some kind of garbage collection (e.g. Java, Python, C#). Supporting features like a garbage collector and runtime bounds checking also means the language has to depend on a sizeable runtime, making them unsuited to run on many embedded, resource-constrained devices. One might conclude there is a fundamental trade-off to be made between high-level features with type-/memory-safety on the one hand, and low-level control on the other hand.

It was only in 2015 that Rust 1.0 was released [3]. As a language that tries to give the programmer both high-level features guaranteeing safety and low-level control, Rust has gained a lot of popularity. Some of the main domains Rust is trying to improve on in 2018 are network services, webassembly, and embedded devices. One notable aspect about its approach on memory safety is that it provides conventional type safety, but it also eliminates data races in threaded programs and prevents invalid iterators. Additional features of the language include type inference, pattern matching and efficient C bindings.

Ownership and Borrowing

A core concept of Rust's approach to type safety is the notion of ownership. It states that an object must always be owned by a single alias during the entire execution of a program. Consequently, the object can be deleted as soon as the owning alias goes out of scope and no garbage collection is necessary. Assigning one variable to another *moves* ownership to the latter variable and invalidates the former. Functions that need temporary access to an object of type `T` can *borrow* this object. This borrow can be either explicitly immutable (`&T`) or mutable (`&mut T`). Rust's borrow checker statically verifies that there exist at most a single mutable reference or one or more immutable references to a single object at the same time. In other words, it prevents both aliasing and mutation to occur at the same time. This restriction effectively eliminates data races, as there is no way for two aliases to the same object to be valid where at least one of them has write access.

Listing 2.1 demonstrates the compiler rejecting an invalid program. In the example, vector `v` is constructed in `main` with value 1. Then, a borrow is created for function `borrow`. As this function takes an immutable reference, it only has read-only access to the vector. After the value 2 is pushed to the vector, function `take` requires a `Vec<i32>` and as a result, ownership of the vector is transferred to the argument of `take`. Any attempt by `main` to use the variable `v` after this function call will therefore fail.

```rust
fn main() {
    let mut v = Vec::new();
    v.push(1);
    borrow(&v); // OK, v borrowed
    v.push(2);
    take(v);
//       - value moved here
    v.push(3); // ERROR
//  ^ value used here after move
}
```

```rust
fn borrow(v: &Vec<i32>) {
    // ...
} // Borrow ends here

fn take(v: Vec<i32>) {
    // ...
}
```

Listing 2.1: Code snippet demonstrating the compiler-enforced ownership mechanism of Rust.

The following paragraphs describe how the unsafe memory access patterns listed in Sect. 2.1 will be rejected by the Rust compiler.

Buffer Over-flow/-read. To eliminate memory accesses outside the bounds of an array (or `slice`), arbitrary indexing will be accompanied by runtime bounds checks. In a large amount of cases however, Rust programmers will make use of iterators, sometimes combined with a for loop. As a result of using iterators, invalid memory accesses can not occur. Additionally, the compiler can also omit bounds checks in cases where it can derive the knowledge that the array index will never exceed the array length.

Use After Free and Double Free. Both of these mistakes are prevented by not relying on the programmer to free memory. Rather, the compiler will free the memory (called `drop`) as soon as the owning variable goes out of scope. In the rare case that a variable is explicitly dropped, the value is moved and thus can no longer be used afterwards.

Data Race. Data races are prevented by the ownership system, as it makes sure it is impossible to create aliases to mutable references. When sharing data between threads, the type system enforces only objects guarded by proper synchronization mechanisms can be sent between threads.

Unsafe Code

The key challenge in developing this borrow checker is making sure it is *sound*, meaning it rejects all unsafe programs, but also *useful*, meaning it should accept enough valid programs. Accepting all safe programs while rejecting all unsafe programs is simply too big of a challenge. Therefore, the rust compiler is quite strict and there are many advanced low-level programming constructs that cannot be encoded in Rust's type system. To overcome this restriction, Rust allows the use of the `unsafe` keyword. Many of Rust's standard libraries rely on the `unsafe` keyword for their efficient low-level implementation. Code inside a block marked `unsafe` is allowed to perform a few additional operations (e.g. raw pointer dereference, access and modification of a mutable static variable) that are considered unsafe in regular code.

With `unsafe`, Rust's enforcement of ownership is relaxed in an encapsulated and controller way. The aim is that programmers that make use of APIs implemented with `unsafe` code never experience unsafe behavior in their programs. Concerns about whether Rust's type system is actually sound have been addressed by Jung et al. [36], who give a formal safety proof for a realistic subset of the Rust language.

2.2 Protected Module Architectures

Computer systems often have software deployed on them by multiple mutually distrusting software providers. Especially in cloud environments, developers have very little control over the used infrastructure. To enforce basic security guarantees, systems can employ various software isolation techniques. Conventional operating systems are tasked with enforcing isolation of different processes. In practice, however, trusting an entire OS for this task has proven to be an issue, as a particular error in any part of the very large OS code base compromises the entire system.

This section first provides an outline of the desirable properties of software isolation [42]. Then, it closely examines two implementations of Protected Module Architectures (PMAs) that aim to achieve these properties: Intel SGX and Sancus.

2.2.1 Properties and Features

Attacker Model. In the attacker model of trusted computing architectures, attackers are able to manipulate all software and hardware components outside of the TCB. This includes full control of privileged software like the OS and the ability to deploy their own malicious software or protected modules. Architectures that do not employ any protection for main memory consider physical attacks to be out of scope, while others consider only some hardware components to be in the TCB and allow physical attacks on components that are not. In addition to controlling software on the device, they are also in control of the network communication to the device. Communication can be sniffed, modified and even generated by attackers. As for cryptographic primitives, the Dolev-Yao model [26] is used. Attackers are assumed to be unable to break cryptographic primitives but can execute attacks against communication protocols.

Trusted Computing Base. The Trusted Computing Base (TCB) of a system consists of its most privileged components w.r.t. to security. Any compromised component within the TCB puts the security properties of the entire system at risk. Conversely, any compromised component outside of the TCB cannot violate the security properties of a trusted component. The security of a system therefore depends upon the absence of bugs and vulnerabilities inside the TCB. For this to be easily verifiable, the size and complexity of the TCB should be kept minimal.

In conventional OSs, applications often execute in user-mode on top of a monolithic or hybrid kernel. Since kernel mode is the most privileged mode in a modern OS, any source code that runs in this mode is part of the TCB for these applications. Having most of an OS as part of the TCB is cause for some concern. First, it should be noted that common operating systems today have an very large code base. As of Linux kernel version 4.14, it contains over 20 million lines of code [2], while Windows 10 presumably contains between 40 and 60 million lines of code. Additionally, being system software with relatively strict performance requirements, modern OSs are still primarily written in unsafe languages like C or C++. From both of these observations, it can be concluded that it is near impossible to verify the correctness of these code bases.

Isolation. Module isolation is achieved by including hardware mechanisms designed for this purpose in the architecture. No party outside of the TCB should be able to inspect the internal state or make changes to the code of a module. To enforce legitimate execution of a modules' code section and protect against Return-Oriented Programming (ROP) attacks, entry to a module is limited to a number of so called *entry points.*

Attestation. Attestation allows a module to provide proof that it is in a certain state. The isolation mechanisms should be enabled for the module to prevent tampering with this state after providing the proof. Attestation can be requested either by another software module running on the same platform (so called *local attestation*), or by some remote party (called *remote attestation*). To enable attestation, different properties of the module like code, memory location, etc. are often measured by combining them in a single hash. If the module can provide proof it possesses this measurement, the other party knows it is communicating with a successfully initialized and isolated module.

Sealing. Sealing enables a module to store internal state securely to disk. PMAs provide strong isolation guarantees while the system is running. Whenever a system goes down because of a crash, reboot or power outage however, it must be able to recover its state. Of course, this state has to be integrity and confidentiality protected. The key that is used for sealing this data is commonly derived from the module's measurement.

State Continuity. With the guarantees of sealing in place, a secure but stale versions of the state can still be replayed. When a module requires freshness of its state, additional security measures are required to ensure the module rejects an old state. More specifically, *state continuity* requires: (1) rollback prevention, (2) the module either executes with accepted input or stops executing, and (3) an unexpected crash never leads to an invalid state of the system. Typically, a trusted source of monotonically increasing values is used and the counter value is included in the state. One such approach specific to SGX is Ariadne [55].

2.2.2 Intel SGX

The first PMA this master's thesis looks at is Intel's Software Guard Extensions (Intel SGX) [1, 18]. SGX is a set of instructions added to the x86 instruction set architecture. The technology was introduced in 2015 as part of Intel's sixth generation *Skylake* microprocessor architecture. It allows user-level protected modules, called *enclaves*, to be isolated from all other software on the system. The memory used by this enclave is protected by the hardware and can not be accessed from outside the enclave. This includes privileged software like the OS, hypervisor, and other SGX enclaves.

Isolation

SGX reserves a region in main memory called Processor Reserved Memory (PRM), which is protected from all non-enclave accesses. It contains both the Enclave Page Cache (EPC) and Enclave Page Cache Map (EPCM). The EPC stores pages containing enclave code and data, while the EPCM tracks ownership of each page. SGX is an architecture that does not include all hardware in the TCB. Specifically, only the processor package is considered trusted. Enclave pages that are stored in main memory must therefore be protected. To achieve this, SGX incorporates the Memory Encryption Engine (MEE) [31], a hardware unit responsible for encrypting and integrity protecting data passing between the processor and main memory. When the EPC runs out of capacity, the OS kernel is responsible for evicting pages to another location in main memory outside of PRM. The `EWB` instruction makes sure the page is encrypted, integrity and replay protected.

Attestation

Attestation in SGX is based on an enclave's measurement. During initialization of an enclave, the measurement is computed by taking a secure hash over all page content. Once initialization is complete, hardware protects these pages inside the EPC from modifications and a remote party is able to identify the software that is executing inside the enclave based on its measurement. This is achieved by comparing the reported enclave measurement with the expected one. If both match, the remote party is confident the expected code is correctly initialized and isolated in the enclave.

Sealing

SGX supports two modes for sealing data. In the first mode, the key derivation policy is set to `MRENCLAVE`. This results in the derived key to be tied to the enclave's measurement. This is useful to allow the enclave to store its internal state so it can survive power cycles when no other enclave should have access to this data. The second mode sets the key derivation policy to `MRSIGNER`. In this case, the key is tied to the key pair used to sign the enclave. Consequently, it is possible for multiple enclaves signed by the same entity to access the same sealed data.

2.2.3 Sancus

The second PMA this master's thesis focusses on is Sancus [47], a security architecture aimed at extensible, low-cost and low-power embedded devices with a hardware-only TCB. Just like SGX, Sancus provides hardware support for isolating an application's code and data at runtime. The practical work in this thesis makes use of the prototype based on the open-source OpenMSP430 processor implementation.

Key Derivation

The physical embedded devices are deployed and managed by a trusted infrastructure provider IP and run one or more protected software modules from possibly multiple distrusting software providers. A specific software provider SP must be allowed by IP to deploy its software on the nodes managed by IP. SP is then able to establish a secure communication channel with a specific node using a shared symmetric key.

Every node has a unique node-specific key baked in silicon that is known to IP. From this key and a unique public identifier for SP, IP can derive key $K_{N,SP} = kdf(K_N, SP)$ and deliver it to SP. With this software provider key $K_{N,SP}$, SP can compute the module-specific key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$. Here, SM represents the software module identity. This identity consists of a hash of both the text section and the module layout, which describes the start and end addresses of the text and data sections.

The key derivation function kdf is one of the cryptographic primitives implemented in hardware on a Sancus node. The node is therefore able to derive the software module key $K_{N,SP,SM}$ itself. Software on the nodes is never able to access this key directly however, it is only indirectly available to protected software modules running on the node. As this key is only known to SP and SM, it provides a base for remote attestation and establishing a secure communication channel for provisioning secrets.

Module Isolation

Sancus modules are divided in a public text section and a private data section, on which strict access rules are enforced. The text section is immutable, while the data section contains application-specific data and runtime metadata. Sancus' counter-based memory access control model [56] makes sure the data section of a module is only accessible while code in the corresponding text section is being executed. This mechanism is implemented by a dedicated Memory Access Logic (MAL) circuit. The maximum number of modules that can be protected, and consequently the number of MAL units, on the node is configured when synthesizing the processor. The layout information for each module is maintained by the processor in specialized protected storage region only accessible from hardware.

Loading the actual module starts with the code section being loaded in the appropriate location in memory. Afterwards, the special `protect` instruction is used to perform the necessary checks and enable memory protection on the layout. The module-specific key $K_{N,SP,SM}$ will also be derived as a result of the `protect` instruction. Any malicious change to the code section or layout of the loaded module will result in a change to $K_{N,SP,SM}$. As a result, any future attestation requests will fail. In terms of entry points, Sancus modules only have a single physical entry point. Multiple user-defined, logical entry points are dispatched through this single physical one.

2.3 VulCAN

The last section on this background chapter is dedicated to VulCAN, a design for message authentication on vehicular control networks that makes use of PMAs. First, this section describes the details of the medium often used in those networks. Then, authenticated communication and the notion of authentic execution is presented.

2.3.1 Controller Area Network

The Controller Area Network (CAN) bus is an ISO defined multi-master message broadcast system used to connect Electronic Control Units (ECUs) [17]. Since its official release in 1986, the CAN bus has been primarily used in the automotive industry. It allows car manufacturers to avoid complex wiring systems by using a single two-wire bus. This thesis will zoom in on two of the defining features: the message format and bus arbitration mechanism.

Message Format

CAN messages consist of an identifier, followed by the message payload. The CAN 2.0B specification defines two different message formats: a 'standard' format which uses an 11-bit identifier, and an 'extended' format which adds space for an additional 18 bits to the identifier, resulting in a 29-bit identifier. Both frame formats allow for up to 8 bytes of data. After that, a CRC and ACK field are added. Figure 2.1 shows these elements in the layout of a standard and extended CAN frame.

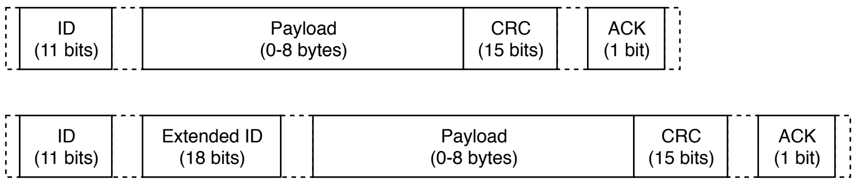

Figure 2.1: CAN frame format. Standard (top) and Extended (bottom).

Bus Arbitration

The CAN communication protocol is a carrier-sense multiple access (CSMA) protocol. This means that any node may start transmitting a frame when it notices there is no other traffic on the bus. When multiple nodes start sending at the same time, the conflict is resolved by bit-wise arbitration on the identifier field. A logical zero is defined as 'dominant', while a logical one is defined as 'recessive'. If all nodes are transmitting a recessive bit, the value of the bus will be recessive. On the other hand, if any node is transmitting a dominant bit, the value on the bus will be dominant. While a node is transmitting, it continuously compares the bit it is sending with the bit that it monitors on the bus. If these bits are equal, the node continues to transmit its frame. Only when a recessive bit is transmitted and a dominant bit is being monitored on the bus, the node immediately stops transmitting. It now has to wait until the bus is free to retry sending its frame. The node sending the lowest message identifier will win arbitration because it contains more zeroes at the start of the frame. As long as the used identifiers are unique, this arbitration mechanism always allows for a single message to be sent and no time on the bus to be lost.

2.3.2 Authenticated Communication

The CAN bus is a broadcast medium which, by design, does not provide confidentiality, authentication, or replay protection for its messages. Many existing attacks in the literature [37, 44, 15] rely on these features being absent. Especially the lack of authentication allows an attacker to inject messages onto the bus via any ECU, and make them appear legitimate.

In an attempt to standardize authentication on CAN, the AUTOSAR [8] standard includes guidelines for adding authentication to CAN messages. Previous to the standard, a variety of other approaches has been proposed [30, 33, 38]. Two protocols that were developed in reaction to the standard are LeiA [52] and VatiCAN [48]. Both use 128-bit keys, 64-bit MACs and provide freshness using counters. In their approach, MACs are sent separately after the original message on a different CAN identifier. Unmodified legacy applications that do not require authenticated messages can just act on the plain text messages and ignore MACs sent on the authentication identifier. The MAC itself is calculated using a unique connection key over the message payload and monotonic counter value.

2.3.3 Authentic Execution

PMAs provide strong security guarantees for software running inside protected modules. However, software running in isolation without any side effects is not useful. Execution of a software module must result in some output being written (e.g. to a screen or file) or some action being performed (e.g. turning on a LED or actuator). Moreover, a software module rarely operates without any input, be it physical inputs or signals from other software. Therefore, the objective is to provide security guarantees for a modules' I/O channels, in addition to what PMAs provide for their internal state.

When integrity and freshness is established for channels between protected modules, these guarantees can be extended to the entire distributed application. *Authentic execution* then states that a sequence of observable I/O events must have been caused by actual input events to the system as a result of the high-level source code of the individual modules [46]. While confidentiality can be a goal in addition to integrity, it is not essential to achieve authentic execution. In some applications, it is not desirable to have confidentiality on produced events when for example interoperability with legacy components is required. Lastly, authentic execution does not enforce any availability guarantees. With an attacker model that includes full control of the network used by nodes in a distributed application, this is a challenging problem [58] that is out of scope for this master's thesis.

Practically, an infrastructure owner will deploy nodes according to a deployment descriptor, which specifies which modules to deploy on which nodes and how to connect their channels. The work of Noorman et al. builds upon Sancus for a prototype implementation. It achieves confidentiality and integrity of inputs and outputs by assigning a symmetric key to each connection. Outputs can then be encrypted before leaving the module and inputs are decrypted upon entering the module. Code for these operations is generated by the compiler.

While Sancus' secure I/O channels are generated at compile time, VulCAN allows for dynamic authenticated communication channels between different ECUs in an application. It leverages existing protocols that add message authentication on CAN, such as LeiA and VatiCAN, and puts them inside a protected module. In doing so, the attacker model can include arbitrary code execution on participating nodes. VulCAN also makes use of symmetric communication keys that are used by the message authentication protocols as the basis for their key derivation. These keys would be freshly distributed by an attestation server each boot cycle. Consequently, the attestation server must have complete knowledge of participating modules and authenticated channels between them. Interoperability with legacy components is ensured by placing Sancus-enabled gateways before legacy ECUs to transparently perform the necessary authentication for this legacy component. In theory, VulCAN allows the notion of authentic execution to be dynamically extended over an entire application of Sancus-enabled ECUs.

2.4 Conclusion

This background chapter covered many of the important aspects w.r.t. software isolation that will be relevant throughout this master's thesis. First, it is argued that existing backwards-compatible memory safety approaches for C all incur a significant performance and/or memory overhead. Rust is presented as an alternative systems programming language that eliminates these errors at the language level.

Memory safe code is paramount when using PMAs to enforce software isolation guarantees. The two architectures that are of interest in this case are Sancus and Intel SGX. The former targeting resource-constrained embedded systems, the latter targeting modern x86 hardware.

Finally, these architectures are used to provide authenticated communication for automotive control networks using VulCAN. The next chapter will define the requirements for interaction between two PMAs, this will allow to extend the previously introduced notion of authentic execution from Sancus to SGX.

Chapter 3

Safe Interaction Between SGX and Sancus

Embedded PMAs like the Sancus architecture are well suited to give strong security guarantees in absence of conventional software isolation techniques. At the same time, being an embedded platform with strict limitations, for example in terms of power usage, also limits the computational power available for deployed software. Any software module running on these microcontrollers will have to take into account that it does not have the same resources available to it as it would have on a modern general purpose x86 processor. For more complex use cases, this extra computational power could be required. An example being a distributed application with many Sancus modules running on many nodes which have to be attested and provided with secrets by a central attestation server. This use case is explored in-dept in chapter 4.

In such distributed applications, the application design may call for a connection between one or more Sancus modules on the one hand and another software module running on a more powerful platform, like a single x86 processor, on the other hand. This connection can be established over any type of network media (e.g. TCP/IP, CAN). It is important to note that no mather what network media is chosen, it is fundamentally untrusted because the attacker model includes full control over any network communication. While is this fine for noncritical communication, extra measures have to be put in place to achieve any of the stronger security properties like confidentially and integrity. More specifically, this master's thesis explores an extension of the notion of authentic execution to include the interaction between two different PMAs, Sancus and SGX.

With the absence of efficient public key cryptography capabilities on Sancus, the best way of establishing a communication channel that is integrity and confidentiality protected is to make use of the built-in encryption primitive together with the unique module-specific key. The rest of the chapter is structured as follows. First the implementation of the authenticated encryption primitive in software for use in the SGX enclave is discussed in Sect. 3.1. Then, Sect. 3.2 zooms in on the first case study, a secure log of authenticated traffic on the CAN bus. Finally, this chapter is concluded in Sect. 3.3.

3.1 Spongent

Sancus limits the cryptographic primitives to authenticated encryption with associated data. Therefore, using this primitive is the most efficient way to establish security guarantees on the communication with a software module deployed on Sancus. Sancus implements the SpongeWrap [10] construction with Spongent [13] as the underlying sponge function. When implemented directly in hardware, this is an efficient operation in terms of both power usage and runtime. On SGX, this specific cryptographic primitive is not part of the architecture and thus has to be implemented in software. Using Rust for this task increases the trustworthiness of the authentic execution property.

3.1.1 Spongent in Rust

The memory safety guarantees provided by the Rust compiler seem particularly useful for implementations of cryptographic functions. To demonstrate this, this thesis includes a Rust implementation of SpongeWrap and the Spongent cryptographic functions. This section compares the Rust implementation with an existing C implementation in terms of runtime performance and programmer effort.

Runtime Performance

When Rust is to be used as a systems programming language in applications where C++ is typically used, it should show similar runtime performance. To that end, the Rust implementation of Spongent is compared with the existing C++ implementation of the Sancus compiler. They are compared in terms of clock cycles needed to compute a 128-bit MAC of a 128-bit and 1024-bit input. Using the x86 instruction RDTSCP, the amount of processor cycles since reset is queried before and after the call to SpongeWrap, and the difference is taken. Table 3.1 shows the results of this comparison.

First of all, these results are similar to previous benchmarks of this algorithm on x86 in [41]. There, is it also shown that the hardware implementation of Sancus outperforms the software implementation by five orders of magnitude. Based on the fact that performance between C++ and Rust is very similar, two things can be concluded. The first being that the bad software performance of Spongent is due to characteristics of the algorithm that can't be efficiently expressed in software, making it much better suited for implementation in hardware. The second conclusion is that Rust closely matches C++ 's performance. They are both compiled to efficient machine code and as front-end to LLVM, Rust can make use of over a decade of fine-tuned optimizations. Indeed, security is achieved at a minimal cost using zero-cost abstractions and carefully choosing required guarantees.

Programmer Effort

A first measure of programmer effort can be expressed in terms of source lines of code (SLOC). Table 3.2 indicates the Rust crate contains a total of 296 lines of code, while

	C++		Rust	
Input length	256-bit	1024-bit	256-bit	1024-bit
Avg. cycles	54,368,526	201,213,242	50,396,303	184,812,155
Std. dev.	336,584	9,694,415	504,136	2,175,737

Table 3.1: Runtime performance comparison between C++ and Rust of SPONGEWRAP. C++ is compiled with '`-O3`' and Rust with '`--release`'.

the C++ implementation consists of 439 lines of code (source and header definitions). This approximate 30% reduction can be partially attributed to the fact that Rust provides high level language features that increase expressiveness and reduce overall code complexity. In Rust, there is also no need to separate the declaration and definition of symbols, which cuts down the line count as well.

At the same time, a large portion of SPONGENT consists of pure functions that operate on a single integer or fixed-length bit sequence. Because of the similarity in low-level control between the two languages, those functions require very little changes when porting from one language to the other. On the other hand, functions that do a lot of pointer manipulations are less easily translatable in Rust. However, as soon as the meaning of a certain piece of C code is understood, the Rust equivalent is often easy to express. Listing 3.1 shows a comparison between C and Rust of a code snippet that consumes some input byte array. In many such cases, input is consumed monotonically from beginning to end with a predefined step size. In C, a programmer has to manually keep track of where the current iteration is situated in the input and how much of the input is left to process. While in Rust, a programmer can make use of a standard function available for slices called `slice::chunks(size: usize)`, which aggregates the iterator in chunks of length `size`. When using such functions, the compiler is able to statically verify an iterator will never be invalid at runtime. The compiler can therefore eliminate the bounds checks in many cases and the generated machine code will be very close to optimal.

```c
char* input = ...;
int inputLeft = inputLength;
while (inputLeft) {
    // Use RATE bytes of input
    ...

    input += RATE;
    inputLeft -= RATE;
}
```

```rust
let input: &[u8] = ...;

for block in input.chunks(RATE) {
    // Use 'block'
    ...
}
```

Listing 3.1: Comparison between C (left) and Rust(right) of a common code pattern.

[1]Lines of code measured with `cloc` not including blank lines and comments.

Language	SLOC[1]
C++	439
Rust	296

Table 3.2: Source lines of code (SLOC) comparison of Spongent implementation between C and Rust.

3.2 Case Study: Logging Authenticated Communication

With a software implementation of SpongeWrap in place, an SGX enclave is now able to participate in secure communication between Sancus modules. As a demonstration, this thesis includes an enclave capable of listening in on the authenticated communication sent in the context of a VulCAN application. While listening to the communication on the CAN bus, the enclave logs all messages and whether they were successfully authenticated or not. When the log itself is protected by SGX's sealing mechanism, it provides strong a non-repudiation guarantee for the messages exchanged by entities connected on the CAN bus. One example use case this could be used for is in case of an accident involving a vehicle. The log of traffic could provide an auditor with clues about if one of the components inside the vehicle was misbehaving and sending incorrect commands. If so, this could lead to the manufacturer of this component to be held accountable.

Recall that the protocols discussed in Sect. 2.3.2 add authentication to communication on the CAN bus by sending the MAC over the original message in a second CAN message. In addition, they protect against replay attacks by adding nonces and a nonce synchronization mechanism to the protocol. The focus is on LeiA in this thesis as, in contrast to VatiCAN, LeiA connections are entirely self-contained, while VatiCAN relies on an external global nonce generator. To strengthen the confidence in the security of the implementation, the LeiA implementation present in VulCAN has been ported to Rust as part of the practical contributions.

This LeiA implementation in Rust is then used in the SGX enclave tasked with logging traffic on the CAN bus. Because most of the logic is contained inside the LeiA protocol library, the code responsible for logging the observed events is less than 100 lines of Rust code. It only forwards all CAN payloads received from the untrusted context to the VulCAN context and logs the returned events. Table 3.3 displays all possible events that can be returned from a VulCAN context. When developing an enclave that is not only tasked with logging traffic but that must also actively participate in communication, this same VulCAN context can be used to send messages on the registered connection identifiers as well.

3.2.1 Security Evaluation

The aim of the log produced by the enclave is to provide a non-repudiation guarantee on the data that appeared on the CAN bus. Therefore, it is important to consider

Event	Description
Received(*id*, [*mac*])	A message was received on *id* without any errors. If a MAC is expected, its value is returned in *mac*.
Authenticated(*id*)	Returned in response to a MAC that successfully authenticated a previous message on *id*.
MissingMAC(*id*)	Returned in response to a message on *id* while a previous message has not yet been authenticated.
UnexpectedMAC(*id*)	Returned in response to a MAC for *id* while no MAC was expected.
IncorrectMAC(*id*)	Returned in response to an incorrect MAC for *id*.
Desync(*id*)	The local nonces have become out of sync with received communication for *id*. A resynchronization procedure should be started for this connection.
Resynced(*id*)	The resynchronization procedure has been successfully completed for *id*.
UnknownId(*id*)	A message has been received in *id* but no connection was set up for this.

Table 3.3: Source lines of code (SLOC) comparison of LeiA implementation between C and Rust.

ways in which an attacker is able to manipulate the contents of this log.

First, an attacker should be prevented from directly tampering with the contents of the log. To that end, the data should be confidentiality, integrity, and replay protected. The SGX sealing feature could be used to cryptographically provide integrity and confidentiality. It does however not protect against replay attacks. While full state continuity approaches like Ariadne [55] may be excessive in this case because the data is not used as input to the module, including a monotonically increasing counter in every batch of logs would provide an auditor with a chronological view of logged events.

Second, it is important to note that the trusted context in this system is relatively small. The enclave consists of less than 100 lines of code responsible for the actual logging and another approximate 350 lines in the VulCAN implementation responsible for managing connections and computing MACs. The untrusted context is all of the CAN bus, CAN controllers, and software stack up to and including the untrusted part of the logging application that loads the SGX enclave. All of this untrusted software is subject to tampering by an attacker. Specifically, as availability is not addressed by this design, an attacker is able to simply prevent any message from reaching the logging enclave. An attacker that succeeds in manipulating CAN communication would make sure these messages are not present in any log. This reduces the practical usefulness of such application in a forensic setting drastically.

3.3 Conclusion

An essential requirement for another party to interact with a Sancus module is that it has to implement Sancus' authenticated encryption primitive. Such code is inevitably part of the security critical components of a system and thus should ideally be memory safe. To that end, a Rust implementation of the Spongent cryptographic functions was developed. In terms of runtime performance of cryptographic algorithms, Rust is shown to be very similar to C++ . Moreover, it is argued that porting existing C++ code requires a small to moderate amount of programmer effort. As a case study, this thesis includes an implementation of a logging enclave tasked with keeping a log of all observed authenticated CAN communication. Because the attacker is in control of the software stack between the communication bus and the enclave however, the practical usefulness of such enclave in the presence of a malicious actor is questionable.

Chapter 4

Attestation Server for VulCAN

Unauthenticated communication on CAN combined with increased connectivity for automotive vehicles has proven to be a real problem in the industry [65, 54]. Solutions that add authentication to CAN are a step in the right direction. However, by themselves they only defend against network-level attackers that do not have compromised the OS or another privileged component of the platform the code is running on. As the key used to cryptographically encrypt and authenticate messages must in that case also be considered compromised. VulCAN aims to provide a solution to this issue by making use of PMAs. It provides implementations of the LeiA and VatiCAN protocols inside a Sancus protected module. This way, the hardware protection provided by the Sancus architecture eliminates tampering by attackers with full remote control of code execution on the node.

Both protocols require initial symmetric connection keys to be securely distributed to all participants of a connection. VulCAN includes a generic design for an attestation server that would simultaneously attest and distribute connection keys to participating modules. This chapter presents a more complete design and implementation of this attestation server. First, the design goals are outlined in Sect. 4.1. According to these goals, a detailed version of an attestation and key distribution protocol is introduced in Sect. 4.2 and Sect. 4.3 respectively. The third responsibility of the attestation server, supporting network changes, is discussed in Sect. 4.4. Section 4.5 then provides a security discussion on both the attestation and key distribution protocols. Thereafter, Sect. 4.6 discusses the actual implementation as an extension to the existing VulCAN implementation. Finally, Sect. 4.7 contains a conclusion to this chapter.

4.1 Design Goals

The purpose of the attestation server (AS) is threefold. First, it should remotely attest the distributed protected modules. Before providing the protected modules with any secrets, AS needs to guarantee they are properly loaded and isolated on their platform. Second, AS should provide successfully attested modules with the session keys necessary for their authenticated communication protocol. Third, AS should support changes to the network, like addition and replacement of nodes, by an untrusted party.

Note that all of these goals need to be realized over an untrusted network and thus need to be protected against attackers controlling the network.

4.2 Attestation

The goal of the remote attestation process is for AS to establish a fresh guarantee that the expected software is correctly loaded and isolated on each of the nodes in the network. More precisely, the attestation process aims to guarantee that the expected source code is presently running with the expected memory layout on the expected device. Because AS needs a fresh guarantee, attestation must be completed for each protected module on each boot cycle.

The attestation protocol is a challenge-response based protocol. A protected module is considered attested by AS if and only if it can prove possession of a shared secret. This secret acts as the 'attestation secret' K_{PM} and should be unique for each of the protected modules. Therefore, it has to be derived by or provided to both AS and each protected module in advance of the attestation process.

4.2.1 Specification

In a first step, AS sends a message containing the challenge value n to a particular protected module PM. This challenge value is a random 48-bit value generated by AS using either a cryptographically secure pseudo-random number generator or true random number generator. In response, PM uses its authenticated encryption primitive to calculate the 128-bit authentication tag over n using the attestation secret K_{PM} as the key. The resulting tag is first truncated to a 64-bit MAC by discarding the eight least significant bytes. This MAC is then sent back to AS over the untrusted network. AS is also able to calculate this MAC as it has knowledge of K_{PM}. If both match, PM has proven possession of the expected attestation secret and so AS has a strong guarantee that PM is running uncompromised. This protocol is visualized in Fig. 4.1.

Research on protocol design uses a specific notation for authenticated encryption [14]. Typically, $\{X\}_K$ represents some data X encrypted under key K. The result is both a cipher text and authentication tag for X. Throughout this master's thesis, this notation is extended to account for authenticated encryption with associated data used by Sancus. Specifically, $\{X; A\}_K$ represents some data X encrypted under key K while simultaneously providing authenticity for data A. Here, the result is cipher text for X and the 128-bit authentication tag for both X and A. In addition, $\{X; A\}_K^T$ represents the 64-bit MAC obtained from truncating the tag.

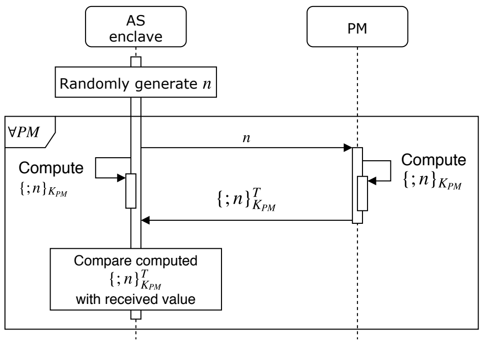
AS enclave
PM
Randomly generate n
$\forall PM$
n
Compute $\{;n\}_{K_{PM}}$
Compute $\{;n\}_{K_{PM}}$
$\{;n\}^T_{K_{PM}}$
Compare computed $\{;n\}^T_{K_{PM}}$ with received value

Figure 4.1: Attestation of protected modules (PM) by the attestation server (AS).

4.3 Key Provisioning

In addition to attesting a protected module, the attestation server has to provide successfully attested modules with the necessary secrets. In this case, all PMs that are participating in authenticated communication with each other need one or more symmetric connection keys. For each of the connections C present in the network, AS randomly generates a unique connection key K_C. It can then start the process of distributing each of these keys to the PMs participating in the corresponding connection.

For the communication to be properly authenticated, the communication key has to be confidentiality and integrity protected on its way to PM. Otherwise, an attacker is able to modify the key that is used by the modules or use the key to generate and send traffic himself. These guarantees are enforced by using the modules' authenticated encryption primitive so AS is able to encrypt K_C with their module-specific key. The resulting combination of cipher text and tag will make sure only a properly loaded and unmodified module is able to successfully decrypt the connection key. A visual representation of this protocol is presented in Fig. 4.2.

Note however, that AS needs a fresh guarantee that each of its keys is successfully installed. If this were not the case, an attacker could force PMs to reuse a connection key from a previous boot cycle and then replay all recorded communication from that connection. Like during the previously discussed attestation protocol, this would typically be achieved by sending a nonce to which PM has to respond with a MAC over this nonce using K_{PM}. In this case though, K_C can be used instead of a separate nonce, as it already provides 128 bit of fresh data. PM simply responds with the MAC over a magic attestation constant M under K_C to prove to AS it successfully received and installed K_C.

The problem of properly installing connection keys and preventing unauthorized key reinstallation is further explored in section 4.3.1. More formally, it must be the case that as soon as AS considers a module initialized, the module has installed all connection keys provided by AS and only those keys.

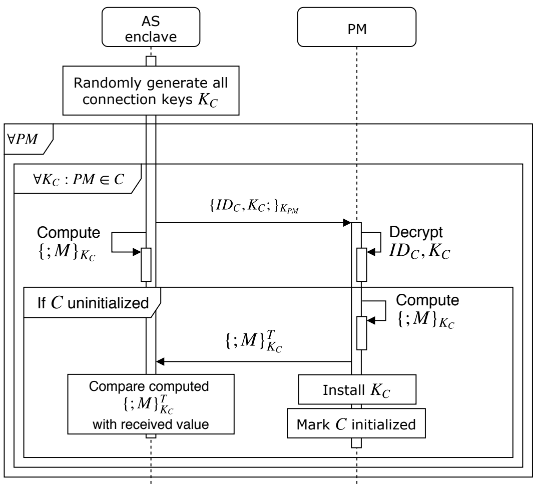

Figure 4.2: Distribution of connection keys K_C to protected modules (PM) by the attestation server (AS).

4.3.1 Key Reinstallation

Proper key installation, together with protection against reinstallation, has proven to be a non-trivial problem. A notable example being the recent discovery of a key reinstallation attack against WPA2 [61]. In this case, the problem is a weakness in the standard rather than the implementations. The other way around is also possible, as an inconsistency was discovered during research for this thesis in the implementation of authentic execution for Sancus. Here, the specification [46] states a new key can only be installed for a specific connection id when its current value in the key table is still zero. Yet this check is missing from the implementation[1] which is shown in Listing 4.1. Received keys are unconditionally unwrapped on line 10-11 and installed in the key table `__sm_io_keys`. In addition, the absence of a mechanism to enforce freshness makes this implementation vulnerable to replayed keys. These incidents justify the need for a thorough analysis of the requirements of key installation and the protection against key reinstallation. The remainder of this section discusses those requirements and how the design presented in this chapter achieves them.

[1]https://github.com/sancus-pma/sancus-compiler/blob/5d5cb/src/stubs/sm_set_key.c#L11

```c
void SM_ENTRY(SM_NAME) __sm_set_key(
    const uint8_t* ad, const uint8_t* cipher,
    const uint8_t* tag, uint8_t* result)
{
    uint16_t conn_id = (ad[2] << 8) | ad[3];
    // ...
    if (conn_id >= SM_NUM_CONNECTIONS)
        code = IllegalConnection;
    else if (!sancus_unwrap(ad, 4, cipher, SANCUS_KEY_SIZE, tag,
            __sm_io_keys[conn_id]))
    {
        code = MalformedPayload;
    }
    // ...
}
```

Listing 4.1: Vulnerable code snippet from Sancus' secure I/O implementation responsible for installation of connection keys.

Reinstallation Defenses in the VulCAN Attestation Server

One approach to make sure an attacker is not able to reinstall a recorded connection key to PM is to never use the same encryption key K_{PM}. Remember that this key is derived from (1) a node-specific hardware key, (2) the software provider identity, and (3) the module identity (text section and layout information). The easiest way to cause a change in K_{PM} is to make a single change in the text section of the module. This change in module-specific key must then be reflected in the network topology information of AS. Both redeploying a module and subsequently changing the topology information requires the involvement of the trusted software provider. Depending on the application, this may be undesirable. In the context of the automotive example, a car owner should not be required to return his or her car to the manufacturer after each use.

Clearly, there should be a way for AS to establish the guarantee that its freshly generated connection key has been successfully installed at PM, without encrypting it with a new encryption key every boot cycle. To enforce this guarantee, AS requires PM to respond with a MAC over the magic attestation constant M under K_C. Successful decryption of the key distribution message (and thus possession of K_C) implies possession of K_{PM}. PM can subsequently use K_C to compute the MAC over M. If AS receives the correct MAC for its value of K_C, it knows PM has correctly installed the connection key. This communication is still vulnerable to reinstallation attacks, however, as an attacker could provide PM with a replayed connection key after AS has completed its key distribution. In that case, AS considers PM to have installed the correct connection key, as it has responded with the correct MAC. Unbeknownst to AS, PM is actually using a replayed key provided at a later time by the attacker.

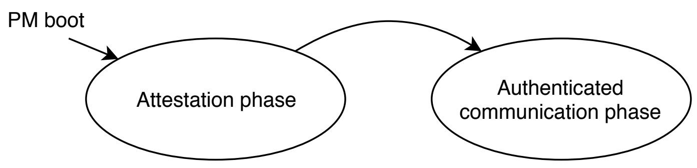

Figure 4.3: State diagram for each authenticated communication connection.

To eliminate the case where an attacker replaces PM's connection keys with replayed ones, PM should only accept a single connection key for each of its connections each boot cycle. Specifically, all of PM's connections should start out uninitialized, indicating no connection key has yet been provided by AS. As soon as one is received and responded to, the corresponding connection should be considered initialized and its connection key considered final. The state of each connection can therefore be described as a state machine with only two states, as depicted in figure 4.3. In the initial state, PM still requires a connection key and thus cannot engage in authenticated communication. PM transitions to the authenticated communication state for a connection as soon as the communication key is received. Notice that there is no way for PM to go back to accepting a connection key for this particular connection. This way, even if an attacker replays a recorded key distribution sequence, PM will not replace the existing key. Of course, an attacker is also able to send a recorded sequence before AS is able to do the same. In this case, PM will reject all legitimate key distribution sequences coming from AS and never respond with the correct MAC over the attestation constant. Consequently, AS will not consider PM to have all connection keys correctly installed and must prevent the system from performing any (safety critical) actions. In the automotive example, an incomplete initialization of a module with a critical responsibility (e.g. breaking or accelerating) will prevent AS from starting the vehicle.

4.3.2 Combination with Attestation

To prevent malicious key reinstallation by an attacker, the response in the key distribution protocol is constructed by PM using K_C which is contained in the key distribution sequence. It can now be shown that the knowledge AS gains from the response includes the aim of the original attestation protocol. Indeed, every time PM responds with the MAC using the connection key, AS reconfirms the knowledge originally gained from attesting the module.

During the attestation protocol, AS receives proof that PM was able to derive K_{PM}, which it could have only done when the expected source code is running with the expected memory layout on the expected device. Freshness of this proof is ensured by sending a randomly generated challenge PM must calculate the MAC of, with K_{PM}. During key distribution, PM replies with the MAC over the attestation constant using K_C as the key, which proves possession of K_C. In turn, possession of K_C proves possession of K_{PM}, as PM was able to decrypt the key distribution sequence. In this case, freshness is ensured by the randomly generated connection

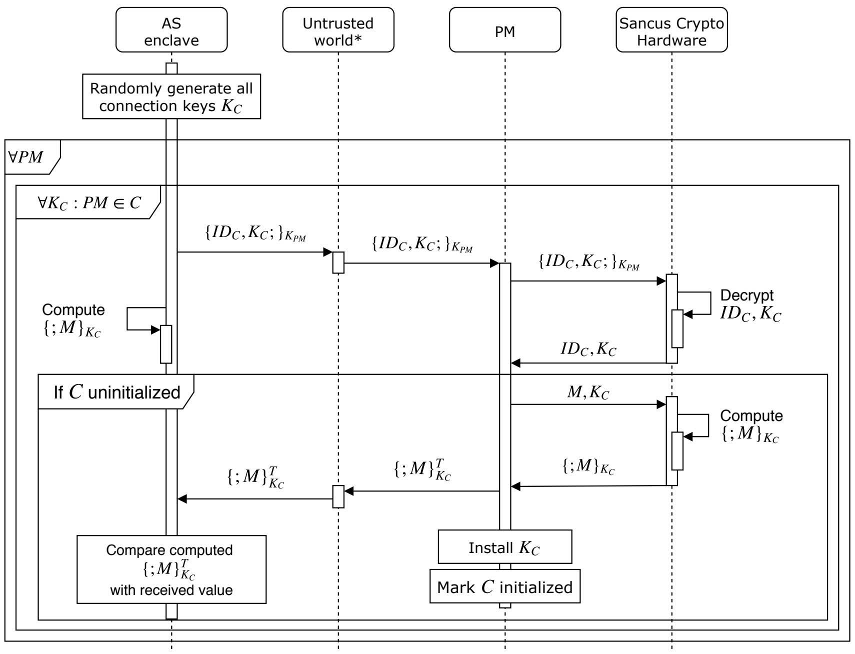

Figure 4.4: Extended view of key distribution and attestation protocols. *Comprises any intermediary software and hardware entities (e.g. network media and corresponding software stacks)

key K_C. An extended view of the protocol across all actors involved is shown in Fig. 4.4. Both protocols provide AS with proof PM was able to derive K_{PM}. As a result, when PM requires one or more connection keys, AS is allowed to omit the attestation protocol entirely. The length of the entire setup phase can thus be reduced to only the required key distributions.

4.4 Network Changes

The third responsibility of the attestation server is to support changes to the network of nodes. First, this section defines how the network topology will be represented and what information needs to be stored by the attestation server. Then, it describes what needs to happen when a node is added, deleted or replaced.

4.4.1 Network Topology

For an attestation server to be able to attest and provision connection keys K_C to participating PMs, it has to have a complete view of the topology of the network. This topology conceptually needs to consist of two pieces of information. First, AS needs a set of K_{PM}s representing all PMs that it needs to attest during the attestation process. Each K_{PM} is used to calculate the MAC over the challenge during attestation, and to encrypt the key distribution sequence during key provisioning. Second, the topology contains a set of connections. Each of these connections is defined as a set of PMs that need to participate in authenticated communication with other members of this set. All of these PMs need to be in possession of the connection key K_C if they want to send or receive messages on this connection.

Notice that this topology is fairly static. Node additions, replacements, or deletions are assumed to be infrequent. Additionally, the attestation server needs this information during every boot cycle. Therefore it makes sense for this information to be stored on persistent storage. Looking at the information that is present in this topology however, it is clear that some of it is confidential data. Specifically, all module-specific keys K_{PM} should not be stored in plain text on untrusted storage. This data should be stored in such a way that only the attestation server itself is able to read it. The attestation server therefore requires the underlying PMA to allow sealing of data. This enables AS to wrap the topology data in a secure way to disk, enforcing that it is the only entity that is able to unwrap the data at a later point in time. In this case, AS seals its secret keys K_{PM} every time changes are made and unseals them whenever it boots up to start the attestation and key provisioning processes.

4.4.2 Obtaining module-specific keys

Storing K_{PM} directly is not the only way for AS to obtain this key. In the Sancus key derivation scheme explained in Sect. 2.2.3, there are multiple ways for AS to derive K_{PM} from its components. This section evaluates all of these options in terms of risk when the contents of the topology are leaked.

The key derivation scheme starts with the node-specific key K_N. To be able to derive K_{PM} from this, one also needs access to the software provider identity and the module identity, both of which are publicly available, non-confidential data. A problem with this approach, however, is that now K_N is not only known by IP, but also by applications requiring a secure communication channel to PM. If this key is compromised at any point in time by any of these applications, attackers are able to derive the module-specific key of all current and future modules deployed on N. As a result, they are able to decrypt and tamper with any traffic encrypted with K_{PM}.

Another option is to provide the enclave with the provider key $K_{N,SP}$, a key specific to the node and software provider. Only the last step in the key derivation scheme needs to be performed, which is to derive K_{PM} from $K_{N,SP}$ and the module identity. Now, the application needs to keep the provider key for this node secure. In case this key leaks, attackers are not able to derive the key of all modules on the node like with the previous approach, but they are able to do this for all modules of this specific software provider.

The third option is to provide K_{PM} directly, so no more keys have to be derived by the enclave itself. In contrast to the other approaches, a compromised key now only enables attackers to interfere with communication to and from PM. Attackers would not be able to derive the keys for other existing or future modules on N with this knowledge. For PM to be secure again, SP would need to redeploy it with a change in its code section. Based on these observations, the third approach where an application such as AS stores K_{PM} directly involves the least risk when keys do get compromised.

4.4.3 Topology Changes

Rarely will a system of distributed nodes be free of any changes during its operation: physical microcontrollers break down over time and need replacement, changing requirements cause new nodes to be created, old nodes to be deleted, and existing software to be updated with new features and bug fixes. Any addition, deletion, or replacement of a node in the network leads to the same change happening on all PMs deployed on that node. The following paragraphs describe what should happen to the topology in each of these cases.

Regarding state continuity of the topology data it is important to note that, apart from confidentiality and integrity, no additional security measures must be taken as long as the K_{PM} of each module is assumed to be confidential at all times. Any replay attack against the topology data will in that case result in an old set of valid connections being initialized. AS will also not be able to complete attestation for any new or changed modules as their K_{PM} will no longer be known, hampering only the availability of the system. If protection against a possible leak of K_{PM} is required, some state continuity solution should be applied to the data for reasons explained below.

Addition. The introduction of a new PM requires that its K_{PM} is added to the set used for attestation. The added PM may also need to communicate with one or more other PMs. If this is the case, either the topology needs to be extended with new connections, or PM is added to existing connection sets.

Deletion. Whenever a PM is removed from the network, the corresponding K_{PM} should be removed from the set that AS uses to attest all PMs. This is especially the case if AS links some critical actions to the event that all PMs have been successfully attested. For example, in the context of vehicular ECUs, engine ignition may require attestation of all PMs, as suggested by Van Bulck et al. [57]. Usability of the system would be severely impacted when the topology is out of date and contains a nonexistent PM, as the car would simply refuse to start. There are also zero or more connections that the removed PM was part of. The topology should be updated to remove PM from these sets as well. This is not as critical as with the attestation set however, because sending out an encrypted key distribution sequence that no PM can decrypt and use does no harm. This is of course assuming K_{PM} was not compromised. If the key was compromised, removing it from existing connection sets is very important. Otherwise attackers can learn the connection key other PMs in the set are currently using.

Replacement. Replacing an existing PM changes the node-specific key of the device modules are running on. Specifically, it means that K_{PM} of those modules is now different as well, as it is derived from both the hardware and software module identity. Consequently, the old K_{PM} has to be substituted for the new one in both the set used for attestation and any connection set PM is a participant in. Same as with deletion, state continuity of this change to the topology is important when the old K_{PM} might be compromised.

4.5 Security Evaluation

What remains is to make sure an attacker is not able to perform man-in-the-middle or brute force attacks against the attestation process. For the attestation server to conclude PM is properly loaded, it has to receive the correct 64-bit MAC as a response to the challenge it sent. This means that an attacker's goal is to provide the correct MAC to AS while, for example, having tampered with the software running inside PM. Multiple ways an attacker could try to achieve this are discussed below.

Brute-Force the Attestation Secret

Obviously, an attacker is able to imitate PM in the attestation process if he possesses the attestation secret K_{PM}. An off-line brute force attempt could be launched to obtain this secret. Guesses can be made by enumerating all possible secret values or otherwise generated. To verify each guess, an attacker needs one or more previously issued challenges and their observed corresponding MAC values. Any previous boot cycle where this specific PM was involved has produced such a pair during the attestation process. Verifying the guess is then done by comparing the attacker's computed MAC to the observed MAC. Such an attack can be mitigated by using a secret that is large enough (e.g., a 128-bit key) so that trying to find it this way becomes highly impractical.

Spoof Challenge Values

Note that AS only needs to be presented with the correct MAC to its challenge for it to conclude PM is attested. This means that an attacker can trick AS into believing PM is attested if he gets hold of the MAC in a different way. There are multiple ways an attacker could possibly go about this. The off-line approach would be to brute force all possible challenges under all attestation secrets, which is also considered impractical. An on-line approach also exists, however, and relies on being able to send many challenge values to PM and, in doing so, obtaining the corresponding MACs. PM may start listening for incoming challenges as soon as it is loaded and stops listening as soon as it is provided its connection keys.

It seems that an attacker could simply use this mechanism to subsequently spoof all challenge values and, in doing so, obtain the responses containing the corresponding MACs. This is considered to be infeasible, as it would take 32 days of this setup time to obtain only 1% of all 2^{48} challenge MACs assuming each request takes 1μs to complete.

$p(n, 2^{48})$	n
50%	1.98×10^7
90%	3.60×10^7
99%	5.09×10^7

Table 4.1: Number of boot cycles needed for collision probabilities of 50%, 90% and 99%.

Observe Recurring Challenges

Finally, when using randomly generated challenge values, eventually an attacker will observe recurring challenges. This is an issue since the correct response to this challenge has already appeared on the network and thus can easily be reused by the attacker. This is important to consider as VatiCAN, an alternative protocol for authenticated communication, was shown to be vulnerable to this kind of attack in the appendix of [57]. It can be argued though, that such a collision in this protocol is sufficiently rare to be of any practical use to an attacker. The probability of collision within a group of n numbers selected from a total of d randomly generated numbers, can be described as an instance of the "Birthday problem" [64, 43]. Its approximation is given by:

$$p(n, d) \approx 1 - e^{\frac{-n(n-1)}{2d}} \tag{4.1}$$

Table 4.1 shows the values of n for collision probabilities 50%, 90%, and 99% when applying it to 2^{48} total challenges. These numbers support the claim that a collision event is sufficiently rare, as the boot cycle would need to be completed almost 20 million times to have a 50% chance of challenge reuse.

An initial straw-man proposal might have been to use a monotonically increasing counter instead of a randomly generated challenge value. The upside of this is that recurring values are impossible and these birthday attacks are thus not an issue. The problem with this approach is that it introduces predictability in the challenges appearing on the network. An attacker would be able to observe the current value of the counter and predict the challenge for any future boot cycle. As previously explained, any particular challenge to a PM is easily spoofed. Consequently, an attacker is able to record the response to a challenge that would be sent by AS in the future. He is then able to trick AS into considering PM as attested by responding with the obtained MAC.

4.6 Implementation

This thesis includes a prototype implementation of the attestation design outlined in this chapter. The implementation itself can roughly be split into three parts, of which one has already been discussed and the two others are discussed in this section. First, there is the actual attestation server implementation as an SGX enclave using Rust.

Underlying that is the Rust Spongent implementation discussed in Sect. 3.1. Third, to achieve a working prototype, the VulCAN implementation on Sancus was updated to execute the receiving part of the key distribution protocol and subsequently install connection keys for its authenticated communication protocol.

4.6.1 Attestation Server

At the core of the attestation server application is the attestation server enclave, implemented as an SGX enclave. When developing software for SGX, one has to take into account some restrictions. For example, there are many functions in the C standard library that are not supported [5]. To enable Rust code to be compiled for SGX, the library has to be marked `#![no_std]`, which disables use of the Rust standard library. As a result, many useful features like collections (e.g. `Vec`, `HashMap`, `String`) become unavailable. To ease SGX development in Rust, Ran Duan et al. at Baidu X-lab developed a Rust-SGX SDK crate [25, 4]. The crate implements many features in the Rust standard library in terms of SGX primitives (e.g. collections, secure file I/O, RNG, synchronization mechanisms). The general algorithm implemented for the attestation server is outlined in Alg. 1.

Algorithm 1: Key distribution by attestation server

```
   Data: participation: map of PM to connections
   Result: Set of attested PMs
1  Randomly generate $K_C$ for all $C$;
2  for $(PM, connections) \in participation$ do
3      for $ID_C \in connections$ do
4          Send $\{ID_C, K_C\}_{K_{PM}}$;
5          Receive $\{;\texttt{0xA77E57ED}\}_{K_C}$;
6      Add PM to attested;
```

In the prototype, *participation* is a map from unique PM identifiers to a set of unique connection identifiers. PM identifiers are 16 bit numbers and defined by the application. Connection identifiers are also 16 bit long and application-defined. Because the CAN bus is used in this prototype, connection identifiers map directly to CAN identifiers. Consequently, although they are defined by the application, extra care should be taken when selecting connection identifiers so they don't collide with existing CAN identifiers used on the bus.

The actual messages that are sent to PM are detailed in Fig. 4.5 and Fig. 4.6. In addition to $\{ID_C, K_C\}_{K_{PM}}$, the first message contains the identifier of PM in plain text. This way PM is able to quickly discard sequences that are not meant for it and thus can save on clock cycles and consequently power usage as it's not trying to decrypt the entire sequence. Note that this is purely an implementation optimization and does not affect the security properties of the protocol. Following this identifier, the cipher text and 128-bit tag of the connection identifier and key are sent. To be able to send this resulting 320-bit payload over the CAN bus, it

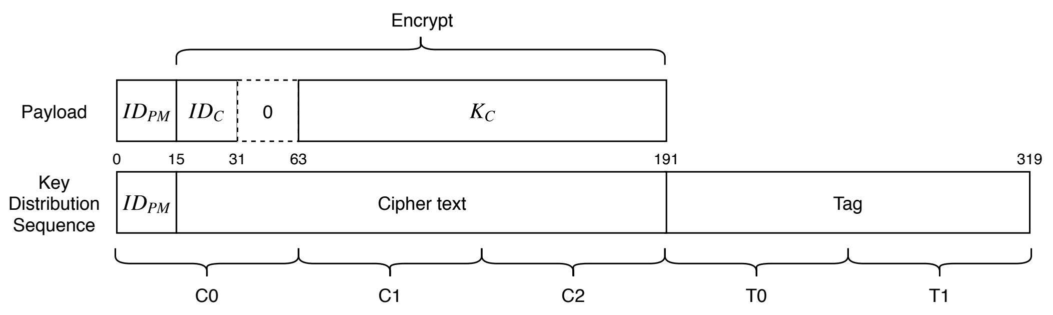

Figure 4.5: Decomposition of the payloads of each Key Distribution Sequence (KDS).

must be split up in five separate CAN messages (denoted C0, C1, C2, T0, and T1 in Fig. 4.5). The hex constant `0xA77E57ED` is used as the magic attestation constant. In addition, between the AS enclave and PM in Fig. 4.6, CAN messages are represented as "***id,payload***", where ***id*** denotes the CAN id on which the message is being sent and ***payload*** denotes the CAN payload. The identifiers CAN_{Send} and CAN_{Recv} are reserved CAN identifiers for the two channels used by AS to send and receive its traffic.

Security Evaluation

While the implementation complies with the specification from Sect. 4.3, there are certain details specific to this implementation of which the security implications have to be addressed.

First, communication between AS and all PMs happens over a standard CAN bus. Because of the 8 byte payload length limitation on CAN, key distribution has to be split over a sequence of messages. This does introduce additional possibilities for attackers to tamper with the communication on the network. In particular, they are able to change the order the messages arrive in at PM in an arbitrary way, possibly changing the meaning of the entire sequence. However, each message contains a part of the cypher text or MAC of the payload. Any reordering will therefore result in a failed attempt by PM to decipher the sequence. Attackers can also remove messages or insert their own messages into the sequence. This will all lead to a failed decryption attempt as well, as without the proper key, the attackers are not able to provide the correct MAC for this new sequence. Consequently, no connection key will be installed by PM. This only has an effect on availability, which is a non-objective of this design.

A second change in this implementation is the addition of the plaintext module identifier as the first 16 bits of the first message of the key distribution sequence. Its purpose is to allow modules to quickly discard message sequences that are not meant for them. A module will only try to decrypt a sequence if the identifier matches its own, which helps the module to conserve clock cycles and power. Because the identifier is sent in plain text, attackers are able to redirect a sequence to another

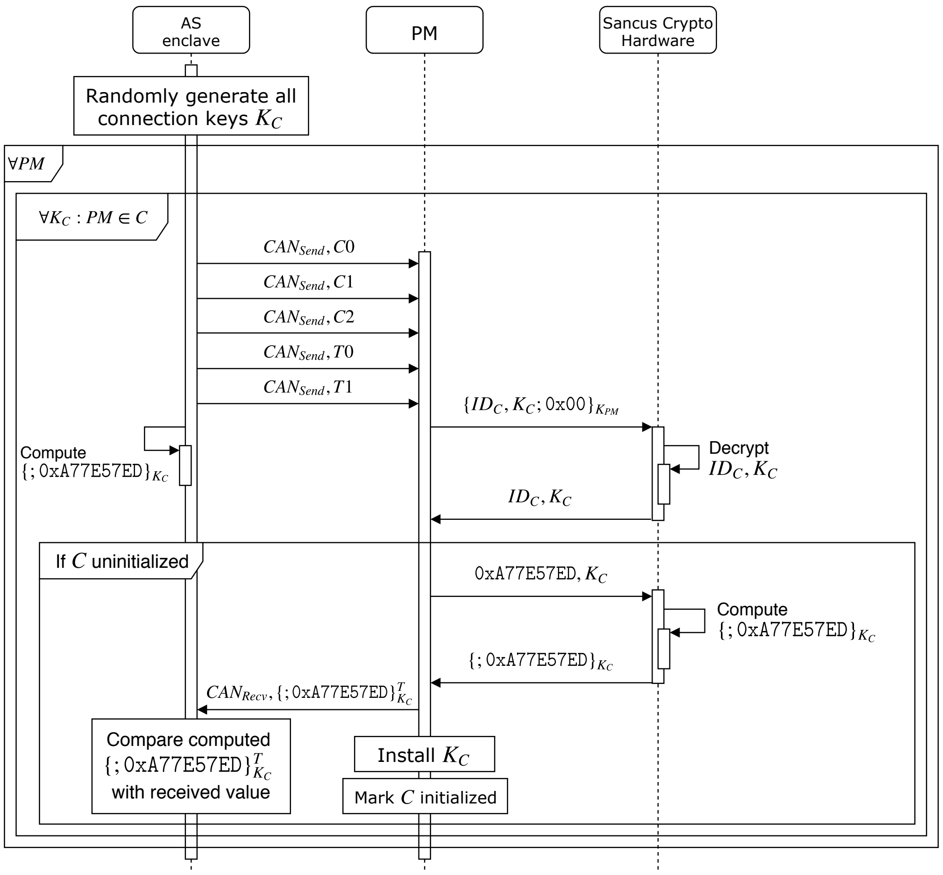
AS enclave
PM
Sancus Crypto Hardware
Randomly generate all connection keys K_C
$\forall PM$
$\forall K_C : PM \in C$
$CAN_{Send}, C0$
$CAN_{Send}, C1$
$CAN_{Send}, C2$
$CAN_{Send}, T0$
$CAN_{Send}, T1$
$\{ID_C, K_C; \texttt{0x00}\}_{K_{PM}}$
Compute $\{;\texttt{0xA77E57ED}\}_{K_C}$
Decrypt ID_C, K_C
ID_C, K_C
If C uninitialized
$\texttt{0xA77E57ED}, K_C$
Compute $\{;\texttt{0xA77E57ED}\}_{K_C}$
$\{;\texttt{0xA77E57ED}\}_{K_C}$
$CAN_{Recv}, \{;\texttt{0xA77E57ED}\}^T_{K_C}$
Compare computed $\{;\texttt{0xA77E57ED}\}^T_{K_C}$ with received value
Install K_C
Mark C initialized

Figure 4.6: Extended view of the key distribution implementation. The untrusted world between the AS enclave and each PM is left out for legibility.

module. This other module would then fail to decrypt the sequence, because it is encrypted with the original module's K_{PM}. The aim of this change is only to reduce the computational load of participating nodes in the benign case. An attacker would still be able to make a node decrypt way more traffic than is necessary, but this as well is an availability concern. Moreover, an attacker does not gain any advantage when learning which cipher texts are meant for which PM.

4.6.2 Attestation in VulCAN Protected Module

The existing C VulCAN implementation contains hard-coded connection keys for its demo applications. Therefore, this master's thesis also provides a prototype implementation of the receiving end of the key distribution protocol. As this is code running inside a Sancus module, it is written entirely in C. Allowing Sancus modules to be developed in Rust is suggested as a future work opportunity. The algorithm used here to initialize each connection is outlined in Alg. 2. The total number of lines changed in the C implementation is about 200[2].

Algorithm 2: Receiving key distribution on VulCAN PM

```
Data: A connection $C$
Result: $C$ marked initialized
1  Read single CAN message;
2  if first 16 bits match own ID then
3      Read full key distribution sequence (4 more CAN messages);
4      if unwrap $\{C, K_C;\}_{K_{PM}}$ successful then
5          if $C$ marked uninitialized then
6              Install $K_C$ for connection $C$;
7              Send $\{;\texttt{0xA77E57ED}\}_{K_C}$;
8              Mark C initialized;
9  else
10     Discard next 4 CAN messages;
```

4.7 Conclusion

This chapter started by outlining the design goals for an attestation server for VulCAN. Based on these goals, two protocols are developed: an attestation protocol and a key distribution protocol. Both are supplemented with a security evaluation. For attestation, the focus is making sure an attacker cannot trick AS in considering a particular PM as attested while it has been tampered with. Key distribution, and key reinstallation attacks in particular, have proven to be a non-trivial issue. Therefore, special care was taken to prevent these kind of attacks against the key distribution protocol. It was also shown that because of the freshness requirement on key distribution, it provides the attestation server with strictly more knowledge than attestation alone. As a result, performing the separate attestation protocol for a PM is considered optional when PM also requires one or more connection keys. The chapter also presented a brief discussion on network topology changes and their impact on AS. The prototype implementation of the attestation server once again proves the viability of using Rust for the development of SGX enclaves.

[2] https://github.com/stenverbois/vulcan/tree/sten

Chapter 5

Conclusion

This final chapter reflects on the work presented in this thesis while summarizing its contributions and limitations. The introduction started off by arguing the value of isolating software modules with a minimal TCB using PMAs. Both the SGX and Sancus architectures succeed in achieving strong security guarantees for software running on systems where attackers have arbitrary code execution capabilities. However, they both have limitations as well. Where SGX lacks built-in support for trusted I/O paths [63], embedded Sancus nodes lack SGX's computational power. Therefore, the work in this thesis is inspired by the need for heterogeneous networks of PMAs. To that end, it details the requirements to achieve secure communication between SGX enclaves and Sancus modules. As a proof of concept, prototype implementations for setting up and listening to secure communication on a CAN bus using VulCAN are included. Additionally, it is shown that using Rust to develop SGX enclaves requires very little extra programmer effort while achieving similar runtime performance and much better security guarantees.

This chapter first discusses the limitations and challenges of the developed prototypes in Sect. 5.1. Then, its contributions are summarized in Sect. 5.2. Finally, Sect. 5.3 outlines opportunities for future work in this area.

5.1 Limitations and Challenges

Initially, the logging enclave was developed to be used in a forensic setting. The log would be retrieved after a car accident to allow an audit to determine if a component sent unusual traffic on the CAN bus. However, as there is no trusted path set up between the CAN bus and the enclave, an attacker might have been able to prevent these messages from being captured in the log. While logging could still be of use in a benign case, this attacker capability must be considered when using this application in practice.

In this thesis, SGX was chosen as the platform for the VulCAN attestation server because it is readily available on modern x86 hardware. In the network of ECUs, AS acts as the root of trust responsible for attesting all other, initially untrusted, participants. Trust in the AS enclave is established by a number of dedicated enclaves developed by Intel. This means an SGX enclave can only be deployed in production after Intel has whitelisted the developer's singing key. It should be notes that the design of both enclaves is in no way tied to SGX. Other PMAs that have support for the required features such as attestation and sealing may be considered [28, 19].

For the attestation server prototype, it is assumed every PM in the prototype needs one or more connection keys to interact with other PMs on the network. The attestation protocol is not separately implemented, as each PM is able to complete attestation using the key distribution protocol.

5.2 Contributions

- Chapter 3 details the requirements needed for setting up a secure communication channel between an SGX enclave and a Sancus module. This chapter includes a prototype implementation in the form of an SGX enclave used to securely log authenticated traffic using LeiA on the CAN bus.
- Using the same motor vehicle application as the previous chapter, Chapter 4 presents the design and implementation of an attestation server capable of attesting PMs over a CAN bus. First, the attestation and key distribution protocols and their security properties are discussed separately. Then, it is shown that the guarantees the attestation server obtains from the response to the key distribution sequence are strictly stronger than those obtained by the attestation protocol. Therefore, the latter can often be omitted to save time and resources.
- The implementation and evaluation sections of both Chapter 3 and 4 show that developing such security critical code in Rust is often easier than C or C++ because of the safe high-level abstractions. Porting certain modules to Rust from these languages is also deemed to require very little programmer effort. Additionally, Sect. 3.1.1 measured Rust's runtime performance and concluded it is similar to C/C++ for the Spongent cryptographic functions.

All source code of both the Spongent cryptographic functions and SGX enclave implementations for VulCAN in Rust are publicly available at `https://github.com/stenverbois/spongent-rs` and `https://github.com/stenverbois/vulcan-rs` respectively.

5.3 Future Work

Using Rust in Sancus modules. Improving support and tooling for embedded devices is part of Rust's focus for 2018. Eventually, an effort should be made to allow compilation of Rust code to Sancus modules on the MSP430 microprocessor. Then, (parts of) the existing VulCAN implementation for Sancus can also be ported to Rust. This would serve the goal of developing all application code running inside a PMA in a safe programming language.

Rust type	EDL annotation
T	T
&T	[in] *T
&mut T	[in, out] *T
&[T]	[in, count=*len*] *T
&mut [T]	[in, out, count=*len*] *T

Table 5.1: Mapping of a few common Rust type signatures to EDL annotations. The unsafe wrapper function that constructs a slice from the raw pointer input would take *len* as an additional parameter.

EDL generation from Rust function definitions. The Enclave Definition Language (EDL) file describes an SGX enclave's trusted and untrusted functions and types. The correctness of the EDL file and the C code that the `Edger8r` tool generates from it is of critical importance, as it was shown to facilitate side channel attacks against the enclave [34]. Apart from writing memory safety vulnerabilities directly in the enclave code (in an unsafe language), accidentally using the wrong EDL annotations on pointer arguments can also introduce them. With insufficient or wrong annotations, the generated C wrappers will not contain the proper checks on these pointer arguments. An extension to the EDL definition has been proposed by van Ginkel et. al. [60]. It requires the programmer to specify the enclave interface in a separation logic based language. Because of Rust's expressive type system however, it may well be possible to deduce all information needed to generate the EDL annotations (and also the wrapper functions generated from them) directly from the function prototype. While it was ultimately deemed out of scope for this master's thesis, early research indicates this is indeed the case (see Table 5.1).

Fully abstract compilation for Rust. The expressive type system enables the Rust compiler to enforce strong safety guarantees on the Rust code it compiles. In the context of SGX however, this code is compiled to a dynamic library that interacts at runtime with an untrusted context. Many of the static type level checks are completely discarded during compilation. There is therefore a risk of triggering undefined behavior by providing malformed input to the module. Listing 5.3 shows a concrete example of such behavior. A function `accelerate` is shown which defines some change in velocity of some entity based on a `turbo` boolean. The intended behavior here is that this function would return the integer 11 when the entity is accelerating with turbo and return the integer 10 when it is accelerating normally. When this function is on the boundary between the trusted and untrusted contexts, an attacker can provide some value that does not follow Rust's internal representation of a boolean. The x86 assembly generated by the compiler at the bottom of Listing 5.3 shows that because the internal representation of a boolean is `0x01` for `true` and `0x00` for `false`, the compiler can effectively just add `0xa` or decimal 10 to whatever the input boolean was. This means that when an attacker calls this function from

C with an input value of 50, this results in the function returning the value 60 even though this would never by possible in a pure Rust context. This can be considered very problematic when these concepts of velocity and acceleration are directly controlling some real world object such as a car or an industrial machine. Note that while this is a carefully crafted and highly specific example, it supports the idea that extra security measures should be implemented to check the validity of input values regardless of language semantics when interacting with an unsafe context.

```rust
#[no_mangle]
pub extern "C" fn accelerate(turbo: bool) -> u32 {
    if accelerate { 11 } else { 10 }
}
```

```asm
# <velocity_change>:
lea     0xa(%rdi),%eax
retq
```

Listing 5.1: Code snippet showing compiler optimizations that can be misused by an attacker (top) and resulting assembly with rustc 1.26.0 (bottom).

Bibliography

[1] Intel Developer Zone: SGX. URL: `https://software.intel.com/en-us/sgx`, last checked on 2018-05-14.

[2] Lines of code of the Linux kernel versions. URL: `https://www.linuxcounter.net/statistics/kernel`, last checked on 2018-05-10.

[3] Rust. URL: `https://www.rust-lang.org`, last checked on 2018-04-10.

[4] Rust-SGX-SDK by Baidu X-Lab. URL: `https://github.com/baidu/rust-sgx-sdk`, last checked on 2018-05-05.

[5] Unsupported C standard Functions in SGX. URL: `https://software.intel.com/en-us/node/709259`, last checked on 2018-05-05.

[6] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[7] T. M. Austin, S. E. Breach, and G. S. Sohi. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[8] AUTOSAR. Specification of Secure Onboard Communication, 2017.

[9] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *International Conference on Computer Aided Verification*, pages 178–183. Springer, 2011.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[12] H.-J. Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN Notices*, volume 28, pages 197–206. ACM, 1993.

[13] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. Spongent: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62(10):2041–2053, 2013.

[14] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proc. R. Soc. Lond. A*, 426(1871):233–271, 1989.

[15] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[16] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[17] S. Corrigan. *Introduction to the Controller Area Network (CAN)*. Texas Instruments, May 2016.

[18] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[20] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[21] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *NDSS*, 2013.

[22] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 97–106. ACM, 2010.

[23] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, volume 37, pages 57–68. ACM, 2002.

[24] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[25] Y. Ding, R. Duan, L. Li, Y. Cheng, Y. Zhang, T. Chen, T. Wei, and H. Wang. Poster: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2491–2493. ACM, 2017.

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

[27] D. Gay and A. Aiken. *Memory management with explicit regions*, volume 33. ACM, 1998.

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

[29] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. *ACM Sigplan Notices*, 37(5):282–293, 2002.

[30] B. Groza, S. Murvay, A. Van Herrewege, and I. Verbauwhede. Libra-can: a lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security*, pages 185–200. Springer, 2012.

[31] S. Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive*, 2016:204, 2016.

[32] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144. ACM, 2012.

[33] A. Hazem and H. Fahmy. Lcap-a lightweight can authentication protocol for securing in-vehicle networks. In *10th escar Embedded Security in Cars Conference, Berlin, Germany*, volume 6, 2012.

[34] Intel Corporation. Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits. 2018.

[35] B. Jacobs, J. Smans, and F. Piessens. The Verifast program verifier: A tutorial. *Tech. Rep.*, 2014.

[36] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66, 2017.

[37] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[38] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata. CaCAN - Centralized Authentication System in CAN. 11 2014.

[39] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[40] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732. ACM, 2013.

[41] A. Madalin Ghenea. A Security Kernel for Protected Module Architectures. Master's thesis, KU Leuven, 2017. `https://distrinet.cs.kuleuven.be/software/sancus/publications/madalinghenea17thesis.pdf`.

[42] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, PP(99), 2017.

[43] E. H. McKinney. Generalized birthday problem. *The American Mathematical Monthly*, 73(4):385–387, 1966.

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

[45] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[46] J. Noorman, J. T. Mühlberg, and F. Piessens. Authentic Execution of Distributed Event-Driven Applications with a Small TCB. In *13th International Workshop on Security and Trust Management (STM'17)*, volume 10547 of *LNCS*, pages 55–71, Heidelberg, 2017. Springer.

[47] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7:1–7:33, September 2017.

[48] S. Nürnberger and C. Rossow. –vatiCAN–Vetted, Authenticated CAN Bus. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 106–124. Springer, 2016.

[49] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *arXiv preprint arXiv:1702.00719*, 2017.

[50] A. One. Smashing the stack for fun and profit. Phrack 49, 1996.

[51] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[52] A.-I. Radu and F. D. Garcia. LeiA: a lightweight authentication protocol for CAN. In *European Symposium on Research in Computer Security*, pages 283–300. Springer, 2016.

[53] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[54] Stanley Law Group. Class Action Lawsuit Filed To Hold Toyota, Ford And GM Accountable For Dangerous Defects Allowing Cars To Be Hacked And Drivers To Lose Control. `https://www.prnewswire.com/news-releases/class-action-lawsuit-filed-to-hold-toyota-ford-and-gm-accountable-for-dangerous-defects-allowing-cars-to-be-hacked-and-drivers-to-lose-control-300048163.html`, 2015.

[55] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[56] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 344–361. Springer, 2010.

[57] J. Van Bulck, J. T. Mühlberg, and F. Piessens. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 225–237. ACM, 2017.

[58] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. Towards Availability and Real-Time Guarantees for Protected Module Architectures. In *Companion Proceedings of the 15th International Conference on Modularity (MASS'16)*, pages 146–151. ACM, 2016.

[59] V. Van der Veen, L. Cavallaro, H. Bos, et al. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, pages 86–106. Springer, 2012.

[60] N. van Ginkel, R. Strackx, and F. Piessens. Automatically Generating Secure Wrappers for SGX Enclaves from Separation Logic Specifications. In *Asian Symposium on Programming Languages and Systems*, pages 105–123. Springer, 2017.

[61] M. Vanhoef and F. Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.

[62] P. Wagle, C. Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255. Citeseer, 2003.

[63] S. Weiser and M. Werner. SGXIO: Generic trusted I/O path for intel SGX. *CoRR*, abs/1701.01061, 2017.

[64] E. W. Weisstein. Birthday problem. 2003.

[65] A. Wright. Hacking cars. *Communications of the ACM*, 54(11):18–19, 2011.

Fiche masterproef

Student: Sten Verbois

Titel: Safe Interacting Enclaves for Heterogeneous Protected Module Architectures

Nederlandse titel: Veilige Interactie van Enclaven voor Heterogene Beveiligingsachitecturen

UDC: 681.3

Korte inhoud:

There has been a big increase in connected computing devices in recent years. Some of them are handling privacy-sensitive information in the cloud or performing safety-critical actions in modern automotive systems. Assuring all of those devices are secure and performing as expected is a big challenge. To that end, different software isolation techniques have been widely implemented in high-end systems. On embedded devices however, these techniques are often omitted because of resource constraints.

Recent research on Protected Module Architectures (PMAs) aims to provide efficient isolation of software modules from any compromised software running on the system. Concrete implementations of PMAs exist both for high-end systems and low-end embedded devices. In practice, many large applications will likely consist of a heterogeneous set of platforms and thus a heterogeneous set of PMAs.

This master's thesis looks into the secure interaction of different PMAs. A first important observation is that there is no isolation mechanism that can protect against a modules' own source code when it contains memory safety vulnerabilities like the ones that are common in unsafe languages like C and C++ . Taking into account these memory safety issues, a first main contribution of this master's thesis is to propose the use of Rust as an alternative to C/C++ for writing code that executes inside a protected module. Two other, more practical, contributions focus on the specific PMAs Intel SGX and Sancus in the context of automotive control networks. The first one being an SGX enclave containing a rust port of the LeiA message authentication protocol. This enclave is able to keep a secure log of all authenticated traffic it observed on the CAN bus. The second enclave is a realization of an attestation server for such networks, a central entity responsible for attesting participating Sancus modules and providing them with fresh connection keys. Both show the viability of interaction between PMAs via secure communication channels, implemented in a safe and fast programming language.

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

Promotor: Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessor: Dr. R. Strackx
Dr. G. Sedrakyan

Begeleider: Ir. J. Van Bulck