KU LEUVEN

FACULTEIT WETENSCHAPPEN

Security Enhanced LLVM

Hans Winderix

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

Assessoren:
Ir. J. Van Bulck
Dr. S. Delbruel

Begeleiders:
Dr. J.T. Mühlberg
Ir. N. van Ginkel

Proefschrift ingediend tot het
behalen van de graad van
Master of Science in toegepaste informatica,
specialisatie software-ontwikkeling
en gedistribueerde systemen

Academiejaar 2017-2018

© Copyright by KU Leuven

Without written permission of the supervisors and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to KU Leuven, Faculteit Wetenschappen, Geel Huis, Kasteelpark Arenberg 11 bus 2100, 3001 Leuven (Heverlee), Telephone +32 16 32 14 01.

A written permission of the supervisor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Preface

Software security is a rapidly evolving field of research. This master's thesis has given me the opportunity to catch up in some areas and to get acquainted with interesting novel ideas like secure compilation and PMAs. The result of this endeavour is a modest contribution to the software security community in the form of a proposal for a unified secure compiler infrastructure.

There are some people who contributed directly or indirectly to this master's thesis and I would like to take the occasion to thank them for helping me along the way. Foremost, I would like to thank my daily advisors Jan Tobias Mühlberg and Neline van Ginkel for their continuous support. Thanks for letting me explore in a very independent way this exciting intersection between compilers and software security. A special thanks goes to JT for convincing me to do a master's thesis in the field of software security, which opened up a fascinating world to me.

My gratitude also goes to my other supervisor, prof. Piessens, for providing me with this opportunity and for being such a great teacher. It is a privilege to have been one of your students. I would also like to thank Jo van Bulck for explaining, discussing and sharing some insights with me on the Sancus security architecture and to give me some pointers to related work.

Thanks to Iebe, my best friend.

And last but not least, I would like to thank Wannes, Kato and Marie, who have been very supportive of their father's crazy idea of getting a university degree at the age of 39 while raising three such amazing children. I could not have done this without your support.

Hans Winderix

Contents

Abstract

In today's technology-driven world, computers are ubiquitous and we expect them to behave in a correct and secure way. We rely on computing devices to control critical infrastructure. An increasing number of applications are being trusted with security-critical information. The computer security community is continuously looking for innovative defensive measures against a wide range of security threats to protect our computing infrastructure from tampering and theft.

Secure compilation and PMAs are two such novel research areas. Secure compilers aim to preserve the security properties of high-level programming language abstractions after compilation, allowing to reason about application security at a comfortable abstraction level. PMAs are able to provide an efficient mechanism to enforce some of these properties. This type of security architectures makes it possible to isolate security-critical modules to protect them from their enclosing application. Implementations are typically realized with a low-level security mechanism and a product-specific compiler.

This master's thesis explores the feasibility of a secure compiler infrastructure where security properties can be represented and manipulated in a generic way at the different layers of abstraction. A generic compiler infrastructure where common programming models and common algorithms for analysis and transformation can support a wide range of programming languages and target architectures. The main contribution of this master's thesis is the proposal of such a generic secure compiler infrastructure. A proof of concept implementation of this proposal is provided in the form of an LLVM extension for the property of software module isolation, supporting the C and Rust programming languages and the Sancus and Intel SGX PMAs.

The work presented in this master's thesis demonstrates that by sharing a common infrastructure, improvements in uniformity, reusability, programmability and performance can lead to more secure applications.

Samenvatting

Computers zijn alomtegenwoordig en niet meer weg te denken uit onze huidige gedigitaliseerde samenleving. Zo vertrouwen we een toenemend aantal toepassingen privacygevoelige informatie toe en zijn we afhankelijk geworden van computersystemen die instaan voor het beheer van vitale infrastructuur. We verwachten dan ook dat ze zich op een juiste en veilige manier gedragen. Onderzoekers zijn onophoudelijk op zoek naar innovatieve maatregelen die onze computerinfrastructuur moeten beveiligen tegen een aanzienlijk aantal gevaren.

Twee nieuwe onderzoeksdomeinen die hieruit zijn voortgekomen, zijn veilige compilatie en PMAs. Met veilige compilatie beoogt men om de eigenschappen omtrent beveiliging, die uitgedrukt zijn door concepten in een hogere programmeertaal, te behouden nadat de programma's zijn omgezet naar uitvoerbare machinecode. Deze techniek laat ontwikkelaars toe om na te denken over de bescherming van computertoepassingen op een comfortabel abstractieniveau. PMAs vertegenwoordigen een nieuw type beveiligingsarchitecturen dat toelaat om sommige van deze eigenschappen op een efficiënte manier af te dwingen. Ze laten toe om de beveiligingsgevoelige delen van een toepassing te beschermen door ze in afzondering van de rest van de toepassing uit te voeren. PMAs worden gewoonlijk verwezenlijkt door een primitief beveiligingsmechanisme en een architectuurspecifieke compiler.

Deze masterscriptie onderzoekt de haalbaarheid van een platform voor veilige compilatie waar de beveiligingseigenschappen op een generieke manier op de verschillende abstractieniveaus kunnen worden voorgesteld en verwerkt. Een platform waar gemeenschappelijke programmeermodellen en gemeenschappelijke algoritmes voor analyse en transformatie tal van programmeertalen en computerarchitecturen ondersteunen. De belangrijkste bijdrage van deze thesis is een voorstel voor zo een eengemaakt platform en een prototype dat de eigenschap van isolatie van software modules, uitgedrukt in de C en Rust programmeertalen, bewaart na compilatie naar de Sancus en Intel SGX PMAs.

Het werk dat hier uiteengezet wordt, toont aan dat door een eengemaakt platform voor veilige compilatie verbeteringen in gelijkvormigheid, herbruikbaarheid en gebruiksvriendelijkheid kunnen leiden tot beter beveiligde toepassingen.

List of Figures and Tables

List of Figures

List of Tables

List of Listings

List of Abbreviations

ACL	Access Control List
API	Application Programming Interface
ASLR	Address Space Layout Randomization
AST	Abstract Syntax Tree
CC	Calling Convention
CIA	Confidentiality, integrity and availability
DSL	Domain Specific Language
EABI	Embedded Application Binary Interface
EDL	Enclave Definition Language
FFI	Foreign Function Interface
HIR	High-level Intermediate Representation
IDL	Interface Definition Language
IP	Infrastructure Provider
IR	Intermediate Representation
ISA	Instruction Set Architecture
MIR	Middle-level Intermediate Representation
MMU	Memory Management Unit
OS	Operating System
PM	Protected Module
PMA	Protected Module Architecture
ROP	Return Oriented Programming

SDK	Software Development Kit
SGX	Software Guard Extensions
SLLVM	Security Enhanced LLVM
SM	Software Module
SP	Software Provider
TCB	Trusted Computing Base
TLB	Translation Look-aside Buffer
XD	eXecute Disable
XN	eXecute Never

Chapter 1

Introduction

In today's digitized society, computing devices and software are omnipresent in our everyday lives. More and more we depend on these technologies and we expect them to behave in a correct and secure way. An increasing number of applications are being deployed and trusted with security-critical and privacy-sensitive information. We rely on secure computer systems to control critical infrastructure such as the facilities for railway networks, water supply and power plants.

The reality is that many contemporary computer systems that are used in sectors such as healthcare and transportation are distressingly insecure [58]. The majority of software that facilitates software developers and end users to interface with computing hardware, so-called system software, has been written in inherently insecure programming languages that are susceptible to a large number of low-level software vulnerabilities [41], [47], [24]. Increased network connectivity and software extensibility have made computing devices even more vulnerable to security threats than ever before. Developing software in a safe programming language can eliminate a wide range of these vulnerabilities but does not provide absolute security guarantees either.

This master's thesis builds on two novel research areas in the secure software community. First, the emerging field of *secure compilation* [14], [42], [43] studies how to preserve the security properties of high-level programming language abstractions in the low-level representation of programs, allowing to reason about the security of applications at the abstraction level of the application's source code. Second, to achieve this goal we need mechanisms that enforce the security properties of the high-level abstractions in an efficient way. Recent research on *PMAs* [50], [52], [39], [37], [40] has proposed such an efficient enforcement mechanism.

This master's thesis explores the feasibility of a unified secure compiler infrastructure. By unifying security abstractions and programming models, by sharing an infrastructure for analyzing and transforming the representations at the different abstraction layers, improvements in uniformity, reusability, programmability and performance can lead to more secure applications.

The property of secure compilation is summarized in Section 1.1. Section 1.2 thereafter briefly explains PMAs and in section 1.3 the contributions of this master's thesis are summarized, revolving around the idea of a generic secure compiler infrastructure. The structure of the thesis text is outlined in 1.4.

1.1 Secure Compilation

Abstraction is an important technique for managing the complexity of computer systems. Declarative statements in a program are used to hide details that are irrelevant at a particular abstraction level. However, when the assumptions and the intuitive properties of an abstraction do not correspond to the concrete implementation, this can lead to security vulnerabilities that are exploitable by attackers interacting with the application at the level of the implementation.

Software developers typically reason about the properties of an application at a high level of abstraction. The root cause of a large number of software vulnerabilities can often be explained as a high-level abstraction in a software application that can be broken in a lower-level representation of that application. Familiar programming abstractions such as *process*, *module*, *function*, *stack*, *type* and *lexical scoping* often only exist in the source code or they are implemented in a way that does not enforce the abstraction at stake.

However, recent research has demonstrated that the security properties that are expressed in the source code of an application can be preserved, even after this high-level representation has been transformed into low-level machine code [14]. The field of secure compilation studies compilation schemes that perform transformations on software entities that result in representations that are as secure as their source-level original [14], [42], [43]. A compiler with this property makes it possible to reason about the security of an application at the abstraction level of the source code.

1.2 Protected Module Architectures

Memory protection is a well-established security technique where the memory of a computer system is partitioned into a number of segments. Access to the individual segments is then managed according to a security policy, usually expressed in terms of some higher level abstractions such as *isolation*, *process* and *module*. The isolation of computer processes, for example, prevents one computer process from accessing memory segments that are allocated to other processes. On conventional computer systems, process isolation is typically realized with separate address spaces through a tight cooperation between an omnipotent monolithic kernel and the computer hardware.

A Protected Module Architecture (PMA) [50], [52] is a novel class of security architectures with a finer grained memory protection scheme to protect small isolated compartments that share an address space. These architectures allow software developers to create an "enclave" in the shared address space to be protected from a potentially malicious surrounding environment. Such an enclave is characterized by a protected code section, a protected data section and a limited number of interaction points between the isolated module and the enclosing application. The PMA prevents unauthorized access to the enclave's memory sections and ensures that the isolated module can exclusively be invoked via one of its interaction points, reminiscent of software design concepts such as information hiding and interfaces.

PMAs can guarantee the isolation properties of a Protected Module (PM) even when the enclosing application has been compromised by an attacker. Hardware based PMAs take this idea a step further and are also able to exclude the Operating System (OS) from the application's Trusted Computing Base (TCB), the set of hardware and software components that has to be trusted not to contain any security vulnerabilities. The larger the TCB, the more likely that it contains security holes that can be exploited [41], [47], [24] and the harder it is to prove that an application is correct and secure.

1.3 Contribution

This master's thesis explores the feasibility of a unified secure compiler infrastructure that allows security properties to be represented and manipulated in a generic way at the different layers of abstraction. For each supported programming language, a generic programming model to express the security properties at stake must be defined, independent from any target architecture. At the level of the Intermediate Representation (IR), the security properties must be expressed independent from any source language and independent from any target architecture. Maximizing genericity in the representation not only facilities smooth language interoperability but also encourages the development of generic algorithms for transformation and analysis. Furthermore, uniformity also reduces the efforts that are required to formally specify and verify the generic models and transformations. The limited time frame of a master's thesis narrows down this exploration to the property of software module isolation, but the principles should be applicable to other security properties as well.

More specifically, the following contributions are made:

- A proposal for a unified secure compiler infrastructure where security properties are represented and manipulated in a generic way at the different abstraction layers, independent from any source language and target architecture.

- A proof of concept of these ideas is implemented for the property of software module isolation that supports the C and Rust programming languages and the Sancus and Intel Software Guard Extensions (SGX) security architectures. Supporting two different programming languages not only demonstrates the genericity of the solution but it also demonstrates that this approach leads to a seamless language interoperability between the supported languages.

- A proposal and a prototype for a new Sancus toolchain with an improved design that adheres to generally accepted software engineering principles. The new toolchain facilitates the development of Sancus-enabled applications with a smaller TCB. Furthermore, the generated machine code exhibits an improved runtime performance when compared to the legacy toolchain. The new toolchain is also an improvement in terms of user-friendliness as it results in improved compile times and it removes the need for a Sancus-specific linker.

- A generic programming model for software module isolation for the LLVM intermediate representation, that expresses the isolation properties independent from any source language and target architecture.
- A generic programming model for software module isolation for the C programming language, that expresses the isolation properties independent from any target architecture with an emphasis on programmability.
- The introduction of the Rust ecosystem to the Sancus security architecture. The legacy Sancus toolchain can only be used for the development of protected modules in C. Safe programming languages like Rust are an effective countermeasure against low-level attacks, a property that makes them complementary to PMAs.
- The foundational work on an alternative to the Baidu Rust SGX Software Development Kit (SDK) for developing Rust SGX enclaves. This alternative approach provides more opportunities for using information from Rust's advanced type system.

The source code of the secure compiler infrastructure is publicly available at `https://github.com/hanswinderix/sllvm`.

1.4 Outline

The remainder of this text is structured as follows:

Chapter 2 Background The background chapter provides the reader with the security background that is required to read the rest of this text. First, a basic vocabulary and some fundamental concepts of computer security are introduced. Thereafter, the security properties, the security mechanisms and the security architectures that are relevant for this master's thesis are presented. Finally, this chapter concludes with a discussion on safe programming languages.

Chapter 3 Security Enhanced LLVM This chapter discusses the proposal of a unified secure compiler infrastructure and a proof of concept implementation for software module isolation. First, it provides the argumentation why we need such an infrastructure. Then, the programming models for software module isolation in the LLVM IR language and in the C programming language together with their proof of concept implementation are explained. Thereafter, the Sancus and Intel SGX security architectures are discussed as targets of this secure compiler infrastructure. This approach is compared with the officially supported toolchains. Finally, the genericity and the extensibility of the proposal is demonstrated by adding support for the Rust programming language.

Chapter 4 Conclusion This chapter contains concluding remarks about this master's thesis. It presents a summary of its contributions, recognizes limitations, discusses related work and provides some suggestions for future work.

Chapter 2

Background

This chapter provides an overview of background knowledge that can be instructive for reading the rest of this text. First, common vocabulary and concepts in the field of computer and software security are introduced. Then, the security properties, the security mechanisms and the security architectures that are relevant for this master's thesis are described. Finally, the chapter concludes with a discussion on safe programming languages.

2.1 Computer Security

Like any other discipline, the discipline of computer security comes with its own vocabulary. This section aims to introduce some basic terminology and informally defines computer security concepts that are used throughout this document. The purpose of this section is to merely give a brief introduction and not to give an exhaustive overview. Pointers to relevant literature are included for a more elaborate discussion.

The field of computer security encompasses, among others, hardware security, network security and software security. The latter is the focus of this master's thesis. Software security is a sub field of computer security that studies how intelligent adversaries can exploit design flaws and software defects and how to deal with these exploits. Security is an important aspect of every phase in the software development life cycle and it should receive the necessary attention in all those phases, starting from requirements gathering over design and implementation all the way to testing.

Since we increasingly rely on computing systems in our daily lives, securing these systems by preventing and detecting unauthorized access is of considerable importance [58]. Computer systems that enable electronic banking and electronic commerce are just two of the many examples where security mechanisms are necessary to protect financial data against thieves. Data loss, as another example, can severely impact the operation of a corporation.

First, let us try to come up with a definition of security itself. The main goal of security, according to Coulouris et al. [19], is "to restrict access to information and resources to just those principals that are authorized to have access". Computer security (or IT security [25]) investigates how to address this problem in computer systems.

A principal in this definition, is applicable to any entity that is involved in an operation that needs to be secured. Humans, computers, services, processes and threads are all possible principals. They need to be identified and authenticated in order to take part in security critical operations. A principal with the intention of breaking the security of a system is called an attacker. Commonly used synonyms for an attacker are adversary and malicious user.

The capabilities and the restrictions of an attacker are captured by an attacker model. Even a powerful attacker is faced with some restrictions since it would be impossible to defend against an omnipotent adversary. The attacker model allows security experts to reason about the effectiveness of a security solution. Consider for example that we assume that an adversary does not have physical access to the hardware of a computing platform nor that he can break cryptographic primitives, then these limitations should be part of the attacker model. Conversely, it is reasonable to expect that the adversary can falsify the source field of a message that travels on a public network or that she has access to the source code of an open source component.

To be able to manage security, it is important to make an overview of the security-relevant resources. An asset is a valuable resource for an organization that needs to be protected and secured against enemies. Security is concerned with the protection of these assets. Assets may be computing infrastructure and computing resources (such as servers, laptops, CPU, memory segments and I/O devices) or software (such as applications, database management systems, and services). But an asset can also be sensitive data like credit card numbers and cryptographic keys or privacy-related information like personal email and patient records that are stored in the database of a hospital.

The security properties one wishes to maintain are called security objectives and they are included in the security policy of an organization. A classification of security properties can often be helpful in the elicitation of the security objectives for a specific computer system. Confidentiality, integrity and availability (CIA) are typical examples of such categories. Authenticity, the property of being authentic is another one.

Assets are endangered by security threats. A security threat is the potential for breaching security. An attack is an actual attempt to break security which manifests itself as a sequence of steps. The subject of an attack is called a target. A weakness in a computer system that allows an attacker to break one of the security properties of an asset is called a vulnerability, which can be, among others, a flaw in the design, a defect in the software or a misconfiguration of a security component. An attack exploits a vulnerability and when attackers exploit vulnerabilities, countermeasures are created and deployed.

Assets are protected by access control mechanisms. Generic access control mechanisms enforce the appropriate use of the assets according to an application-specific security policy.

A protection domain defines a set of assets that can be accessed in that domain and the operations that are allowed on each of the assets. A computer process executes in a specific protection domain and is allowed to use the domain's access rights when accessing resources. A protection domain is an abstraction and can correspond to different entities. Users, software components, software modules or procedures are all possible domains. For each of these cases, it is necessary to be able to switch from one protection domain to another, effectively changing the rights of the running process. For example, in the case that a domain corresponds to a process, domain switching happens when a process sends a message to another process resulting in an action from the other process that needs to be executed by the computer system.

There are fundamentally two approaches to implementing access control, both having their strengths, weaknesses and trade-offs: capabilities and access control lists [25], [19], [49]. The first approach, a capability list, is associated with a particular domain and defines the list of resources that can be accessed in that domain together with the permitted operations on each of these resources. A capability is like an unforgeable key held by a principal which is authorized to access the corresponding resource. It suffices to possess the key to gain access. Of course, capabilities themselves are protected resources. They must be attack-proof and must be managed by a trusted component like the OS. Otherwise, their ability to secure resources can be compromised. The second approach, an Access Control List (ACL), is associated with a particular resource and maintains the list of the domains that can access that resource together with the permitted operations within that domain.

Traditionally, security is one of the responsibilities of the OS. It permits or rejects access to a protected resource. The computational cost of software protection can be high. Hardware support can significantly reduce these protection costs. The NX feature in modern versions of the AMD and Intel X86 chips, for example, can prevent the execution of code from a stack section in memory.

Security mechanisms are used to minimize the risks of potential security violations. They are employed to implement the security policy of a computer system. Security mechanisms can be preventive (prevent assets from being damaged) detective (detect when an asset has been damaged) or reactive (recover from damage to an asset, if possible) [25].

Cryptography, the science of keeping information secure, is a technology that lies at the basis of many security mechanisms. It is an important technology for proving the authenticity of information and for ensuring its privacy and its integrity.

Security engineers have a rich arsenal of security mechanisms at their disposal to realize the security policy of a computer system. The mechanisms can be selected based on the estimated cost of a security violation. A cost is associated with each mechanism and this cost must be balanced against the damage of a successful attack. To demonstrate the effectiveness of the security mechanisms that are used in a given computing platform, the system's designers must show that under the given attacker model each possible attack is prevented by an effective countermeasure and that for each security goal an appropriate security mechanism is in place. Each of these reasonings is called a security argument.

The security-critical components of a computer system are the components that are responsible for the implementation and the enforcement of the security policy of a computer system. These components, together with all the hardware, firmware and software elements that they depend on, need to be trusted and are commonly referred to as the Trusted Computing Base (TCB). Any vulnerability in this TCB can produce severe security violations compromising the entire system. It is therefore an important design principle to minimize the TCB as it reduces the attack surface of a system, the number of opportunities for an adversary to break the security.

2.2 Security Properties

This section gives an overview of the security properties that are relevant for this master's thesis. Since this thesis is mainly about software security, the properties that are discussed in this section are related to software security. Almost all of them apply to processes, programs that are being executed with a specific memory layout for storing their code and data. For this reason, they depend on some form of memory protection mechanism that enforces their security.

This section is organized as follows. First the primitive properties that can be classified in one of the CIA categories are discussed. Thereafter, the compound properties are discussed. Compound properties are properties that represent more than one primitive or compound property.

2.2.1 Confidentiality

Confidentiality can be described as the protection against unauthorized disclosure of information. Safeguarding the privacy and secrecy of sensitive information is a highly desirable property in many computer systems. Confidentiality is more than just hiding the contents of data. It is also about making sure that adversaries cannot learn information via a side channel, where information is derived from events that might seem to be unrelated at first sight. Although privacy and secrecy are often used as synonyms, privacy is more about the protection of personal information, while secrecy is more about the protection of information belonging to an organization [25].

Secrecy of Process Data

In order not to compromise the security of a system, it is important to keep secret the security-critical data of a security mechanism while the corresponding code is running. Cryptographic keys are a good example of such data. Some classes of sensitive data, like credit card information, need to be secured to prevent criminals from making purchases that will be charged on somebody else's card. Other categories of sensitive data, like personal information about individuals, can be subject to specific rules and regulations that companies have to comply with.

Secrecy of Process Code

Sometimes it is necessary not to keep the data but the machine code of the running process from being disclosed to unauthorized parties. For example, access to the machine code could allow competitors to reverse engineer the innovative algorithms that were developed by a company over many years. Many corporations want to protect the investment they made in this intellectual property.

Secrecy of Persisted Information

While the previous two properties dealt with information that resides in main memory, this property is concerned with how to store sensitive information securely on untrusted secondary storage. Sealed storage is a technique that typically relies on encryption to guarantee that the confidential information can only be revealed under certain conditions. For example, when information is sealed, it can be bound to the hardware and the software that is being used. Unsealing the information will only succeed under exactly the same configuration.

Secrecy of Communication

In order to exchange messages containing sensitive information with the outside world, it is required that the communication channel preserves the secrecy of the exchanged messages.

2.2.2 Integrity

Where confidentiality is concerned with the prevention of unauthorized disclosure of information, integrity is concerned with the prevention of unauthorized modification. Integrity encompasses both the accidental and malicious modification and destruction of data. In many cases, integrity is required to be able to guarantee some of the other security properties. For example, in order to keep information secret, it is important to keep the access control system from being compromised. Otherwise an attacker could violate confidentiality by gaining unauthorized access to the information that is being protected.

Integrity of Process Data

Data integrity is about protecting information from being corrupted. An attacker might want to do this to circumvent the security checks that are build in the software of an application. As another example, violating data integrity can also be effective for an attacker that wants to mount a denial of service attack. The destruction of data can result in a service from becoming unavailable when that data is required for its proper operation.

When data is loaded into the main memory of a computer system, data integrity is also concerned with preventing the execution of data.

Integrity of Process Code

Once code has been loaded in memory, it is often necessary to prevent the code from being modified. For example, an attacker can compromise a computer system by modifying code of a security-critical component.

Restriction of Entry Points

Restricting the number of entry points of a memory segment containing executable code is an effective countermeasure against several low level software attacks. Because this property prevents an attacker from jumping to arbitrary code, it makes attacks like Return Oriented Programming (ROP) [47], where existing code snippets (gadgets) are used, very unlikely. As another example, bypassing authorization checks by directly jumping to security-sensitive code [24] can be prevented.

Integrity of Process Metadata

A program in execution maintains metadata. This metadata is stored in processor registers and in the activation records that live on the call stack. The stack pointer keeps track of the top of the stack, the frame pointer contains the base address of the current stack frame. Each stack frame typically contains the value of the previous frame pointer and the return address. If an attacker is able to override this data, he can take complete control of the computer system [41], [47], [24], [25].

Integrity of Communication

The property of not being able to modify, insert, delete or replay transmitted messages is an integrity property.

2.2.3 Authenticity

Being able to prove and verify the authenticity of claims is an important security property for most computer systems. For example, when somebody logs on to a computer, she makes a claim about her identity that is verified by the computer system. The process of proving and verifying the validity of something is commonly referred to as authentication. Attestation is often used to refer to the authentication of hardware and software, including their configuration. Attestation can provide a high level of assurance to another party that a computing entity is in a trustworthy state.

Measuring is a technique for establishing the identity of software components. A measurement is the result of the computation of a MAC or hash of identifying information of a software component, such as its binary representation and its memory layout. The measurement can be used for attestation purposes.

Local Attestability

Software entities that are deployed on the same machine typically communicate with each other through function invocation. One entity, the caller, initiates the communication by invoking a function on the other entity, the callee. Since the caller might pass sensitive information during the invocation in the form of parameters, he needs to be able to assess the trustworthiness of the callee. The process of verifying the identity of the callee is commonly referred to as callee authentication. Similarly, when sensitive information is returned by the callee, the callee needs to have trust in the caller. Caller authentication can provide this trust.

Remote Attestability

With remote attestation, a computer system is able to present reliable evidence about its hardware and software configuration to a remote machine. This property requires a mechanism to perform integrity measurements, a secure communication channel and a remote attestation protocol. To give an example where this might be a useful property, consider a host with the task of provisioning a remote machine with secret information. In order not to disclose this sensitive information to unauthorized parties, the host needs convincing evidence that the configuration of the remote machine has not been compromised by an attacker.

2.2.4 Compound Properties

Isolation

Isolation is a well-established security-engineering principle. By preventing components from interfering with each other, they are less susceptible to security attacks. Isolation offers both confidentiality and integrity of the entities that are isolated. Isolation can be expressed as part of a linguistic abstraction in a high-level programming language to be enforced by a compiler.

Memory isolation aims to protect the memory segments that are allocated to a running software entity. Memory isolation can be used at different levels of granularity. For example, it is possible to protect processes, software components, software modules, procedures and closures.

Process isolation Traditionally, one of the tasks of an OS is to make sure that all processes are executed in their own virtual address space, effectively isolating the OS and the processes from each other. No process has access to the address space of another process and communication with other processes is only possible through primitives provided by the OS.

Software module isolation A system supporting the property of software module isolation allows for isolation at the granularity of a software module. The software modules of an application can be isolated to protect them against attacks that come from other parts of the application that share the same address space. In some cases, it is even possible to protect them against a potentially malicious OS.

Recently, this form of isolation has seen a lot of activity in the research community [50], [52], [39], [59], [34], [40]. Juglaret et al. [28] refer to this property as compartmentalization.

Secure Communication

The property of security communication requires that both the confidentiality and the integrity of the messages are guaranteed.

Secure Linking

Secure linking is a combination of secure local communication and mutual local attestability.

Sealing

Sealing requires confidentiality and integrity of persisted information and local attestability.

2.3 Security Architectures

Security involves a wide array of properties and mechanisms that vary on a number of axes, including the following:

Type of access control As mentioned in section 2.1, there are fundamentally two approaches to implementing access control. With capability-based access control, on the one hand, it is easy to pass on a capability, which makes it hard to get an overview of who has permission to access a given object. Furthermore, capabilities need to be protected too and are hard to revoke. With access control lists on the other hand, it is easy to revoke a permission but it is hard to get an overview of permissions given to a specific user.

Size of the TCB On conventional computer systems, security properties are realized through a tight cooperation between the OS and the computer hardware. Hardware-only security mechanisms do no require the OS to enforce the security properties, which significantly reduces the size of the TCB as modern OSs have a monolithic design and consist of several millions lines of code.

Complexity Simple generic mechanisms are easy to verify and can provide for a high degree of assurance. However, such a simple generic mechanism may be too general to use as a building block for the enforcement of a very specific protection requirement [25].

These degrees of freedom in the design space for security mechanisms have yielded a wide range of protection mechanisms in computing architectures. This section first discusses protection in conventional architectures. Then the Protected Module

Architecture, a novel type of security architectures that was briefly introduced in section 1.2, is presented in more detail,

2.3.1 Protection in Conventional Architectures

Hardware Support

An attacker needs to have physical access to the computing platform in order to compromise the security properties that are provided by a protection mechanism that is built into the hardware of a computer system. For this reason, hardware protection can be considered more secure than protection that is implemented in software. Although recently, the Spectre [29] and Meltdown [33] attacks have reminded us that also hardware protection is not absolute and that there are hardware vulnerabilities that can be exploited.

Another advantage of a hardware protection measure is that it can be more efficiently implemented compared with an equivalent solution in software. The computational cost of a software mechanism can be high and hardware support can significantly reduce these protection costs.

Most modern processors have different modes of operation. At least two separate modes of execution, supervisor mode (privileged mode) and user mode (unprivileged mode) are made available by these types of processors. The current mode of operation is managed by a hardware mode register. The hardware enforces that some instructions, the so-called privileged instructions, can only be executed when the processor is in privileged mode. Whenever a software or hardware interrupt occurs, the hardware puts the processor into supervisor mode and executes a piece of supervisor code, a so-called interrupt handler. The idea of this setup is to give the exclusive right to execute privileged instructions to a trusted component like the operation system.

The Memory Management Unit (MMU), a memory mapping hardware device that is part of modern higher-end computer platforms makes it possible to separate the logical (or virtual) address space of a process from its physical address space. The task of the MMU is to map logical addresses as generated by the CPU to physical addresses as seen by the memory hardware [49]. The mapping information is maintained into the page table which is backed by the Translation Look-aside Buffer (TLB) a hardware cache that is used to reduce the increased latency when accessing memory.

Although a programmer views memory as a single contiguous address space, this technique allows for a program to be scattered around in physical memory which make memory easier to manage in a multi-programming environment. Another benefit to this approach, is that it allows the logical address space to be larger than the physical address space which can be a convenience for the programmer as it can simplify writing programs. The instructions that determine how to set up the page table are privileged instructions. Together with the consequence that it is not possible to access physical memory that cannot be reached through the page table, it allows the MMU to be used as a hardware security mechanism too.

On systems with an MMU, it is also possible to restrict access to the individual pages in a page table. With such an access control mechanism in place, individual pages in the page table can be marked read-only, read-write, execute-only or non-executable. This can effectively be a cheap run-time countermeasure against low level attacks such as those that exploit a buffer overflow vulnerability [24]. To give an example, when an attacker tries to execute his carefully crafted code after injecting it on the non-executable stack, it will be detected by the hardware which can flag the memory violation with a hardware trap allowing the OS to terminate the process. The eXecute Disable (XD) bit, on Intel systems and the eXecute Never (XN) feature on the ARM architecture are incarnations of such a mechanism.

Operating System Support

Protection is considered to be one of the core services of an operation system [49]. To give an example, it is the task of the OS to protect itself and the user processes from each other's activities. The OS uses the security primitives made available by the hardware architecture at stake as the building blocks for the implementation of its security mechanisms.

When the OS schedules the processes to execute, it arranges that the kernel-level processes are executed in privileged mode and that the user-level processes are executed in unprivileged mode. System calls provide user programs with an interface to services of the OS. A system call interrupts the processor and triggers the processor to enter privileged mode. This mechanism allows the OS to perform privileged operations on behalf of a user program in a controlled way.

Memory protection is configured by the OS but enforced by the hardware. The memory addresses that are generated by the processor have to be validated and mapped to physical addresses. Access rights on memory (read-only, read-write, non-executable, execute-only) must also be checked when the instructions are executed. A program in execution generates many addresses when instructions and data are loaded from main memory. Checking the validity of these addresses and checking the access rights on the memory would incur a huge overhead if this would have to be implemented in software. Hardware support is necessary.

The OS protects the different processes from each other's activities by isolating them from each other. The kernel makes sure that each process has its own separate address space and that it is unavailable to other processes. On conventional computer systems, process isolation is realized through a tight cooperation between the kernel and the computer hardware. Being a privileged process, the kernel has control over the page table and the different process registers. It can configure the CPU in such a way that other processes can only access the machine's physical memory in acceptable ways. The OS maintains one page table per process and each process owns the memory as dictated by its page table. The mapping to physical memory is controlled by the OS. It is the responsibility of the OS not to map virtual memory of different processes to the same physical memory. A context switch enables the correct page table.

Recent research by the security community [39], [37], [59], [30] [40], has proposed several alternatives to the conventional approach of realizing isolation. The following subsection discusses PMAs, one of these alternatives.

2.3.2 Protected Module Architectures

The granularity of protection in traditional systems, where the unit of protection is a complete process, does not offer very strong security guarantees. Plenty of use cases would benefit from a more fine-grained protection strategy. It makes sense, for example, to have a mechanism that restricts access to memory segments at the granularity of a software module.

Protection at the level of a software module enables the execution of the security critical parts of an application in their own protection domain, not only isolating them from the rest of the application but also from potentially malicious system software, such as the OS. This approach would reduce the attack surface significantly. If one of the other modules is compromised, it would not affect the protected components.

The canonical example is the implementation of a cryptographic library. If such a library maintains the keys that are used by its cryptographic primitives, it is of paramount importance for security to protect the secrecy of these keys. Additional protection against other potentially malicious modules will increase the level of assurance that the secret keys will not be compromised.

The need for a more fine-granular form of protection triggered the development of PMAs such as Flicker and TrustVisor [52]. These architectures implement the idea of an inverse sandbox. A sandbox is typically employed to enclose an untrusted piece of software in a constrained environment with very limited privileges to prevent a potentially malicious component from contaminating the healthy pieces of a computer system. With PMAs, this idea is turned around. In this setting, the software modules are the entities that need protection against an untrusted environment. Every PM corresponds to a separate protection domain and the containing application runs with minimal privileges, adhering to the well-known principle of least privilege [46].

This kind of protection is complementary to the protection provided by traditional OSs. By combining both types of protection, a process can be associated with several protection domains that all share a single address space.

Another issue with the protection in conventional systems is the size of the TCB. As mentioned in the first section of this document, minimizing the size of the TCB is an important security design principle. Since protection is considered to be one of the core services of the OS, the OS is often included in the TCB.

However, for performance reasons, commodity OSs are typically written in low-level, unsafe system programming languages like C++. This makes them vulnerable and a beloved target for attackers, looking for low-level vulnerabilities to exploit [24]. Furthermore, since modern OSs have a monolithic design and consist of several millions of lines of code, it is hardly feasible to defend them against these attacks. PMAs offer also a solution for eliminating the need to trust the OS and other elements of the execution infrastructure.

A third issue with the conventional approach to protection and isolation is that the mechanisms are too resource-intensive for low end mobile and embedded devices. Many of these lightweight systems do not have the advanced hardware supported features of privilege levels and virtual memory that are available in higher-end computer systems. However, these systems can have strong security requirements.

With the advent of execution aware access control schemes, PMAs have been proposed for low-cost, low-power embedded devices. According to Strackx et al., program counter based memory protection provides a lightweight hardware mechanism that can be implemented in an efficient way [50], [39], [40].

Implementations

PMAs all provide some form of isolation and some of them have cryptographic support and key management built-in. Implementations are typically realized by a compiler and a low-level security mechanism.

They have been proposed for very different types of systems, at different system abstraction layers, each of them suited for different application scenarios [52], [34]. Some PMAs have support for only one enclave, such as ARM's TrustZone [11], while others can run several of them. Implementations have been proposed for high-end systems, whereas others target low end embedded platforms. There are PMAs that position themselves as solutions for building secure applications in a way similar to microkernels. Others can be used for protecting entire VMs with OS and applications into an enclave to gain security from, for example a malicious cloud provider.

There are PMAs that make use of an innovative and efficient security mechanism to enforce the confidentiality and integrity of a PM. This novel execution-aware mechanism realizes isolation and protection by mediating access to protected memory sections, based on the value of the program counter [50]. Strackx et al. [52] report on their experiences of three of these PMAs, each of them implemented at a different system layer. More information on the authors' experiences with Sancus [40], a hardware-based PMA, Fides [53], a hypervisor-level PMA, and Salus [16], a kernel-based PMA, and for a discussion on typical application scenarios, their advantages and limitations, see [52].

The focus of this master's thesis lies on two PMAs that are based on a program-counter based access control mechanism that is entirely implemented in hardware. Sancus [40] is designed for devices that are situated on the low-end side of the computing spectrum. Intel SGX [37] runs on high-end computing devices such as desktops and servers.

Attacker Model

For the remainder of this text, we will only consider hardware-based PMAs with cryptographic support and key management built-in. We assume the following very powerful attacker model, a consequence of the small TCB of hardware-based PMAs.

First, since this type of architectures removes the dependency on an OS kernel for ensuring the security of the system, the threat model considers an attacker that is able to inject arbitrary machine code in the main memory of a computer system, including the parts where the code and the data of the kernel reside. With this ability, an attacker can bypass all security mechanisms implemented by the kernel and by the layers above the kernel.

There are plenty of exploits and attacks that are documented in the security literature, showing that this is a realistic assumption. Consider for example an attacker exploiting a memory-safety bug in the kernel. Such a vulnerability makes it possible to alter the intended control flow, enabling the execution of arbitrary code in a privileged protection domain [24].

Once an attacker has gained full control over the computer system, including unlimited access to the file system, she can deploy and execute malware. She can tamper with installed software, including the OS and system libraries like the C standard library. An attacker can also communicate with the PMs by sending them input and by receiving output from them, e.g. by installing additional modules. Having access to the unprotected memory sections of running applications, code and data residing in memory can also be modified.

However, an attacker cannot compromise the memory access control mechanism employed by the PMA. Neither can she compromise or tamper with the memory that is allocated to the protected software modules. Consequently, this means that the considered threat model assumes that there are no low level vulnerabilities in the code of a PM that an attacker can exploit. Similarly, it is assumed that there are no flaws in the design of a PM, such as an Application Programming Interface (API) function that leaks secret information. In other words, the TCB of a PM comprises the PMA and the PM itself.

Second, an attacker can monitor and modify any traffic on networks that connect the PMA with other, potentially malicious, computer systems enabling him, for example, to mount man-in-the-middle attacks. An attacker cannot break cryptographic primitives but can perform protocol level attacks, according to the Dalev-Yao attacker model [22].

Third, some PMAs, such as Intel's Skylake architecture where the PMA functionality is implemented in the SGX extensions, allow the attacker to have physical access to some parts of the hardware, such as the system memory. This means that an attacker is supposed to be able to place probes on the memory bus or disconnect the memory chips and read out their contents on another system.

Of course, PMAs do not provide a complete solution against security threats and have their limitations. As mentioned before, they do not provide a defence against attacks that exploit weaknesses within the implementation of a PM. A PM is only protected from a malicious context. Safe programming languages are an effective countermeasure against low-level attacks which makes them a complementary technique to PMAs.

Furthermore, PMAs do not offer any availability guarantees. They are no defence against denial of service attacks.

Sancus

Noorman et al. designed Sancus [40], a hardware only PMA targeting applications for devices that are situated at the low end side of the computing spectrum, low-power embedded devices operating with a limited set of resources.

A simple but effective program-counter based access control mechanism [50] guards the PM's memory sections from unauthorized access. Additionally to the protection of security-sensitive data, the Sancus architecture also includes support for secure linking and and remote attestation, so that devices can prove their identity to a remote party.

The sweet spot for Sancus in the application domain space are extensible networks of small embedded devices. The network is extensible in the sense that software can be installed by third-party software providers. It is the system's responsibility to ensure that these mutually untrusted extensions do not interfere with each other.

Consequently, the cryptographic scheme of this PMA is designed with this application scenario in mind. In this scenario, an Infrastructure Provider (IP) manages a network of nodes. Every node N has a fixed key K_N, etched into the silicon and only accessible by the hardware, and shared with the IP. To be able to deploy software modules, a Software Provider (SP) has to request a key from the IP. A key is derived based on K_N and the SP's public identifier: $K_{N,SP} = \mathit{kdf}(K_N, SP)$. The key derivation function *kdf* does not rely on software as it is implemented entirely in hardware. This approach allows for the exclusion of any software from the trust boundary.

The identity of a Software Module (SM) includes a measurement of its code and its layout, the address bounds of the SM's protected text and data sections when loaded into main memory. The identity of a SM is likewise used to derive a key $K_{N,SP,SM} = \mathit{kdf}(K_{N,SP}, SM)$.

Since the node key K_N is only accessible by the hardware, the module key $K_{N,SP,SM}$ can only be derived on node N. Furthermore, $K_{N,SP,SM}$ is only accessed indirectly by specific processor instructions and the hardware guarantees, also based on the value of the program counter, that the software module SM, deployed by SP, has the exclusive right to do so. Therefore $K_{N,SP,SM}$ can be used for secure linking, secure communication and remote attestation. If the code of a SM has been tampered with, the code measurement will be different and the hardware will derive a wrong key.

A Sancus prototype has been developed by its designers as a set of extensions to the TI MSP430 architecture, but other low end architectures can be transformed into a PMA in a similar way. The implementation of the security kernel extends the MSP430 architecture with a memory access logic (MAL) circuit and a protected storage area. The protected storage area is used by the security kernel for storing cryptographic keys and some bookkeeping information used by the MAL to enforce the access control rules.

Figure 2.1 illustrates the layout of a PM on the Sancus security architecture.

Because the text section of a PM needs to readable in order for the Sancus hardware to be able to measure the code, Sancus does not offer confidential loading.

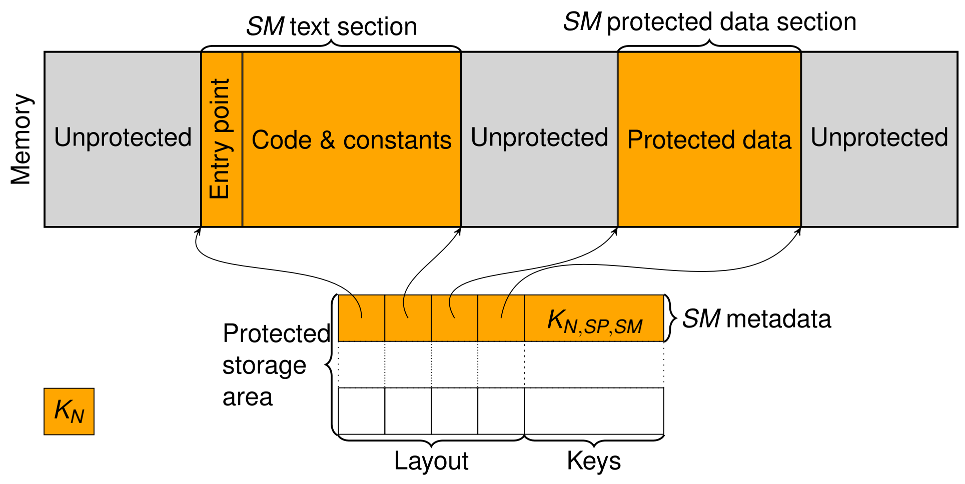

FIGURE 2.1: Layout of a Sancus PM

An illustration of the Layout of a PM on the Sancus security architecture (courtesy of [39]).

The Sancus attacker model excludes denial-of-service and physical attacks.

The Sancus toolchain is based on the LLVM compiler infrastructure. It is implemented as an LLVM compiler transformation that frees the developer from dealing with the low-level details related to the Sancus architecture.

It takes care of the following tasks [40]:

Logical entry points The Sancus hardware limits access to a protected module to a single physical entry point. It is the task of the compiler to generate code that creates the illusion of multiple logical entry points, on top of the sole physical entry point. The compiler generates code for a dispatcher function that is mapped onto the physical entry point.

To realize the implementation of the logical entry points, the compiler introduces a new Calling Convention (CC) for which it reserves three additional MSP430 registers. Every entry function is assigned a unique identifier by the compiler. As part of the new Sancus CC, to call a logical entry function, the caller has to put the identifier of the function in one of the dedicated registers. The return address is put in one of the other registers. This gives the dispatcher the necessary information to forward the call to the correct entry point and to return to the correct call site.

When a PM has invoked a function that is external to the module, transferring control from the callee to the caller is equally only possible in an indirect way via the physical entry point. The compiler ensures that returning from out calls is handled in a correct and secure way.

Stack switching The stack frames of PM cannot be placed on a globally accessible stack as this would enable adversaries access to information that should be kept confidential or protected from malicious modification. It would also make a PM vulnerable to attacks that manipulate control data that are stored in a stack frame.

For this reason, PMs should maintain their own stack. The Sancus compiler reserves space for a protected stack in the data section of a PM. On module entry, the global stack is replaced by the module's private stack by copying the address to the top of the private stack to the stack pointer register. On module exit, the stack pointer is saved in the protected area and the global stack pointer is restored.

Exiting a PM To avoid leaking security-sensitive information that might be left in one of the registers of the processor as a residual of computation, the compiler generates code, to be executed whenever a module exits, to clear the registers that are not part of the CC. In other words, only the registers that hold a parameter or a return value are the registers that keep their value when changing the protection domain.

Secure linking The Sancus compiler generates code that verifies the authenticity of a PM when one of its entry functions is called. This feature is out of scope for this master's thesis.

Intel Software Guard Extensions

SGX [37] is technology that was introduced by Intel in 2015 when they released the sixth generation of its Core microprocessors, based on the Skylake architecture.

SGX represents a set of new instructions, a new mode of execution (enclave mode) and a new memory access control scheme that is activated when the processor enters enclave mode. With these new instructions hardware PMs can be created, managed and executed in a secure way. In Intel SGX terminology, PMs are metaphorically referred to as enclaves, armored entities entirely surrounded by an untrusted environment, including possible privileged malware.

Intel SGX' approach is complementary to the conventional coarse-grained approach to isolation, where processes are the unit of isolation, each having a separate address space, as discussed in section 2.3.1. SGX extends this classical way of protection with additional hardware enforced isolation guarantees, at the granularity of a software module.

As opposed to Sancus, the SGX based PMA targets applications running on powerful desktops and servers, characterized by features like privilege levels, virtual memory and an almost unlimited amount of computing resources, at least when compared to lightweight embedded devices. According to McKeen et al. [37], the SGX technology is very scalable as it can execute several protected applications concurrently and it supports the creation of extremely large enclaves.

With SGX' new protection model, system software can still be made responsible for managing the system resources, but without having to be part of the trust boundary. Enclaves are built and loaded by privileged software and once this is done, it is protected by SGX' access control mechanism. A process of measuring establishes the enclave's identity, which is used to proof that the code has not been tampered with.

The virtual to physical address mappings, typically managed by the OS, are made tamper-resistant by a hardware protection mechanism making sure that the mapping cannot be changed, effectively preventing an attacker from bypassing the protection and placing the OS outside the TCB. When an hypervisor or OS puts enclave pages in unprotected memory or swaps them out to secondary storage, their contents gets automatically encrypted. Because of the complexity of conventional operating systems, this approach also opens opportunities for a new type of side channel attacks [60].

SGX guarantees the security properties also in the presence of interrupts, faults and exceptions. When one of these events occurs, the state of the processor is securely saved to a memory location inside the enclave and consequently replaced by a synthetic state to prevent leaking sensitive information [37]. SGX supports multithreaded enclaves, where each thread in each enclave has dedicated structures for storing its execution context, such as the contents of CPU registers and metadata like control flow information.

The SGX extensions also comprise hardware and instructions for secure communication and remote attestation, allowing the secure provisioning of enclaves with sensitive information. Sealing is also supported by the hardware, allowing an enclave to store data on storage external to the enclave, while still guaranteeing confidentiality and integrity.

SGX assumes a threat model where attackers have physical access to all the hardware that is outside the processor package. Since enclave data is automatically encrypted once it leaves the processor package, the protection of sensitive information is guaranteed even when hardware like the memory is compromised.

2.4 Safe Languages

A safety property is a property that guarantees that certain types of errors are absent, a guarantee that something bad will never happen. Safety in the context of programming languages aims to avoid undefined and implementation-dependent behavior, which is well-known to give rise to a wide range of vulnerabilities [24].

Language based protection is a security mechanism that belongs to the preventive category of countermeasures. The semantics of a safe language are completely defined and, by consequence, the assurance is given that the execution of a program written in such a language is always well-defined. It will never be exposed to undefined behavior [44]. The safety guarantees rely on mechanisms that are implemented by the compiler and by the run-time system of the programming language.

In *Types and Programming Languages* [44], Benjamin C. Pierce informally defines a safe language as “one that protects its own abstractions”. Safe languages not only protect the integrity of their linguistic abstractions, abstractions that are built in the programming languages themselves, but they also guarantee the integrity of the in-language domain-specific abstractions, abstractions that are introduced by the programmer and that are built on top of the primitive linguistic building blocks.

Language safety is effective only under a relatively weak attacker model. The attacker model of language safety assumes that adversaries can only interact with programs according to the semantics as expressed in the high-level source code of the programs. To give an example, a programming language with procedures typically allows to enter a procedure via a single entry point. However, after compilation the same procedure can typically be entered from anywhere as most instruction sets have instructions for jumping to arbitrary locations in memory.

There are different types of language safety. Memory safety, for example, guarantees the absence of memory violations. Type safety, as another example, makes sure that no violations against the language’s type system occur and thread safety prevents race conditions from happening.

A very coarse classification of programming languages is one where we only have two categories: low-level programming languages and high-level programming languages. Machine code and assembly languages belong to the family of low-level programming languages while languages like Java and Rust are instances of high-level programming languages.

The extent to which a programming language provides abstraction determines its membership to one of the these two categories. A low-level programming language is characterized by its lack of advanced abstraction mechanisms. The programming constructs in these languages are shaped around a particular type of hardware architecture and therefore reflect that architecture’s instruction set. So because of their close relationship with a particular kind of hardware, these languages are referred to as being "low-level" or "close to the hardware".

One of the benefits of using a programming language that offers more advanced linguistic abstractions is that it can provide the programmer with a powerful tool to declaratively express some security properties of his or her program, making it easier to express security intent. By statically analyzing the source code, a compiler of a safe language can then enforce some of these abstractions, although run-time safety in general is not achievable by static analysis alone.

Low-level programming languages and unsafe programming languages offer little or no such protection mechanisms. For example, in these types of languages there are typically no limitations to accessing arbitrary memory locations, which leaves the door wide open for a wide range of severe vulnerabilities [41], [47], [24].

Developing software in a safe programming language can eliminate these vulnerabilities but does not provide absolute security guarantees either [56], [43], [45], [57]. Often an attacker is able to interact with the system at a lower abstraction level than that of the high-level programming language. It is therefore essential to understand how an attacker can compromise security goals when interacting with the low-level layers of a computer system.

The C and C++ programming languages do not provide memory safety. To give an example of memory unsafe behavior, in these languages a programmer can construct pointers to arbitrary memory locations. This is a useful feature for a system programming language but it can also lead to the corruption of memory, giving an attacker an opportunity to make the program misbehave.

Programs that have been written in a memory-safe programming language are guaranteed not to access memory locations they are not authorized to access. A well-known example of a programming language that is considered to be memory-safe is Java. The Java virtual machine dynamically enforces memory safety, preventing errors such as buffer overflows. Rust is another example of a memory safe programming language. Like C and C++, Rust is designed for system programming such as writing device drivers and programming embedded systems, where control and performance are important requirements. Rust can provide the same memory safety guarantees as Java, but without any run-time overhead.

PMAs such as Sancus and Intel SGX come with toolchains for writing PMs in C, an unsafe programming language. Safe programming languages and PMAs are complementary in the security guarantees they provide. PMAs do not provide a defence against attacks that exploit weaknesses within the implementation of a PM. A PM is only protected from a malicious context. Safe programming languages are an effective countermeasure against low-level attacks. This makes them very appealing for developing PMA applications.

Chapter 3

Security Enhanced LLVM

This chapter presents Security Enhanced LLVM (SLLVM), a proposal for a secure compiler framework with a focus on uniformity, reusability and programmability. It also discusses a proof-of-concept implementation that demonstrates the genericity of the framework. SLLVM has the ambition of evolving into a secure compiler infrastructure that unifies the different compilation schemes that were proposed since the dawn of the research field of secure compilation.

SLLVM extends LLVM, an open source compiler framework for lifelong program analysis and transformation [32]. LLVM is used for the development of compiler front ends and back ends. The LLVM project [5] provides a set of reusable libraries. Its modular design corresponds to the typical phases of a compiler, as explained in textbooks on compiler theory, which is in strong contrast with other open source programming language implementations that were developed as special-purpose tools which a monolithic architecture [17].

At the heart of the LLVM compiler infrastructure is the Intermediate Representation (IR), a typed instruction set that is independent from any source language and independent from any target architecture [7]. A programming language's front end translates a source program, written in the language at stake, into IR. The components that make up LLVM's middle end can then analyze and transform the IR in a generic way. This typically results in an optimized IR. A target specific back end is responsible for converting the IR into machine-dependent executable code.

SLLVM also extends Clang [1], the C front end for LLVM, and Rustc [8], the Rust compiler, both targeting LLVM's IR.

The chapter is organised as follows. Section 3.1 first provides a motivation for a unified secure compiler framework. It compares the design of secure compilers with the design of conventional compilers, or compilers that don't preserve the security properties of high-level language abstractions. Next, section 3.2 presents SLLVM-IR, this master's thesis' approach to software module isolation in LLVM IR. Section 3.3 thereafter discusses SLLVM-C, this master's thesis' approach to software module isolation in the C programming language. Section 3.4 discusses a new Sancus toolchain and how it is integrated with SLLVM. Section 3.5 thereafter discusses how the SGX security architecture is supported as one of the targets of SLLVM. Finally, section 3.6 demonstrates the genericity of SLLVM by extending the Rust compiler to target SLLVM-IR.

3.1 Motivation

An Intermediate Representation (IR) is a language for an abstract machine that is independent from a specific source language and independent from a specific Instruction Set Architecture (ISA). The front-end of a compiler translates a source language to the IR while the compiler's back-end translates an optimization of the IR to machine language. Although it is possible for a compiler to translate the source code directly to the actual machine code, translating to an IR is beneficial for software engineering properties such as modularity, reusability and portability [15].

This design allows to significantly reduce the implementation effort of the compiler. Suppose that N different source languages have to be translated to M different target machines. Not having a translation phase to an IR and directly targeting the machine language would require N x M compilers, as illustrated in figure 3.1. Furthermore, it would be necessary to add machine specific details to the compiler's front-end and source language specifics to the back-end which violates the *separation of concerns* principle.

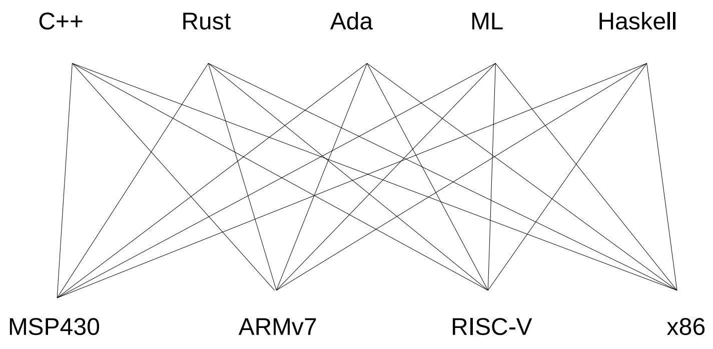

Figure 3.1: Compiler implementation without IR

A translation from five different source languages to four different target languages without an IR would require 20 different compilers. (based on [15])

An IR-based compiler infrastructure translates the source code in two passes: the first pass translates the source code to IR and the second pass translates the IR to the machine language of the target architecture. With this approach, the compiler only needs to provide N front-ends for N source languages and M back-ends for M target machines, or N + M components. This idea is depicted in figure 3.2.

Hiding specific targets from the different front-end components and hiding the specifics of the source languages from the different back-end components simplifies their implementation. It allows for the effective reuse of large parts of the implementation effort: intricate code for analysis, optimization and other purposes does not

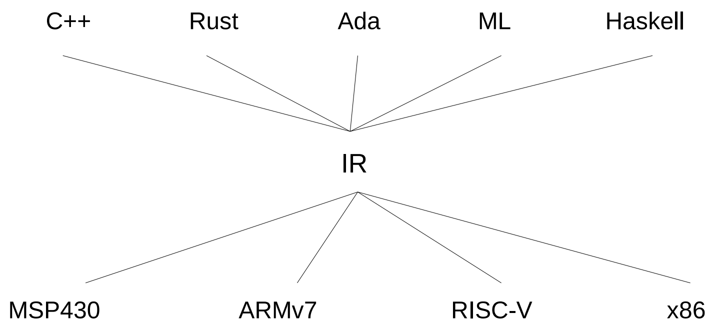

Figure 3.2: Compiler infrastructure with IR

An IR-based compiler infrastructure only requires five front-ends for five source languages and four back-ends for four targets. (based on [15])

have to be duplicated. Furthermore, adding support for a new source language or for a new target architecture requires minimal effort.

The same idea is applicable in the context of security. Extending programming languages with security features, adding support for new security architectures and techniques like secure compilation can all benefit from making a clear separation between these three concerns. A language-specific front-end deals with issues related to a specific programming language and implements a mapping onto the primitives provided by the IR. The IR provides a way for generically expressing the security properties through a set of primitives. The back-end, finally, encapsulates the details related to a specific security architecture.

However, secure compilation schemes seem to be implemented as special-purpose tools, leaving little opportunity for sharing between the different secure compiler initiatives. Security architectures such as Sancus [40] and Intel SGX [37], for example, come with an architecture-specific compiler implementation, illustrated in 3.3, although there is a significant overlap in the offered security features. If it is possible to design a common compiler infrastructure for a wide range of programming languages and target architectures, then it should also be possible to extend such an infrastructure with generic support for a wide range of security models, such as PMAs and capability-based security architectures, effectively unifying the different special-purpose implementations.

This chapter formulates a proposal for the design of a secure compiler infrastructure with the ultimate aim to unify the different schemes that exist today. SLLVM is designed as a set of LLVM extensions for the generic representation of security properties in computer programs. A proof of concept implementation for software module isolation should demonstrate that it is possible and beneficial to generically represent security properties at the different levels of abstraction. Because of the restricted time-frame of a master's thesis, the implementation is limited to two programming languages, C and Rust, and to two security architectures, Sancus and

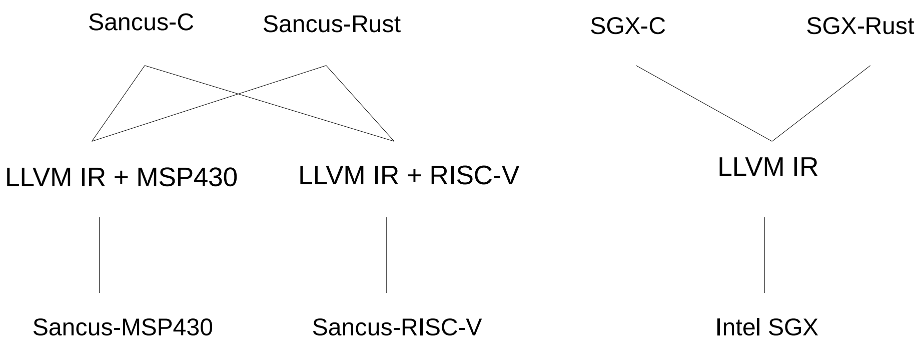

Figure 3.3: Product-specific Secure Compilers

Illustration of the current approach towards secure compilation where SDKs and toolchains are designed as special-purpose tools.

Intel SGX, as depicted in figure 3.4.

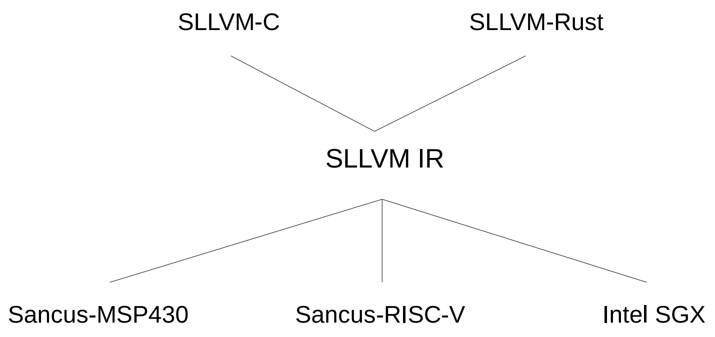

Figure 3.4: Security Enhanced LLVM

Schematic overview of the proof of concept implementation of the SLLVM secure compiler infrastructure.

3.2 Software Module Isolation in SLLVM-IR

SLLVM's IR is at the heart of the SLLVM secure compiler infrastructure. SLLVM IR extends the LLVM IR with security-related abstractions, that can be expressed independent from any source language and independent from any target architecture.

The LLVM documentation is very explicit about extending LLVM with new instructions [2]. It discourages to add instructions to the IR because this modifies the binary format and will require a significant effort to ensure backward compatibility.

Also according to the documentation, the recommended way to extend the IR is with intrinsics. It is advised to start an extension to LLVM as an intrinsic function and to turn it into an instruction later, if warranted.

This section first presents an approach for representing software module isolation in LLVM IR. SLLVM aims to stay as close as possible to the native programming model of the IR by avoiding the addition of concepts that can be expressed with existing language constructs. Next the target independent analysis and transformation passes at the level of the middle end are explained. The section concludes with a discussion of the approach.

3.2.1 Programming Model

Software module isolation revolves around the concept of an isolated module. An isolated module in SLLVM-IR is a monolithic software entity with the following properties:

- An isolated module can have public data associated to it. Public data contains security-insensitive information that can be accessed by software entities external to the isolated module. It concerns data that logically belongs to the isolated module but that does not need to be protected from unauthorized access.
- An isolated module typically has private data. Private data contains security-sensitive information that needs to be isolated from all software entities that are external to the isolated module at stake. Private data can only be accessed by software entities that belong to the same isolated module and that are within the protection boundaries of the isolated module. This means that entities that logically belong to the isolated module but are not within the protection boundaries cannot access the private data either.
- Public code is code that logically belongs to the isolated module, but that does not need to be protected from unauthorized access.
- The private code of an isolated module is code that can only be invoked by software entities that logically belong to the same isolated module and that are within the protection boundaries of the isolated module.
- The invocation of an isolated module is restricted to a number of entry points. This property prevents an attacker from jumping to arbitrary locations within the isolated module.

SLLVM intends to preserve these properties after compilation. When the SLLVM compiler cannot guarantee the preservation of these properties for the selected target architecture, then it should abort the compilation process.

An isolated module in SLLVM-IR is represented by an ordinary LLVM IR module, enriched with SLLVM extensions. Symbols with local linkage are considered to be within the protection boundaries of the isolated module. SLLVM adds three extensions to LLVM IR to fully support isolated modules.

Firstly, there needs to be a way to convey to the different LLVM components that an IR module needs to be securely compiled according to the semantics of module isolation as specified in the beginning of this section. However, the LLVM IR is not very well suited to communicate information about an IR module as a whole to the LLVM subsystems, such as the target-independent code generator. For this reason, the LLVM infrastructure has created the `llvm.module.flags` named metadata, a dictionary of key-value pairs that facilitates the LLVM subsystems that are interested in a flag to look it up [7]. SLLVM adds the `sllvm-module` module flag to indicate that the security-related semantics of software module isolation should be preserved by the compilation process for the IR module at stake.

Secondly, since not all public functions of an isolated IR module can be considered entry points of the isolated module, SLLVM extends LLVM IR with a function attribute, `sllvm-entry`, which can be used to indicate that an IR function is an entry point of the isolated module.

Thirdly, SLLVM adds four new intrinsics that are meant to mark points in SLLVM IR programs where protection domains are changed. These *domain crossing intrinsics* are the following:

sllvm_ecall Indicates that an isolated module is entered via an entry call, an invocation of one of its entry functions.

sllvm_exit Indicates that an isolated module is exited via a return from an entry call.

sllvm_ocall Indicates that an isolated module is exited due to an out call, an invocation of a function that does not belong to the protection boundary of an isolated module.

sllvm_return Indicates that an isolated module is entered via a return from an out call.

How each of the generic domain crossing intrinsics is lowered into target-specific assembly instructions is dependent on the selected target architecture and the responsibility of the corresponding code generator.

Listing 1 contains an example of an isolated module in SLLVM-IR. The listing is based on code that is generated by a modified Clang front end (section 3.3) from the code in listing 5. To keep it short and readable, the for this purpose irrelevant parts of the generated code are removed. Comments in the code listing further illustrate the SLLVM-IR programming model.

3.2.2 Design and Implementation

The generic part of the SLLVM middle end consists of two main components: a generic analysis pass and a generic transformation pass. The structure of the SLLVM middle end is illustrated by the class diagram in figure 3.5.

```
source_filename = "enclave1.c"
; The following line contains a declaration of an entry function
; of another isolated module. #0 refers to an attribute list that
; contains the sllvm-entry attribute.
declare i16 @enclave2_entry(i16) #0

; The defintion of a public variable
@enclave1_shared = global i16 0, align 2
; The 'internal' keyword makes variables private
@secret = internal global i16 0, align 2

; The 'internal' keyword makes functions private
define internal void @internal_func() {
entry:
  ; Private data can be accessed from internal functions
  %0 = load i16, i16* @secret, align 2
}

; The defintion of an entry function.
define void @enclave1_entry1() #0 {
entry:
  call void @internal_func() ; Invocation of internal function
  call i16 @enclave2_entry(i16 @secret) ; Out call to enclave2
}

; Multiple entry functions can be defined.
define void @enclave1_entry2(i16 %p) #0 {
entry:
  call i16 @unprotected_f(i16 p) ; Out call to unprotected code
}

; The following attribute list contains the sllvm-entry attribute.
; Functions that are associated with this list are treated as
; entry functions.
attributes #0 = {"sllvm-entry"="true"}

; The following line is the defintion of an entry of the
; llvm.module.flags named metadata to indicate that this concerns
; an isolated module. The module name is derived from the name of
; the source file.
!0 = !{i16 2, !"sllvm-module", !"enclave1"}

!llvm.module.flags = !{!0}
```

Listing 1: An isolated module in SLLVM-IR

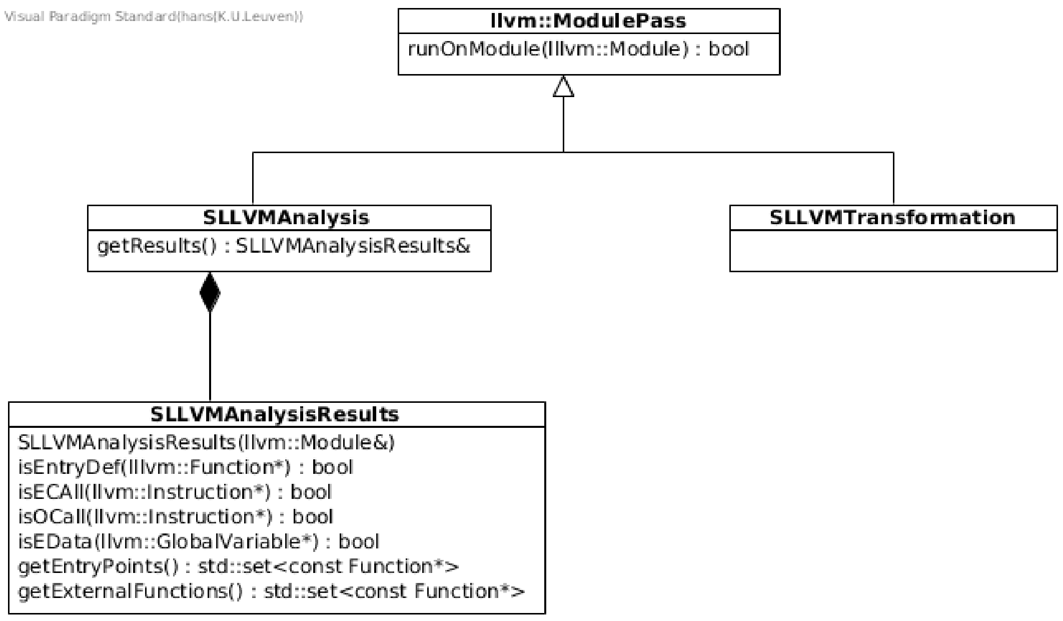
Visual Paradigm Standard(hans(K.U.Leuven))
llvm::ModulePass
runOnModule(llvm::Module) : bool
SLLVMAnalysis
getResults() : SLLVMAnalysisResults&
SLLVMTransformation
SLLVMAnalysisResults
SLLVMAnalysisResults(llvm::Module&)
isEntryDef(llvm::Function*) : bool
isECAll(llvm::Instruction*) : bool
isOCall(llvm::Instruction*) : bool
isEData(llvm::GlobalVariable*) : bool
getEntryPoints() : std::set<const Function*>
getExternalFunctions() : std::set<const Function*>

Figure 3.5: Class diagram of the generic part of the SLLVM middle end

The generic part of the middle end of the SLLVM compiler consists of an analysis pass and a transformation pass.

The analysis pass is represented by `SLLVMAnalysis` and `SLLVMAnalysisResults`. The `SLLVMAnalysis` class inherits from the LLVM `ModulePass` class and overrides the `runOnModule` method, acting as the entry point for the analysis process.

The analysis pass infers pertinent information about isolated modules. Firstly, an IR module is considered to be an isolated module if and only if it provides an implementation of an LLVM function that has the `sllvm-entry` function attribute attached to it. As explained previously, the `sllvm-entry` attribute is used in SLLVM IR to mark an LLVM function as an entry function.

Secondly, `SLLVMAnalysis` infers the out calls of an isolated module. Every call site is analyzed and when the callee resolves to a function declaration instead of a function definition, then it must be an out call.

Thirdly, global variables with local linkage not resolving to a declaration are considered a part of the protected data of the isolated module.

The generic SLLVM transformation, `SLLVMTransformation`, depends on the analysis pass. `SLLVMTransformation` also inherits from `ModulePass`. It implements an IR-to-IR transformation that is performed on isolated modules independent from any target architecture. Currently, its functionality is limited to two operations on isolated modules, both implemented in the body of the `runOnModule` method. First, it derives the module name based on the name of the LLVM IR source file. Then it adds the `sllvm-module` flag to the LLVM `llvm.module.flags` name metadata to convey to the other LLVM subsystems that it concerns an isolated module.

3.2.3 Discussion

SLLVM makes an important separation in the middle end. It separates the design and the implementation of SLLVM's intermediate abstraction layer into generic components and platform specific components. The common code in the generic analysis and transformation passes minimizes duplication of logic and maximizes code reuse. This section dealt with the generic components. Sections 3.4 and 3.5 discuss architecture specific transformation passes.

SLLVM extends LLVM's IR with a limited amount of new constructs. The reason for this is that it aims to avoid adding concepts that can be expressed in the existing programming model of LLVM IR.

The IR's module construct is extended to express which modules should preserve the properties of isolation after compilation. In an environment where this is the case for every module, the extension would not be needed at all. However, in practice, almost all security architectures are limited in the number of isolated modules that they can support, which warrants the extension.

The existing IR concept of linkage is used to express which programming elements are within the protection boundaries of an isolated module, and which ones are not. All entities with local linkage are hidden from all other modules. Because SLLVM maps an isolated module onto an ordinary module, no extensions are required to denote encapsulation.

The domain crossing intrinsics raise the level of abstraction. Lowering the domain crossing intrinsics into architecture specific assembly code is one of the responsibilities of the code generator in the back end.

Although the domain crossing intrinsics are defined as linguistic abstractions in SLLVM's IR, which is an element of the generic part of the compiler's middle end, they are ignored by the generic transformation pass. This is because the exact location where they should appear in the list of IR instructions is platform specific. The Sancus specific transformation pass for example, discussed in section 3.4, adds `sllvm_enter`, `sllvm_exit`, `sllvm_return` in the body of the generated dispatcher.

3.3 Software Module Isolation in SLLVM-C

One of the requirements of SLLVM is to stay as close as possible to the native programming models of the programming languages it supports, without changing the semantics of the existing language concepts. The number of introduced annotations or language extensions should be kept to an absolute minimum and when they are needed, they should be as non-intrusive as possible. Meeting this requirement will ease the adoption of this new technology by the developers at stake.

This section presents an approach for representing software module isolation in C. It explains the changes to Clang and concludes with a discussion of the approach.

3.3.1 Programming Model

Software module isolation is a property of computer systems that protect software modules against attacks that come from other parts of the enclosing application.

How does this fit in the existing C programming model? Paragraph 6.2.2.3 of the C specification [26] states:

> "If the declaration of a file scope identifier for an object or a function contains the storage-class specifier static, the identifier has internal linkage."

This means that the `static` storage-class specifier for entities at file-scope is used to limit the scope of identifiers. The function or the object that is denoted by an identifier with internal linkage can only be referred to by identifiers belonging to the same translation unit. Given that a translation unit consists of a preprocessed source and that it compiles into a single object file, an entity with internal linkage can not be referenced by code that does not belong to the same source file where the entity is defined, unless it has been included by a preprocessor directive.

Because the C specification already provides a mechanism to hide functions and objects within a compilation unit, it seems natural to consider the compilation unit as the unit of modularity. In this perspective, using the `static` storage class specifier for file-scope identifiers is equivalent to expressing the idea of data hiding and encapsulation within a modular unit. Since C programmers are already familiar with these language concepts and use them to modularize their programs, introducing another concept with similar semantics leads to concept redundancy and is an unnecessary source of confusion.

Based on paragraph 6.2.2.3 of the C standard and the implications for data hiding, we can provide a definition of an isolated module in the SLLVM programming model. However, there are practical limitations that prevent us from preserving the security properties of isolation at the lowest abstraction layers for every module in an application. When the target of compilation is a PMA, for example, compiling every single module as an isolated module is not something that is generally feasible because not all of these architectures scale well enough to support tens or even hundreds of PMs. In the case of Sancus, for example, the total number of protected software modules at any particular time has a fixed upper bound [40].

For this reason, a developer should be able to indicate when the compiler has to preserve the confidentiality and integrity of a module's code and data, according to the semantics of the corresponding source code. Conveying that information with a compiler option would avoid extending the C programming language with a new language concept. This way, a programmer could control the secure compilation of a translation unit in the build system, where the rest of the build configuration is maintained.

However, depending on the selected security mechanism to enforce a module's isolation properties after compilation, interfaces at a lower abstraction layer can be different for isolated modules than for ordinary, unprotected modules. At the ABI level, for example, the CC for entry functions of isolated modules might be different, as is the case for the Sancus architecture [40], in which case the compiler has to generate different code for invocations of functions that are part of ordinary modules than for invocations of functions that are part of an isolated module.

It would be cumbersome and error-prone to communicate this information to the compiler via a command-line option because, as made clear by the previous paragraph, this information is not only needed when compiling an isolated module but also when compiling a translation unit that depends on an isolated module.

The obvious location then, to mark a translation unit as an isolated module, is the header file. The header file represents the public interface of a C compilation unit and it is clear by now that marking a translation unit as an isolated module is information that belongs in that public interface. Typically in C, a header file is included by its corresponding implementation file and by all the files that depend on it. However, after preprocessing, the associations of the declarations to their originating header file are lost, and the declarations end up as an integral part of the including translation unit.

This means that every public function declaration needs to be decorated with an annotation to indicate that it is part of the public interface of an isolated module. To avoid doing this manually, which would be error-prone and not at all programmer-friendly, the preprocessor can be instructed to automate this task by means of a pragma. The pragma is added to the header file, expressing that the corresponding implementation file is to be securely compiled and that the translation units that depend on it might need special care when calling functions from the isolated module. When processing the header file, the preprocessor then attaches the required annotations to the declarations before embedding them in the translation units, effectively freeing the developer from this burden.

Now we have everything we need to specify the SLLVM programming model for software module isolation in C. To start with, the public interface of an isolated module is specified in a header file. The header file is included by the implementation of the isolated module at stake and by all the translation units that depend on it. This is important because the preprocessor performs a transformation to the function declarations. A pragma in the header not only instructs the compiler to preserve the security properties of the isolated module but it also implicitly marks all function declarations in that file as entry points to the isolated module. The translation unit that provides the definitions for these entry points will be considered as the implementation of the isolated module and will be compiled as an isolated module. Consequently, an isolated software module in this model corresponds to a complete compilation unit.

The semantics of the software entities within an isolated module are as follows. Objects with external linkage are not protected at all and can be accessed directly by code that does not belong to the isolated module. Function definitions with external linkage that do not have a matching entry point declaration are not considered to be part of the isolated module. Consequently, these functions cannot access the protected entities of the isolated module. Objects of isolated modules can only be accessed via the function definitions that do have a matching entry point declaration. They are the single points of entry into the isolated module. Objects and functions with internal linkage constitute the protected elements of an isolated module. They can only be accessed by entities that belong to the isolated module.

Listings 2, 3, 4 and 5 contain an example of an isolated module in SLLVM-C. Comments in the code listings illustrate the SLLVM-C programming model.

```c
#ifndef ENCLAVE1_H
#define ENCLAVE1_H

/* The following pragma is the only C language extension that is
 * required to support the SLLVM programming model for software
 * module isolation. The pragma instructs the C preprocessor to
 * mark every function declaration in this header as an entry
 * function. Every compilation unit that provides an
 * implementation for one of these functions will be compiled
 * as an isolated module.
 */
#pragma sllvm module

/* Declaration of public data */
extern int enclave1_shared;

/* Declaration of an entry function. */
void enclave1_entry1(void);

/* Multiple entry functions can be declared */
void enclave1_entry2(int p);

#endif
```

Listing 2: The interface of an isolated module (enclave1) in SLLVM-C

3.3.2 Implementation

Changes to Clang to support the programming model of SLLVM-C are summarized in table 3.1.

File	Added lines
include/clang/Basic/Attr.td	5
lib/CodeGen/CodeGenModule.cpp	3
lib/Sema/SemaDeclAttr.cpp	3
Sum	11

Table 3.1: Changes to Clang to support isolated modules in SLLVM-C

```c
#ifndef ENCLAVE2_H
#define ENCLAVE2_H

/* The following pragma instructs the C preprocessor to mark every
 * function declaration in this header as an entry function.
 */
#pragma sllvm module

/* Declaration of an entry function. */
int enclave2_entry(int k);

#endif
```

Listing 3: The interface of an isolated module (enclave2) in SLLVM-C

```c
#ifndef UNPROTECTED_H
#define UNPROTECTED_H

int unprotected_f(int p);

#endif
```

Listing 4: The interface of some unprotected module in SLLVM-C

The changes are limited to the definition of the `sllvm-entry` attribute in the table description file *Attr.td*, which is used to mark the entry points of an isolated module in the preprocessed compilation units. The SLLVM pragma to flag a C module as an isolated module, is expanded by the preprocessor into code with the `sllvm-entry` attribute attached to the proper entry functions. Although support for the pragma is not yet supported in the proof of concept implementation of the compiler, the code it expands to is fully supported.

The files *CodeGenModule.cpp* and *SemaDeclAttr.cpp* contain the code on how to map the C-level `sllvm-entry` attribute onto the LLVM IR-level `sllvm-entry` function attribute.

3.3.3 Discussion

People typically do not like change. By limiting changes, by not unnecessary disrupting the habits of developers, the chances of adoption of a new technology increase. Concept redundancy is a source of confusion for seasoned programmers, wondering why a new concept is needed. Programmer comfort is an important aspect

```c
/* The header of this implementation file indicates that this
 * concerns an isolated module. This does not have to be repeated.
 */
#include "enclave1.h"
#include "enclave2.h"
#include "unprotected.h"

/* The following variable contains security insensitve data that
 * logically belongs to this isolated module. This type of data
 * can be accessed by code that resides outside the protection
 * boundaries of the isolated module.
 */
int enclave1_shared;

/* The declaration of the following file scope identifier contains
 * the storage-class specifier static. According to the C standard,
 * this means that the identifier has internal linkage. The
 * variable can only be accessed by code of the isolated module.
 */
static int secret;

/* The declaration of the following file scope identifier contains
 * the storage-class specifier static. According to the C standard,
 * this means that the identifier has internal linkage. The
 * function can only be invoked by code of the isolated module.
 */
static void internal_func(void)
{
  /* Private data can be accessed from internal functions */
  secret++;
}

/* The definition of an entry function requires no annotation. */
void enclave1_entry1(void)
{
  internal_func();        /* Invocation of an internal function */
  enclave2_entry(secret); /* An out call to protected code */
}

/* Multiple entry functions can be defined */
void enclave1_entry2(int p)
{
  unprotected_f(p); /* An out call to unprotected code */
}
```

Listing 5: The implementation of an isolated module (enclave1) in SLLVM-C

and not paying attention to it may lead to disapproval by a group of people that have an important vote in the decision process of the adoption of an otherwise excellent piece of technology.

To securely compile isolated modules written in the C programming language, no fundamental changes are required to C's native programming model. This is a result of SLLVM's design goal to stay as close as possible to the original programming model of the languages it supports. By unifying the boundaries of isolated modules with the boundaries of a C compilation unit, the C language extension to support isolated modules can be limited to the addition of a single pragma. In an environment where every module would be compiled as an isolated module, no annotation would be required at all.

By including the definition of the `sllvm-entry` attribute in the *Attr.td* table description file, the SLLVM attribute is added to the list of natively supported attributes. The definition declaratively specifies that the entry function annotation can only be attached to functions. This makes it impossible to make errors by attaching it to other constructs, such as global variables. The Sancus toolchain, by contrast, defines its extensions to the C programming model as string annotations, which does not prevent attaching the annotation to the wrong constructs, decreasing programmer comfort and leading to more debugging efforts.

3.4 SLLVM and Sancus

PMAs are a class of security architectures that are designed to ensure the secure execution of software modules [52]. This makes them an attractive target for this master's thesis as it is focused on the security property of software isolation. It should be relatively straightforward to map higher-level programming language abstractions related to isolation to the low-level instructions of these architectures.

An important contribution of this master's thesis is a proposal for an alternative implementation of the Sancus toolchain. Currently, this toolchain is implemented in a rather ad-hoc way and does not provide a good starting point for SLLVM, a platform that has the ambition to become a generic secure compiler infrastructure.

This section is organized as follows. First, the C programming model for module isolation of the legacy Sancus toolchain and its implementation are briefly explained. The C programming model of SLLVM was previously discussed in section 3.3.1. Thereafter, the design and the implementation of SLLVM-Sancus are described. Finally, the different approaches are contrasted and discussed.

3.4.1 Legacy Programming Model

In the programming model of the legacy toolchain, a PM is declared with the `DECLARE_SM(module_id, vendor_id)` macro, which provides an identification of the module and its vendor. Additionally, this model introduces three explicit annotations that have to be used by the developer of a PM to indicate which functions should be part of the PM (`SM_FUNC(module_id)`), what its entry functions are

(`SM_ENTRY(module_id)`) and what data should be part of the protected data section (`SM_Data(module_id)`) [40].

The entry function annotations have to be added to the function declarations in the header file and to the corresponding function definitions in the implementation file. In the Sancus C programming model, one C compilation unit does not correspond to one PM. The elements of one protected module are linked together via the `module_id` that is part of every annotation and this mechanism makes it possible to spread the parts of one protected module over several source files.

Code listings 6, 7 and 8 are the Sancus-C equivalents to the SLLVM-C 2, 3 and 5 listings respectively. Listing 4 does not change. In Sancus-C, every programming element that is a member of a PM requires an explicit annotation with the unique name of the PM attached to it. Comments in the code listings further illustrate the Sancus-C programming model and contrast it with SLLVM-C.

```c
#ifndef ENCLAVE1_H
#define ENCLAVE1_H

/* The Sancus support header must be included in order to develop
 * Sancus protected modules with the legacy toolchain.
 */
#include <sancus/sm_support.h>

/* The SancusModule declaration needs to be manually added. This
 * variable contains metadata of the module.
 */
extern struct SancusModule enclave1;

/* Declaration of public data. */
extern int enclave1_shared;

/* The SM_ENTRY annotation paramterized with the name of the
 * protected module is required for entry functions.
 */
void SM_ENTRY(enclave1) enclave1_entry1(void);

/* The name of the protected module must be repeated in every
 * Sancus annotation.
 */
void SM_ENTRY(enclave1) enclave1_entry2(int p);

#endif
```

Listing 6: The interface of an isolated module (enclave1) in legacy Sancus-C

```
#ifndef ENCLAVE2_H
#define ENCLAVE2_H

/* The Sancus support header must be included in order to develop
 * Sancus protected modules with the legacy toolchain.
 */
#include <sancus/sm_support.h>

/* The SancusModule declaration needs to be manually added. This
 * variable identifes the module and the vendor.
 */
extern struct SancusModule enclave2;

/* The SM_ENTRY annotation paramterized with the name of the
 * protected module is required for entry functions.
 */
int SM_ENTRY(enclave2) enclave2_entry(int k);

#endif
```

Listing 7: The interface of an isolated module (enclave2) in legacy Sancus-C

3.4.2 Legacy Implementation

The legacy Sancus compiler consists of two major parts. First, a set of MSP430 assembly language stub files contain the generic parts of the Sancus protection code. For example, Sancus implements the logical entry points on top of the sole physical entry point by means of a jump table. The code to look up a logical entry point in the jump table is part of the generic *sm_entry.s* stub file while the jump table, which is different for every PM, is not.

Second, the `SancusModuleCreator` class implements an LLVM transformation pass. It implements an IR-to-IR transformation that realizes the functionality as described in the section 2.3.2. For example, the `SancusModuleCreator` instruments the entry function invocations according to the Sancus CC. It also generates the jump table that is used by the generic *sm_entry.s* assembly stub. The transformation injects LLVM IR inline assembly blocks, that express the desired behavior in MSP430 assembly code.

`SancusModuleCreator` depends on a number of helper classes. The helper class `AnnotationParser` is responsible for parsing the Sancus-specific annotations. `AnnotationParser` provides a method for retrieving all the annotations of a given C entity, such as a function or a global variable. `SancusModuleInfo` provides a number of utility methods for producing unique assembly language labels that are

```
#include "enclave1-sancus.h"
#include "enclave2-sancus.h"
#include "unprotected.h"

/* A protected module is declared with the DECLARE_SM macro. The
 * macro expands into a SancusModule defintion that identifies
 * the module and the vendor.
 */
DECLARE_SM(enclave1, 0x1234)

int enclave1_shared;

/* The SM_DATA annotation is used to define private data. This
 * annotation is redundant with the 'static' specifier.
 */
static int SM_DATA(enclave1) secret;

/* The SM_FUNC annotation is used to define private functions. This
 * annotation is redundant with the 'static' specifier.
 */
static void SM_FUNC(enclave1) internal_func(void)
{
  secret++;
}

/* The SM_ENTRY annotation must also be attached to the
 * definition of an entry function.
 */
void SM_ENTRY(enclave1) enclave1_entry1(void)
{
  internal_func();
  enclave2_entry(secret);
}

void SM_ENTRY(enclave1) enclave1_entry2(int p)
{
  unprotected_f(p);
}
```

Listing 8: The implementation of an isolated module (enclave1) in legacy Sancus-C

derived from the module's name. Finally, the `FunctionCcInfo` utility class provides information on which of the CC registers are actually used by a given function. This information is used when a PM transfers control to code external to the module to avoid that security-sensitive information is leaked through one of the registers. Figure 3.6 contains the corresponding UML class diagram.

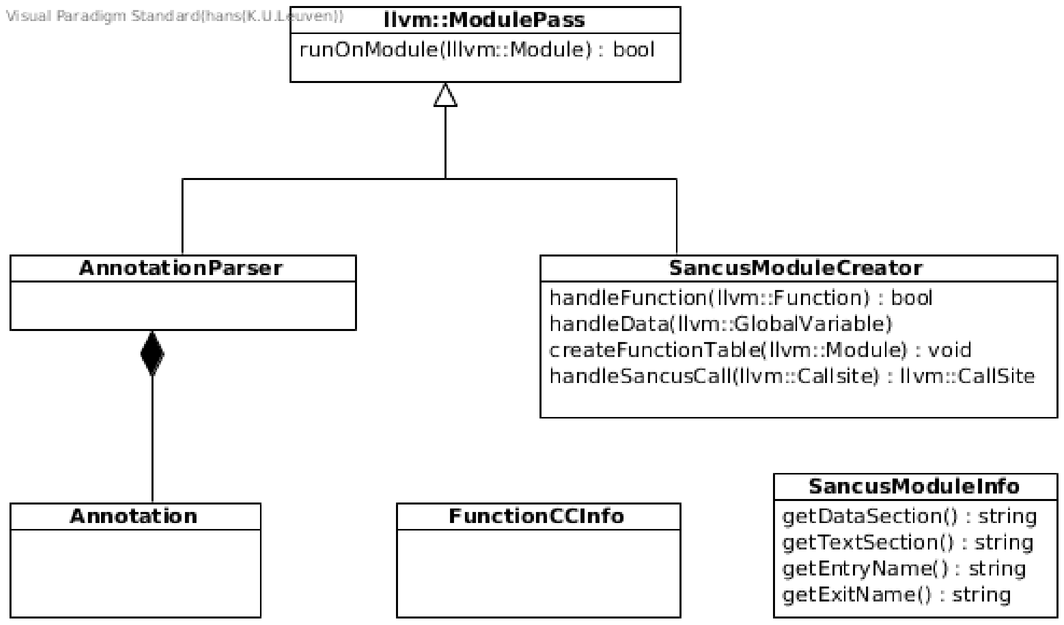
Visual Paradigm Standard(hans(K.U.Leuven))
llvm::ModulePass
runOnModule(llvm::Module) : bool
AnnotationParser
SancusModuleCreator
handleFunction(llvm::Function) : bool
handleData(llvm::GlobalVariable)
createFunctionTable(llvm::Module) : void
handleSancusCall(llvm::Callsite) : llvm::CallSite
Annotation
FunctionCCInfo
SancusModuleInfo
getDataSection() : string
getTextSection() : string
getEntryName() : string
getExitName() : string

FIGURE 3.6: Class diagram of the legacy Sancus LLVM transformation pass

The Sancus transformation is implemented by the SancusModuleCreator class. The other classes are helper classes.

A developer using the Sancus toolchain to compile his or her application does not directly deal with the LLVM-based executable. Instead, the Sancus toolchain is installed as a pair of Python scripts, sancus-cc and sancus-ld, that wrap the interactions with the LLVM compiler and the GNU linker respectively. The main task of the Sancus linker, sancus-ld, is to preprocess the object files to replace the placeholder labels from the generic assembly stubs with labels that are derived from the name of the PM with the purpose of making them unique. After this preprocessing step, the actual linking is performed by the MSP430 GCC linker.

3.4.3 SLLVM Implementation

SLLVM-Sancus is largely based on the secure compilation scheme for PMAs that was proposed by Agten et al. [14] and that is also implemented by the legacy Sancus compiler [40].

The C front end of the SLLVM Sancus compiler is implemented as a Clang extension that translates SLLVM-C to SLLVM-IR, as previously discussed in section

3.3.

The middle end of the SLLVM Sancus compiler consists of two passes. First, a generic IR-to-IR transformation is applied to each isolated module as explained in section 3.2, where an isolated module corresponds to a PM in the context of Sancus. Second, a Sancus-specific transformation pass converts a generic SLLVM-IR module into one that contains Sancus-specific information. SLLVM extends the LLVM IR for this purpose by adding an abstraction for the Sancus CC, `SANCUS_ENTRY`. Figure 3.7 shows the UML class diagram of the parts of the SLLVM middle end that are relevant for Sancus.

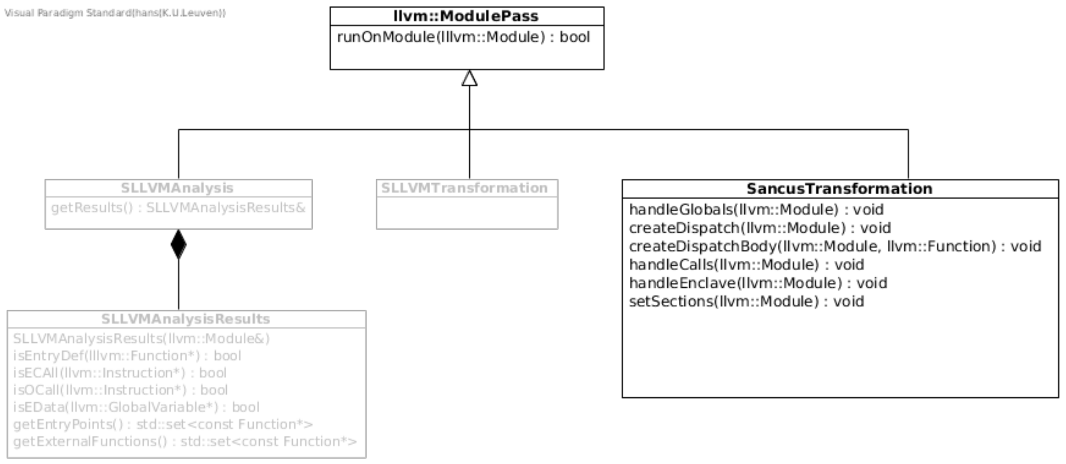
Visual Paradigm Standard(hans(K.U.Leuven))
llvm::ModulePass
runOnModule(lllvm::Module) : bool
SLLVMAnalysis
getResults() : SLLVMAnalysisResults&
SLLVMTransformation
SancusTransformation
handleGlobals(llvm::Module) : void
createDispatch(llvm::Module) : void
createDispatchBody(llvm::Module, llvm::Function) : void
handleCalls(llvm::Module) : void
handleEnclave(llvm::Module) : void
setSections(llvm::Module) : void
SLLVMAnalysisResults
SLLVMAnalysisResults(llvm::Module&)
isEntryDef(lllvm::Function*) : bool
isECAll(llvm::Instruction*) : bool
isOCall(llvm::Instruction*) : bool
isEData(llvm::GlobalVariable*) : bool
getEntryPoints() : std::set<const Function*>
getExternalFunctions() : std::set<const Function*>

Figure 3.7: SLLVM middle end class diagram for the Sancus security architecture

The middle end of the SLLVM Sancus compiler consists of three main components: the generic `SLLVMAnalysis` and `SLLVMTransformation` classes and the Sancus-specific `SancusTransformation` class. The greyed out components are explained in section 3.2.2.

Finally, SLLVM modifies the existing LLVM MSP430 back end to lower the Sancus CC and the SLLVM domain crossing intrinsics, previously discussed in section 3.2.

SLLVM Middle End

The Sancus transformation is implemented as an IR-to-IR transformation in less than 300 lines of C++ code, including the generic components that were discussed in section 3.2. `SancusTransformation` (figure 3.7) is implemented as an LLVM pass that transforms a generic SLLVM module, independent from any specific security mechanism, into one that is specific to the Sancus architecture, but still independent from a specific ISA.

Because the `SancusTransformation` class inherits from `ModulePass`, it needs to implement the `runOnModule` method to realize the desired transformation. Its functionality is roughly decomposed into three methods: `handleEnclave`, `handleCalls` and `setSections`.

Firstly, `handleEnclave` is invoked on all the modules that need to preserve the property of isolation, as indicated in the source code and inferred by the analysis pass. 3.2. Plain old LLVM modules are skipped.

The method `handleEnclave` itself depends on two other methods: `handleGlobals`, and `createDispatch`. The first one, `handleGlobals` ensures that all global variables that are categorized by the analysis pass as state that is private to the module, are kept private indeed. It does so by overriding the linkage type of these variables to `InternalLinkage`. Furthermore, this method also creates a local stack, a local stack pointer variable that is initialized with the top of the local stack and some other module-private globals that are used for storing the callee save registers before invoking external methods. This step is required to be able to restore these resisters when control flow returns, as they are cleared when performing the out call to prevent unauthorized information flows.

The second method that the top level `handleEnclave` method depends on is `createDispatch`, whose main responsibility is to generate the dispatcher, the code that implements the multiple logical entry points on top of a single physical entry point. The dispatcher is mapped by the linker onto the physical entry point. SLLVM extends LLVM IR with a new CC, `SANCUS_ENTRY`, which is attached to the dispatcher.

As opposed to the legacy Sancus toolchain, the implementation of the SLLVM dispatcher is expressed exclusively in LLVM IR with a switch instruction, keeping it independent of any specific ISA. In LLVM IR, the switch instruction is used to specify a table of values and destinations [7]. The SLLVM compiler assigns a unique identifier and label to every entry function, a pair that is added as an entry in the table described by the switch instruction. The label's destination contains the instructions that delegate the call to the corresponding logical entry point.

For each entry point, the compiler also generates two global external constants whose names are derived from the name of the entry point at stake. One constant is assigned the identifier of the entry function, the other is an alias of the dispatcher whose purpose is to improve the readability of the generated code which makes debugging the compiler easier. To prevent other modules from bypassing the dispatcher by calling an entry function directly, `SancusTransformation` sets the linkage of each entry function to `InternalLinkage`.

The transformation is also responsible for marking the code points that correspond to the crossing of a protection domain. Most of them occur in the dispatcher. The SLLVM domain crossing intrinsics, explained in section 3.2, are added at the appropriate places in the code: `sllvm_ecall` to mark the module's entry via an entry call, `sllvm_exit` to mark the module's exit by returning from an entry call and `sllvm_return` to mark the module's entry by returning from an out call.

Secondly, the top-level method `handleCalls` operates on all the modules, not only the protected ones. This method replaces every entry call by a call to the module's dispatcher. It copies the arguments of the original call instruction, adds one additional parameter, the identifier of the logical entry function, and sets the CC to `SANCUS_ENTRY`.

For out calls, `handleCalls` injects a call to the SLLVM intrinsic `sllvm_ocall` right before the out call, to mark the appropriate protection domain switch. Just like the other domain crossing intrinsics, `sllvm_ocall` is an abstraction to avoid having to commit to a specific architecture early in the compilation process. It is the back end's task to generate the corresponding assembly code for a specific target.

Thirdly, the top-level method `setSections` is responsible for assigning the module's global variables and function definitions to explicit memory sections. The code and the data of every PM is placed in one of the following dedicated sections, where *pmName* stands for a unique name that is derived from the PM's name:

sllvm.data.*pmName*: For storing the data that is private to the module, isolated from the other modules.

sllvm.text.*pmName*: For storing the module's constants and code.

sllvm.text.dispatch.*pmName*: Only the dispatcher is assigned to this section. The Sancus architecture requires that the physical entry point is placed at the beginning of the module.

Finally, `SancusTransformation` generates a reference to the global variable that stores the global stack pointer. Access to this variable is required to be able to implement the stack switching feature, explained in section 2.3.2.

SLLVM Back End

The LLVM infrastructure provides a target-independent code generator, a framework that contains a set of reusable components for translating LLVM IR into assembly code for a specified target [6].

LLVM converts the IR into a selectionDAG, a directed acyclic graph that is used for code generation. Each node in the graph represents an IR instruction. During code generation, the instruction selection phase of the compilation process maps the IR instructions to architecture-specific instructions through a number of passes. Eventually, the instruction selection phase turns the LLVM code into a DAG of target instructions [6].

The MSP430 back end operates on the LLVM IR to convert it to the MSP430 assembly language. The SLLVM modifications to support Sancus are mainly located in the `MSP430AsmPrinter` class, the `MSP430ISelLowering` class and in the `MSP430InstrInfo` target description. The `MSP430ISelLowering` modifications by SLLVM to support Sancus, represent the bulk of the changes in the MSP430 back end. Roughly 75% of the SLLVM-Sancus code in the back end is located in `MSP430ISelLowering`.

The first task of the SLLVM Sancus back end is to lower the four SLLVM domain crossing intrinsics, explained in section 3.2. To achieve this, it modifies the `MSP430InstrInfo` target description file, which contains the definitions of all the instructions for the MSP430 target. The lowering is specified in a declarative way by pattern matching on these domain crossing intrinsics in the SelectionDAG.

The targets of the call instructions, that are part of the result of this lowering process, are emitted by the `MSP430AsmPrinter`. This class overrides the

`EmitEndOfAsmFile` virtual method of its `AsmPrinter` parent to achieve this. Each PM has its own copy of the lowered intrinsics and they are placed in the module's dedicated text section. The `EmitEndOfAsmFile` method also generates for each PM the appropriate Sancus instructions for activating the hardware protection of the PM at stake.

The second task of the SLLVM Sancus back end is to lower the Sancus CC. The `MSP430ISelLowering` class is responsible for replacing or removing operations and data types in the SelectionDAG that are not supported natively by the MSP430 target architecture [6]. Lowering the LLVM IR calling conventions is a part of this responsibility. The `lowerSancusCall` method is the entry point in the `MSP430ISelLowering` class for lowering the Sancus CC.

SLLVM reduces the required additional number of registers in the Sancus CC from three, as is the case in the legacy Sancus toolchain, to two, which results in more efficient code. The Sancus CC follows the MSP Embedded Application Binary Interface (EABI) for passing the function arguments. Registers R12 through R15 are used for this purpose. No changes are required for this part of the CC.

Register R6 is used for passing the identifier of the logical entry function to the dispatcher. Register R7 for passing the return address, the instruction immediately after the call to the dispatcher. The method `lowerSancusCall` and its helper methods are responsible for generating the appropriate instructions to make sure that these registers are assigned the proper values before invoking the dispatcher. Furthermore, this method takes care of saving and restoring the global and local stack pointers where necessary. For out calls, the legacy Sancus CC uses an additional register to pass the number of arguments of the callee to the generic assembly stub code. SLLVM derives this information from the function signature at compile-time, and generates ad-hoc code for each out call.

SLLVM Linker

SLLVM-Sancus does not require a custom linker, as is the case with the legacy toolchain.

As discussed previously, every PM in Sancus only has a single physical entry point. Therefore, the compiler generates a dispatcher that implements multiple logical entry points on top of the physical entry point. At link-time it must be arranged that the dispatcher is mapped onto the module's physical entry point.

This is done as follows. The middle end of the SLLVM compiler assigns the dispatcher to the sllvm.text.dispatch.*pmName* section, where *pmName* is a name that is derived from the name of the protected module. The linker script is used to specify that the dispatch section is to be placed at the beginning of the text section of the PM. The Sancus architecture requires that the physical entry point is placed at the beginning of the module.

3.4.4 Discussion

The discussion contrasts the approach of the legacy Sancus toolchain with the approach of SLLVM using the following criteria: the programming model for module isolation in the C programming language, software design, performance and security.

Programming Model

Unnecessary duplication in the programming model of the legacy Sancus toolchain does not contribute to programmability. Sancus annotations in the header files need to be repeated in the corresponding implementation files, a practice with little added value. Furthermore, having to repeat PM names in every single Sancus annotation is cumbersome and error prone. Simple typographical errors, for example, are enough to assign elements to the wrong PM, requiring more debugging effort.

Concept redundancy in the Sancus programming model is another issue that degrades programmability and programmer friendliness. Most annotations of the Sancus programming model express ideas that are already available as linguistic abstractions in the C programming language. This redundancy in programming constructs is confusing for new Sancus developers getting familiar with its programming model and tedious for experienced Sancus developers.

The C programming model for software module isolation that is proposed by SLLVM provides an alternative to this legacy model. It is sufficiently expressive to capture all of the semantics of the legacy model.

SLLVM's programming model for software module isolation simplifies programmability. It is easier to write PMs for the Sancus architecture in SLLVM-C than it is the case for Sancus-C. One of the requirements of SLLVM is to stay as close as possible to the native programming model of the programming languages it supports. With the addition of a single pragma, it definitely satisfies this requirement for the C programming language.

Furthermore, the SLLVM model encourages separation of concerns and modular decomposition, important principles of software design. Adhering to these principles leads to an improved software quality and is beneficial for software properties such as readability, understandability and maintainability. While Sancus allows spreading the elements of one protected module over several compilation units, SLLVM-C requires that an isolated module's elements are part of the same compilation unit.

Software Design

The design and the implementation of the legacy Sancus compiler commits itself to a specific target architecture very early in the compilation process. Consequently, this ad-hoc design does not adhere to some generally accepted software engineering and compiler principles.

By adding MSP430 assembly code to the LLVM IR, the Sancus compiler mixes two abstraction layers, effectively collapsing them into a single one. This is clearly a violation against basic architectural principles such as separation of concerns and maintaining high cohesion in software components.

The principle of abstraction is at the core of a good design. Not abstracting the target architecture in the middle end of the compiler prevents code from being shared when supporting more than one underlying ISA for Sancus. Currently, the Sancus prototype is based on the MSP430 instruction set, but the Sancus ideas can be applied to other low-cost, low-power micro controllers.

Abstraction is a useful design principle when retargetting new architectures. To give an example, porting the Sancus PMA from MSP430 to RISC-V hardware would require a complete rewrite of the MSP430 specific implementation of the toolchain, involving duplication of a significant part of the compiler logic. Factoring out the common logic in architecture agnostic components would be a significant improvement to the compiler's design.

SLLVM leverages the elements of the LLVM compiler infrastructure such as the abstract IR and the target independent code generator to support the Sancus security architecture. SLLVM's design adheres to the important principle of separation of concerns [31]. By tightly integrating with the LLVM compiler infrastructure, SLLVM's architecture modularizes separate distinct concerns into different components, each having a cohesive responsibility.

The SLLVM approach maintains a clear distinction between the traditional three abstraction layers in a compiler, by keeping the IR independent of any specific source language and independent of any specific target architecture. The SLLVM compiler postpones the commitment to a specific instruction set as late as possible by expressing the concepts of isolation and how to enforce them in the Sancus context at a higher level of abstraction in an abstract representation of the actual machine code, the SLLVM IR, an extension of LLVM IR.

Abstraction renders the SLLVM toolchain portable. The Sancus dispatcher logic, for example, is written in LLVM IR and translated during the code generation phase of the compilation process into MSP430 assembly code by the existing LLVM machinery. However, it can be translated to other architectures without any additional effort because of its independence of any specific ISA. Central to the dispatcher's implementation is the LLVM IR switch instruction. Depending on properties of the target machine the switch instruction may be converted to assembly code in different ways. For example, it could be translated into a number of chained conditional branches or into a lookup table [7].

The Sancus CC is another example of how SLLVM leverages abstraction to strive for high cohesion, low coupling and separation of concerns. By creating a linguistic abstraction for the CC, expressing that a function or a function invocation conforms to a specific CC can be done declaratively in IR, separating it from the concern of how to lower the CC at stake for a specific architecture.

SLLVM's approach is very different compared to the approach of the legacy toolchain, where low level MSP430 assembly code is injected in the higher level LLVM IR. Lacking important abstractions and architectural structure by implementing the Sancus compilation as a single LLVM transformation pass, maintainability, extensibility and readability of the legacy Sancus toolchain source code is hard, increasing the chance of introducing bugs and vulnerabilities.

Because of the presence of MSP430 assembly in the middle end, understanding the `SancusModuleCreator` code suffers similar drawbacks as reading and maintaining assembly code. Furthermore, the reader has to deal with three different languages. The transformation pass itself is written in C++ and it generates LLVM IR and MSP430 assembly. SLLVM limits the number of languages one has to deal with at the same time to two.

Performance

SLLVM for Sancus leads to better performance in three areas.

Firstly, SLLVM-Sancus comes with a compile time performance improvement compared to the legacy Sancus toolchain. That is because the latter requires that every PM is preprocessed by a custom Python linker script before the actual linking occurs. This additional step introduces an overhead in the linking process which is absent when compiling with SLLVM.

Secondly, compared with the legacy toolchain, the SLLVM code generator produces less machine instructions for switching protection domains, resulting in an improved runtime performance. Entry calls and out calls in Sancus are significantly more expensive compared to normal calls. This is due to the limitation of having only a single entry point and because of the precautions that have to be taken to avoid leaking security-sensitive information. Optimizing domain crossing costs is important, even more so on the resource constrained systems that Sancus is targeting.

For example, to invoke an entry function without parameters from unprotected code and return from it, it costs 52 MSP430 assembly instructions in code that is generated by the legacy Sancus toolchain. SLLVM reduces this cost to 38 instructions, an improvement of more than 25 percent.

Thirdly, the reduction from seven to six registers in the Sancus CC for out calls alleviates the pressure on the limited number of available registers. The legacy Sancus CC uses an additional register to pass the number of arguments of the callee to the generic assembly stub code. SLLVM derives this information from the function signature at compile-time, and generates ad-hoc code for each out call, resulting in less instructions and lower register pressure.

A lower register pressure implies that less memory spills and reloads from memory are needed, which has a direct impact on the runtime performance of applications. Accessing the CPU's registers is significantly faster than accessing memory, giving rise to executables that run faster. This means that we can expect a performance improvement as SLLVM makes an additional MSP430 register available for other purposes, a reduction of the register pressure with more than twelve percent.

Security

As mentioned in the background chapter of this document, reducing the size of the TCB is an important security design principle. Any vulnerability in the TCB can have severe consequences regarding the security of the system. A larger TCB is more difficult to read and understand, and contains more bugs. By limiting the size of the

TCB, there are less opportunities for an attacker to exploit vulnerabilities that can compromise the security of the whole computer system.

However, size is not the only property to consider when evaluating the security of the TCB. Any component of the TCB that is optimized for size, without considering aspects like source code readability and good design, does not really contribute to an improved security.

Convoluted or inferior design is definitely not beneficial for security. Security favors simplicity. Poor software architecture and bad design render a software implementation difficult to understand. A less-structured piece of software is harder to read because the reader must derive more information and structure herself, leading to more misinterpretations and mistakes. It makes it hard to reason about the correctness of a software product, also in terms of security, which increases the chances of introducing vulnerabilities.

The TCB consists of the components of a computer system that are responsible for the implementation and the enforcement of the security policy of a computer system. Clearly, this means that both the Sancus compiler and linker belong to the TCB. The Sancus compiler, for example, generates code that clears the CPU's registers when exiting a PM to avoid leaking security-sensitive information. Clients of the Sancus compiler have to be certain that the compiler is doing its job properly.

It is thus important from a security point of view to strive for a small implementation size of both the Sancus compiler and linker without compromising the quality of its design. SLLVM has a positive impact on the trustworthiness of the Sancus TCB because it realizes a significant reduction of the TCB size of the Sancus compiler compared to the legacy Sancus toolchain and, at the same time, it adheres to well-established software engineering principles.

The design and the implementation of SLLVM-Sancus and how this compares with the design and the implementation of the legacy Sancus compiler is discussed in sections 3.4.2 and 3.4.4.

Table 3.2 provides an overview of the number of files and the amount of code required to support Sancus in the legacy toolchain, organized per file type. Table 3.3 provides this overview for SLLVM. A summary of the existing LLVM files that are changed, and the number of added lines of code to these files to support Sancus in SLLVM is given by table 3.4.

Language	files	blank	comment	code
C++	4	166	37	645
Python	4	166	90	589
C/C++ Header	6	97	237	328
Assembly	6	31	56	184
C	4	15	2	74
Sum	24	475	422	1820

Table 3.2: Lines of code for Sancus support in the legacy Sancus toolchain

Language	Files	Blank	comment	Code
C++	3	72	38	274
C/C++ Header	5	60	5	185
Sum	8	132	43	459

Table 3.3: Lines of code for Sancus support in SLLVM

File	Added lines
lib/Target/MSP430/MSP430ISelLowering.cpp	159
lib/Target/MSP430/MSP430InstrInfo.td	30
lib/Target/MSP430/MSP430AsmPrinter.cpp	15
lib/Target/MSP430/MSP430ISelLowering.h	11
lib/Target/MSP430/MSP430InstrInfo.cpp	5
include/llvm/IR/Intrinsics.td	4
lib/Target/MSP430/MSP430MCInstLower.cpp	3
include/llvm/IR/CallingConv.h	2
Sum	229

Table 3.4: Added lines of code to existing LLVM files for Sancus support in SLLVM

The security-critical code that is generated by the compiler is also an integral part of the TCB for applications that rely on the Sancus security architecture. This means that the degree of confidence one can have in the security of the generated MSP430 assembly increases with smaller code size and better readability. As previously discussed, the code that is generated by the SLLVM back end is smaller in size than that from the legacy compiler. Furthermore, the legacy compiler produces code that is more difficult to understand because it makes more use of unstructured control flow.

3.5 SLLVM and Intel SGX

This section discusses how SLLVM supports Intel SGX, another PMA. It is organized as follows. First, the C programming model for module isolation of the Intel SGX SDK and its implementation are briefly explained. The C programming model of SLLVM was previously discussed in section 3.3.1. Thereafter, the design and the implementation of SLLVM-SGX are described. Finally, the different approaches are contrasted and discussed.

3.5.1 Legacy Programming Model

The SGX SDK is a collection of APIs, libraries and tools enabling software developers to create and debug SGX enclaves for C and C++ applications [3].

This section discusses a subset of Intel SGX SDK's programming model, sufficient to define basic enclaves with similar features as the protected modules from the

Sancus programming model. Advanced features, such as the EDL attributes that can be used with pointers, are out of scope of this master's thesis. They are briefly discussed in the future work section at the end of this document.

An enclave is typically built as a shared library, a DLL on Microsoft Windows or a shared object on Linux. This implies that an enclave can be created from several compilation units. Any compilation unit is either an integral part of an enclave or completely outside of it.

The SGX runtime provides a loader to dynamically load enclaves that are compiled as shared libraries. After successfully loading an enclave, a handle or an enclave ID is returned, which should be passed as a parameter when entry calls are performed [4].

When developing SGX enabled applications with the Intel SGX SDK, enclaves have to be described in the Enclave Definition Language (EDL), a Domain Specific Language (DSL) developed and maintained by Intel to describe trusted and untrusted software entities that are used in the enclave's function prototypes. EDL is an Interface Definition Language (IDL) used to specify the enclave interface. An EDL file specifies which functions are exported (trusted) and which ones are imported (untrusted) by the enclave under description. Each enclave is represented by one EDL file [4].

Syntactically, an enclave specification in EDL consists of two parts: a mandatory trusted block and an optional untrusted block. The trusted block specifies the enclave entry functions, the exported functions that constitute the enclave's public interface. The SGX hardware enforces that these functions are the only entry points of the enclave. The untrusted block specifies which functions of the outside application are invoked from within the enclave.

Listing 9 is the EDL specification for a part of the isolated module that was previously specified in the SLLVM-C listings 2 and 5. Comments in the listing illustrate the basic SGX programming model.

3.5.2 Legacy Implementation

The Edger8r tool, pronounced "edgerator", is a component of the Intel SGX SDK, that generates C/C++ edge routines. Edge routines are proxy functions that provide the implementation of the interface between the untrusted application and the enclave. The edge routines are C wrapper functions for the enclave exports, used for the entry calls, and for the imports, used by the out calls [4]. Edger8r generates the edge routines according to the enclave's interface as described by its corresponding EDL file.

For every SGX enclave, Edger8r generates four files. For example, it generates the following files for listing 9:

enclave1_t.h Provides the proxy declarations for the trusted edge routines. To be included by the hand-written source code of the enclave.

enclave1_t.c Provides the proxy definitions for the trusted edge routines. To be compiled as part of the shared library representing the enclave.

```
enclave
{
  // Export functions (ECALLs)
  // Trusted function prototypes
  trusted
  {
    public void enclave_entry1(void);

    public void enclave_entry2(void);

    void internal_func(void); // Private function
  };

  // Import functions (OCALLs)
  // Untrusted function prototypes
  untrusted
  {
    int enclave2_entry(int k);

    int unprotected_f(int p);
  };
};
```

Listing 9: An isolated module in EDL

enclave1_u.h Provides the proxy declarations for untrusted edge routines. To be included by the source code of the untrusted application.

enclave1_u.c Provides the proxy definitions for untrusted edge routines. To be compiled as part of the untrusted application.

For listing 9, Edger8r will generate two untrusted proxy functions, called by the application, for the entry calls and two trusted proxy functions, called by the enclave, for the out calls.

```
/* Prototypes of trusted edge routines */
sgx_status_t SGX_CDECL enclave2_entry(int* retval, int k);
sgx_status_t SGX_CDECL unprotected_f(int* retval, int p);
```

```
/* Prototypes of untrusted edge routines */
sgx_status_t enclave_entry1(sgx_enclave_id_t eid);
sgx_status_t enclave_entry2(sgx_enclave_id_t eid);
```

The SGX runtime provides the generic functions `sgx_ocall` and `sgx_ecall`, that take care of the necessary actions to cross protection domains, such as executing the appropriate SGX instructions and managing the thread control structures. The function `sgx_ocall` is invoked by the trusted edge routines while the untrusted edge routines use `sgx_ecall`.

For these generic runtime functions to execute the enclave specific functionality, a layer of indirection is added. The enclave's exported and imported functions have to be described in data structures that are used by the SGX runtime to lookup and invoke the correct functions.

As previously discussed, the SGX C programming model for isolation allows enclaves to be composed of several compilation units that are eventually compiled into a single shared object, the binary representation of the enclave. Like Sancus PMs, SGX enclaves have a single physical entry point. Logical entry points are supported as follows. The binary interface between the enclave and the SGX runtime requires that the shared object exports two symbols, `g_ecall_table` and `g_dyn_entry_table`. The symbol `g_ecall_table` provides a description of the enclave's exported interface and the symbol `g_dyn_entry_table` provides memory to store runtime metadata. The Edger8r tool generates C code to define these tables, based on the information in the enclave's EDL description.

3.5.3 SLLVM Implementation

The C front end of the SLLVM compiler is implemented as a Clang extension that translates SLLVM-C to SLLVM-IR, as previously discussed in section 3.3.

Similarly to SLLVM-Sancus (section 3.4), the middle end of the SLLVM compiler for SGX consists of two passes. First, a generic IR-to-IR transformation is applied to each isolated module as explained in section 3.2, where an isolated module corresponds to an enclave in the context of SGX. Second, an SGX-specific transformation pass converts a generic SLLVM-IR module into one that contains SGX-specific information. Figure 3.8 shows the UML class diagram of the parts of the SLLVM middle end that are relevant for SGX.

Contrary to SLLVM-Sancus, no extensions to the LLVM IR and no changes to the back end are required to add support for Intel SGX to SLLVM. The SGX runtime encapsulates the SGX specific instructions that need to be executed when crossing protection domains. Furthermore, the SGX runtime offers useful services such as the dynamic loading of enclaves and the management of multi-threaded enclaves.

For this reason, SLLVM-SGX targets the SGX runtime library. The SGX transformation is implemented as an IR-to-IR transformation in less than 300 lines of C++ code, including the generic components that were discussed in section 3.2. `SGXTransformation` (figure 3.8) is an LLVM pass that transforms a generic SLLVM module, independent from any specific security mechanism, into an SLLVM module that is specific to the Intel SGX security architecture.

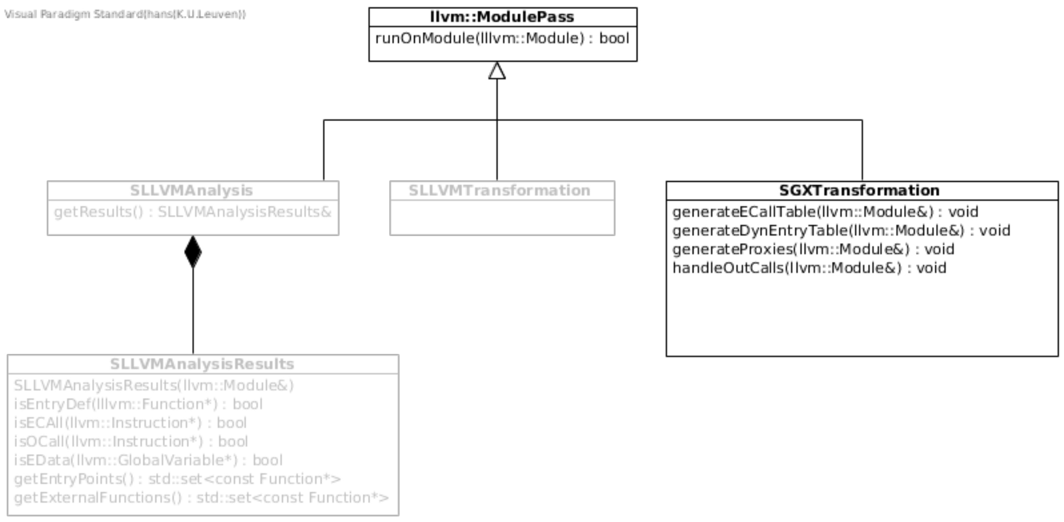
Visual Paradigm Standard(hans(K.U.Leuven))
llvm::ModulePass
runOnModule(lllvm::Module) : bool
SLLVMAnalysis
getResults() : SLLVMAnalysisResults&
SLLVMTransformation
SGXTransformation
generateECallTable(llvm::Module&) : void
generateDynEntryTable(llvm::Module&) : void
generateProxies(llvm::Module&) : void
handleOutCalls(llvm::Module&) : void
SLLVMAnalysisResults
SLLVMAnalysisResults(llvm::Module&)
isEntryDef(lllvm::Function*) : bool
isECAll(llvm::Instruction*) : bool
isOCall(llvm::Instruction*) : bool
isEData(llvm::GlobalVariable*) : bool
getEntryPoints() : std::set<const Function*>
getExternalFunctions() : std::set<const Function*>

Figure 3.8: SLLVM middle end class diagram for the SGX security architecture

The middle end of the SLLVM compiler for SGX consists of three main components: the generic `SLLVMAnalysis` and `SLLVMTransformation` classes and the SGX-specific `SGXTransformation` class. The greyed out components are explained in section 3.2.2.

Because the `SGXTransformation` class inherits from `ModulePass`, it needs to implement the `runOnModule` method to realize the desired transformation. Its functionality is roughly decomposed into four methods: `generateECallTable`, `generateDynEntryTable`, `generateProxies`, and `handleOutCalls`.

Firstly, the methods `generateECallTable` and `generateDynEntryTable` generate, as their names suggest, the `g_ecall_table` and `g_dyn_entry_table` tables, discussed in section 3.5.2. Both tables are generated based on the SLLVM-IR programming model for software module isolation. They are expressed in LLVM IR, independent from any source language, and lowered to x86 assembly by the unmodified LLVM x86 back end.

Secondly, the method `generateProxies` generates LLVM IR edge routines that provide the implementation of the interface between the untrusted application and the enclave.

Thirdly, the method `handleOutCalls` replaces the calls to the functions that are outside of the module by calls to the appropriate edge routine.

3.5.4 Discussion

The SGX transformation pass of SLLVM generates semantically equivalent code as the Edger8r tool from the Intel SGX SDK, but at a lower abstraction layer. The Edger8r tool is language-specific as it generates C code based on an IDL that is designed to compensate for the poor expressivity of the C programming language.

The SGX transformation pass of SLLVM, on the other hand, generates LLVM IR, a representation that is not too high-level, as in closely attached to a specific source language, nor too low-level, as in close to a specific target architecture.

The Intel SGX SDK only supports the development of applications in C and C++. The Edger8r tool and the Enclave Description Language are closely entangled with the C programming language. Application developers that want to use a safe programming language for writing SGX enabled applications have to resort to a Foreign Function Interface (FFI) to be able to invoke the generated edge routines.

By defining a programming model for software module isolation at a lower abstraction layer, such as the compiler's IR, and by generating IR code instead of C code, both the enclave's specification and the code generator can be decoupled from the C programming language to become language independent.

By supporting the more advanced features of EDL in SLLVM-IR, such as the attributes that can be used with pointers (see section 4.4), the resulting code that is generated for enforcing these features can be recycled in a non SGX context by lowering it to other target architectures, such as Sancus-MSP430.

Front ends for other programming languages than C/C++ that natively support (some of) these concepts can target the extended SLLVM-IR as well. This would pave the way for developing SGX enabled applications in safer languages, such as Rust and Ada, without depending on FFIs or other sub-optimal language interoperability techniques.

The proof of concept implementation of SLLVM-SGX shows that this is possible. SLLVM-SGX generates LLVM IR versions of the edge routines and the enclave interface description tables, compatible with the SGX runtime. However, instead of relying on an external description language, it does so based on the information that is contained in the SLLVM-IR programming model for isolation, as discussed in section 3.2. The existing x86 LLVM back end is used to lower the generated LLVM IR code into x86 assembly to be further compiled down into a shared object file that can be loaded with the loader from the SGX runtime library.

Another issue with the approach of the SGX SDK, is that it increases the degree of duplication. C and C++ developers already have to deal with duplicated information. They have to maintain the declaration of external software entities, such as functions and variables, in a separate header file, in parallel with their implementation. Languages like Java don't require this duplication as they derive the interface from the implementation. The Intel SGX SDK requires the interface of SGX enclaves to be described in yet another file, in the Enclave Definition Language. This unnecessary duplication does not contribute to programmability.

3.6 Software Module Isolation in SLLVM-Rust

Hardware based PMAs like Sancus and Intel SGX are designed to provide an efficient enforcement mechanism to restrict the access of secret information to trusted software modules. They facilitate the protection of these individual software modules against attacks from the enclosing application or from a potentially malicious OS.

Furthermore, these security architectures have built-in encryption facilities and offer useful security features such as remote attestation for setting up a trust relationship with applications that run remotely on untrusted platforms.

However, these technologies are useless as a defense against attacks that exploit flaws in the design or vulnerabilities in the implementation of the modules themselves. For example, the assurance one has in the protection that is provided by PMAs is fallacious when the information that needs to be kept a secret is simply exposed by the return value of an entry function.

Both the Intel SGX SDK and the Sancus toolchain are for the development of C and C++ applications only. These programming languages are not particularly known for their safety properties and might lead to applications compromising the security properties of Sancus PMs and SGX enclaves. For example, programs written in these languages are susceptible to low-level attacks that exploit memory safety vulnerabilities such as buffer overflows and dangling pointers, elaborately discussed by Erlingsson et al. in [24].

Since C/C++ applications are likely to have security vulnerabilities that defeat the purpose of developing Sancus or SGX enabled applications in the first place, we need better memory safety guarantees. By writing the security-critical software modules of these types of applications in a safe programming language, we can eliminate a whole class of vulnerabilities that could otherwise compromise the security properties that are provided by these promising new security architectures.

Like C and C++, the Rust programming language is a system programming language with excellent performance characteristics, making it a suitable alternative to these languages. Unlike C/C++, Rust provides very strong security and safety properties, guaranteeing memory safety and thread safety.

The security guarantees of Rust are complementary to those of PMAs. Rust is a safe programming language that is able to protect a software module against a wide range of vulnerabilities that manifest themselves from within the module while security architectures such as Sancus and Intel SGX are able protect a software module against a possible hostile environment, like the application it is a part of or an untrusted OS.

By adding support for Rust to the SLLVM secure compiler infrastructure, application developers are offered an alternative to C/C++ for writing Sancus PMs and SGX enclaves, but with similar performance characteristics. SLLVM-Rust enables the developers to write the security-critical modules in a safe programming language to obtain an even higher level protection.

3.6.1 Programming Model

The purpose of this master's thesis is not to develop and implement a sound programming model for software module isolation in SLLVM-Rust. There simply isn't enough time to develop such a sound model, implement it and test it for Sancus and Intel SGX. Developing and implementing such a sound programming model, based on Rust's module system and its visibility modifiers, is considered future work. At the end of this section, a Rust listing is provided to show what this model might look like.

To demonstrate the genericity and language-independence of SLLVM, this master's thesis defines an ad-hoc model for module isolation in Rust, similar to what the legacy Sancus toolchain does for the C programming language. It should be straightforward to port the implementation of this model to a model that uses native Rust concepts.

In Rust, declarations can be annotated with 'attributes' [9]. Any item declaration may have an attribute attached to it. SLLVM-Rust uses the `linkage` attribute to indicate that a programming element is within the protection boundaries of an isolated module or not: the `external` value is used for public elements, the `internal` value is used for private elements. Entry functions have to be annotated with the custom attribute `sllvm_entry`.

This programming model suffices to express the ideas related to SLLVM's isolated modules, to demonstrate the mapping from Rust to the SLLVM IR and to test the execution of Rust PMs on the Sancus architecture.

Listing 10 provides an example of an isolated module in ad-hoc SLLVM-Rust, a model that can be used as a starting point for developing and implementing a sound SLLVM-Rust programming model for module isolation. It seems that no SLLVM-specific attributes are necessary in such a model when defining it in terms of Rust's module system, as illustrated in listing 11.

```rust
#![no_std]
#![feature(linkage)]
#![feature(custom_attribute)]

#[linkage = "internal"]
static SECRET: i16 = 42;

#[linkage = "internal"]
fn internal_func() -> i16 {
  SECRET
}

#[linkage = "external"]
#[no_mangle]
#[sllvm_entry]
fn enclave3_entry() -> i16 {
  internal_func()
}
```

Listing 10: An isolated module in SLLVM-Rust (ad-hoc model)

```
#![no_std]

pub mod enclave3 {
  static SECRET: i16 = 42;

  fn internal_func() -> i16 {
    SECRET
  }

  #[no_mangle]
  pub fn enclave3_entry() -> i16 {
    internal_func()
  }
}
```

Listing 11: An isolated module in SLLVM-Rust

3.6.2 Implementation

The Rust compiler, Rustc, performs a number of basic operations. First, it parses a Rust file and produces the corresponding Abstract Syntax Tree (AST) that closely matches the lexical syntax of the Rust language. A number of operations, such as macro expansion, are performed on the AST. Before type checking the code, the AST gets converted into High-level Intermediate Representation (HIR), a desugared version of the AST that is better suited for semantic analysis. After type checking, the HIR is lowered into Middle-level Intermediate Representation (MIR), a representation that is operated on by the borrow checker to enforce memory safety. Finally, the MIR is translated into LLVM IR, where LLVM takes care of the rest of the compilation process.

Changes to Rustc for supporting the ad-hoc SLLVM-Rust programming model for isolation are limited to the last step of the Rust compiler, the translation to LLVM IR. Only three additional lines of Rust code to the file *attributes.rs* are required for defining the mapping of the `sllvm_entry` SLLVM-Rust attribute onto the `sllvm-entry` SLLVM-IR attribute. The `internal` and `external` linkage values are translated by the Rust compiler to the `InternalLinkage` and `ExternalLinkage` LLVM linkage values respectively.

Adding the `no_std` attribute to isolated modules can dramatically reduce the size of the compiled isolated modules and, consequently, of the size of the application's TCB. Size is also a concern on the resource constrained embedded devices that Sancus is targeting. Much of the functionality that is made available by Rust's standard library is also available via the core crate, which is automatically brought into scope when compiling with `no_std`. This means that for compiling isolated modules without Rust's standard library, a library of the core crate, *libcore* needs to be available for the target platform.

For example, to be able to write isolated modules in Rust and compile them for the Sancus security architecture, an MSP430 *libcore* is required. The Xargo [10] build tool was used for this master's thesis to create a *libcore* for the MSP430 platform. Xargo was developed to make it easy to cross compile Rust crates for targets that don't have binary releases of the standard crates. There are plans to integrate Xargo's functionality into Cargo, Rust's package manager and build tool.

3.6.3 Discussion

This master's thesis provides the foundation for writing Sancus protected modules in a safe programming language. SLLVM has shown that it is feasible to run Rust PMs on the Sancus security architecture. Since the modified Rust compiler generates SLLVM-IR, isolated modules that are written in Rust can also be compiled as SGX enclaves. Because of time constraints, SLLVM compiled SGX enclaves have not been tested on platform.

To write SGX enclaves in the Rust programming language, Intel recommends the Baidu Rust SGX SDK [21] on their SGX website. The Baidu Rust SGX SDK is a framework that facilitates the development of safe SGX enclaves. SLLVM-Rust proposes an alternative to develop SGX enabled applications.

SLLVM-Rust is a proof of concept for the genericity of SLLVM and important challenges remain. Firstly, as already mentioned before, the ad-hoc programming model should be replaced by a sound model that uses native Rust constructs. SLLVM aims to stay as close as possible to the native programming model of a programming language. Rust already has a module system with visibility modifiers, so it seems logical to base the programming model for module isolation on that.

However, there are some issues with the Rust module system that need to be addressed. For example, the Rust module system allows multiple modules to reside in one source file while the Rust compiler translates separate Rust modules that are implemented in the same source file into a single LLVM IR module. This clearly breaks the abstraction of isolation and the associated security properties that are guaranteed in the originating Rust program. As another example, the Rust programming language allows nested modules, a concept that does not exist in the programming model of SLLVM-IR.

Secondly, developing isolated modules in a safe programming language may give a false sense of security. In a malevolent environment, pointer parameters to entry functions have to be handled with care. For example, validation code has to be written to ensure that every memory address that is derived from a pointer parameter does not belong to one of the module's protected memory regions. Intel EDL provides a partial solution for not having to manually write this validation code. In EDL, enclave writers are given a set of pointer attributes that can be used to declaratively specify additional constraints on pointers, such as indicating that parameters are input or output parameters. The Edger8r tool then generates the required validation code. Van Ginkel et al. further illustrate this problem and report on a work-in-progress towards a comprehensive solution [56], [57].

Thirdly, the paper on the Baidu Rust SDK [21] identifies some challenges related to multithreaded applications that probably have to be addressed in SLLVM-Rust as well if threading is to be supported. Multithreaded applications are out of scope of this master's thesis.

Chapter 4

Conclusion

Recently we have seen the emergence of interesting research areas that promise to improve the poor state of the security of our computing infrastructure. Secure compilation [14], [42], [43] and PMAs [50], [52], [39], [37], [40] are two such novel research areas. Secure compilers generate code that preserves the security properties of the corresponding source code. PMAs can offer an efficient mechanism to enforce some of these properties. This type of security architectures makes it possible to isolate security-critical modules from the enclosing application.

PMAs such as Sancus [40] and Intel SGX [37] have defined their own programming models and have created special-purpose SDKs that facilitate the development of isolated modules in C for the architecture at stake. This master's thesis has shown that it is possible to represent a security property in a generic way at the different levels of abstraction, independent from any source language and independent from any target architecture. A common programming model at the different abstraction layers and a common compiler toolchain maximize uniformity, reusability and programmability and contribute to safer language interoperability.

In the context of this master's thesis, a proof-of-concept secure compiler for software module isolation has been designed and implemented. First, a new implementation of the Sancus toolchain has been proposed, with improvements in overall design, programmability, usability, performance and security. This demonstrated the possibility to develop Sancus-enabled applications in the C programming language with a generic toolchain. Thereafter, the genericity of the design has been confirmed by adding support for another security architecture, Intel SGX, and another programming language, Rust.

As such, SLLVM lays the foundations of a unified secure compiler infrastructure. SLLVM encourages reusability as it avoids having to deal with different programming models and different toolchains for every possible combination of security architecture and programming language, a strategy that has proven its success in the development of conventional compilers.

This chapter is organised as follows. Section 4.1 summarizes the contributions of this master's thesis. Section 4.2 discusses the challenges that were encountered and recognizes the limitations of the proof-of-concept. Section 4.3 thereafter discusses related work. Finally, section 4.4 concludes by outlining future work directions.

4.1 Contribution

This master's thesis has explored the feasibility of a unified secure compiler infrastructure that allows security properties to be represented and manipulated in a generic way at the different layers of abstraction. The limited time frame of a master's thesis has narrowed down this exploration to the property of software module isolation but the principles should be applicable to other security properties as well.

More specifically, the following contributions are made:

- A proposal for a unified secure compiler infrastructure where security properties are represented and manipulated in a generic way at the different abstraction layers, independent from any source language and target architecture.
- A proof of concept of these ideas is implemented for the property of software module isolation that supports the C and Rust programming languages and the Sancus and Intel SGX security architectures. Supporting two different programming languages not only demonstrates the genericity of the solution but it also demonstrates that this approach leads to a seamless language interoperability between the supported languages.
- A proposal and a prototype for a new Sancus toolchain with an improved design that adheres to generally accepted software engineering principles. The new toolchain facilitates the development of Sancus-enabled applications with a smaller TCB. Furthermore, the generated machine code exhibits an improved runtime performance when compared to the legacy toolchain. The new toolchain is also an improvement in terms of user-friendliness as it results in improved compile times and it removes the need for a Sancus-specific linker.
- A generic programming model for software module isolation for the LLVM intermediate representation, that expresses the isolation properties independent from any source language and target architecture.
- A generic programming model for software module isolation for the C programming language, that expresses the isolation properties independent from any target architecture with an emphasis on programmability.
- The introduction of the Rust ecosystem to the Sancus security architecture. The legacy Sancus toolchain can only be used for the development of protected modules in C. Safe programming languages like Rust are an effective countermeasure against low-level attacks, a property that makes them complementary to PMAs.
- The foundational work on an alternative to the Baidu Rust SGX SDK for developing Rust SGX enclaves. This alternative approach provides more opportunities for using information from Rust's advanced type system.

The source code of the secure compiler infrastructure is publicly available at `https://github.com/hanswinderix/sllvm`.

4.2 Limitations and Challenges

This section discusses some challenges and limitations that were encountered during this endeavour.

First of all, the limited amount of time has certainly been a challenge. A master's thesis roughly corresponds to 500 hours of work, the equivalent of 13 man-weeks. This short time frame in combination with the vast amount of addressed technologies has prevented an in depth treatment of most of the subjects that are covered, an important limitation. Investigating the genericity of a unified secure compiler infrastructure requires an in breadth approach and involves a significant amount of study on a wide range of topics. Implementing the proof-of-concept of the secure compiler necessitates the reading and understanding of a considerable amount of source code, the embodiment of the different compilers, code generators and runtime libraries, some better documented than others.

Another challenge has been the ad-hoc design of the legacy Sancus toolchain. Originally, it had been planned to refactor the design and the implementation of the legacy toolchain to use it as the basis for the proof-of-concept implementation of the unified secure compiler infrastructure. Thereafter, support for more security properties, target architectures and programming languages could be added. When it became clear that MSP430 assembly code was intertwined with the IR, that there was no classical middle-end/back-end separation in the design, it was decided to start over and rewrite the Sancus toolchain from scratch. Although it meant that less security properties could be covered than originally planned, it eventually led to new ideas for a more performant, user-friendly and secure Sancus toolchain.

This brings us to the following limitation. The proof-of-concept implementation of the unified secure compiler infrastructure only covers a limited number of security properties, security architectures and programming languages. Furthermore, secure compilation of software module isolation to PMAs has been extensively investigated. Other security properties of high-level programming language abstractions and efficient mechanisms to enforce them have not been looked at. To really show its value as a generic and unified platform, more properties, architectures and languages have to be supported.

Another important limitation of the SLLVM prototype is that it does not provide a scheme to guarantee secure interrupts and secure multithreading. Multithreaded applications with isolated modules that can be interrupted at any time must be supported in order to be convincing as a secure compiler infrastructure. Incompatibilities between the threading models of high-level programming languages and low-level security architectures, as is the case with Rust and Intel SGX [21], have to be addressed.

4.3 Related Work

Vulnerability mitigation techniques have been devised as defences against very specific exploits. Stack canaries, for example, are deployed to detect attempts to smash the stack with the purpose of overwriting return addresses [24]. Address Space Layout Randomization (ASLR) as another example, is a technique that aims to mitigate attacks that depend on knowledge of an application's memory layout. The idea of ASLR is to randomise the memory layout in such a way that the different code and data segments are put at unpredictable addresses. ASLR has been implemented in commodity OSs such as Windows and Linux and has been the subject of scientific study [48], [54], [27], [13].

Since McCune et al. introduced the concept of a PMA in 2008 [36], more than a decade of research in this area has yielded a significant body of work with several prototypes from both academia and industry [36], [35], [53], [23], [39], [37], [51], [40]. Strackx et al. have demonstrated that PMAs can be realized at any layer of a computer system [52], from the lowest level, tightly integrated with the hardware architecture, over a hypervisor-based implementation to the OS kernel.

Maene et al. published an overview of hardware-based trusted computing architectures for isolation and attestation [34]. The survey compares twelve designs ranging from high-end architectures such as Intel SGX [37] to low-end embedded devices like Sancus [40], supporting different security properties.

Another class of security architectures that is promising as a target for secure compilers are so-called capability architectures, although systems like Hydra [49] and Cambridge CAP System [38] are not widely used. Capability systems have been around for a very long time. The term *capability* has been introduced in 1966 [20]. A capability is like an unforgeable key that is needed to be able to access a corresponding resource. It suffices to possess the key to gain access. Recent research has revived interest in capability architectures. CHERI [18], [59], [28] is a capability model that offers byte-granular memory protection facilities.

An extensive body of work that studies secure compilation schemes has been published [12], [13], [14], [42], [43], [55], although no research on a unified infrastructure seems to be available. Most research is focused on a single programming language and a single type of security architecture. Secure compilation techniques for PMAs are proposed by Agten et al. [14]. Tsampas et al. propose a secure compilation scheme for automatic compartmentalization of C programs on capability machines [55].

Secure compilation is not only about programming languages, compilers and system architectures. Its roots are closely entangled with the research area of fully abstract compilation. Formal specification and verification of the alleged security properties have always had a prominent role in this research area. The idea to formalize the compiler property of secure compilation with the notion of full abstraction was first proposed by Abadi [12] . SLLVM's striving for genericity and unification creates opportunities for reusability in this area as well. To give an example, available secure compilation proofs from SLLVM-C to SLLVM-IR can be reused when proving that the compilation from SLLVM-C to Sancus is secure.

4.4 Future Work

With this master's thesis, the research effort for a unified secure compiler infrastructure has just begun. SLLVM needs to be improved and extended in several areas before it is ready to be used a platform for secure compilation.

The proof-of-concept implementation of SLLVM only targets the Sancus and Intel SGX security architectures, two instances of a hardware-based PMA. Maene et al. published an overview of hardware-based trusted computing architectures for isolation and attestation [34] that are all potential target architectures for a secure compiler. Furthermore, hypervisor-based and kernel-based PMAs [52] and other types of security architectures such as capability machines like CHERI [59], have to be supported as well to really demonstrate the genericity of a unified compiler infrastructure.

Currently, SLLVM only supports a single security property, software module isolation. There are many others that have to be included in order to succeed and to be accepted as a common secure compiler infrastructure. Secure linking, sealing, secure communication and remote attestation are all examples of important properties that are required to guarantee the confidentiality and integrity of many applications that we as a society have come to depend on.

This master's thesis has shown with SLLVM-Rust that by targeting SLLVM-IR adding support for a new programming language immediately makes that language available for all the supported target architectures. To really succeed as a unified compiler infrastructure, SLLVM must support more programming languages. First of all, the ad-hoc Rust programming model should be replaced by a sound model that uses native Rust constructs. Rust already has a module system with visibility modifiers, so it seems logical to base the programming model for module isolation on that. Ada is a programming language that is used in projects where a software bug can lead to severe damage. It is used to develop safety-critical applications in industries such as space, defense, aerospace, railway transport and banking, where secure compilation techniques can definitely provide a valuable contribution to the overall safety of the applications. C++, as another example, is more and more used for developing embedded applications, the type of applications that Sancus was developed for. Modern C++ and more efficient implementations of the standard have made the language an attractive alternative to C.

The current prototype does not support secure interrupts and multithreaded applications, with isolated modules that can be interrupted an any time. To be able to securely support multiple threads on some architectures, the corresponding call stacks have to be separated and protected. Other architectures, like Intel SGX for example, are designed with multithreaded enclaves in mind which in turn can lead to incompatibilities with the threading models of some programming languages, as described by Ding et al. in [21].

Developing isolated modules may give a false sense of security. In an environment where the enclosing application and the isolated modules share the same address space, pointer parameters to entry functions have to be handled with care. Validation code has to be written to ensure that every memory address that is derived from a pointer parameter does not belong to one of the module's protected memory regions. It is possible to automate this process by generating the corresponding validation code. Van Ginkel et al. further illustrate this problem and report on a work-in-progress towards a solution for SGX [56], [57]. If such a solution can be generalized by porting it to a source and target independent representation such as SLLVM-IR, the additional defensive measures can also be reused in other contexts, such as the development of Sancus-enabled applications in Rust.

Finally, to be able to reason about the security properties and to formally prove the preservation of those properties after compilation, the programming models at the different abstraction layers have to be formalized in both their syntax and their semantics. Proving secure compilation requires a formalization of the different model-to-model transformations as well.

Bibliography

[1] Clang: A C language family frontend for LLVM. `https://clang.llvm.org`. Accessed: 2018-06-03.

[2] Extending LLVM: Adding instructions, intrinsics, types, etc. `https://llvm.org/docs/ExtendingLLVM.html`. Accessed: 2018-06-03.

[3] Intel Software Guard Extensions (Intel SGX) SDK. `https://software.intel.com/en-us/sgx-sdk`. Accessed: 2018-06-03.

[4] Intel Software Guard Extensions (Intel SGX) SDK - Documentation. `https://software.intel.com/en-us/sgx-sdk/documentation`. Accessed: 2018-06-03.

[5] The LLVM compiler infrastructure. `https://llvm.org`. Accessed: 2018-06-03.

[6] The LLVM compiler infrastructure - documentation. `https://llvm.org/docs`. Accessed: 2018-06-03.

[7] LLVM language reference manual. `https://llvm.org/docs/LangRef.html`. Accessed: 2018-06-03.

[8] The Rust programming language. `https://github.com/rust-lang/rust`. Accessed: 2018-06-03.

[9] The Rust programming language. `https://doc.rust-lang.org/book/first-edition/index.html`. Accessed: 2018-06-03.

[10] The sysroot manager that lets you build and customize ‘std‘. `https://github.com/japaric/xargo`. Accessed: 2018-06-03.

[11] ARM Security Technology - Building a Secure System using TrustZone Technology. `https://developer.arm.com/technologies/trustzone`, 2005.

[12] M. Abadi. *Protection in Programming-Language Translations*, pages 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[13] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012.

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

[15] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[16] N. Avonds. Implementation of a state-of-the-art security architecture in the linux kernel. Master's thesis, KU Leuven, 2013.

[17] A. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Achictecture of Open Source Applications. CreativeCommons, 2011.

[18] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130, New York, NY, USA, 2015. ACM.

[19] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[20] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, Mar. 1966.

[21] 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*, CCS '17, pages 2491–2493, New York, NY, USA, 2017. ACM.

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

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

[24] Ú. Erlingsson, Y. Younan, and F. Piessens. *Low-Level Software Security by Example*, pages 633–658. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[25] D. Gollmann. *Computer Security (3. ed.)*. Wiley, 2011.

[26] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[27] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 161–174, June 2011.

[28] Y. Juglaret, C. Hritcu, A. A. D. Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 45–60, June 2016.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. January 2018.

[30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[31] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[32] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. January 2018.

[34] P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, PP(99):1–1, 2017.

[35] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158, May 2010.

[36] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, Apr. 2008.

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

[38] R. M. Needham and R. D. Walker. The cambridge cap computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 1–10, New York, NY, USA, 1977. ACM.

[39] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., 2013. USENIX.

[40] J. Noorman, J. V. 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 Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017.

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

[42] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, Apr. 2015.

[43] M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 17–30, June 2016.

[44] B. Pierce. *Types and Programming Languages*. Types and Programming Languages. MIT Press, 2002.

[45] F. Piessens, D. Devriese, J. T. Mühlberg, and R. Strackx. Security guarantees for the execution infrastructure of software applications. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 81–87, Nov 2016.

[46] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975.

[47] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[48] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[49] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2014.

[50] R. Strackx. Efficient isolation of trusted subsystems in embedded systems. Springer, 2010.

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

[52] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. *Protected Software Module Architectures*, pages 241–251. Springer Fachmedien Wiesbaden, Wiesbaden, 2013.

[53] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[54] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[55] S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017*, pages 1–14, 2017.

[56] N. van Ginkel, R. Strackx, T. Mühlberg, and F. Piessens. Towards safe enclaves. In *4th Workshop on Hot Issues in Security Principles and Trust (HotSpot 2016)*, pages 1–16, 2016.

[57] N. van Ginkel, R. Strackx, and F. Piessens. Automatically generating secure wrappers for sgx enclaves from separation logic specifications. In *APLAS 2017: Programming Languages and Systems*, pages 105–123. Springer International Publishing, 2017.

[58] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security Privacy*, 10(5):68–70, Sept 2012.

[59] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[60] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

FACULTEIT WETENSCHAPPEN
Kasteelpark Arenberg 11 bus 2100
3001 HEVERLEE, België
tel. + 32 16 32 14 01
fax + 32 16 32 19 95
www.wet.kuleuven.be