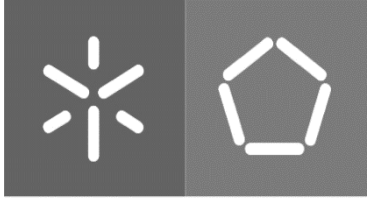
Universidade do Minho
Escola de Engenharia

José Pedro Silva Lopes

Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Runtime Security and Data Integrity

Dezembro de 2017



Universidade do Minho
Escola de Engenharia

José Pedro Silva Lopes

Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Runtime Security and Data Integrity

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Adriano Tavares
Professor Doutor Sandro Pinto

Dezembro de 2017

DECLARAÇÃO

Nome: José Pedro Silva Lopes
Endereço Eletrónico: lopes.joseps@gmail.com
Telefone: +351910162580
Bilhete de Identidade/Cartão do Cidadão: 14349380

Título da dissertação: Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Runtime Security and Data Integrity
Ano de conclusão: 2017

Orientador:
Professor Doutor Adriano Tavares

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos e Computadores
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, _____/_____/_____

Assinatura:

Abstract

One of the most popular cyber-attack vectors to compromise computer systems is related to memory corruption. Memory corruption is one of the most prevalent and devastating vulnerabilities. The widespread adoption of virtualization technology in embedded systems generally and naively accepts *Virtual Machine Manager (VMM)* or hypervisor software as the *Trusted Computing Base (TCB)*. As a software component, vulnerabilities can still be present, allowing attackers to subvert it alike *Operating Systems (OSs)*. Virtualization empowers mixed-criticality embedded systems by executing critical and non-critical tasks under the same hardware. Therefore, security and safety are critical in their design as attacks on real-time embedded systems software can put lives in danger and/or cause enormous financial losses.

Disregarding code-injection attacks, memory corruption exploits consist of: control- and non-control-data attacks. In practice, code-injection attacks are prevented with a W$\oplus$E policy which defines memory regions either as writable or executable, as *Memory Protection Unit (MMU)* hardware is now commonly available. Throughout this work, the focus is mainly on non-control-data attacks. Nevertheless, control-data attacks are also tackled with *Control-Flow Integrity (CFI)* enforcement.

This thesis uncovers a tailor-made security solution enforcing data integrity in the μRTZVisor VMM, according to a specification devised by the developer. The Zynq-7000 *System on Chip (SoC)* was leveraged to isolate a remote integrity monitor from the hypervisor, in a separate core. Through compile-time instrumentation, execution traces are collected, recording updates to critical static variables on μRTZVisor. The monitor audits these traces by searching for violations of data integrity rules, concurrently to hypervisor's execution.

Automating the deployment of the devised security mechanism is required to facilitate its adoption. Using ontologies for knowledge representation, information related to the security mechanism and the data aspect of the μRTZVisor software is modeled into a specifically designed meta-model. Ontologies uniformize knowledge representation and aid maintainability. By inserting the modeling efforts into the SeML modeling infrastructure, code generation capabilities are leveraged to generate implementation-specific files.

Keywords— Security, Data Integrity, Ontologies, Remote Monitor, Embedded, Memory Vulnerabilities, ARM, Meta-model, hypervisor, SeML, μRTZVisor, CFI

Resumo

Um dos mais populares vetores de ataque a comprometer os sistemas computacionais é a exploração de vulnerabilidades de corrupção de memória. Estas vulnerabilidades, além de bastante comuns, podem ter efeitos devastadores. A difusão da tecnologia de virtualização em sistemas embebidos assume, ingenuamente, o *software* VMM como pertencendo à TCB. No entanto, podem ainda existir vulnerabilidades, permitindo aos atacantes subverter os mecanismos de segurança. A virtualização permite a criação de sistemas embebidos de criticidade mista, executando funções criticas e não criticas na mesma plataforma. Por esta razão, a segurança é essencial na conceção dos mesmos. Ataques a sistemas embebidos críticos podem ter efeitos devastadores como a perda de vidas humanas e/ou enormes perdas económicas.

Não considerando ataques que injetam código novo no programa, existem duas maneiras de explorar vulnerabilidades de corrupção de memória: ataques a dados de controlo e aos restantes dados do programa. Na prática, ataques que inserem código novo são prevenidos com uma política de W$\oplus$E, em que segmentos da memória são classificados como executáveis ou passiveis de serem escritos. Essa política é aplicada pelo *hardware* MMU que se encontra atualmente presente numa vasta gama de sistemas. O foco deste trabalho inside principalmente em ataques não direcionados aos dados de controlo. No entanto, estes ataques também são considerados através da implementação de um mecanismo de CFI.

Esta tese propõe uma solução de segurança, especialmente concebida para o μRTZVisor, que providencia integridade de dados de acordo com uma especificação concebida pelo desenvolvedor. Utilizando o SoC Zynq-7000, o monitor é isolado num *core* diferente do utilizado pelo *software* de virtualização. Através da inserção de instrumentação em tempo de compilação, é efetuado um registo das operações de escrita em variáveis criticas do μRTZVisor. O monitor remoto audita o registo à procura de violações na especificação de integridade de dados providenciada.

A automação da aplicação do mecanismo de segurança proposto é necessária para facilitar a sua adoção. Utilizando ontologias como linguagem de representação de conhecimento, informação relacionada com os mecanismos de segurança e o plano de dados do *software* de virtualização são modelados num meta-modelo desenvolvido neste trabalho. A utilização de ontologias uniformiza a representação de conhecimento e a manutenção do mesmo. Através da inserção dos esforços de modelação na infraestrutura de modelação SeML, são ainda utilizados mecanismos de geração de código para gerar ficheiros de implementação.

Palavras-chave **—** Segurança, Integridade de Dados, Ontologias, Monitor, Sistemas Embebidos, Vulnerabilidades de Memória, ARM, Meta-modelo, hypervisor, SeML, μRTZVisor, CFI

Contents

List of Figures

List of Listings

Acronyms

2D-DFG	two-dimensional data-flow graph.
API	Application Programming Interface.
ASLR	Address Space Layout Randomization.
BE	Back End.
CFG	Control-Flow Graph.
CFI	Control-Flow Integrity.
COTS	Commercial Of The Shelf.
DFG	Data-Flow Graph.
DFI	Data-flow Integrity.
DIFT	Dynamic Information Flow Tracking.
DL	Description Logic.
DOP	Data Oriented Programming.
DoS	Denial of Service.
DSL	Domain Specific Language.
FE	Front End.
FIQ	Fast Interrupt Request.
GCC	Gnu Compiler Collection.
HASK-PP	High Assurance Security Kernels Protection Profile.
IDE	Integrated Development Environment.
IL	Intermediate Language.
IPC	Inter-Process Communication.
IRQ	Interrupt Request.
JOP	Jump Oriented Programming.
KB	Knowledge Base.
LHS	Left-Hand Side.
LOC	Lines of Code.

ME	Middle End.
MILS	Multiple Independent Levels of Security.
MMU	Memory Protection Unit.
NS	Non-Secure.
NX	non-executable.
OCM	On-chip Memory.
OS	Operating System.
OWL	Web Ontology Language.
PIE	Position Independent Executable.
RHS	Right-Hand Side.
ROP	Return Oriented Programming.
RTL	Register Transfer Language.
RTOS	Real-Time Operating System.
SCR	System Control Register.
SMC	Secure Monitor Call.
SMI	System Management Mode.
SoC	System on Chip.
SWRL	Sematic Web Rule Language.
TCB	Trusted Computing Base.
TOE	Target of Evaluation.
TSF	TOE Security Functionality.
VM	Virtual Machine.
VMI	Virtual Machine Introspection.
VMM	Virtual Machine Manager.
WIT	Write Integrity Testing.
WP	Write Protect.

CHAPTER 1

Introduction

Nowadays, safety critical functions are being increasingly performed by embedded systems, ranging from automotive controlling systems to IoT-enabled devices. Failures in applications in safety- and security-critical environments can often be disastrous, either from an economic or individual point-of-view. With embedded systems becoming pervasive in our key infrastructures (e.g., in creating smart homes and cars) there is a demand for devices to provide multiple functionalities and abilities to users [6]. With added features, there are new risks and threats that must be tackled. Typically, faults are caused by software errors and can be exploited by an intelligent adversary. Designing certified secure, industrial products is a demanding task. For example, regarding the Common Criteria standard [7], high assurance levels translate to increased design and verification costs and architectural adjustments. The highest assurance levels (i.e., *Evaluation Assurance Level (EAL)* 6 and EAL 7) enforce formal verification, most valuable for highly critical systems. For example, seL4 [8] depicts a formally verified OS that uses a formal specification to ensure implementation's correctness. Mixed-criticality systems - not demanding of such assurance levels - can relax formal verification, introducing a runtime monitor to audit potentially security-critical events.

Multiple Independent Levels of Security (MILS) is an evolving component-based architecture intended for high-assurance environments, fully reliant in a separation kernel (e.g., VMMs) [9]. It considers a product the *Target of Evaluation (TOE)* and the separation kernel the *TOE Security Functionality (TSF)*. TSF is defined as the hardware or software that implements the security functional requirements for a system. The TSF is the only controlling entity a MILS-compliant system. A Common Criteria profile (e.g., Euro-MILS and *High Assurance Security Kernels Protection Profile (HASK-PP)*) specifies guidelines for the design of secure and trustworthy applications for critical environments. For example, HASK-PP states that the security kernel must be capable of generating information to verify its integrity (evidence of integrity), which can be provided by a runtime monitor. High-assurance systems may require formal and static verification for these components to achieve higher assurance levels (EAL 7); however, some system classes can afford to be less compliant to these standards, acquiring increased flexibility and higher security with an isolated runtime integrity monitor.

Attacks towards all internet-connected computer systems are often performed by exploiting memory corruption vulnerabilities in software (e.g., buffer overflow, double free). Aiming to control program's execution, attackers ambition is to modify program's control-related structures (e.g., stack return address) with spurious data. Several countermeasures, tackling this type of attacks, have already been proposed due to their high expressiveness. When control-flow protection techniques shut attackers down, they will be incentivized to pursuit and employ non-control-data attacks [12]. These attacks, although less expressive, cannot be neglected, as demonstrated in [13] with the creation of Turing-complete attacks. Frequently, countermeasures designed for general-purpose systems do not shift well to the embedded domain, since these systems have stricter requirements (e.g., determinism) and present limited or constrained hardware resources.

The widespread adoption of virtualization technology, in embedded systems, prompted a new wave of research [14, 15, 16, 17, 18, 19], due to its unique security advantages in isolating commodity OSs as virtual machines. The fundamental assumption is that the hypervisor is a trustworthy component, included in the TCB. Security engineers cannot assume there are no mechanisms by which system's security and integrity can be compromised, even in the present of correctly written software [8]. Recursive security mechanisms create successive protection levels, limiting an attacker in its malicious actions. A hypervisor must adopt these mechanisms to provide its expected trustworthiness.

Altogether, security must be seen as a new dimension that embedded system designers should consider throughout the design process [20]. Cyber-security must be integrated into systems as it is impractical and too expressive to secure a system after design, as advised by the US Department of Defense [21]. A paradigm-shift from reactive to proactive security is fundamental. Engineers must tackle security throughout system development, specially due to increasing systems' complexity and connectivity. Considering that all employed security countermeasures can be made ineffective against one simple security flaw, nothing can be overlooked. This work is embodied in a "secure by design" approach to secure the μRTZVisor hypervisor [22]. Even with a small codebase, VMM software can contain vulnerabilities which can be detected statically or, when static analysis does not suffice, at runtime.

This work proposes a Data Integrity mechanism for real-time embedded systems. The main efforts are towards securing the virtualization layer software, namely, the in-house developed μRTZVisor hypervisor, deployed on a TrustZone-based platform. Common wisdom dictates that to secure an application, monitoring software must run a layer below the application. The VMM provides the lowest level in the software stack thus, this principle is not applicable here. This thesis introduces a monitor, executing with the same privilege as the hypervisor, that enforces integrity to its sensitive data structures. While the ideal solution would be complete memory safety [23], which prevents memory errors in the first place, existing solutions are not adequate for em-

bedded systems. For example, Cyclone [3] and CCured [24] are C type-safe dialects that provide memory safety but endure in high performance overhead and nondeterminism. Furthermore, this work automates, to a certain extent, the deployment of the Data Integrity countermeasure using semantic technology.

Nowadays, design automation is pivotal to reduce time-to-market. Evermore, creating new embedded systems - specially IoT-based - requires a consistent and increasing effort from a development team, ranging from hardware to cloud-based designs [25]. Model-based design is recognized has a key technology to improve productivity as it highlights important information and helps to manage complexity. Semantic technology, particularly ontologies, provides a modeling framework extended with *Description Logic (DL)*, allowing to create an artificial intelligence-assisted system design platform, performing adequate design choices according to specific metrics and requirements.

1.1 Thesis Contributions

In summary, this thesis makes the following technical contributions:

- **Novel Threats Highlighting** – State of the art data-oriented attacks are described as a method to identify vulnerable software and the adequacy of the proposed security mechanism.
- **Specialized Monitoring Techniques** – This thesis presents a runtime monitor to ensure data-plane integrity of the μRTZVisor VMM. This work is tailored to the specific requirements of this VMM.
- **Isolation Techniques** - Several isolation techniques are enforced to effectively protect critical memory content of the security mechanism.
- **New Program Analysis and Instrumentation Techniques** – At compile-time, the program is analyzed for possible sources of data-oriented attacks. Then, instrumentation is inserted to provide data for the runtime monitor.
- **Automation Techniques** – A meta-model and respective model is devised to facilitate the deployment of the security mechanism. The key idea is to automate through model-driven software programming.

1.2 Contextualization

Introducing secure engineering practices earlier in the product development cycle can reduce costs, as found in [26], [27] and [28]. Until recently, vendor's response to attacks was to provide patches to fix the most recent vulnerabilities. Nevertheless, patches are not the ideal solution [29] and a different paradigm towards security must be embraced. The solution is to develop secure software from the beginning, enforcing security during the whole product's life cycle. Developers' paradigm must shift from implementing the highest number of functionalities before the deadline to a "secure by design" paradigm [30].

Real-time high-assurance systems promote four properties: security, safety, real-time and fault-tolerance. This work aims to increase the security level of the μRTZVisor VMM, which seeks to be compliant with the Common Criteria standard. This thesis focus on ensuring the integrity of μRTZVisor (*MILS Separation Kernel* in Fig. 1) which, in turn, provides system's spacial and temporal resource separation supporting the aforementioned properties. Promoting both data and execution flow integrity in μRTZVisor increases its assurance, which brings the hypervisor closer to the desired EAL, according to the Common Criteria standard: EAL 4.

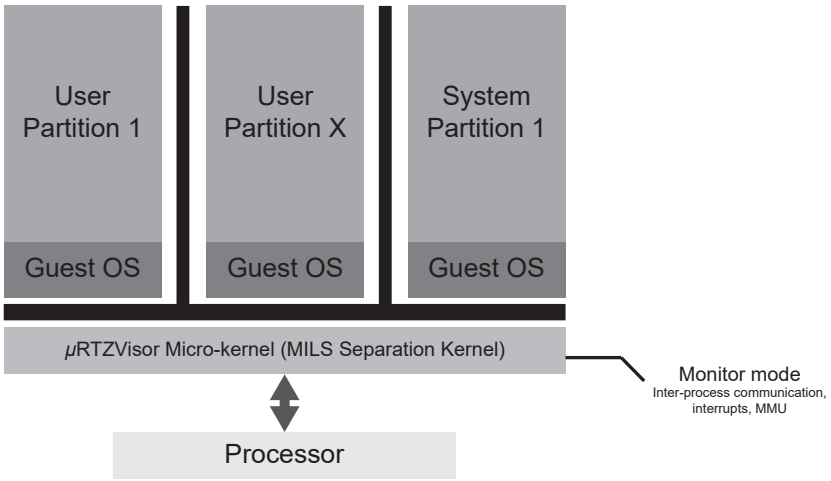


Fig. 1: MILS architecture. MILS architecture with μRTZVisor acting as the Separation Kernel and executing in the ARM's Monitor mode.

This thesis is part of a major project to design a secure hypervisor. The project comprises of several small projects aiming to satisfy hypervior's strict security requirements. In this "security by design" approach, the VMM software is being designed following a security-oriented architecture and coding guidelines. Another thesis is focusing on providing a secure boot mechanism. As this thesis focuses on runtime defenses for the data-plane of the VMM code, a similar thesis focuses on protecting the control-plane. The latter work will be integrated with this work

to provide complete data- and control-plane protection to the software. Ultimately, all works are integrated using Model-driven software development [31], using ontologies for knowledge representation. A modeling framework, SeML, is also being designed as part of this project. Using ontologies, models can be reasoned upon to verify if required security properties are met by design and checked for consistency [32].

1.3 Thesis Outline

The remaining of this thesis is structured as follows. Chapter 2 provides background information about virtualization technology, the ARM architecture, GCC and semantic technology. Chapter 3 discusses the evolution of attack and countermeasure design. This chapter provides a special focus on non-control-data attacks. Chapter 4 introduces the work, from other projects, that is used by this thesis. Chapter 5 describes the proposed implementation to provide data integrity to the μRTZVisor hypervisor, against exploitations of memory corruption vulnerabilities. Chapter 6 delves into the ontological modeling efforts. Chapter 7 denotes the security analysis and performance evaluation performed on the proposed countermeasure. Primarily, it discusses the effectiveness of the proposed solution in terms of security, evaluating its own security and resilience facing state of the art attacks. Chapter 8 summarizes the contributions of this thesis, discussing open problems for future work.

CHAPTER 2

Background

Four pivotal concepts are introduced in this chapter. Firstly, virtualization is discussed as it is one of the pillars of this work. In chapter 5, the Data Integrity security mechanism is introduced, which aims to protect the µRTZVisor virtualization software against the exploitation of memory-related vulnerabilities. An overview of the ARM architecture and TrustZone is provided since µRTZVisor is implemented by leveraging specific ARM technologies. Next, the internals of the GCC compiler are discussed since this thesis takes advantage of GCC to insert instrumentation. Finally, ontologies are examined as they provide the backbone for the modeling part of this work.

2.1 Virtualization

At the end of the 1960s, virtualization emerged as a software-abstraction layer [33]. Platform virtualization partitions a hardware platform into one or more *Virtual Machines (VMs)*, supporting the execution of multiple OSs. Advances in computer architecture now enable the transposition of virtualization technology, from its commonplace in server computing, onto the embedded domain [34]. Embedded systems are highly heterogeneous and commonly combine different OS categories. In mixed-criticality systems, a *Real-Time Operating System (RTOS)* is usually used for real-time execution of critical tasks, while a feature-rich OS carries the execution of complex applications (e.g., user interfaces). Nowadays, the automotive industry is using virtualization to integrate real-time control with infotainment environments [35, 36, 37]. Likewise, the aeronautics and aerospace industries are also using virtualization to provide isolation for safety-critical components [38, 39].

Embedded virtualization solutions follow two different approaches: full-virtualization and paravirtualization. The former presents a higher performance cost, while the later presents a higher design cost [40]. Using full-virtualization, guest OSs are unaware of the underlying hypervisor. Without an interface for VM-VMM communication, the implicit guest privilege reduction works by trapping the execution of sensitive instructions to the hypervisor thus, guest VMs

can execute unaltered. In paravirtualization the running guest is modified to be VMM-aware and operate in the virtualized environment [41]. With this technology, the VMM is rather simple, closing the performance gap between virtualized and non-virtualized hardware. Here, the created virtual environment communicates with the underlying hardware via high-level hypercalls (similar to an OS system call), enabling non-virtualizable instructions to be executed directly, achieving near native performance [42]. Furthermore, hypercalls are a communication channel between a hypervisor and the respective guests, allowing the VMM software to provide higher-level functionalities to its guests. Alike OSs' system calls, hypercalls can possess vulnerabilities, allowing well-known attacks such as privilege escalation.

One of the main drivers to implement the VMM as a microkernel is to minimize the TCB. Generally, microkernel implementations obey to the Liedtke's Minimality Principle: A concept is tolerated inside the microkernel only if moving it outside the kernel (i.e., permitting competing implementations) would prevent implementation of the system's required functionality [43]. A reduced TCB is desired when advocating for safe and secure systems, as a smaller program is easier to be assured or verified for correctness [44]. To reduce the TCB, components not affecting system's functionality are extracted from the VMM and considered untrusted. Usually, such components are moved into high privileged VMs, which then interact with untrusted system components via memory sharing or *Inter-Process Communication (IPC)* mechanisms. A reference architecture of a microkernel-like VMM is depicted in Fig. 2, where the fundamental components of a system belong to the VMM while hardware drivers are positioned in the dedicated *Driver Server*.

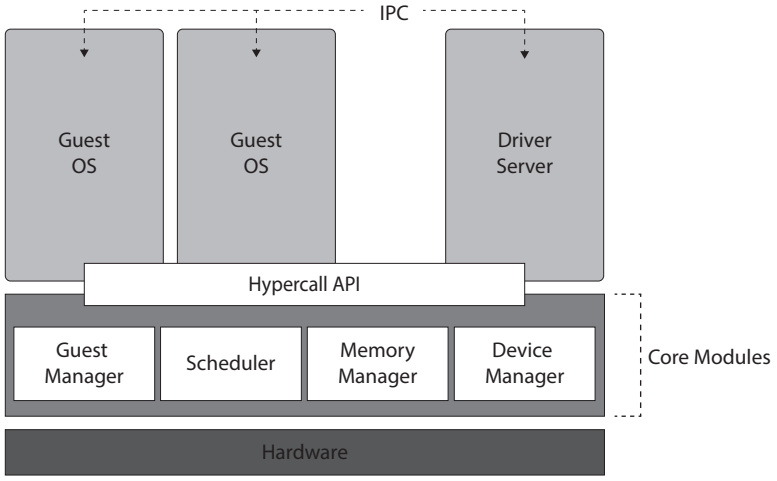


Fig. 2: Reference architecture of a microkernel-based hypervisor. A microkernel reference architecture using Liedtke's Minimality Principle. Drivers are considered another VM, being non-essential. VMs communicate with the hypervisor via the Hypercall *Application Programming Interface (API)* and between themselves using an IPC mechanism.

2.2 ARM Architecture

An understanding of the ARM architecture is essential to this work. The proposed Data Integrity security mechanism was specifically designed to be deployed in an ARM processor. Furthermore, by understanding part of the ARM Security Extensions, namely, TrustZone technology, the reasons behind certain design decisions can become clearer. μRTZVisor is also a TrustZone-enabled VMM. The focus of this chapter is to understand the implications of an ARM processor equipped with TrustZone technology.

2.2.1 Processing Modes

Before introducing the Security Extensions, which include TrustZone, and the Virtualization Extensions, the ARM architecture [45] contemplated seven processor modes: User, System, Supervisor, *Interrupt Request (IRQ)*, *Fast Interrupt Request (FIQ)*, Abort, and Undefined. User mode is the only non-privileged user mode. Privileged processor modes grant the ability to perform tasks unavailable to User mode (e.g. MMU configuration). With TrustZone, two security states were introduced, for the processor, that work independently of the mode. Also, this technology introduced a new processor mode called Monitor - discussed below - to interface between a secure and non-secure processor state. This distinction between the two states is completely orthogonal to the privilege level of the execution mode.

The ARM architecture specifies sixteen 32-bit general-purpose registers (R0-R15), among other status registers. From these set, some registers have special purposes. R13 is the stack pointer, R14 the link register and R15 the program counter. R15 is available to the programmer, and an explicit write to this register will alter program flow. ARM implements register banking between processor modes. With this technique the same register points to a different physical storage location, depending on the current processor mode. For example, there is a copy of the stack pointer and link register for most processor modes. As stated in [46], processor modes refer to the various ways that the processor creates an operating environment for itself.

After reset and with the Security Extensions, the processor always boots in the privileged Supervisor mode in the Secure world. The start-up sequence must perform the required initialization operations for the system such as stack initialization for each processor mode and definition of secure and non-secure system resources.

2.2.2 TrustZone

This work presents a security mechanism that provides data integrity to a TrustZone-assisted hypervisor. TrustZone hardware architecture aims to provide resources that enable a system designer to build secure systems [47]. This technology divides device's hard and software resources into two different execution domains: the secure world for the security subsystem and the non-secure world for everything else. A secure system design is expected to place all sensitive resources in the secure world, while also executing robust software.

TrustZone adds an additional *Non-Secure (NS)* bit to every memory transaction. Considering this as an extra bit in a 32-bit processor, effectively, two distinct secure and non-secure 32-bit address spaces are created for each world. To switch context between both domains a special core mode called Monitor mode was introduced, as depicted in Fig. 3. When the processor executes in this mode, it is always in the secure state. The software in this mode is implementation specific. Normally, it saves the state of a world and restores the state of the next world to be executed. The separation of these domains creates two virtual cores when, in reality, both worlds are executing in the same core in a time-sliced manner. Entering Monitor mode, from the non-secure world, is only possible through the dedicated instruction, the *Secure Monitor Call (SMC)*, or hardware exception mechanisms [47]. Further, exception sources, such as interrupts, can be configured to be handled in monitor mode.

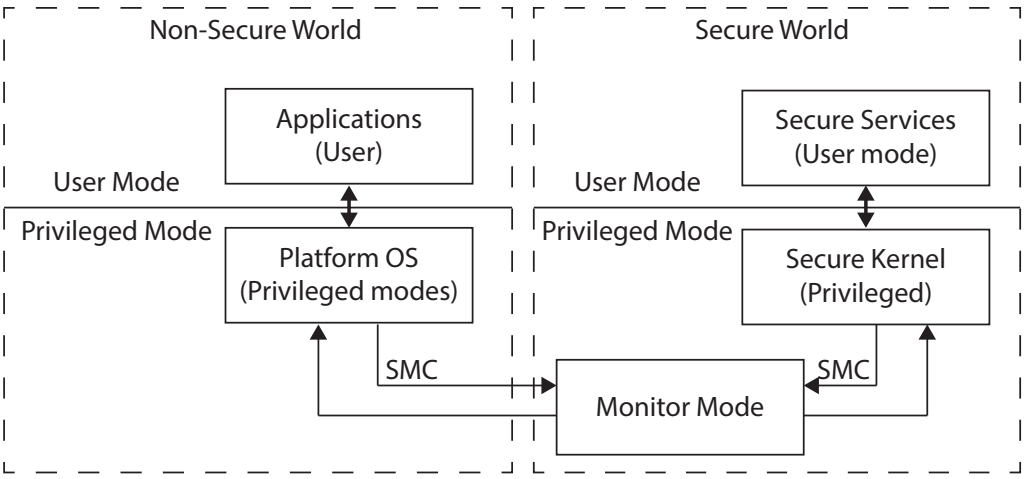


Fig. 3: ARM core with Security Extensions (Adapted from [1]). ARM Security Extensions insert a new mode (Monitor mode) which transfers execution between the Normal and Secure World. Transitions between both worlds are always interceded by the code executing in this mode. Processors modes work independently of the Non-Secure and Secure worlds.

In TrustZone-enabled processors, there are two MMUs interfaces, one for each virtual core. This ensures that secure page tables are made confidential and are protected from the non-secure

world. Essentially, page table entries, mapping to physical memory, include a NS-bit used to determine if the access is made to the non-secure or secure physical memory. When in the non-secure world, the value of this descriptor is forced, by hardware, to 1. This avoids secure accesses in the non-secure world. On the other hand, the secure world can access either secure or non-secure memory.

Processor's state can be modified by enabling the NS bit of the *System Control Register (SCR)*, using the system control co-processor interface (CP15). This interface consists of dedicated instructions to access the co-processor registers. As aforementioned, in Monitor mode, the core always executes in the secure world, disregarding the NS-bit. When in Monitor mode, the processor is highly-privileged as it can access and modify memory and registers of either world [47].

2.3 GNU Compiler Collection Overview

The GNU Compiler Collection is amongst the most popular compilers. It includes multiple front-ends and support for C, C++, Fortran, Ada and other programming languages. Being a free and open software, documentation on its inner-workings is highly available. GCC has three main components: the *Front End (FE)*, the *Middle End (ME)* and the *Back End (BE)*. The internals of GCC are depicted in Fig 4. Each component is characterized by its *Intermediate Language (IL)*, whose abstraction lowers as the program travels through the three previously mentioned components. The Front End is characterized by its GENERIC IL. The purpose of GENERIC is simply to provide a language-independent way to represent functions in tree data structures. While some optimizations can be performed at this level, most are postponed to the next IL, due to its restricted expressiveness and similarities to machine code (i.e., three-operand expressions). The GENERIC IL is created by the respective programming language Front End. The Middle End encompasses the GIMPLE IL. The compiler optimizer/pass which converts GENERIC into GIMPLE is called the gimplifier. This unit recursively generates GIMPLE tuples from GENERIC expressions. Most optimizations are performed in this stage. The Back End comprises the *Register Transfer Language (RTL)* IL. The last one before the code generation (i.e., translation to assembly). RTL is almost human unreadable and some machine specific optimizations are performed upon it (e.g., register allocation).

GCC manages its optimizers through the *Pass Manager*. This unit decides the position in the pipeline where each pass runs. Calling GCC with different optimization flags essentially enables and disables certain passes on the Pass Manager.

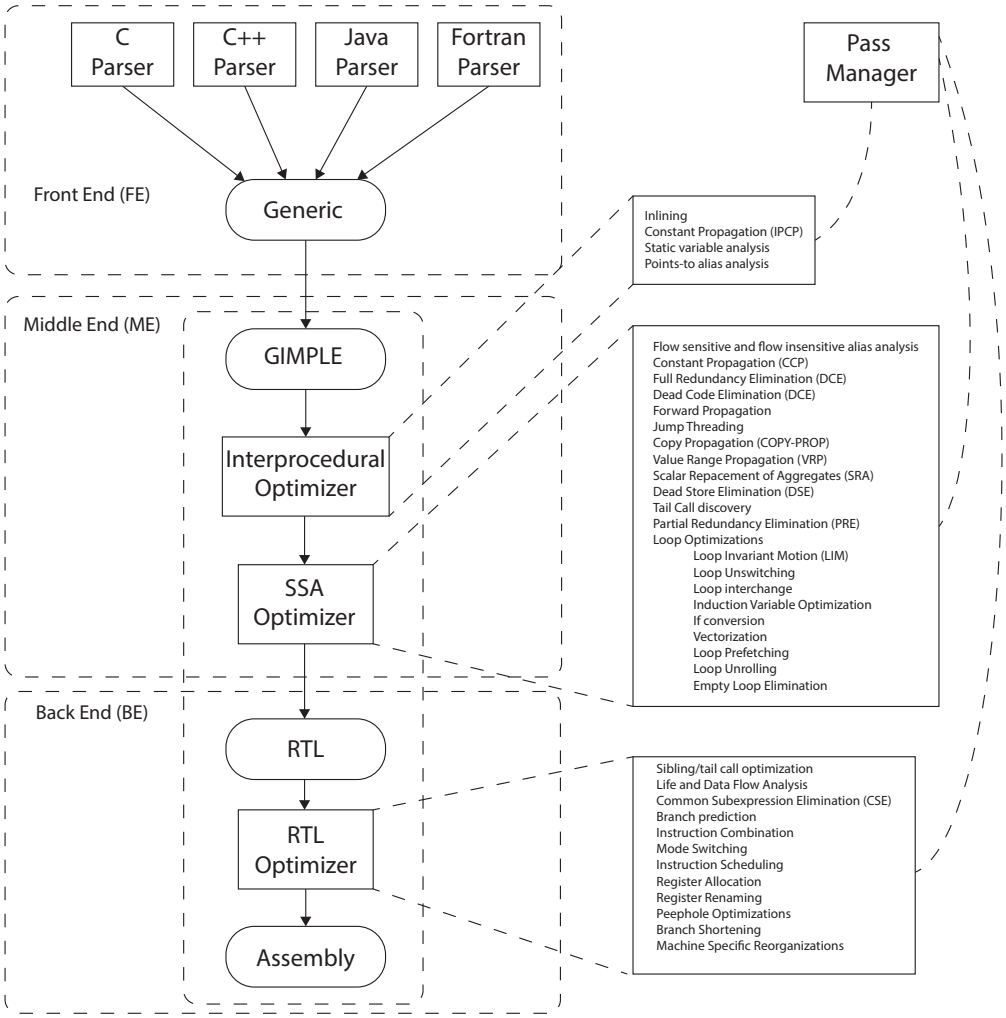


Fig. 4: GCC internals overview (Reproduced from [2]). A scheme depicting the pipelined structure of GCC. Each square, after GIMPLE, performs a series of optimizations on the code. Most optimizations are executed upon GIMPLE expressions in the SSA form [48]. The optimizations on the right side of the figure can be manually enabled/disabled with the Pass Manager. RTL is used to describe data flow at the register-transfer level of an architecture thus, certain optimizations can only be applied upon it.

2.4 GIMPLE

GIMPLE IL is a simplification and a less expressive subset of the GENERIC IL, mostly used for optimizations. It is based on the SIMPLE IL used in the McCAT compiler project [49]. The most conspicuous feature is its three-operand representation, while using temporary variables to store intermediate values. This work focusses on GIMPLE's load-store operations, albeit its extensive instruction set. Specifically, indirect and direct store operations to static variables in C++. Thus, for a more comprehensive view, the GCC Internals document shall be consulted [50]. GIMPLE was chosen since 1) it offers most flexibility for manipulation and 2) most significant optimizations occur in this IL. The latter point is of uttermost importance because an abstract view of all load-store operations, which will, in fact, be executed on the target is provided in GIMPLE. GENERIC IL and C++ are far to abstract to detect load-store operations occurring directly on the target platform.

Every GIMPLE instruction, in a block of instructions, which assigns an expression to the *Left-Hand Side (LHS)* (creating an object that persists beyond a single expression) is called a GIMPLE assign, and it is represented in GCC by a tree node of type GIMPLE_ASSIGN. Every store operation follows this template:

LHS = RHS1 EXPRCODE RHS2

By scrutinizing the LHS operand of an assignment instruction it is possible to infer the destination of the store operation. The type of the LHS operand allows to detect assignments to C++ static variables, since these operands map to GENERIC IL nodes containing more high-level information. In this work, the focus is on four LHS operand types: Variable declaration (VAR_DECL), Memory Reference (MEM_REF), Component Reference (COMPONENT_REF) and Array Reference (ARRAY_REF). As their name suggests, Component References refer to references to class members. In the source code, a reference to a class member or structure field creates a Component Reference type node, during compilation. Logically, accesses to arrays, belonging to classes and structures or not, generate Array References. These references are chained interchangeably to represent complex accesses to variables (e.g., accessing members of classes when themselves are members of other classes). Memory References are created to represent indirect accesses to variables. Inspecting the address used by a Memory Reference it is possible to infer the type of variable being accessed and the variable itself. A Variable Declaration denotes an access to a static variable. When accesses to static variables with a class type are performed, they normally consist of a chain of Array and Component References ending in a Variable Declaration. If the access is indirect, a Memory Reference is used.

2.5 Semantic Technology: Ontologies

A variety of computer science fields, information integration, and the semantic web recognize ontologies as a promising technology for knowledge sharing and reuse [51]. An ontology is an explicit specification of a conceptualization of a domain of interest [52]. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. Every knowledge base, knowledge-based system, or knowledge-level agent is committed to some conceptualization, explicitly or implicitly [52]. A conceptualization must accurately capture the world for the purpose of the knowledge representation. Ontologies seek to represent a reality, in such a manner that different persons understand its terms; thus, learning about entities according to concepts they represent as well as contexts in which they are used. They consist of concepts connected with semantically rich and distinct relationships. The goal of this subchapter is to provide a quick overview of several ontology-related concepts

Ontologies are mostly used for the purpose of reasoning. Reasoning is associated with the process of reaching conclusions. The axioms, present in a *Knowledge Base (KB)*, constitute explicit knowledge, but implicit knowledge can be derived through reasoning. Implicit knowledge logically follows what was stated explicitly [53]. Axioms are assertions, in a logical form, that define concepts on an ontology. Reasoning with ontologies is mainly divided in two categories: the verification of an ontology and the deduction of new axioms. In the former, reasoning is used to detect any contradicting information present in the ontology. Assuming an adequately defined ontology, the process of deduction derives logically correct implicit conclusions or knowledge.

2.5.1 Essential Features of an Ontology

Generally, ontologies are visualized and thought of as semantic networks [54]. A semantic network is a structure for representing knowledge as a pattern of interconnected nodes and arcs. Nodes in the net represent concepts of entities while arcs describe relationships that hold between concept nodes. Labels on the arcs specify relation types. A portion of a conceptual model about digital components is depicted in Fig. 5. ”SoCs and memories are different digital components” and ”Zynq-7000 is a SoC” is knowledge that can be intuitively inferred from this network.

There are several ontology languages with different semantics and constructs for ontology’s formalizations. Nevertheless, some essential characteristics, extracted from [53], common to many ontologies, can be identified using this semantic network. Further, the presented concepts will be connected to the *Web Ontology Language (OWL)* ontology language. By not delving into formalities, this chapter yields enough ontology knowledge without overwhelming the reader.

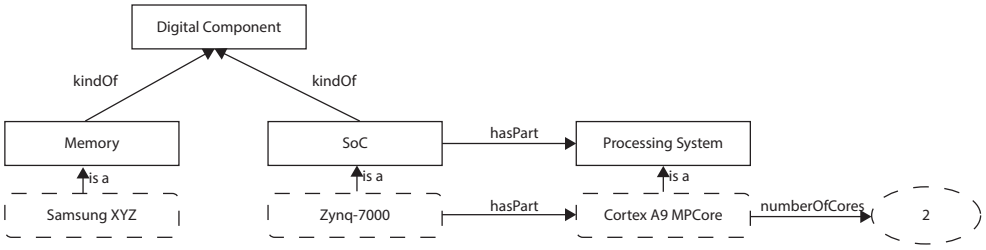


Fig. 5: An example of ontology modeling to represent digital components. In this conceptualization, the Digital Component concept is specialized into two other concepts using the kindOf relation. The ”is a” relationship constitutes instantiation and hasPart aggregation. The dashed rectangle individuals represent a concrete reality captured by this conceptualization.

Some essential characteristics of ontologies are:

- **Interrelation** – Interconnecting semantic network's concepts, through expressive relations between them, enriches the network with extra knowledge essential to create the desired conceptualization. Without relations, the network would be a loose collection of concepts, not portraying all the required knowledge.

- **Instantiation** - Instantiation provides a distinction between concrete objects and abstract categories (i.e., classes) that group akin objects. This mechanism is found across many models (e.g., C++ Object Model [55]). Through this mechanism individual objects are assigned to classes as instances. For example, ”Zynq-7000” is an instance of the *SoC* class.

- **Subsumption** – It can be defined as a specific type of interrelation that reflects generalization/specialization. Taxonomies are a type of KB based on subsumption relationships between terms [54]. In the above example, *Memory* and *SoC* are defined as a *kindOf Digital Component* thus, by being a specialization, they inherit all properties of this concept.

- **Axiomatization** – Usually, interrelated conceptual nodes do not include adequate expressiveness to express rich knowledge. While instantiation and subsumption are simple forms of axiomatization, it is possible and often necessary to create complex statements about the domain of interest [53]. These complex statements are often defined using logic equations and they lack a visualization paradigm.

- **Attribution** – Ontology languages (e.g., OWL) commonly support the definition of statements about datatypes and their values. An attribute is an inherent characteristic of a concept, being defined as a string, integer or other datatypes. In the example, the *Cortex-A9 MPCore* processing system is attributed a number of cores: 2.

- **Exclusion** – Prevents two general conceptual nodes from overlapping. This sophisticated knowledge representation presents itself in the form of class exclusion. In the example, a *Memory* object cannot be a *SoC* and vice-versa.

2.6 Ontologies Languages and Tools

W3C standardization efforts resulted in the OWL. This language allows modelers to express detailed constraints between classes, entities, and properties. OWL was adopted as a recommendation by the W3C in 2003 [56]. OWL is a computational logic-based language. Knowledge represented in OWL can be verified for consistency. Also, explicit knowledge can be leveraged to deduce implicit insights, using computer programs.

As above-mentioned, classes are the main building blocks of an OWL ontology. An empty ontology contains one class called Thing. OWL classes are interpreted as sets of individuals (or abstract categories). The class Thing is the set of all individuals. Consequently, every OWL class is a subclass of Thing [57]. Applying the exclusion feature in OWL is equivalent to defining classes as disjoint, so that two classes have no individuals in common. Classes can be made disjoint using the built-in OWL axiom `owl:disjointWith` between two class descriptions.

Protégé is the *de facto Integrated Development Environment (IDE)* for ontology editing. It is a free, open-source and sophisticated Java ontology editor and knowledge-base framework [53]. It has played an important role in the popularization of OWL. Protégé uses reasoning to support the development and maintenance process of ontologies (e.g., checking for inconsistencies) inferring new knowledge and answering queries [58]. Protégé supports multiple DL-based OWL reasoners (e.g., Pellet [59]).

2.6.1 OWL Properties

OWL properties represent relationships and are of three main types: Object, Data and Annotation properties. Object properties link two individuals through a relationship. Data properties describe relationships between an individual and data values. The latter are used to add meta-information to classes, individuals and object/data properties [60]. If the above semantic-network was translated to OWL, *hasPart* and *numberOfCores* would be an object and data property, respectively. It is also possible to create sub-properties (i.e., inheritance of properties) of both object and data properties. However, data and object properties cannot be mixed (i.e., a data property cannot be a sub property of a object property).

OWL Object Properties

OWL 2 supports two kinds of object property expressions. Each object property may have a corresponding inverse property. If a property p links individual a to b then its inverse will link individual b to a. In Fig. 5, isPartOf could be the inverse property of *hasPart*. Object properties can also be enriched with properties (i.e. meta-properties):

- **Functional** – For a given individual, there can be at most one individual that is related to the individual via the property (e.g., a car can only have one motor.). Likewise, an inverse object property can be defined as functional.
- **Transitive** – If a property P relates individual a to individual b, and also individual b to individual c, then individual a is related to individual c via property P.
- **Symmetric** – If a property P relates individual a to individual b then individual b is also related to individual a via property P. An object property can also be defined as asymmetric, meaning individual b cannot be related to individual a by property P.
- **Reflexive** – A property P is reflexive when it relates an individual a to itself. An object property can also be irreflexive disallowing such a relation from an individual to itself.

Object properties link individuals from the domain to individuals from the range. If the domain and range are not specified, both point to the Thing class (i.e. the set containing all classes). OWL domains and ranges should be seen as axioms used for reasoning to deduce new knowledge and not as constraints that must be verified.

OWL Data Properties

As previously stated, data properties describe relationships between individuals and data values of an individual data type. Data types are entities that refer to a set of data values. Furthermore, data types are comparable to classes. While classes contain individuals, the former contains data values such as strings and numbers. Datatypes are a kind of data range, which allows them to be used in restrictions [61]. The functional property, which assures that only one functional data property links a single individual to a data value, is the only property translating from the previous property type. Data properties can also be specified as disjoint, assuring that an individual does not possess two disjoint data properties.

OWL Annotation Properties

OWL allows classes, individuals, data/object properties and the ontology itself to be annotated with information or meta-data. This information generally consists of comments or references to resources such as websites, etc. As annotation properties are not used in property axioms, sub-properties and domain/range constraints cannot be defined for them. Mostly, annotation properties are filled with data literals (e.g., the string ”hello”). As stated in the OWL 2 Direct Semantics [62]: OWL 2 allows ontologies, individuals and axioms to be annotated. However, these types of annotation have no semantic meaning in OWL 2.

2.6.2 SWRL

Sematic Web Rule Language (SWRL) [63] provides a high-level abstract syntax enhancing OWL's, allowing to define axioms with an antecedent and a consequent. SWRL aims to be the standard rule language of the Semantic Web. Whenever conditions in the antecedent hold, whatever is in the consequent must hold true as well. These two fields consist of zero or more atoms. Multiple atoms are considered conjunctions. SWRL provides many built-in atoms [64] to perform the most basic logical assertions about ontological elements. The Protégé IDE also provides support for editing and executing SWRL rules. In equation 1, the rule states that if a motorized vehicle has two wheels it is, without doubt, a motorcycle as well. The ”?” symbol denotes variables and the ”$\rightarrow$” separates the antecedent from the consequent. The ”swrlb:equal” is a SWRL built-in to check for equality while the other elements of the rule belong to the ontology.

$$\begin{aligned} MotorizedVehicle(?p) \wedge hasWheels(?p, ?wheelnumber) \wedge \\ swrlb : equal(?wheelnumber, 2) \rightarrow Motorcycle(?p) \end{aligned} \tag{1}$$

CHAPTER 3

State of the art

This chapter begins examining the evolution of memory-related attacks and countermeasure design, throughout history. Subsequently, more recent security techniques are discussed, beginning with solutions aiming at complete memory safety (e.g., safe programming languages). As these solutions show inadequacy for the embedded domain, other countermeasures providing partial memory safety are introduced. At the end of this chapter, non-control data attacks are throughly discussed, providing powerful examples to clarify their expressiveness.

3.1 Memory Errors

Memory errors in C and C++ programs are among the oldest classes of software vulnerabilities [65]. Being designed with a performance over security rationale, C and C++ provide several security debilities (e.g., weak type system, pointer arithmetic). While several more secure languages exist, they rarely are an option in the embedded system domain due to timing and resource constrains. Poorly written code in unsafe languages is the main cause of memory errors. Writing unexploitable code, in the aforementioned languages, has performance implications, among others, that must be considered.

Despite decades of research, memory errors still subvert system's security and are prevalent among newly discovered vulnerabilities. Several countermeasures have been developed which either focus on a specific or generic attacks. While these solutions are effective in protecting against some or all memory flaws, they usually are not widely adopted or not well suited for the real-time and performance constraints of embedded systems. In search for the ideal solution to protect the data plane of μRTZVisor, it is important to review countermeasure design on memory errors and attack evolution. This chapter tries to depict the evolution and state of the art on memory-oriented attacks and corresponding countermeasures.

3.1.1 Evolution of memory related attacks throughout history

In the end of the twentieth century, stack smashing and buffer overflow attacks were publicly disclosed [66] [67], creating awareness for stack protection countermeasures by the security community. Stack-based buffer overflows are probably the best-known memory error vulnerabilities. They occur when a stack buffer overflows and overwrites adjacent memory regions in the stack [65] and can be used for extremely expressive attacks (e.g., arbitrary code-injection). The most expressive kind of attack grants to the attacker control over the executing code. The original *non-executable (NX)* stack countermeasure [68] proved efficient against initial stack-related attacks, aiming to modify program's control flow by inserting spurious data on the stack. Shortly after, attackers discovered how to surpass NX stack protections by crafting spurious library function calls. Basically, this allows a vulnerable function to divert from normal execution by returning to any library function, an attack known as return-to-lib-c [69]. Stack canaries [70] were introduced to work alongside NX stack protection, detecting when stack return addresses were altered. Stack canaries can also mitigate return-oriented programming [71, 72] attacks which allow an attacker to exploit memory errors in a program without injecting new code. In a return-oriented attack, the attacker arranges short sequences of existing instructions, being able to induce arbitrary behavior in the target program [71]. Canaries mitigate this kind of attack by detecting changes to return addresses, stored in the stack. Further work went beyond NX stack protection, introducing the W$\oplus$X security feature, which separates data and executable segments to increase security over just a NX stack. This feature mitigates the expressiveness of code-injection attacks as executable segments are not writable and data segments cannot be executed. Stack smashing countermeasures were also added to the GNU GCC compiler [73]. Even with all these protections in place, attacks which alter execution flow are still possible (e.g., by changing pointers used in indirect branches), although more difficult to perform. *Address Space Layout Randomization (ASLR)* is a technique used to thwart attacks which rely on target code or data location knowledge [74]. Customarily, ASLR implementations randomize memory segments at the start of the application and vary from a coarse to a fine-grained type of randomization. Using *Position Independent Executable (PIE)* with ASLR provides an extra layer of security since the code does not use constant relative offsets to a base address, although possibly incurring in performance penalties by adding indirections (e.g., by having to load the address of a function before jumping into it). PIE executables can be positioned anywhere in memory as they do not possess absolute addresses. With the security community focusing on stack-related defenses, attackers shifted their attention to the HEAP. In the absence of a NX HEAP, it's possible to redirect code execution to a HEAP memory location. If desired, the HEAP can also benefit from some of the previously mentioned defenses

(e.g., NX stack). As the focus of this work is not on dynamically allocated data, HEAP attacks and defenses are not discussed any further.

3.2 Countermeasure Design

As the arms race between security countermeasures and attacks endures, several proposals try to concede memory safety to unsafe languages like C and C++. Adopting other programming languages is often not an option in the embedded realm, mainly due to indeterminism caused by powerful run-time environments and limited flexibility. Memory safety can mitigate novel attacks, such as DOP [13], and any other attack relying on memory vulnerabilities to succeed. Providing memory safety, with negligent performance overhead, is still an active field of research, especially when considering an embedded environment. Several methods have been purposed ranging from secure dialects of unsafe programming languages to bound checkers. Other methods trade off complete memory safety for performance improvements. This duality endures on the design of every security countermeasure. Furthermore, most security research aims at general computing ignoring special constraints associated with embedded systems design (e.g., determinism) or lack of supporting hardware (e.g., MMU), commonly available in those platforms.

Distinct security solutions, aiming at memory safety to various extents, are exposed in this subchapter. Hopefully, it will provide context over the devised security solution, uncovered in the next chapter, attained either by analyzing similar approaches or by considering the attacks mitigated by such countermeasures.

3.2.1 C and C++ Dialects

C and C++ use a weak type system, granting flexibility in data representation and pointer usage, which can often be exploited. The lack of memory-safety in C/C++ can lead to vulnerabilities. CCured [24] is a C dialect introducing a safe type system utilizing both static analysis and run-time checks. Cyclone [3] is another dialect also aiming to rule out all memory violations. Dialects attempt to bring safety and security to unsafe programming languages, without transferring the abstractions inherent to high-level languages. Furthermore, some of these dialects try to keep memory management under the hands of the programmer as much as possible, instead of introducing garbage collectors with high performance cost. Therefore, they maintain the flexibility of the corresponding languages. Increased security comes at a cost, as depicted in Fig. 6 for Cyclone. More often than not, dialects give place to alternative security solutions without complete

memory safety but with acceptable performance overhead. Furthermore, a disadvantage of these approaches is the complexity of porting existing C or C++ code to these dialects.

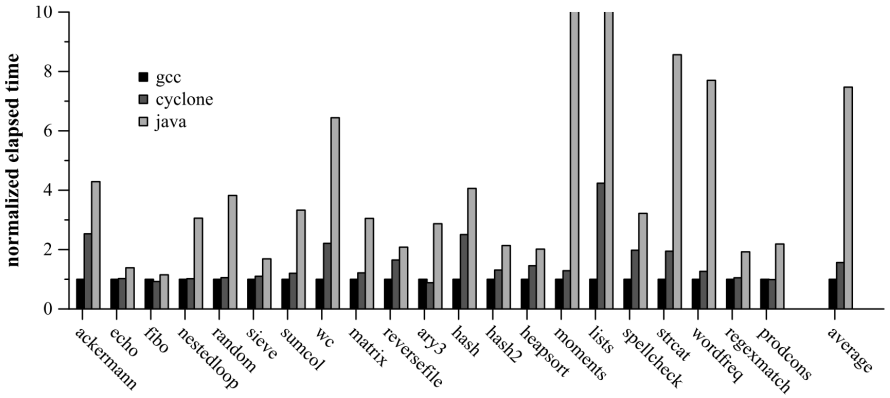


Fig. 6: Comparing Cyclone performance overhead with other programming languages (Reproduced from [3]). A benchmark is performed executing different programs compiled with GCC (optimization flag -O3), Cyclone and Java. Elapsed time is normalized to the C code, compiled with GCC.

These dialects may not be adequate for some particular embedded systems, where total control over the generated code and execution is necessary. In this case, a more specialized solution could be deployed. These dialects aim at high security with the lowest possible performance overhead, without a particular focus on determinism, a key design metric in many embedded systems.

Safer high-level languages, such as Java, also provide strong security guarantees. Unfortunately, these languages only hold such guarantees at source level while their implementation may not enforce them or may implement them naively [75]. Furthermore, many system and embedded software rely on hand-written, optimized machine code which are not able to enjoy the full benefits of such languages or dialects.

3.2.2 Bound Checkers

A major benefit of bound checkers [76, 77, 78] is not requiring source code modification. This methodology consists of inspecting pointer operations, namely, arithmetic and dereference, to safeguard that they remain within the bounds of the same objects, providing spatial safety. Bound checkers can be divided into two categories: Object- and pointer-based approaches [77]. With the first approach, disjoint metadata is stored in a separate data structure, allowing to map addresses, used in the above operations, to objects. Generally, this metadata consists of object's base addresses and bounds, being used to perform run-time out-of-bounds checks. AddressSanitizer [79], Dr. Memory [80] and Memchecker [79] are object-based approaches that rely on a shadow memory to target which addresses can be accessed and which cannot. Pointer-based

approaches use some sort of fat pointer representation, retaining base and bound information with each pointer. This representation transforms some or all pointers to a multi-word structure, containing both pointer value and the low and upper bounds of an object. The previously mentioned Cyclone and CCured dialects use this approach. PAriCheck [77], J&K [81], Baggy Bounds Checking [78] and CRED [82] verify if pointer arithmetic operations do not result in addresses out of object's bounds. Both approaches have their respective strengths and weaknesses. Object-based approaches do not change the memory layout and, in generality, cannot provide complete spatial safety due to overflows inside complex data types (i.e., structures or classes). On the other hand, pointer-based approaches change program's memory layout, possibly causing source code compatibility problems. Regardless of the approach, bound checkers must be complete and check every vulnerable pointer operation, aiming at full memory safety. Primarily, bound checkers aim at spatial safety with some disregard for temporal safety [76].

3.2.3 Control Flow Integrity

CFI aims at blocking attacks targeting programs' control flow, ensuring that these follow a pre-computed execution model: the CFG. CFI is not a complete memory safety scheme like, for example, *Data-flow Integrity (DFI)* or bound checkers, as it only detects control-flow modifications disregarding non-control-data attacks. Thus, it decreases performance overhead by not being a complete solution. CFI intents to detect incorrect values in indirect calls using runtime checks. This scheme assumes a powerful attacker willing to do anything to subvert program's legal control flow, by modifying code pointers. A code pointer can be defined as any variable whose value is used, at anytime, as the address for an indirect call. Subsequently, variables defined as code pointers or stack stored values are included. CFI main concern is indirect calls since the destination address for the call is calculated at runtime instead of compile-time. These branches define the entire attack surface for CFI, assuming program's code is stored in a memory partition protected against writes and with permission to be executed. CFI implementations have three main goals, as stated in [83]:

1. From the CFG, extract a labeling scheme for all indirect jump targets.
2. At each indirect jump target in the binary, insert a label.
3. At each indirect jump instruction, insert instrumentation that ensures the jump target contains the expected label.

A CFG is a directed graph whose nodes represent basic blocks in a program, and the edges depict legal control-flow transfers from one basic block to another, as defined in [83]. In a program,

most instructions capable of altering the control-flow are direct (i.e., their address are defined statically). Nevertheless, indirect calls are present in almost anywhere and are the primary target for attackers. With indirect calls whose value is calculated at runtime, it is critical to perform some sanity checking over the possible destinations of those indirect calls. The CFG generation algorithm intents to reduce possible destinations for an indirect jump while, at the same time, labeling all destinations. For example, a function (Fig. 7) steps through a list, using a loop, to calculate the shape with highest area and, due to C++ polymorphism, there are two area functions - one for circles and another for squares - which create an indirect call to calculate the area depending on the object being analyzed. A possible CFG is depicted on the left side of the Fig. 7. The destination of the indirect call must match the labels associated to both *area()* methods. This verification can vary between implementations. The original CFI scheme uses instrumentation to insert label checks where indirect calls occur and label all destinations of indirect branches (e.g., function's beginning or prologue).

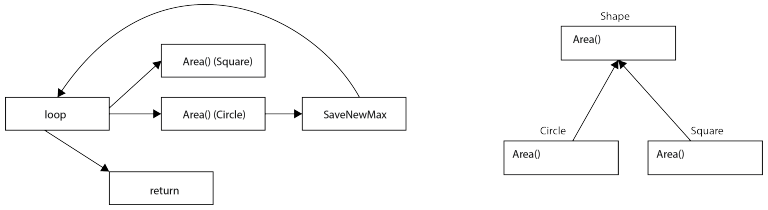


Fig. 7: Using polymorphism to create an indirect branch instructions inside a C++ function.

Yet in the original implementation, the instrumentation tries to read a label from the branch target address, offsetted by a constant value, comparing it to an hard-coded label. If a match occurs, execution continues to the target address. Otherwise, the program can, for example, jump to an error handler.

These initial efforts were complemented with a runtime protected call stack [84] [85] [86] [87] [88]. This stack is required because CFI is limited by the nature of the CFG. The original CFI implementation cannot ensure that a function call returns to its callsite [75] and a runtime call stack can improve CFI enforcement. However, unlike the main stack, this stack must be protected. The stack must be protected both against direct attacks or corruption resulting from program's execution [75].

3.2.3.1 *Enforcing CFI with HyperSafe*

Wang *et al.* introduced HyperSafe [89], an approach to ensure lifetime control-flow integrity to a hypervisor. HyperSafe employs two key techniques to ensure control data's integrity: 1) *non-bypassable memory lockdown* and 2) *restricted pointer indexing*. Non-bypassable memory lockdown consists of using the MMU to enforce W⊕E protection on pages containing legitimate

code. However, W⊕E integrity enforcement relies heavily on the integrity of page tables. For this purpose, page tables are made read-only. Locking writes to page tables also disallows benign page table updates, as the hypervisor can only access memory through virtual addresses, under pages' access policies. To solve this problem, HyperSafe leverages the *Write Protect (WP)* bit, available in x86 CPUs. With this bit, write-protection is ignored and the page table can be updated. At run-time, any attempt to modify page tables will be trapped into a page fault handler that verifies and executes the page update - by disabling the WP bit -, in order to avoid violations of the W⊕E property.

Restricted pointer indexing is a technique that substitutes indirect branches by "index-based" branches. This technique creates static tables with the real destination addresses for indirect branches. This allows to substitute the addresses of indirect branches by indexes to tables, followed by the branch. This limits the set of possible destinations for a branch. Control-flow related instructions are divided into two categories. *Source instructions* generate the control-data used by the hypervisor program (e.g., inserting the return address on the stack). These instructions are instrumented to insert indexes in the program instead of addresses. *Sink instructions* consume the data inserted by *sink instructions* (e.g., returning from a function). These instructions are instrumented to map from indexes to actual function addresses to, then, perform the branch. In practice, indirect instructions can only transfer control to the targets allowed by the CFG. The CFG is computed offline, allowing to protect the tables, at run-time, with 1).

3.2.4 Data Flow Integrity

Castro *et al.* introduced DFI [90], a technique enforcing legal data-flow over the program, recurring to instrumentation and static analysis. Essentially, it computes a *Data-Flow Graph (DFG)*, using static analysis, and instruments the program to ensure that the flow of data - at runtime - follows the data-flow graph [90]. In more detail, it identifies every instruction that reads a variable and uses static analysis to compute the set of instructions that are allowed to write the variable. Afterwards, it instruments writes and reads to ensure that the variables read at runtime were written by allowed instructions [91]. DFI can avoid attacks which cannot be prevented by bound checkers. The latter checks for out-of-bounds accesses and only tries to provide spatial safety. Attacks that overwrite data using pointers to dead objects whose memory was reused (i.e., dangling pointers) go undetected by bound checkers, but they can be detected by DFI. The main goal of this approach is to enforce a safety property (data-flow integrity) that is automatically derived from source code [90]. Furthermore, DFI prevents both control and non-control-data attacks, while CFI only avoids the former type of attacks.

DFI relies on reaching definitions analysis to generate the static data-flow graph. Reaching definition analysis is a static analysis technique, based on data-flow analysis. It concretely deals with reaching the definition (i.e., assignment) and use (i.e., read) of variables. From another perspective, the DFG depicts which instructions may assign values to which variables. DFI employs two types of static analysis to generate reaching definitions in order to diminish computing overhead, as it's easier to analyze definitions to variables inside the function they were declared at than variable definitions outside that same function. After DFG generation, instrumentation is inserted to perform a run-time program check before every variable use. The program gets terminated if the data-flow integrity safety property does not hold. DFI uses a Runtime Definitions Table (RDT) to store the last definition for each variable. This value is checked against the static DFG before a variable use or read.

3.2.5 Write Integrity Testing

Write Integrity Testing (WIT) [91] provides another practical solution capable of defending against exploitation of memory errors in vulnerable programs. Similarly to DFI, it uses static analysis techniques and instrumentation to prevent instructions from illegally modifying variables and ensure that program's execution doesn't divert from the statically-defined control-flow graph. WIT relies on points-to analysis [92] [93] to compute both the control-flow graph and an equivalent graph for data accesses, containing the set of objects each instruction in the program can write [91]. At runtime it enforces CFI, ensuring that programs do not divert from legal execution, that is, follow the control-flow graph. Nevertheless, the focus of WIT is on impeding instructions from modifying objects not included in the statically devised object set, by enforcing a security property called *write integrity*.

When objects are allocated or deallocated, WIT updates a *color table*. The color is devised from the points-to analysis, which assigns a color to all objects and each write instruction such that all objects that can be written by an instruction have the same color [91]. The instrumentation allows to generate colors as objects are created and to check that instructions write to the right color. In short, WIT stores a data structure with colors associated to allocated memory locations. When a write occurs, it checks if the instructions write to the right color. WIT manages stack allocated variables by allocating a color at function's entry point. Deallocation is performed at function exit. An attempt is performed to appoint different colors to all unsafe objects. Nevertheless, each instruction must have the same color as the objects it writes. WIT also tackles dynamic allocated memory. A disadvantage of WIT is its field-insensitive points-to analysis which means that fields inside structures are not distinguished which can make this approach unsuited for C++

programs. Furthermore, WIT also assigns colors to indirect call instructions and function entry points, so that all functions that can be called by the same indirect call have the same color [91].

WIT can be considered a solution with lower performance overhead when compared to the aforementioned DFI method. Both solutions aim at blocking memory exploits in vulnerable programs. Unlike DFI, WIT only instruments writes to variables, making it far more superior in terms of performance. Nevertheless, WIT does not ensure data confidentiality, allowing leaks of private, critical information.

3.2.6 Enforcing Data Flow Integrity on the Kernel

Song *et al.* [94] leverages CFI and DFI to prevent spatial access errors on indirect memory accesses. KENALI is designed to protect against privilege escalation attacks on the kernel. System software, such as operating systems and hypervisors, is typically implemented in unsafe programming languages like C and assembly, making memory corruption exploits the most popular. Non-control-data attacks easily bypass CFI-only protection schemes. Thus, such protection schemes must be complemented with other defense schemes, specifically designed to mitigate non-control-data attacks. KENALI proposes a defense system both *principled* and *practical*, to protect against future attacks while maintaining reasonable performance overhead [94]. While this scheme can be used to enforce various security invariants, KENALI focuses on implementing security invariants related to privilege escalation attacks and kernel access control mechanisms. In particular, *complete mediation* and *tamper proof* are enforced. The former ensures that access control checks are not bypassed, while the latter attests that attackers are not able to tamper with the reference monitor. KENALI proposes a system employing two techniques: *INFERDISTS* and *PROTECTDISTS*.

INFERDISTS is the process that automatically detects kernel-wide security-related data associated with the deployed access control mechanism. The resulting set of data is referred as *distinguishing regions*. Note that all control-data must be included in these regions to enforce CFI and, subsequently, the complete mediation security invariant. KENALI retrieves data belonging to distinguishing regions by analyzing function return codes related to the access control mechanism (e.g., -EACCESS). INFERDISTS retrieves these error codes and, using dependency analysis on conditional variables belonging to security checks, infers the distinguished regions [95].

PROTECTDISTS employs DFI over distinguishing regions, using a two-layer protection scheme. The first layer provides coarse-grained protection to prevent illegal data-flow from non-distinguishing regions to distinguishing regions. The second layer arranges fine-grained protection for data-flow between distinguishing regions. PROTECTDISTS coarse-grained im-

plementation is based on paging and virtual memory. For details on the MMU-based approach for DFI enforcement, the original article should be consulted in [94]. PROTECTDISTS employs a modified version of WIT [91] for fine-grained DFI, only instrumenting writes for data accesses in distinguishing memory regions and using a field-sensitive points-to analysis instead of the field-insensitive approach proposed in WIT.

In summary, KENALI makes two major contributions: the automatic inference of security-related data to enforce the security invariants and a two-layer DFI protection scheme that reduces performance overhead when compared to the original DFI [90] implementation.

3.2.7 Dynamic Information Flow Tracking

Suh *et al.* pioneered DIFT, an approach to protect against software attacks by identifying spurious values, from I/O, and restricting their usage [4]. The method was devised from a rationale that, due to the wide range of security vulnerabilities that overwrite memory locations, it is impossible to detect them all. Thus, an alternative is to catch the unintended use of I/O inputs [4]. I/O inputs can either be managed by the OS (e.g., files) or created by it (e.g., inter-process communication).

This approach relies on both software and hardware components. Namely, DIFT uses a software module, in the OS, that marks inputs from untrusted sources as spurious, as depicted in Fig. 8. Using the processor, it checks every operation for spurious results based on both the inputs and performed instruction. Any use of spurious values is automatically redirected to a software handler that analyzes the alert based on a security policy. For example, checking if an instruction uses a stored spurious value as address for an indirect write prevents changes to the data flow, from potentially malicious inputs and data generated from them. DIFT is named so because it tracks the usage of spurious values. Spurious inputs can propagate to instructions' results depending on the executed operations. Thus, these results are also marked as spurious, by the processor.

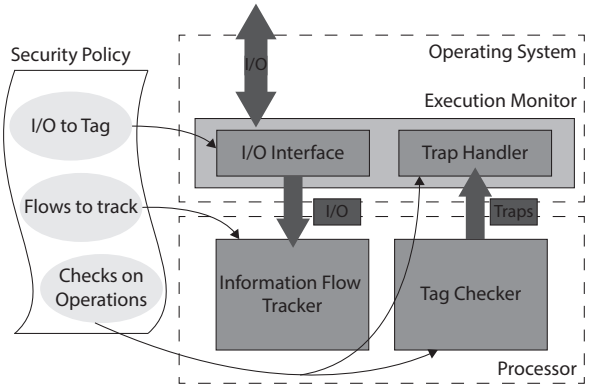


Fig. 8: DIFT Architecture (Reproduced from [4]).

DIFT adds two mechanisms to the processor. On an instruction basis, the Information flow tracker determines if the result is spurious depending on inputs and executed operation. The tag checker looks at the input operands and instruction performed, and executes the software handler if spurious values are used for the operations specified in the security policy, redirecting execution to the monitor [4].

Defining an adequate security policy for an application is essential to capture illegal uses of I/O values and avoid false positives. The security policy defines untrusted I/O channels, trap conditions and software checks when traps are triggered. Traps indicate potential violations of the security policy. With a more general policy, it's natural to trap legitimate uses of spurious values. The software handler exists to explicitly check traps (e.g. by performing bound check) and eliminate false alarms while interrupting execution in case of a real attack.

DIFT requires several processor architectural changes that aren't currently available in *Commercial Of The Shelf (COTS)* solutions (e.g., ARM processors). Mainly, this creates a big setback to its adoption, as it may not be easy to justify the extra cost for the added security, in most projects. Nevertheless, the solution provides a very low performance overhead for certain applications. Primarily, DIFT is only able to rival against the above software-based approaches by providing low performance overhead due to hardware adaptations.

3.2.8 Specification-based Approaches

Specification-based detection relies on program specifications that describe the intended behavior of security-critical programs [96]. Specification-based detection schemes [96, 97, 98, 99] attempt to remove *ad hoc* approaches for codifying system behavior. They are based on the premise that a program adheres to a benign behavior and any deviation from it is considered a security breach. Generally, these approaches include an execution monitor that collects traces from program's execution and enforces security policies, derived from program's semantics. The deployment of the execution monitor varies between approaches. In [98] the monitor executes when system calls are performed by programs and before they are deployed to the kernel thus, reacting before any damage is caused to the system. However, in [99], *Virtual Machine Introspection (VMI)* [15] is suggested, while on a virtualized environment .

The approach in [99] devises a specification for kernel data to detect attacks. It is of special interest as it proposes a somewhat general specification architecture, consisting of five components:

- **The low-level monitor** - The monitor corresponds to the entity that captures data from kernel execution. For example, using VMI or other technique.

- **The model builder** - This entity uses raw data from the low-level monitor and maps it to a model from where the constraints can be applied.
- **The constraint verifier** - This component corresponds to the runtime specification checker.
- **Response mechanisms** - If a security violation occurs this entity is responsible for an adequate response. For example, correcting an error or logging.
- **The specification compiler** - This is an off-line component that translates an high-level specification to a form that can be used by the constraint verifier.

The general ideas introduced above can be easily associated with this work. Namely, every component except the model build and the response mechanisms can be directly mapped to the implementation exposed forward. Response mechanism are not the focus of this work.

3.2.9 Concurrent Security Monitoring

A different approach to security, diverging from the aforementioned memory protection mechanisms, is concurrent monitoring. Several schemes [14, 100, 101] have been devised to concurrently monitor critical programs, while isolating the security mechanism from the target program. Co-Pilot [100] detects kernel integrity violations by running the monitoring software on a separate PCI card. VMWatcher [14] performs VMI to gain semantic insight alike security tools deployed in the OS. [101] uses both VMI and hooks inserted in critical code sections (e.g., system calls) to analyze kernel integrity violations. These propositions aim at securing the kernel but many of the techniques can be easily adopted or adjusted to protect hypervisors or other kinds of software.

Azab *et al.* proposed HyperSentry [89], a framework for runtime integrity measurements on modern VMMs. This work differs from others as it tries to verify integrity of a VMM instead of VMs, thus being extremely relevant for this thesis. Unfortunately, HyperSentry was designed to run on Intel x86 architecture, depending highly on architectural aspects available only on Intel processors. This thesis aims at providing Data Integrity to μRTZVisor, an hypervisor restricted to execute on ARM processors with ARM's Security Extensions, more specifically, TrustZone.

HyperSentry introduces an isolated and stealth software component to execute integrity measurements. Conventional wisdom defends the introduction of a software a layer below the protected software. Defying this, HyperSentry inserts the integrity measurement agent in the same layer as the VMM, since the VMM already executes in the lowest software layer. Leveraging Intel's architecture, the hidden measurement agent can securely perform any requested integrity measures. HyperSentry's TCB consists only of the hardware, firmware and the properly isolated software component (i.e., the *System Management Mode (SMI)* interrupt). This method devises

a technique to create an Out-of-band channel to stealthily trigger hypervisor's integrity measurement agent. The agent requires access to hypervisor's code, data and CPU state. This channel triggers the SMI interrupt which then puts the system in System Management Mode, where, using novel techniques, in-context integrity measurements can be performed to the VMM. In terms of security, HyperSentry lacks protection against transient attacks. This is a limitation of all periodic invocation integrity verification tools. Transient attacks do not perform persistent changes code and data. Thus, they can hide their traces and go undetected by these tools.

Donghai *et al.* presents a concurrent security monitoring method for virtualization environments [5]. To protect a VM, a security collector and analyzer are employed. These components are characteristic of common security software and deployed under an OS. This method decouples both components into two concurrent units, to avoid any attempt of tampering by attackers and diminish performance overhead.

The security collector uses system call events as input to detect intrusions, under the Linux OS. Firstly, a Linux system is modified to insert a trampoline, in every system call, on-the-fly, using the VMM. As execution jumps from program to OS, a kernel module, *interceptor*, is called to store data related to the system calls in a data structure. Furthermore, this kernel module is also dynamically inserted. Other techniques are employed to ensure that this "in-the-box" component stays appropriately isolated from privilege escalation or other kernel related attacks.

The security analyzer is an "out-of-the-box" component deployed in a different core and VM. Data captured from the *interceptor* is analyzed within this component. Using different cores, it's possible to achieve concurrent execution between the monitored program or, in this case, VM and the monitor software (i.e. security analyzer). The *interceptor* stores its data using the Lamport's ring buffer algorithm [102]. Furthermore, the security analyzer has two threads: the event collector and event analyzer [5], as depicted in Fig. 9. The former creates a copy from the ring buffer filled by the security analyzer. The latter analyzes previously copied data for irregularities, performing logging operations whenever necessary. The *event analyzer* uses the method exposed in [103] to detect abnormalities in system behavior from system calls. This method creates a model using static analysis and compares the run-time behavior against this model to detect irregularities. Basically, it can monitor any process of the *DomU VM*.

3.3 Expressiveness of Non-Control Data Attacks

Despite their general applicability, non-control-data attacks are not as straightforward to construct as control-data attacks, mainly due to the required semantic knowledge about target applications [12]. A survey on memory errors shows that most attacks aim at control data. Control-data can be defined as a subset of program's data which can control the execution flow of the same.

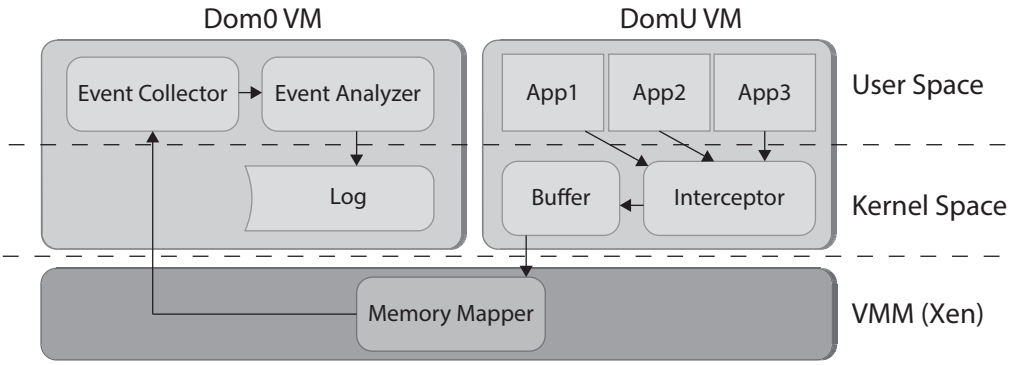


Fig. 9: Architecture of a concurrent security monitor architecture for virtualized environments (Adapted from [5]).

Therefore, defenses have been designed to guarantee CFI. Challenged by these countermeasures, attackers became motivated to search for new attack vectors. Non-control data attacks emerged as an alternative to the previous ones since they are not based on rewriting control data or performing inadmissible control transfers, which can be detected at runtime [19]. Non-control data attacks are not recent [12], however, they are gaining attention as countermeasures against control-data (e.g., stack return address) become available [13]. Non-control data attacks can vary on their expressiveness.

These attacks still exploit memory errors, such as, buffer overflows, integer overflows or heap corruption, among others, so, logically, any defense which guarantees total memory safety (e.g., safe programming languages) can invalidate non-control data attacks as well. As we delve into non-control data attacks, it is extremely important to categorize the different types of data associated with a program. Extending the data categorization provided in [95], a small taxonomy for program's data is provided:

Non-control data

- **Function arguments** – Giving spurious arguments to certain system functions might compromise the system.
- **Decision making variables** – Stack allocated variables or global data that directly influences function behavior, without changing from the legal control flow.
- **Configuration data** – Files or non-volatile data can be compromised and, in turn, compromise the system.
- **User input data** – Attackers can change user input data after a sanity check, by exploiting a posterior memory vulnerability.

- **Security mechanisms data** – Data associated with security mechanisms can be altered to cover an attack, eluding in-place security mechanisms.
- **Passwords and private keys.** – Unintentionally revealing of system secret data can damage the system as data confidentiality is lost.

Control data

- **Data used for indirect branches**– Changing data such as code pointers of tables which posteriorly influence a change in program's execution flow.
- **Stack corruption** – Corrupting the stack (e.g. return address) is one of the most well-known kinds of data attacks.

3.3.1 Data Oriented Programming

Hu *et al.* [13] proposed DOP, a method to perform arbitrary computations on a program's memory, via non-control data attacks. Using this method it is possible to build very expressive attacks such as stealing of private information (e.g., private keys) or privilege escalation attacks. DOP has similarities to both *Return Oriented Programming (ROP)* [72] and *Jump Oriented Programming (JOP)* [71, 104], using gadgets to perform computations. However, in DOP, the structure of gadgets slightly differs from the other two methods, as a program does not divert from the legal execution flow.

DOP requires two gadget types: 1) data-oriented and 2) dispatcher gadgets. The first type consists of sequences of instructions that perform specific computer operations (e.g., arithmetic, assignments, dereferences). Subsequently, dispatcher gadgets are logic segments that chain disjoint data-oriented gadgets together. Using both, an attacker can make a program carry out computation of its choice on program's memory [13]. Data-oriented gadgets must use memory to perform its operations. Hardware registers are not a viable alternative for storage, as the original program uses registers for its normal operations. Furthermore, operations that modify hardware registers and memory state outside attacker's control are likely to occur between two gadgets.

Gadget dispatchers are sequences of instructions that allow an attacker to selectively and repetitively invoke gadgets. A loop with a selector is a common sequence of instructions that simulates a gadget dispatcher, where each iteration executes a subset of gadgets using outputs from gadgets executed in the previous iteration [13]. The load and store addresses of both iterations are controlled by the attacker, via the memory error, as data between gadgets is exchanged

in memory. Such an example is provided by Hu *et al.* in [13] and depicted in Listing 3.1 and 3.2. Using an overflow vulnerability in the `readData` function, listed in 3.1, an attacker can control loop iterations and the data used in the following selector to invoke specific gadgets. A data-oriented gadget can be found in Line 12 and 13 of Listing 3.1, that corresponds to the operation in Line 4 of Listing 3.2. To increment the list pointer in that same listing, the gadget in Line 10 of Listing 3.1 is used. To control the number of times the cycle executes, one only needs to control the `connect_limit` variable through the memory error.

Listing 3.1: Vulnerable code executing operations on connection-related data. To create a DOP attack, the vulnerability in readData is exploited. The while cycle with the selector provide the required dispatcher gadget. The remaining instructions are data-oriented gadgets.

```
1  struct server{ int *cur_max, total, typ;} *srv;
2  int connect_limit = MAXCONN; int *size, *type;
3  char buf[MAXLEN];
4  size = &buf[8]; type = &buf[12];
5  ...
6  while(connect_limit--) {
7          readData(sockfd, buf); // stack bof
8          if(*type == NONE ) break;
9          if(*type == STREAM) // condition
10                 *size = *(srv->cur_max); // dereference
11         else {
12                 srv->typ = *type; // assignment
13                 srv->total += *size; // addition
14         }
15         ... (following code skipped) ...
16 }
```

Listing 3.2: Exemplifying the arbitrary computations that can be performed in the vulnerable code in listing 3.1, using DOP.

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  void updateList(struct Obj *list, int addend) {
3          for(; list != NULL; list = list->next)
4                  list->prop += addend;
5  }
```

By using a vulnerability to manipulate the values of local variables and not ever breaking the legal execution control flow, an attacker can create a list and increment its values. This speaks to the expressiveness of this kind of attacks. Depending on the available gadget set, critical system variables can be modified or disclosed corrupting system integrity, confidentiality and availability.

3.3.2 Data Stitching

Hu *et al.* [105] designed a systematical method to automatically create data-oriented exploits. This scheme introduces the concept of a 2D-DFG to represent data flows in two dimensions: memory addresses and execution time. A 2D-DFG represents data dependencies in a program

for a concrete input [95] and is represented by $G = \{V, E\}$, where V is a set of vertexes and E a set of edges. A vertex is denoted as (a, t), where a corresponds to the address of the variable and t the execution time. A vertex is created when an instruction writes to a memory location a. In the 2D-DFG, an edge (v', v) between two vertexes denotes a data dependency created during the execution. A data edge (v', v) is created for instructions that take v' as input and take v as destination operand. Address edges are created for instructions that use vertex v' as the address of one operand v, i.e. v is calculated from a dereference to v'.

A memory error is required to generate a non-control-data attack, and the set of memory locations affected by it is called I. Data stitching attempts to connect a source vertex v_s to a target vertex v_d. The new data-flow path, connecting both vertexes would have a new 2D-DFG $G' = \{V', E'\}$, with a new set of vertexes and edges, generated by the memory error exploit. The goal of data-flow stitching is to discover a new data-flow edge set E', allowing new data-flow paths from v_s to v_t.

Using the example provided in [105], attackers use a string vulnerability to overwrite the security critical variable `pw->pw_uid` with the root user's id. The `setuid` system call, that is intended to drop process' privileges, makes the program retain its root privileges instead. On the left of Fig. 10, it is the 2D-DFG with benign inputs and on the right with spurious inputs. Numbers on Listing 3.3 correspond to the time-axis on the 2D-DFG. The exploit inserts an edge to write zero from the spurious input to the memory allocated to `pw->pw_id`. This exploits does not subvert execution's control-flow.

More complicated examples are provided in [105] although they are not as common as single-edge stitches. This includes examples using multi-edge stitches, connecting two completely different data flows to achieve the connection between v' and v.

Listing 3.3: Example of vulnerable code.

```
1  struct passwd { uid_t pw_uid; ... } *pw;
2  ...
3  int uid = getuid();
4  pw->pw_uid = uid;
5  ... //format string error
6  void passive(void) { ...
7          seteuid(0); //set root uid
8          ...
9          seteuid(pw->pw_uid); //set normal uid
10         ... }
```

3.4 MELT

GCC MELT is a high-level domain specific language for extending or customizing the GNU Compiler Collection [106]. MELT allows easier customization of the GCC compiler, providing

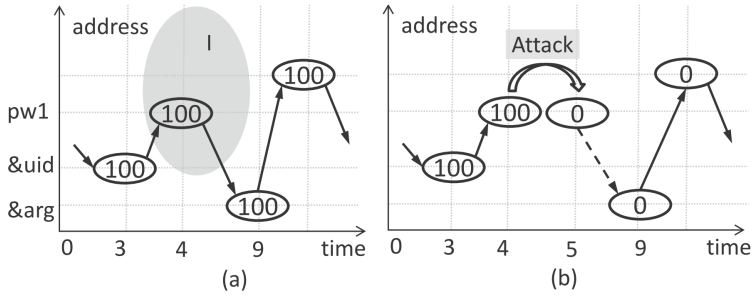


Fig. 10: The 2D-DFG from the code in Listing 3.3. On the fourth instruction, a non-control data attack modifies the value of `pw->pw_uid` in such a way that the root privileged is never revoked.

abstraction over C/C++ coding and interfacing with *GCC* internal representations. Although being a powerful tool, it still requires an understanding of GCC ILs (e.g., GIMPLE and GENERIC). MELT uses the GCC Plugin API, available since version 4.5, which allows to add custom code to GCC. This allow a programmer to take advantage of the existing compiler processing power. Listing 3.4 denotes the power of MELT via a code excerpt provided in [107]. The fifteen *Lines of Code (LOC)* allow to identify functions prefixed with the `bar` string and to pinpoint function calls to `fflush` with `NULL` argument. MELT was not used because it was not specifically designed for what the developed application required from GCC. The choice was to program a plugin directly in C/C++. From an academic point-of-view, this allowed an increase understanding of GCC's inner workings and its intermediate representations. It is believed that the chosen approach was the most efficient, since the manually developed plugin had very specific requirements from the compiler, which might not be easily available through the abstraction provided by MELT. Also, to implement certain functionalities, analysis of GCC source code had to be performed.

Listing 3.4: Example code programmed in MELT.

```
1  (match cfuncdecl
2    ( ?(tree_function_decl_named
3       ?(cstring_prefixed "bar" ) ?_)
4       (each_bb_current_fun () (:basic_block bb)
5         (eachgimple_in_basicblock (bb)
6           (:gimple g)
7           (match g)
8             ( ?gimple_call_1 ?_
9               ?(tree_function_decl_name
10                ?(cstring_same "fflush") ?_)
11              ?(tree_integer_cst 0))
12              (inform_at_gimple g
13                "found fflush(NULL)" ))
14            ( ?_ ()))))))
15 ( ?_ ()))
```

CHAPTER 4

Associated Work

As aforestated, this thesis primarily proposes a security countermeasure to protect the data plane of the µRTZVisor hypervisor. Before delving into implementation details, it is imperative to understand the internals of µRTZVisor itself, specially the data plane. Thus, a special focus will be on explaining its data structures and variables, without forfeiting its implementation based on the ARM TrustZone technology.

The second part of this thesis dwells within ontology modeling. An effort was undertaken to integrate the modeling efforts with the SeML modeling infrastructure. SeML provides enhanced semantic and code generation capabilities that are leveraged to facilitate model representation and augment expressiveness. SeML will be elucidated in its own subchapter. Both µRTZVisor and SeML are under development at the time of writing.

4.1 The µRTZVisor VMM

µRTZVisor [22] is a TrustZone-based embedded hypervisor, programmed in C++. TrustZone allows the deployment of two isolated execution environments, empowering hardware-based system virtualization. By defining a secure and non-secure environment, complex applications (e.g., infotainment) can be deployed in the non-secure world alongside safety/security critical software executing in the secure world. The processor's Monitor mode, introduced in the ARM-v7A architecture, is the hardware highest privilege execution mode, when virtualization extensions are not present. This mode was introduced to perform context-switching between the non-secure and secure worlds, in both directions. As stated by ARM, the monitor code is a security critical component, as it provides the interface between the two worlds [1]. µRTZVisor leverages the Monitor mode and other TrustZone capabilities for virtualization.

The architecture of µRTZVisor is depicted in Fig. 11. This architecture is similar to Fig. 2 since µRTZVisor follows a microkernel design approach. µRTZVisor relies on the SMC instruction to implement its Hypercall API. By executing a SMC instruction in a VM, the core enters

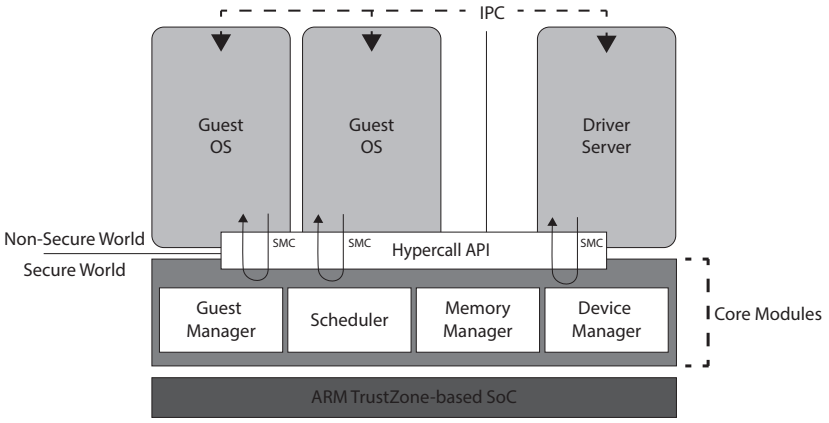


Fig. 11: Architecture of the μRTZVisor hypervisor.

the Secure Monitor Mode to execute VMM code. Having processed the hypercall, execution can be redirected to the requester VM. Amongst guests, the VMM grants spatial isolation through resource partition and multiplexing. Static permissions are defined for each guest, denoting available assets such as memory or hardware peripherals. At runtime, the VMM switches between different hardware access configurations depending on the next VM to be executed. These configurations define guest's resources and what must be protected by TrustZone. In the non-secure world, secure memory blocks and peripherals are protected by the TrustZone hardware. Thus, a VM can only access allocated assets. μRTZVisor diverges from traditional microkernel-like implementations, aiming to minimize modifications to guest OSs. At the same time, it leverages the flexibility of microkernel architectures, providing an IPC mechanism. This ensures that guest OSs only need to be modified to benefit from auxiliary services or shared resources on the kernel's IPC facilities.

Concerning security and memory corruption vulnerabilities, μRTZVisor attack surface is mainly composed by the Hypercall APIs as it's the only source of communication between guests and the VMM. This interface is similar to system calls of OSs, except for the usage of the SMC instruction that, in certain implementations, has already been violated [108, 109, 110, 111, 112]. By executing an hypercall, the core temporarily obtains the highest privileged execution mode, having full access to all system resources. Therefore, VMM data and code can be tampered with, resulting in attacks that could affect confidentiality, integrity and availability of the system. Furthermore, other VMs can be attacked without ever interfering with the VMM. The hardware is considered a trustworthy system component thus, the attack surface is entirely composed of software-based attacks. The Threat Model is further discussed in chapter 5.3.

4.1.1 *µ*RTZVisor Data-Plane Analysis

The main focus of this work is to protect *µ*RTZVisor against non-control-data attacks. Understandings the inner workings of *µ*RTZVisor, in terms of its data-plane, is vital for the remaining of this thesis. *µ*RTZVisor is architected using the singleton design pattern [113], as depicted in Fig. 12. Using singleton, a class only has one instance which is a global point of access to it [113]. The complete class hierarchy is not depicted here, as not all classes are relevant to this work. Also, methods are hidden. Nevertheless, each class implementing the singleton design pattern possesses the usual methods to access their unique static instance.

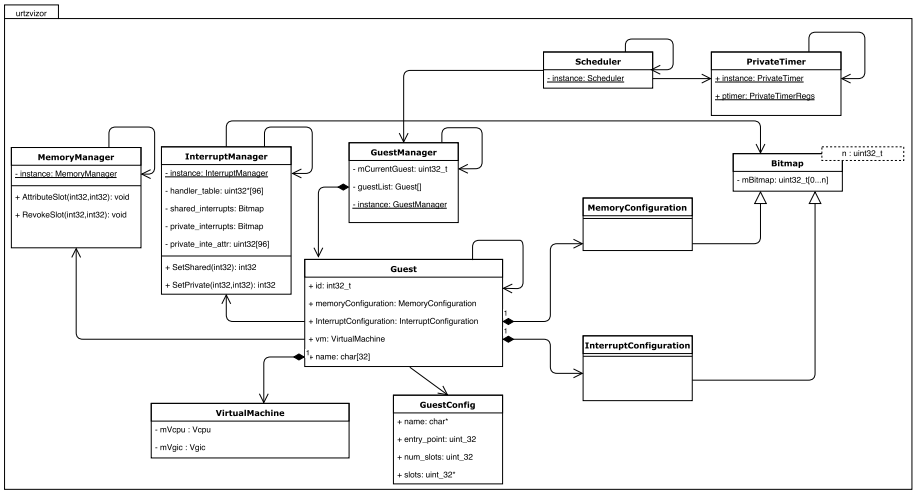


Fig. 12: Simplified UML Class diagram for *µ*RTZVisor.

*µ*RTZVisor has four classes that directly correspond to a component in Fig. 11. *GuestManager* stores all *Guests* in its *guestList* static array. The number of guests in the *guestList* is defined by how many VMs the system designer specifies, at design-time. This manager also aggregates the *mCurrentGuest* integer field, which specifies the currently executing VM. Each *Guest* individual, in the *guestList*, stores information related to a specific *VM* such as an *ID*, a *name* and the memory and interruptions configurations, in the *memoryConfiguration* and *interruptConfiguration* fields, respectively. Configurations of peripherals are not yet implemented, neither is the Device Manager. Both *memoryConfiguration* and *interruptConfiguration* inherit from the *Bitmap* class. The latter class is a template with a variable-sized integer array field called *mBitmap*. For both *MemoryConfiguration* and *InterruptConfiguration* classes, the *mBitmap* has distinct semantics. However, generally, it stores permissions to partitioned system resources (e.g., memory blocks). For *memoryConfiguration*, such array has three elements: the first partitions DDR memory into thirty-two individual blocks and the other two the *On-chip Memory (OCM)* memory into 4KB memory chunks. OCM is used to share memory between guests. The *PrivateTimer* class

has the internally defined *PrivateTimeRegs* structure and a pointer (*ptimer*) of this type. This pointer points to the memory mapped hardware registers corresponding to an hardware timer. This hardware timer controls the *Scheduler*, which possesses a callback function to schedule guest VMs. This data-plane analysis contemplates the μRTZVisor critical structures, which will be consistently mentioned throughout this thesis.

4.2 The SeML Framework

The main goal of SeML is to support the design of a functional system compliant to a model, created using a *Domain Specific Language (DSL)*. To achieve that, domain experts and knowledge engineers first describe system's commonalities and variabilities. Then, system developers map domain theoretical concepts to implementation artifacts. This activities are performed at the metamodel layer using OWL as the knowledge representation language.

With an ontology-based metamodel, the user is capable of using the SeML IDE (Fig. 13) to build a model of the system by interacting with the provided user interface and DSL. DSL keywords for system concepts and relations between them are adopted from the metamodel, allowing the user to immediately start a design-space exploration. Further, metamodels contain variabilities which allow users to configure a system stack suited to their requirements. Note, metamodels embed domain knowledge inserted by domain experts, which accelerates the whole process.

A model can be implemented (i.e., translated into a functional system) at any time, after a coherency verification by the SeML IDE. The DSL engine will invoke methods of the implementation tools, described by the System Developer, since these tools are nothing more than Java programs. The process of code generation is discussed below, with an example. In this work, additional implementation tools are created to cope with the requirements of the developed application. The SeML engine compiles and loads these tools, at runtime. The infrastructure supports OWL and SWRL rules for knowledge description and semantic model verification. As SWRL rules were not expressive enough, the user can also provide custom SWRL built-ins to create more complex rules.

The hierarchy of ontologies introduced in Fig. 13 is detailed in Fig. 14. The *Upper Ontology* is an abstract ontology containing common concepts for both system description and implementation layers. The *Domain Expert* creates a *Domain Ontology* with concepts to represent system artifacts. In turn, the *System Developer* creates individuals and OWL annotation properties for concrete artifacts, specifying the implementation tools to be executed. In fact, *Implementation Ontologies* are based on OWL annotation properties used to provide code generation capabilities to build the final functional system.

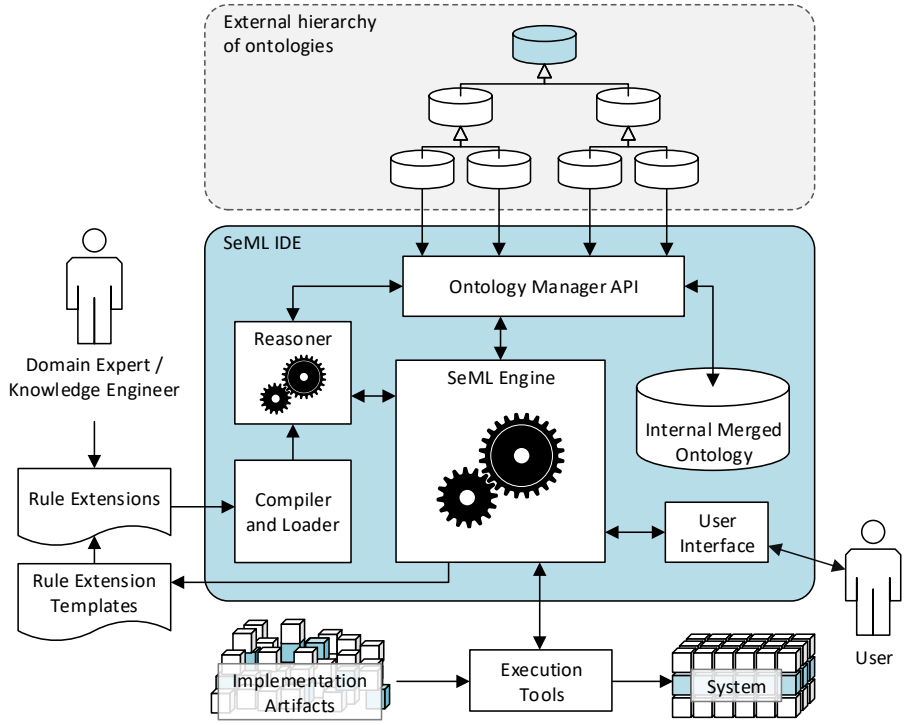


Fig. 13: SeML infrastructure architecture. Going from top to bottom, external ontologies are loaded onto the SeML IDE (implemented in Eclipse). The SeML Engine receives user input through a modeling DSL and uses reasoning capabilities to verify and complete models. At the implementation stage, SeML-related Implementation Artifacts (i.e., Java programs) are executed to translate the model to a functional system.

4.2.1 Code Generation

Mainly, SeML leverages annotation properties, provided by OWL, to incorporate code generation into models. The implementation stage is characterized by a strong relation between a model and user-defined external tools. Annotations are properties that can be associated with every element of an ontological model: individuals, object properties and data properties. SeML uses this feature to strategically execute auxiliary Java programs that, in turn, generate the required system files. Specifically, SeML uses reflection [114] for this purpose, automatically loading and executing Java programs, classified as SeML-related implementation artifacts, as required. Special annotations can be used to map model's ontological elements to implementation artifacts: *ImpInd*, *ImpOP* and *ImpDP*. According to Fig. 15, the first annotation triggers an Implementation Artifact when an individual has a *ImpInd* annotation. The second one is prompted when an object property or an individual possess the *ImpOP* annotation. However, if *ImpOP* is present on an individual, the name of the object property must also be specified, in order to locate it. The same applies to *ImpDP*, but for data properties. The annotation properties are defined in the upper

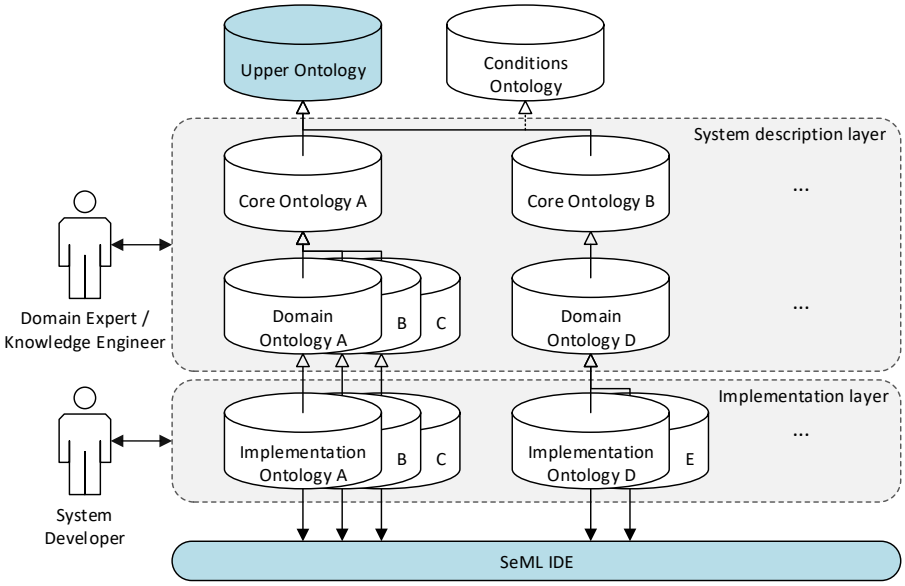


Fig. 14: External hierarchy of ontologies.

ontology. In the implementation stage, the infrastructure will look for the aforementioned trigger conditions to call a specific method of the specified Java program or tool. The order by which Implementation Artifacts are executed is defined by the integer *priority* parameter. A higher value means lower priority. If not specified, the parameter's value is ten (10). By default, the arguments sent to the tool accord to an infrastructure-defined type, *ImplArg*, and are treated as strings, unless the user states otherwise when asserting the annotation. These arguments are taken from predefined ontological elements and sent to the Java methods of the SeML implementation tools or artifacts. Each behavior is asserted in a certain ontological element and contains a string with the Java tool name and method to be invoked.

Annotation Property	Trigger Condition	Arguments	Assertion (individual)	Assertion (Property)
ImpInd	Instantiated Individual	Individual	Tool,method, (,priority(,Annotation Property)	——
ImpOP	Instantiated Individual & OP	Individual1*, individual 2*	Tool,method, property, (,priority(,Annotation Property)	Tool,method, (,priority(,Annotation Property)
ImpDP	Instantiated Individual & DP	Individual*, literal	Tool,method, property, (,priority(,Annotation Property)	Tool,method, (,priority(,Annotation Property)

*Source of arguments are ImpArg annotations (or user-defined) defined in individuals.

Fig. 15: Table denoting OWL annotation properties used for code generation.

Fig. 16 introduces an example with a non-annotated ontology, on the left, and the respective annotated ontology, for implementation purposes, on the right. Existent object and data properties are annotated, in their definition, with their respective annotation properties (i.e., *ImpOP* or *ImpDP*). In this example, individuals do not require annotations, releasing the developer from

annotating every newly created individual that would required an annotation. The only Implementation Artifact present, in this example, is the *Generator* Java program that merely translates an argument from an individual, specified by the already existing *ImpArg* annotation, to text. Then, it prints the name of the object or data property, finalizing with the name of the second individual or data property value that the property connects to. Looking at the priorities, one can expect a text output, for the existing model, of the following form:

pc_1 hasPart ram_1
ram_1 hasSpeed 1333 Mhz
ram_1 hasCapacity 8 Gb

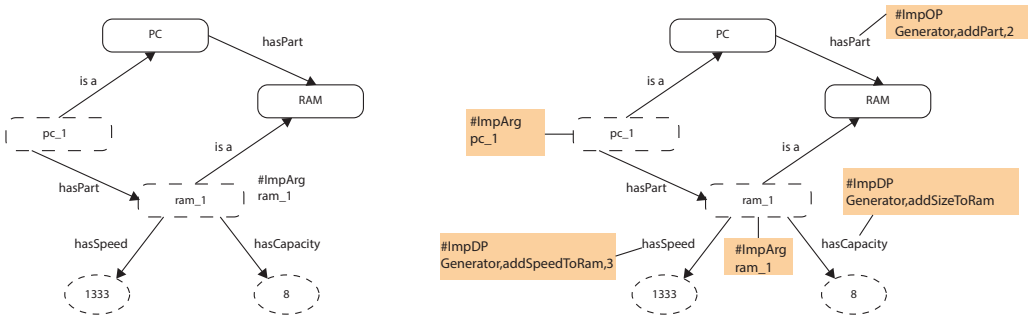


Fig. 16: Annotating ontologies to generate execute Implementation Artifacts.

CHAPTER 5

Data Integrity: Design and Implementation

This chapter encompasses the main contribution of this work: the design and implementation of the Data Integrity security mechanism. After the definition of the threat model, an overview over the specifics of the Data Integrity monitor is provided. Instrumentation is examined preceding a discussion of the data integrity rules. Finally, the Remote Monitor implementation and deployment are detailed. Throughout this chapter, the examples provided are strictly linked with µRTZVisor, as the security mechanism was specifically designed taking into consideration its specificities.

5.1 Design Goals

Designing a hypervisor with tight security requirements for safety-critical and/or real-time applications can be a demanding task. Performance and resource utilization are important metrics for embedded software. Although, due to real-time constraints, determinism is often paramount in embedded systems, creating an extra challenge when designing any security countermeasure oriented towards these systems. Modern defenses focus on protecting the legitimate control-flow of a program, nevertheless they cannot withstand a more subtle type of attack, non-control-data attacks, since they follow the legitimate control flow, and thus leave no trace [13]. Control-flow protection is considered in the holistic approach of securing the μRTZVisor hypervisor. This implementation must be partially designed to work alongside a CFI security countermeasure, providing both control- and data-plane protection.

The main goal is to provide data integrity to μRTZVisor's static critical data structures, following a deterministic approach, while minimizing performance overhead and resource utilization. Several restrictions regarding both hypervisor design and the supported platform were considered during the design and implementation of the proposed approach. Since the hyperviso is still under development at the time of writing, some efforts towards generalization and automation had to be undertaken to keep the security countermeasure up-to-date with newer μRTZVisor versions.

5.2 Platform

The system used in this work is the Digilent ZYBO [115] platform based on a Xilinx Zynq-7000 All Programmable SoC [116]. This SoC is equipped with dual-core ARM Cortex-A9 processors which implement the ARMv7-A architecture [45]. TrustZone is also available, and being leveraged by μRTZVisor for virtualization. Primarily, the memory layout of this SoC contemplates: an OCM, a volatile RAM and memory-mapped peripheral registers. The memory map, comprising these regions, is partially depicted in Fig. 17.

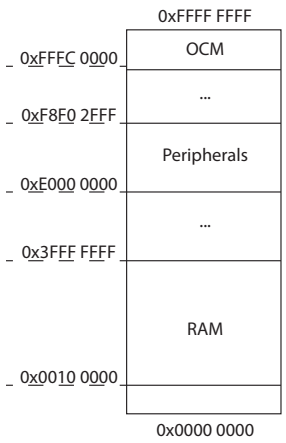


Fig. 17: Generic memory address map for Digilent ZYBO. This memory map is not complete and can be slightly modified according to system requirements. Only important memory section for this work (i.e., OCM, RAM and Peripherals) are highlighted. For more information relate to [116].

The ZYBO is endowed with a 256KB OCM and a 512MB RAM, besides the peripheral allocated memory. As these regions are encompassed in the same address space, an attacker can exploit them with a memory corruption vulnerability to perform arbitrary read and write operations. For fine-grained memory permissions and virtual memory, the ARM Cortex-A provides a MMU. In this platform, the MMU can use a two-level translation, granting granularity for 64KB memory regions. The MMU allows to define read, write and execute policies to physical memory regions. This hardware is pivotal to this work to isolate sensitive components. Violations in the MMU access policies results in a data or instruction abort exception, which allows the processor to recover or reset.

5.3 Threat Model

In this work, the adversary model consists of attackers able to exploit memory corruption vulnerabilities capable of reading and writing arbitrary locations in memory. To successfully launch an

attack, attackers can inject and execute their own code or misuse existing code. More importantly, this threat model considers non-control-data attacks thus, an attacker can commence an attack, without ever modifying the program's control flow. Note this is a powerful threat model considering not only all types of control-flow hijacking attacks, but also sophisticated non-control-data attacks such as Data-Oriented Programming [13].

Similarly to HyperSafe [89], the hardware is considered trustworthy, especially, the secure boot mechanism of the Zynq-7000 SoC and the ARM MMU. Secure boot is essential to guarantee the integrity of the μRTZVisor hypervisor and the proposed security mechanism, at boot-time. The hypervisor code is assumed to be vulnerable, due to its implementation in C++. In this model, it is assumed that the attacker, failing to compromise the hypervisor, will attempt to attack the deployed security mechanisms. Hardware-related attacks are out of the scope of this thread model.

5.4 Proposed Approach

The ascension of virtualization associated with the ever-growing complexity of today's systems makes it impossible, in practice, to determine every failure mode, security vulnerability or to test all possible software behaviors. The following approach presents a runtime defense mechanism against data-plane oriented attacks. Executing at the same privilege level as the hypervisor, a *Remote Monitor*, as depicted in Fig. 18, is introduced to intermittently verify data integrity rules, provided - at design time - by the μRTZVisor's development team. The specification aims to ensure the correct state of all critical static variables at any given time which, in turn, safeguard μRTZVisor's runtime data. The Remote Monitor is isolated from μRTZVisor leveraging the MMU to write-protect sensitive data and enforce a W$\oplus$E policy. Further, the introduction of a CFI scheme ensures that the target program follows a statically-defined CFG.

The Remote Monitor's *Information Collector* retrieves an execution trace, generated by the instrumented μRTZVisor, containing information about write operations to critical static variables to enforce data integrity rules. This execution trace consists of both values written to critical variables as well as addresses used for indirect write operations. This scheme follows a lazy approach, as information is only collected when a write to a critical static variable occurs, in order to minimize performance overhead. The *Ring Buffer* is an intermediary, transferring information from the *Hypervisor* to the *Remote Monitor*. Through compile-time instrumentation, extra instructions are injected into the *Hypervisor*, logging sensitive information in the *Ring Buffer* every time a write to sensitive variables occurs.

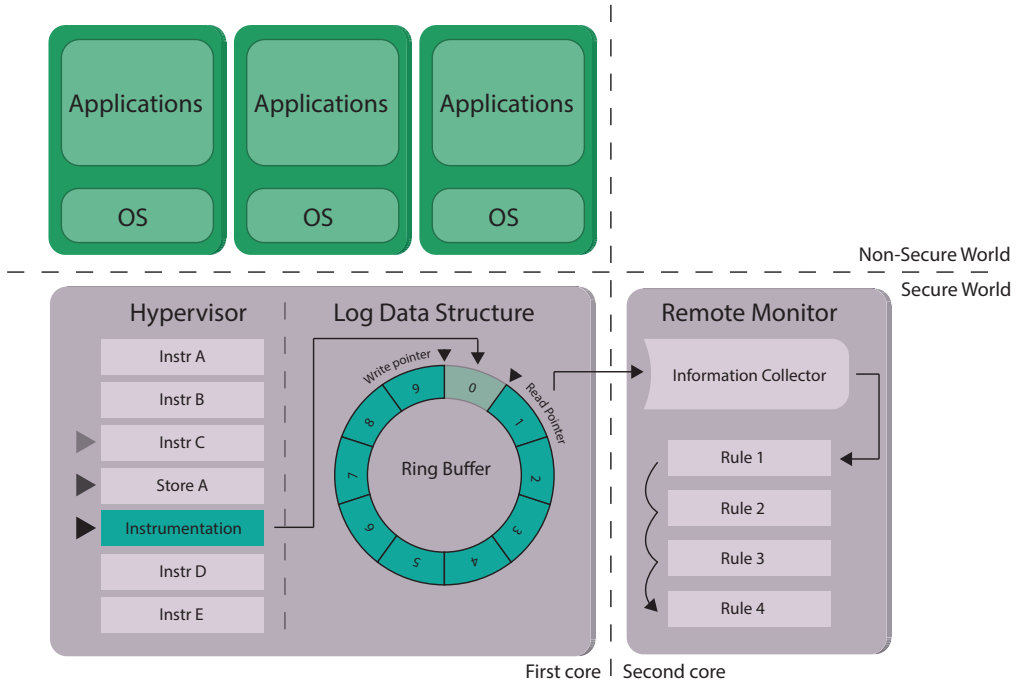


Fig. 18: Architecture of the Data Integrity security mechanism.

5.5 Implementation

The implementation is divided in two parts. μRTZVisor is foremost instrumented, at compile-time, by an extended GCC compiler. Then, the code for the Remote Monitor is automatically generated, using developer-provided abstract data integrity rules. Delving into specifics, two extensions were added to the GCC compiler: the *Type Analyzer* and the *Instrumentation Pass*. The former analyzes every developer-defined data type (e.g, any struct/class definition), essentially creating the memory layout of the program. The layout, containing developer-defined data types and offsets is then stored in a file (*Memory Layout*), as depicted in Fig. 19. This information is subsequently used to translate abstract data integrity rules to C++ code, creating the Remote Monitor (*RM*). Abstract rules will be discussed further down. The *Instrumentation Pass* analyzes code currently under compilation, in its intermediate GIMPLE IL representation, injecting instrumentation as required. Likewise, the *Instrumentation Pass* logs its operations to a file (*Instrumentation Metadata*), used by the *Rule Mapper* to create the aforementioned *C++ Rules*. The *Instrumentation Pass* and *Type Analyzer* are independent entities.

The *Rule Mapper* maps abstract data rules to C++ using information provided by the previous compilation process as well as the developer (i.e., the abstract rules). Rules need to be mapped to C++ since they are developer-defined, in an abstract way, and must be inserted - in the Remote

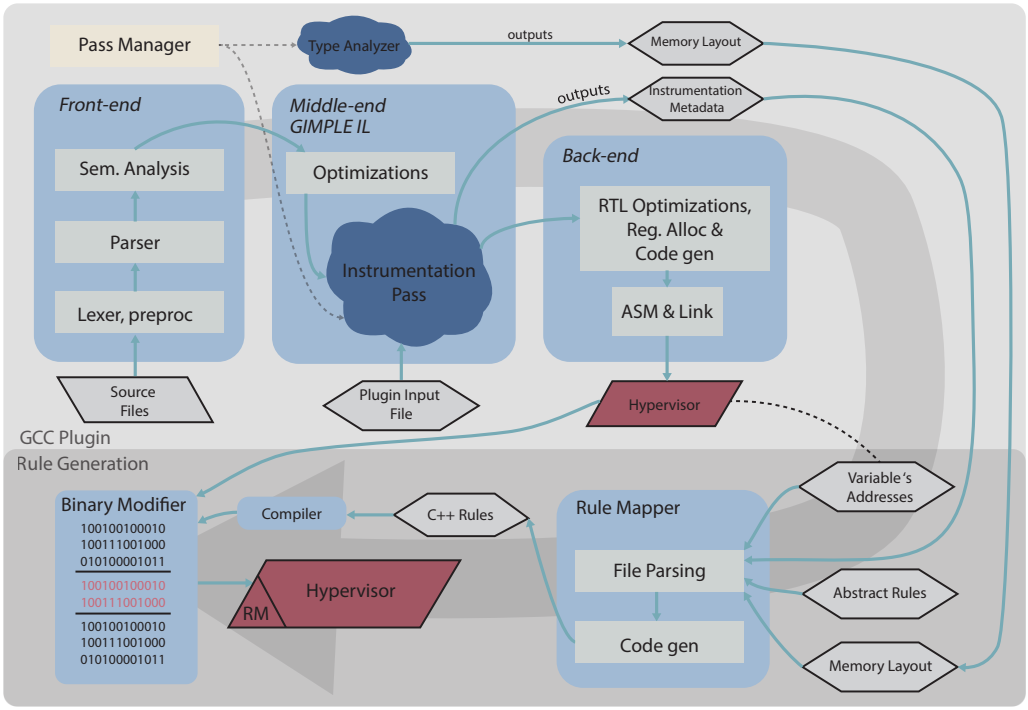


Fig. 19: Implementation steps to generate the Remote Monitor for data integrity. Source files are provided to the compiler's Front-end. During compilation, the Type Analyzer and Instrumentation Pass interpret and modify each source file, respectively. In the second step, Rule Generation, the Remote Monitor (RM) is generated and inserted into the previously compiled Hypervisor code.

Monitor - in the final executable, alongside the hypervisor. To create the Remote Monitor, the C++ rules are compiled and a binary blob is inserted into the hypervisor, in the *Binary Modifier*. This is the final step in the compilation chain. Fig. 19 also provides insight on current chapter's organization. Instrumentation will be discussed at the beginning, followed by the abstract rules and the rule mapping (from abstract to C++). Finally, the generation of the final binary, with the Remote Monitor, will be discussed.

5.6 Structure of the Log Data Structure

A log is a ring buffer data structure, filled by the instrumented hypervisor code, as depicted in Fig. 18. In the current implementation, two virtual logs are present: the *Value Log* and the *Address Log*. In practice, there is only one ring buffer but data flows to one or both virtual logs according to the information being stored: the former logs the values written to critical static variables while the latter logs data used in indirect writes. Note that only information related to data writes is relevant. To fathom the Address Log, a glimpse of the data it stores is fundamental. An indirect write is any

memory write whose final address cannot be calculated statically (i.e., pointers and arrays indexed with variables). An indirect write is any memory write operation whose final address cannot be calculated statically (i.e., pointers and arrays indexed with variables). Since instrumentation is inserted in a GCC's intermediary representation (i.e., GIMPLE IL), information such as the variables used to index arrays is available and can be used for logging purposes. The contents of both logs are depicted in Fig. 20.

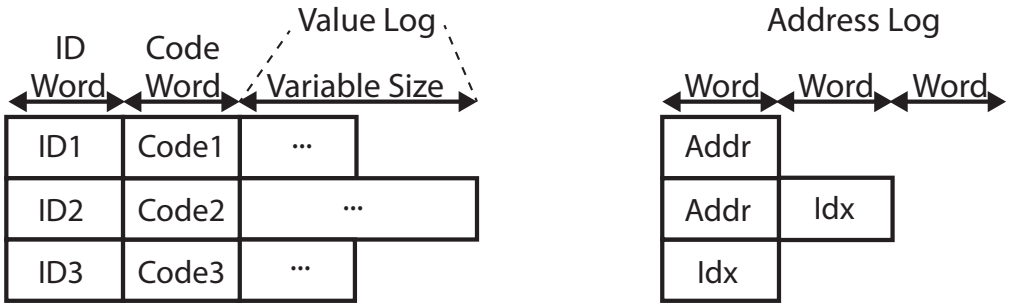


Fig. 20: Structure of the Log Data Structure. Each line represents a log entry.

On each new log entry, a word-sized *ID* is stored. This uniquely identifies a specific location in the code where data is collected. The Value Log makes a copy of the written data immediately after the *Code*. The Address Log stores an address and/or an index (e.g., when a variable is used as an array index). An entry in the Value Log does not require an equivalent entry on the Address Log and vice-versa. Nevertheless, both logs are indexed by the same pointer, effectively creating one log. The Code field can have three values. If the value is zero or one, only the Value Log or the Address Log was filled, respectively. If the value is two, both logs have a new entry. In the implementation of the Value Log, variable-size variables were considered; however, only instrumentation to log a single 32-bit word was implemented. This was enough to enforce the data integrity constraints on the current version of the μRTZVisor hypervisor. The fields of both logs will be made clear when considering the instrumentation, in the next section. The Value Log requires 2KB of memory, encompassing sixteen log entries of 128 bytes each. The Address Log requires 64 bytes, including, at most, three bytes per log entry and sixteen log entries. Both logs are organized in memory as portrayed in Fig. 24. They are a continuous chunk of memory with a *Write Pointer* pointing to the next position to be written. The Log Data Structure memory is configured as write-protected (enforced by the MMU), to disallow log tampering via attacks aiming the security mechanism.

5.7 Hypervisor's Instrumentation

Instrumentation is unavoidable to record the execution trace of the hypervisor, at run-time, and register it to the Log data structure to be further used for data integrity rule enforcement, as depicted in Fig. 18. With two options available, instrumentation at compile-time was chosen instead of binary patching. Compile-time instrumentation consists of extending the compiler with extra functionalities to generate supplementary code. Several reasons support this decision. Firstly, the full source code of μRTZVisor is available. Also, process automation is made simpler than with binary patching. Using a compiler for instrumentation purposes provides an infrastructure to insert code.

With a GCC plugin-based compiler pass, it is possible to provide custom code to GCC, fully interoperable with the remaining compilation process. All internal processing in GCC is controlled by its *Pass Manager* (Fig. 19), while a pass refers to a certain transformation applied to the internal representation of the current compilation unit (i.e., file) [117]. The Instrumentation Pass is a compiler pass registered in the Pass Manager.

The logging instrumentation is straightforward and inserted in assembly, thus being platform specific. The algorithm begins by loading the value of the Log's Write Pointer. After performing arithmetic to calculate the address of the new log entry, the ID and Code are always stored. Then, depending on the data to be logged, the respective logs are updated. In the end, the Write Pointer is incremented to point to the next log entry. Listing 5.1 depicts a commented example of the aforementioned procedure for a new entry on the Value Log. In the case of indirect writes, the Address Log is also updated using a similar algorithm. However, the ID and Code will not be inserted by the Address Log logging procedure if the Value Log is also updated.

Listing 5.1: Example of the assembly logging procedure for the Value Log.

```
1  push {r0-r5}
2  mov      r5, r0              //Copy the integer to be logged (r0) to r5
3  movw     r3, #0              //Load the address of the Write Pointer to r3
4  movw     r0, #31833          //Store the ID in r0
5  movt     r3, #65535
6  movt     r0, #0
7  ldr      r2, [r3]            //Load the current write index value to r2
8  lsl      r2, r2, #9          //Calculate offset from beggining of Value Log
9  movw     r4, #4              //Add the offset to the Value Log base address
10 movt     r4, #65535
11 add      r4, r2, r4
12 str      r0, [r4]            //Store the ID in the log entry
13 add      r4, r4, #4
14 mov      r0, #0              //Store the Code in the log entry
15 str      r0, [r4]
16 add      r4, r4, #4
17 str      r5, [r4]            //Store the written value
18 ldr      r2, [r3]
19 add      r2, r2, #1          //Increment the write pointer and store it
20 and      r2, r2, #15
21 str      r2, [r3]
22 pop {r0-r5}
```

After explaining how data is inserted into the logs, the following section discusses the Instrumentation Pass, focusing on explaining when and how instrumentation is inserted.

5.7.1 Detecting Store Operations in GIMPLE

The Instrumentation Pass works upon GIMPLE IL and after all GCC's Middle-End optimizations. GIMPLE IL was chosen due to its suitable expressiveness, balancing an approximation to machine code (e.g., three operand representation) and enough high-level source code information for the purpose. Running the pass after all Middle-End optimizations results in working with a program surprisingly close to the final. This guarantees that unnecessary instrumentation is not inserted, as all present memory-related operations will translate into the final executable. An empirical analysis on the GIMPLE-based intermediate representation of μRTZVisor led to a categorization of all memory writes: Direct Writes, Partially Indirect Writes and Totally Indirect Writes. This categorization is extensively used throughout this thesis.

Direct Write

A Direct Write is a memory write to a LHS operand, in a GIMPLE assign, whose address is static. Since the location of the write operation is known beforehand, only the Value Log gets filled when a direct write occurs. If the attributed value is a constant, there is no need to even make a copy of the written value to the Value Log, for rule verification. In that case, no log operation occurs whatsoever. Listing 5.2 exhibits a C++ code snippet demonstrating this write type (please refer to line 9), extracted from the μRTZVisor hypervisor source code.

Listing 5.2: Direct write example extracted from μRTZVisor's source code.

```
1  void Scheduler::Schedule(){
2
3  GuestManager& gm = GuestManager::GetInstance();
4  int32_t next = gm.mCurrentGuest + 1;
5  ...
6          /* Check if is a diferent guest */
7          if( gm.mCurrentGuest != next){
8                  gm.ContextSave();
9                  gm.mCurrentGuest = next;
10                 gm.ContextRestore();
11         }
12 }
```

Analyzing the *Right-Hand Side (RHS)* on this attribution, a non-constant integer variable (`next`) is found. Thus, its value must be stored in the Value Log to enforce any rule associated with the `gm.mCurrentGuest` critical variable. On the other hand, the LHS `gm.mCurrentGuest` operand's address is constant – or statically known - because the C++ reference `gm` always points to a GuestManager class static variable named `instance` of this same type, caused by the sin-

gleton design pattern. `mCurrentGuest` is an integer member of the GuestManager class. This can be concluded by analyzing the optimized version of this code, in GIMPLE IL, presented in Listing 5.3. On line 3, the temporary variable `_16` (equivalent to `next` in C++) is being assigned to the `instance.mCurrentGuest` variable. Some insight on the instrumentation procedure can now be provided. Firstly, the LHS operand of a GIMPLE assign is analyzed to detect the write type. Secondly, the RHS is probed for constants. If constants are found they are not logged, since their value will never be spurious and, therefore, harmful.

Listing 5.3: Snippet of GIMPLE code equivalent to lines 8,9 and 10 of listing 5.2.

```
1 <bb 6>:
2 urtzvisor::GuestManager::ContextSave (&instance);
3 instance.mCurrentGuest = _16;
4 urtzvisor::GuestManager::ContextRestore (&instance); [tail call]
```

On the final assembly file, partially depicted in Listing 5.4, the inserted assembly instructions to log the `next` variable into the Value Log are depicted. In fact, the value of the LHS operand of the assignment in line 3, in Listing 5.3, is read and copied to the Value Log after its value gets updated with `_16`.

Listing 5.4: Assembly code generated to insert a word-sized value in the Value Log.

```
1  .LVL22:
2  .loc 1 47 0
3  str      r4, [r5, #1040]
4  .syntax divided
5  @ 47 "src/core/Scheduler.cpp" 1
6  push {r0-r5}
7  mov      r5, r4
8  movw     r3, #0
9  movw     r0, #29204
10 movt     r3, #65535
11 movt     r0, #0
12 ldr      r2, [r3]
13 lsl      r2, r2, #9
14 movw     r4, #4
15 movt     r4, #65535
16 add      r4, r2, r4
17 str      r0, [r4]
18 add      r4, r4, #0
19 mov      r0, #0
20 str      r0, [r4]
21 add      r4, r4, #4
22 str      r5, [r4]
23 ldr      r2, [r3]
24 add      r2, r2, #1
25 and      r2, r2, #15
26 str      r2, [r3]
27 pop {r0-r5}
28
29 @ 0 "" 2
30 .loc 1 48 0
31 .arm
32 .syntax unified
```

Partially Indirect Write

A Partially Indirect Write is a direct write with an indirection in the form of an offset. For example, an access to a static array using a variable as an index can be defined as such. Of course, accessing arrays with constant indexes or accessing non-array C++ class members does not create a Partially Indirect Write. An access of this type can create entries on the Value Log and Address Log. An entry, in the Address Log, will only contain the variable offsets. The statically known base address, associated with a static variable, is used with the offset to determine the final write destination. Listing 5.5 provides an example where the `shared_interupts` static object directly affects its internal array of integers named `mBitmap`, by invoking the inline method `SetBit`.

By analyzing the optimized version of the intermediate representation of this code in GIMPLE (Listing 5.6), it is relatively easy to infer that the integer variable `_23` is being assigned to the `mBitmap` array belonging to the `shared_interrupts` static object, using a variable index `_15`. Thus, for logging purposes both `_23` and `_15` values will be logged to the Value and Address Log, respectively. The logging code, in assembly, is similar to the one present in Listing 5.6 but duplicated, one time for each log.

Listing 5.5: Example of a Partially Indirect Write extracted from *µ*RTZVisor's source code.

```
1 int32_t InterruptManager::SetShared(const uint32_t id){
2
3 ...
4
5 shared_interrupts.SetBit(id);
6
7 ...
8
9 }
```

Listing 5.6: Snippet of GIMPLE code equivalent to line 5 of Listing 5.5.

```
1 _22 = 1 << _9;
2 _23 = _11 | _22;
3 shared_interrupts.mBitmap[_15] = _23;
```

Totally Indirect Write

Totally Indirect Writes encompass any write to a static variable whose address is not known at compile-time. They can be as simple as a pointer containing multiple variable' addresses or more complicated and also contain offsets like in the above-mentioned case. Listing 5.7 denotes such an example: a memory write to a statically unknown object using a variable to index an array. Line 7 demonstrates the usage of the dot operator on the `rguest` C++ reference whose value is determined at run-time, as it can be seen in its declaration in line 5. `GuestManager::GuestCreate` is

a class method and following the C++ convention [118], it uses the "this" pointer to refer to any object of the GuestManager class. Since the `guestList` array belongs to GuestManager, an indirect access via the "this" pointer must occur to determine the object in question. Here, the `guestList` array is indexed via the `guest_num` variable. Thus, to determine the GuestManager object and respective field being accessed, the "this" pointer value and the `guest_num` variable must be analyzed. Listing 5.8 exhibits the optimized code in the GIMPLE IL. In line 3 we can see the "this" pointer (`this_6(D)`) and the `guest_num` variable in a temporary variable (`prehitmp_81`) being used to calculate the write address. The values of both variables are logged in the Address Log and the `guest_num.0_19`, an integer, goes to the Value Log.

Listing 5.7: C++ code performing a Totally Indirect Write (extracted from *µ*RTZVisor).

```
1  void GuestManager::GuestCreate(GuestConfig const &config){
2
3  static int32_t guest_num = 0;
4
5  Guest &rguest = guestList[guest_num];
6
7  rguest.id = guest_num;
8  ...
9  return;
10 }
```

Listing 5.8: Snippet of GIMPLE code equivalent to lines 5 and 7 of Listing 5.7.

```
1  ...
2  guest_num.0_19 = guest_num;
3  MEM[(struct Guest &)this_6(D)].guestList[prephitmp_81].id = guest_num.0_19;
4  ...
```

5.7.2 Identifying Writes to Critical Variables

Identifying all writes to static variables upon which rules must be enforced is a prerequisite. The Instrumentation Pass requires a list of critical variables as an input file (Fig. 19), since these variables vary according to the defined rules. Note, not all writes to static variables are to be instrumented since it would create an overwhelming performance overhead. Above all, hypervisor developers should develop a minimal set of rules which guarantee complete integrity of the hypervisor. Using this file, the Instrumentation Pass tries to match a human-readable C++ style file - with the critical variables -, to the LHS operands of GIMPLE assignments. Listing 5.9 depicts an input file for *µ*RTZVisor. Critical static variables are uniquely identified by their scope (i.e., the "::" operator) and name.

In order to analyze a program that involves pointers it is necessary to have (safe) information about what each pointer points to [119]. Many compiler analyses and optimizations require in-

Listing 5.9: Example of an Input File for the Instrumentation Pass.

```
1 ::urtzvisor::GuestManager::instance.guestList.memoryConfig.bitmap
2 ::urtzvisor::GuestManager::instance.guestList.interruptConfig.bitmap
3 ::urtzvisor::GuestManager::instance.mCurrentGuest
4 ::urtzvisor::InterruptManager::handler_table
5 ::urtzvisor::InterruptManager::shared_interrupts.mBitmap
6 ::urtzvisor::InterruptManager::private_interrupts.mBitmap
7 ::urtzvisor::InterruptManager::private_interrupt_attribution
8 ::urtzvisor::Hypercalls::hypercall_table
9 from F8F00000 to F9F00000
```

formation about which objects each pointer in a program may point to at run-time [120]. With Totally Indirect Writes, the Instrumentation Pass needs to infer the objects a pointer can point-to since rules are only applied to objects. Internally, GCC performs points-to and escape analysis [30] but this data is not complete. To address this problem, a file containing the set of objects pointed by every pointer is provided, as partially depicted in Listing 5.10. The manually created points-to analysis allows the Instrumentation Pass to detect writes to critical static variables on pointers (i.e., Totally Indirect Writes) to add the necessary instrumentation. In Listing 5.10, a filename and the line in the source code uniquely identify a Totally Indirect Write. Each line below the identifier corresponds to an accessed static variable, by the pointer. The `[UNK]` corresponds to an unknown array index (i.e., calculated at run-time) in the chain of references to class member variables. Points-to analysis is not the focus of this thesis; however, it is crucial when evaluating Totally Indirect Writes on the Instrumentation Pass. Without this analysis, the pass could not infer the objects pointed by each pointer and several modifications on critical data would go undiscovered and unverified. To ensure completeness, if the Instrumentation Pass encounters such an access, which is not present in the points-to file, it immediately interrupts compilation.

Listing 5.10: A partial points-to analysis file for *µ*RTZVisor.

```
1 src/core/InterruptManager.cpp:172:
2 ::urtzvisor::GuestManager::instance.guestList.vm.mVgic.vICDISPRx[UNK]
3 src/core/Scheduler.cpp:47:
4 ::urtzvisor::GuestManager::instance.mCurrentGuest
5 src/lib/inc/types.h:94:
6 ::urtzvisor::GuestManager::instance.guestList[UNK].memoryConfig.bitmap[UNK]
7 ::urtzvisor::GuestManager::instance.guestList[UNK].interruptConfig.bitmap[UNK]
```

5.7.3 Instrumentation Metadata

As instrumentation code is inserted in the VMM software, the Instrumentation Pass creates a log of all its operations (the *Instrumentation Metadata* file in Fig. 18). Each possible log entry comprises the filename and the line of the instrumented write in the C++ source code (*Source Code Location*), a C++-similar grammar to identify the written static variable (*Reference*), an ID (*ID*)

to uniquely identify the logging operation and the type of log performed by the instrumentation (*Log Type*). Direct and Partially Indirect Writes are prefixed with *STATIC*, in the *Log Type*. Totally Indirect Writes are prefixed with *INDIRECT*. The *VAL* means that the written value is being copied to the Value Log and each *INDEX*, determines a word-sized entry on the Address Log either with memory addresses or array indexes. Many of the Totally Indirect Writes only access one static variable, due to non-controllable VMM design decisions. The information contained here is used by the Rule Mapper entity, depicted in Fig. 18, to automatically create the C++ integrity rules and the Remote Monitor.

Source Code Location	References	ID	Log Type	
src/core/InterruptManager.cpp:50:	::urtzvisor::InterruptManager::handler_table[UNK]	4888	STATIC_VAL_INDEX	
src/lib/inc/types.h:94:	::urtzvisor::InterruptManager::private_interrupts.mBitmap[UNK]	6102	STATIC_VAL_INDEX	
src/core/InterruptManager.cpp:77:	::urtzvisor::InterruptManager::private_interrupt_attribution[UNK]	23009	STATIC_VAL_INDEX	
src/lib/inc/types.h:94:	::urtzvisor::GuestManager::instance.guestList[UNK].memoryConfig.bitmap[UNK] ::urtzvisor::GuestManager::instance.guestList[UNK].interruptConfig.bitmap[UNK]	31833	INDIRECT_VAL_INDEX	
src/lib/inc/types.h:94:	::urtzvisor::InterruptManager::shared_interrupts.mBitmap[UNK]	6103	STATIC_VAL_INDEX	
src/core/InterruptManager.cpp:124:	::urtzvisor::GuestManager::instance.guestList.vm.mVgic.vICDISPRx[UNK]	21246	INDIRECT_VAL_INDEX	
src/core/InterruptManager.cpp:157:	::urtzvisor::GuestManager::instance.guestList[UNK].vm.mVgic.vICDISPRx[UNK]	29109	STATIC_VAL_INDEX_INDEX	
src/core/InterruptManager.cpp:160:	::urtzvisor::GuestManager::instance.guestList[UNK].vm.mVgic.vICDISPRx[UNK]	18007	STATIC_VAL_INDEX_INDEX	
src/core/InterruptManager.cpp:169:	::urtzvisor::GuestManager::instance.guestList.vm.mVgic.vICDISPRx[UNK]	31795	INDIRECT_VAL_INDEX	
src/core/InterruptManager.cpp:172:	::urtzvisor::GuestManager::instance.guestList.vm.mVgic.vICDISPRx[UNK]	24051	INDIRECT_VAL_INDEX	
src/core/Hypercalls.cpp:18:	::urtzvisor::Hypercalls::hypercall_table[UNK]	28345	STATIC_VAL_INDEX	
src/core/GuestManager.cpp:21:	::urtzvisor::GuestManager::instance.mCurrentGuest	21306	INDIRECT_INDEX	
src/core/GuestManager.cpp:31:	::urtzvisor::GuestManager::instance.guestList[0].id	9941	INDIRECT_INDEX	
src/core/GuestManager.cpp:31:	::urtzvisor::GuestManager::instance.guestList[0].id	9941	INDIRECT_INDEX	
src/core/GuestManager.cpp:56:	::urtzvisor::GuestManager::instance.guestList[UNK].id	3474	INDIRECT_VAL_INDEX_INDEX	
src/core/GuestManager.cpp:56:	::urtzvisor::GuestManager::instance.guestList[UNK].id	3474	INDIRECT_VAL_INDEX_INDEX	
src/core/Scheduler.cpp:47:	::urtzvisor::GuestManager::instance.mCurrentGuest	29204	STATIC_VAL	
src/core/Scheduler.cpp:47:	::urtzvisor::GuestManager::instance.mCurrentGuest	29204	STATIC_VAL	
src/.../cortexA9/PrivateTimer.cpp:44:	PrivateTimerRegs::ANONYMOUS.pt_control_reg	29807	STATIC_VAL	ADDR:f8f00600
src/.../cortexA9/PrivateTimer.cpp:53:	PrivateTimerRegs::ANONYMOUS.pt_control_reg	30599	STATIC_VAL	ADDR:f8f00600
src/.../cortexA9/PrivateTimer.cpp:63:	PrivateTimerRegs::ANONYMOUS.pt_load_reg	19606	STATIC_VAL	ADDR:f8f00600

Fig. 21: Instrumentation Pass meta-data for the critical variables depicted in Listing 5.9.

5.7.4 Protecting the Logs using the MMU

Write-protecting Logs, with the MMU, requires auxiliary code to enable writes in benign situations (i.e., when a log gets legitimately updated). Thus, an extra piece of instrumentation is inserted to both enable and disable the MMU, as required. Since instrumentation is always inserted after a write, the instructions present in Listing 5.11 are inserted after the write and before any attempt to write into a log. The same set of instructions, with the second instruction in line 2, is used to re-enable the MMU after all log operations. These security sensitive instructions are added for every log operation. With a disabled MMU the security mechanism itself is left unprotected. This code also shows that the MMU cannot be disabled by accessing memory mapped registers. Instead, a special interface is used (CP15).

Listing 5.11: Disabling and re-enabling the MMU In line 2, The ”bic” instruction disables the MMU while the ”orr” enables it.

```
1 mrc     p15, 0, r1, c1, c0, 0
2 bic     r1, r1, #1 | orr      r1, r1, #1
3 mcr     p15, 0, r1, c1, c0, 0
4 dsb
5 isb
```

5.8 Extracting Target Program's Memory Layout

Alike the Instrumentation Pass, the Type Analyzer uses GCC's Plugin API to detect new developer-defined data types, after program parsing. All developer-defined types are extracted from the target's program and stored in a file (*Memory Layout* in Fig. 18), to be used afterwards by the Rule Mapper. Classes and structures contain contiguous member fields - excluding padding for alignment purposes -, according to the C++ object model [55]. Furthermore, C++ classes are higher level and feature-enriched versions of C structures. Thus, classes are considered equivalent to C structures, for the purpose of displaying target's program memory layout. The information portrayed in the Memory Layout file is used by the Rule Mapper software, aiding in rule semantic validation and in performing arithmetic to detect specific locations in memory. A fragment of the VMM's developer-defined data types is depicted in Listing 5.12. Firstly, the keyword `struct` defines a class or structure data type, followed by the name of the type and size. Secondly, `struct`'s `Fields` are described by a 1) meta-type, a 2) type, a 3) name and optionally a 4) number of array elements if the type is an array. The `record_type` meta-type includes all C++ structures and classes, `array_type` all array types (a new array type is declare every time an array is declared) and `integer_type` only defines the integer type. Lastly, the offset from the beginning of the structure to a field is specified to calculate the final offset.

Listing 5.12: Partial memory layout file for *μ*RTZVisor.

```
1  struct Guest size 508 [bytes]
2          Field> integer_type int id
3                  Bytes: 0                Bits: 0
4          Field> array_type int* name size 32
5                  Bytes: 0                Bits: 32
6          Field> record_type MemoryConfiguration memoryConfig
7                  Bytes: 32               Bits: 32
8          Field> record_type InterruptConfiguration interruptConfig
9                  Bytes: 48               Bits: 0
10         Field> record_type VirtualMachine vm
11                 Bytes: 56               Bits: 32
12 struct GuestManager size 1020 [bytes]
13         Field> array_type Guest* guestList size 2
14                 Bytes: 0                Bits: 0
15         Field> integer_type int mCurrentGuest
16                 Bytes: 1016             Bits: 0
17 struct ::urtzvisor::InterruptManager::shared_interrupts size 12 [bytes]
18         Field> array_type int* mBitmap size 12
19         Bytes: 0                Bits: 0
20 struct ::urtzvisor::InterruptManager::private_interrupts size 12 [bytes]
21         Field> array_type int* mBitmap size 12
22                 Bytes: 0                Bits: 0
```

While the compiler can be programmed to generate this memory layout information every time it compiles a target program (e.g., μRTZVisor), that is not required, unless source code changes entail modifications on data types. Data type information is persistent between compilations if nothing else is modified (e.g., target platform). Hence, the automatically generated memory layout file is manually edited to add additional information about static variables defined in the source code, which couldn't be yet gathered by the Type Analyzer. This is a relatively easy step compared to manually specifying data types. In fact, all VMM's static variables are depicted in Listing 5.13. The `static` keyword defines a static variable followed by its name. Then the variable's meta-type and type are specified. In the case of arrays, the number of elements is also specified.

Listing 5.13: Manually extended memory Layout file to contemplate declarations of static variables.

```
1  static ANONYMOUS record_type PrivateTimerRegs
2  At      0xf8f00600
3
4  static ::urtzvisor::InterruptManager::shared_interrupts record_type TEMPLATE Bitmap
5
6  static ::urtzvisor::InterruptManager::private_interrupts record_type TEMPLATE Bitmap
7
8  static ::urtzvisor::InterruptManager::handler_table array_type integer_type
9         elements 96
10
11 static ::urtzvisor::InterruptManager::private_interrupt_attribution array_type integer_type
12        elements 96
13
14 static ::urtzvisor::GuestManager::instance record_type GuestManager
15
16 static ::urtzvisor::Hypercalls::hypercall_table array_type integer_type
17        elements 32
```

Every time a declared variable has a type resulting from a C++ template instantiation, a new data type is also manually created. Variables with such types also possess the keyword `TEMPLATE` in their definition. This keyword states that a static variable has a type with the same name as the variable itself. Both `::urtzvisor::InterruptManager::shared_ interrupts` and `::urtzvisor::InterruptManager::private_interrupts` objects have manually defined types, in the Memory Layout file, with the same name as the variables, as depicted in Listing 5.12. Automatically generating the complete file, from the compilation process, will be implemented in future.

5.8.1 Retrieving Static Variable Addresses

The addresses of static variables are modified every time a program is compiled. The memory location of these variables is necessary to enforce data integrity rules. The GNU NM [121] utility allows to automatically obtain those addresses, by inspecting symbols in object files. By providing the target program to NM, this utility is configured to list static variables' location while

also decoding (demangling) low-level symbol names into C++ high-level names. The Remote Monitor uses these addresses to identify the accessed critical static variables in Totally Indirect Writes. The NM utility becomes part of μRTZVisor's compilation process so that, if the target program's source code is compiled, the addresses are updated and stored in a file. For a specific compilation of μRTZVisor, the partial output of NM, depicted in Listing 5.14, highlights several static variables with their respective addresses in bold.

Listing 5.14: Using the GNU NM utility to discover addresses of static variables.

```
1  ...
2  001077e0          B urtzvisor::Gic::instance
3  001077e4 B urtzvisor::GuestManager::instance
4  00107bf8 b urtzvisor::GuestManager::GuestCreate(urtzvisor::GuestConfig const&)::guest_num
5  00107bfc B urtzvisor::Hypercalls::hypercall_table
6  00107c7c B urtzvisor::Hypercalls::instance
7  00107c80 B urtzvisor::InterruptManager::handler_table
8  00107e00 B urtzvisor::InterruptManager::private_interrupt_attribution
9  00107f80 B urtzvisor::InterruptManager::private_interrupts
10 00107f8c B urtzvisor::InterruptManager::shared_interrupts
11 00107f98 B urtzvisor::InterruptManager::instance
12 00107f9c B urtzvisor::MemoryImplementation::instance
13 00107fa0 B urtzvisor::MemoryManager::instance
14 00107fa4 B urtzvisor::PrivateTimer::instance
15 00107fa8 B urtzvisor::Scheduler::instance
16 00107fac B urtzvisor::TzMsgManager::mInstance
17 00109ef8 B urtzvisor::TzMutexManager::mInstance
18 0010a410 B urtzvisor::TzPortManager::mInstance
19 0010a81c B __bss_end
20 0010a81c T __init_array_start
21 0010a820 B _init
22 0010a820 B __init_end
23 ...
```

5.9 Abstract rules

Highly expressive data integrity rules can be created as a result of the sophisticated instrumentation process. The devised rules emerged from developer's requirements to secure the μRTZVisor VMM. Nevertheless, the rule set can be extended as required, considering the modularity of the Rule Mapper software, detailed in Section 5.10.1. The rules applied to μRTZVisor, provided by the developers, are depicted in Listing 5.15. The first rule, Immutable Vector Element (`immutable_vec _element`), write-protects a vector element. Particularly, the first value of an integer array member named `bitmap`, associated with the GuestManager `instance` static variable. This kind of granularity is troublesome to implement using currently available hardware (e.g., MMUs commonly have page-level granularity). Write-protecting an internal part of a continuous memory block [55] is complicated without almost word-level granularity write-protection, except by changing program's memory layout which could induce new errors. The Immutable (`immutable`) rule is a generalization of the former, creating read-only variables. As depicted in line 2 and 3, class member fields can be write-protected. Entire objects with class or structure types can also be protected. Thus, greater granularity over field insensitive methods, like WIT

[91], is provided. Further, the Immutable rule works on basic-type arrays and variables as illustrated in line 4 and 5, as long as they are statically defined. Both variables, on those two lines, are arrays containing function pointers.

Listing 5.15: Abstract rules for μRTZVisor.

```
1 immutable_vec_element ::urtzvisor::GuestManager::instance . guestList . memoryConfig . bitmap [0]
2 immutable ::urtzvisor::InterruptManager::private_interrupt_attribution
3 immutable ::urtzvisor::InterruptManager::shared_interrupts . mBitmap
4 immutable ::urtzvisor::Hypercalls::hypercall_table
5 immutable ::urtzvisor::InterruptManager::handler_table
6 immutable ::urtzvisor::InterruptManager::private_interrupts . mBitmap
7 immutable ::urtzvisor::GuestManager::instance . guestList . interruptConfig . bitmap
8 register_val_pattern ANONYMOUS. pt_control_reg XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX1
9 range_int ::urtzvisor::GuestManager::instance . mCurrentGuest 0 1
```

References whereat rules are enforced have been constructed using a syntax similar to the C++ notation for navigating through class member fields, using the dot operator. These references constitute the second argument of any rule. As rules are applied to references of static variables, the '->' operator is not used. The '::' stands for scope as in C++. Static variables are uniquely specified using scope and name. In Listing 5.15, static variables are highlighted in bold. The Register Value Pattern (`register_val_pattern`) rule is a fundamental rule type for μRTZVisor, which directly interacts with the hardware via memory mapped registers. This rule uses logged update register operations whereupon it employs a bit-wise mask to verify individual bits values. Line 8 presents a case where the first bit of the *ANONYMOUS.pt_control_reg* reference, mapped to the a hardware timer control register, must always be one, while the remaining bits can take any arbitrary value. Internally to μRTZVisor, this rule ensures that the hardware timer, crucial for scheduling the VMs, is never disabled, counterposing, in such scenarios, possible *Denial of Service (DoS)* attacks. The developer has yet to provide a complete set of rules for registers as the VMM is currently being developed. The final rule type, `range_int`, is self explanatory, ensuring that an integer variable stays between a specified range: 0 and 1.

Abstract rule definition is ongoing work, aiming to create a concrete DSL. The above-ment ioned Instrumentation Pass uses an input file (*Plugin Input File* in Fig. 19), manually defined using the references in this set of rules, to capture required write operations, as earlier stated. Listing 5.9 has the direct mapping from this set of abstract rules to the Instrumentation Pass input file. This reduces the required instrumentation inserted into the target program.

5.10 Mapping Abstract to Run-Time Verifiable Rules

Hypervisor developers are required to supply data integrity rules, ensuring the prerequisites for VMM software safeguard. The former rule set is considered abstract considering it's not constructed in C++, therefore it's non-compilable. Instead, a primitive abstracter language is used

and then, rules are automatically translated to C++. Calling this language a DSL would be an overstatement due to its simplicity. The mapping of abstract rules to C++ is performed by the *Rule Mapper* entity, illustrated in Fig. 19. The C++ rules constitute the most important part of the Remote Monitor. Besides the rules, this process demands a target program's memory layout containing data structures and existing static variables. In the Rule Mapper software, this layout generates an in-memory data model of the current target program - here corresponding to μRTZVisor's structure and class definitions and static variables -, allowing to semantically validate abstract rules, (e.g., check if a rule applies to an existing static variable). Another essential part of this process is to retrieve static variables' addresses after VMM compilation. These addresses are crucial to determine the static variable accessed by any Totally Indirect Write. GNU's NM utility is used for this purpose, as detailed in 5.8.1. At last, instrumentation meta-data is paramount, as it provides information about captured data, originated from the compilation of the target program.

5.10.1 An Overview over the Code Generation Process

In the Rule Mapper, code generation consists of two major steps: preamble creation and rule content generation. The former can be further subdivided into rule attribution and preamble code generation. Rule attribution consists of analyzing the instrumentation meta-data, generated by the Instrumentation Pass (e.g., Listing 21), to infer which rules apply to each possible log entry. So far, only one rule can be associated with a log entry. Preamble code generation consists of generating auxiliary code that allows rules to be applied to Direct, Partially Indirect and Totally Indirect Writes. Depending on the access type, different verifications must be performed to detect the exact memory location where a write has occurred. For example, with Partially Indirect Writes, each array element must be unrolled into individual locations for analysis by rules that work upon vector elements (i.e., the Immutable Vector Element rule). Lastly, rule specific integrity verifications can be added in the rule content generation step.

In the first stage, rule attribution, every developer-defined abstract rule is compared against every log entry, listed in Fig. 21, to conclude where rules are applicable. Rules are compared to log entries by matching references from rules to those on log entries (please refer to Listing 5.15 and Fig. 21). Essentially, every possible log entry is sequentially analyzed, to determine what rules are applied to it. Ultimately, every rule will be related to a log occurrence, since the instrumentation was inserted based on the abstract developer-defined rules and every possible log entry corresponds to a write operation to a reference from one of those abstract rules or to non-sensitive fields of critical static variables.

Preamble code generation is possibly the most complex stage of this process. This is due to the three write types, formerly defined in Section 5.7.1: Direct, Partially Indirect and Totally Indirect write operations. References corresponding to Direct Writes (i.e., *LogType* is prefixed by *STATIC_* in Fig. 21) access a statically known memory address; thus, only the Value Log is updated when a write occurs. The log entry itself is uniquely identified by the ID. The preamble for any Direct Write will be similar to Listing 5.16. The generated preamble code is highlighted in bold: a conditional statement identifying the Direct Write by the ID.

Listing 5.16: Preamble code generated to enforce data integrity rules in a Direct Write.

```
1 if(ID == 29204) {
2         //Rule logic below this comment
3         if(*((int*)ptr_val) >= 0 && *((int*)ptr_val) <= 1) {
4                 return true;
5         }
6 }
```

Partially Indirect Writes have a fixed base address for a static variable and an offset of one or more integer indexes. The preamble code produced for a write to `::tzvizor::Hypercalls::hypercall_table`, a thirty-two element integer pointer array, is depicted in Listing 5.17. This can be easily concluded by analyzing Fig. 21 and matching the IDs. By inspection of Fig. 21, it can also be concluded that an unknown integer index is also stored to the Address Log when this writes occurs. This value is used in preamble code generation to provide bound checking.

Considering that `hypercall_table` cannot be modified when the VMM is executing, if any log occurs with an ID equal to 28345, the immutable rule defined in Listing 5.15 is enforced and the Remote Monitor will signalize an error. Nevertheless, rules are only inserted in the next stage. It's important to notice that the size of the `hypercall_table` array is provided by the extended memory layout file (please refer to Section 5.8) supplied to the Rule Mapper software, since this variable is a C++ static array.

Listing 5.17: Preamble code generated to enforce rules in Partially Indirect Writes.

```
1 if( ID == 28345 && index1 >= 0 && index1 <= 32 ) {
2
3         return false;
4
5 }
```

As aforementioned, this implementation supports until two levels of unknown integer indexes, being able to provide bound checking to an array inside an array. This can be seen in Listing 5.18, as the write operation, with an ID of 29109, corresponds to this exact scenario. No rules are being applied to this write thus, the code on Listing 5.18 just performs bound checking by default. Ideally all indirect accesses, partial or total, that can access critical variables must be bound checked as they can be overflowed and modify critical variables with spurious values and

go unnoticed. Inside structures and classes this problem is specially dangerous due to the data contiguous nature. This problem is further discussed in Section 5.13.

Listing 5.18: Preable code for Partially Indirect Accesses with two indirections in the form of integer indexes.

```
1  if( ID == 29109 && index1 >= 0 && index1 <= 2 ) {
2          if( index2 >= 0 && index2 <= 3 ) {
3                  return true;
4          }
5  }
```

Totally Indirect Writes are similar to Partially Indirect Writes, differing only by not having a fixed base address for a static object. This requires to actually verify memory addresses, instead of just indexes, since the accessed static variable is unknown. Furthermore, this write type can also contain integer indexes as above. In Listing 5.19, the preamble code for a Totally Indirect Write, identified by an ID of 31833, is highlighted in bold.

Listing 5.19: Preamble code generated for Totally Indirect Writes. Unrolling two vectors due to lack of information about the indexes.

```
1  if(ID == 31833 && index1 >= 0x1077e4 && index1 <= 0x107be0){
2          if(index1 >= 0x1077e4 && index1<= 0x1079e0){
3                  int addr = 0x1077e4;
4                  if(index1 >= addr + 0x24 && index1<= addr + 0x28 ){
5                          return false;
6                  }
7                  if(index1 >= addr + 0x28 && index1<= addr + 0x2c ){
8                          return true;
9                  }
10                 if(index1 >= addr + 0x2c && index1<= addr + 0x30 ){
11                         return true;
12                 }
13         }
14         if(index1 >= 0x1079e0 && index1<= 0x107bdc){
15                 int addr = 0x1079e0;
16                 if(index1 >= addr + 0x24 && index1<= addr + 0x28 ){
17                         return false;
18                 }
19                 if(index1 >= addr + 0x28 && index1<= addr + 0x2c ){
20                         return true;
21                 }
22                 if(index1 >= addr + 0x2c && index1<= addr + 0x30 ){
23                         return true;
24                 }
25         }
26 }
```

Although this access can write to two distinct static variables, as depicted in Fig. 21, only the preamble for the first (`::urtzvisor::GuestManager::instance.guestList[UNK].memoryConfig.bitmap[UNK]`) is depicted here, due to lack of space. This static variable (`::urtzvisor::GuestManager::instance`) has a rule, available in line 1 of Listing 5.15, protecting the first element of the *bitmap* array, associated with the static variable, against write operations. The memory layout file is used to calculate the offsets which allow to unroll array elements into individual code sections where rules can be applied with array element granularity. Two vectors,

`guestList` and `memoryConfig` with 2 and 3 elements, respectively, are being decomposed to individual elements so that the Immutable Vector Element rule can be applied to the first element of *bitmap*. Without a complete memory layout of the target program this would not be possible. The code in Listing 5.20 also presents a Totally Indirect Writes but, unlike the previous, an array index is logged and used to avoid any vector element unrolling. In this case, memory addresses are used just to detect the static variable.

A question arises as why not always log integer array indexes to avoid performing the operation depicted in Listing 5.19. Optimally, that would always the be case however, the Instrumentation Pass is restricted on the data it can log, since it works with an intermediary code representation of the target program.

Listing 5.20: Preamble code for a Totally Indirect Write without vector unrolling.

```
1  if(ID == 3474 && index1 >= 0x1077e4 && index1 <= 0x107be0){
2              if( index2 >= 0 && index2 <= 2 ){
3                      return true;
4              }
5  }
```

The last step of code generation inserts the actual C++ code, equivalent to the abstract rules, into the Remote Monitor program. In other words, it accordingly fills the conditional statements, highlighted in bold in the previous listings, that denote locations where rules must be inserted. In an abstract way the Rule Mapper tries to provide a modular architecture where every rule is represented by its own class and provides its own code. Thus, rule's logic is separated from the remaining code. The inherited class *Rule* is used to verify if a rule must be enforced against log entries in the Instrumentation Pass meta-data file (Fig. 21), in a generic way using the *verifyApplicability* method (Fig. 22). This class does so by comparing references to variables, on each log entry, with the references in the definition of abstract rules (Listing 5.15). As it can be seen in the Fig. 22, every rule has its own logic aggregated in individual classes, making it easy to extend when necessary.

Immutable Vector Element & Immutable Rule

The generated code for the Immutable Vector Element rule is as simple as a `return false;` statement for vector elements that cannot be written. No code is generated for allowed writes since, by default, the preamble code generation procedure allows any write to variables whose boundaries are in check. This rule identifies where the square brackets are used in the definition of a rule of this type (see Listing 5.15), and where, in the references of each entry in the Instrumentation Pass meta-data file (Fig. 21), it must block the access.

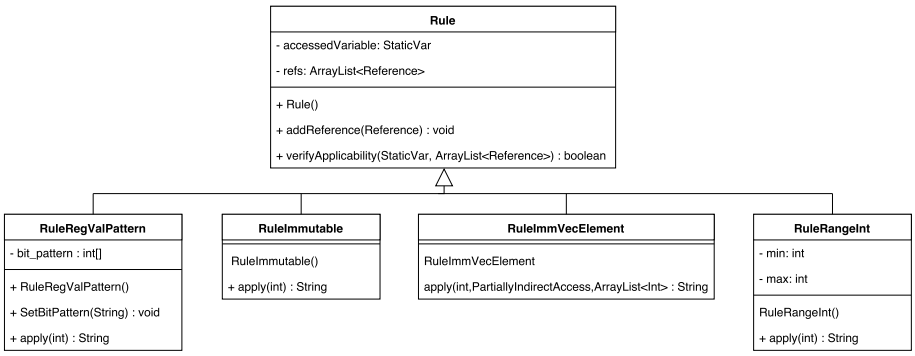


Fig. 22: UML class diagram depicting rule's architecture in the Rule Mapper software. The verifyApplicability method tests if a rule is to be applied to a specific reference. The apply method contains the logic or code associated with each rule. RuleRegValPattern and RuleRangeInt have internal members for their arguments.

The Immutable rule is a generalization and simplification of the previous rule. Likewise, it introduces `return false;` statements in the preamble. However, since a declaration of an Immutable rule cannot contain square brackets (i.e., references to array elements to be protected), different array elements don't need to be discriminated.

Register Value Pattern & Range Integer Rule

This rule is straightforward, it automatically generates a bit mask to ensure that the developer-provided mask, provided in the abstract rule definition (see Listing 5.15), holds on every written register value. It then applies that mask to an "`if`" conditional statement, returning true if the conditions are met or returning an error if not.

The Range Integer rule is similar to the previous one as it also provides a similar `if` statement to verify if written integer values are inside a specific range. This range is also provided in the abstract rule definition.

5.10.2 The Remote Monitor

The Remote Monitor is standalone bare-metal application, distinct from the hypervisor and executing in a separate core from the VMM software. As formerly explained, this application reads both the Address and Value Logs, extracting data for rule enforcement, as required. The Value and Address Logs may not be written at the same time; however, they are synchronized since if one of the logs is not written, an empty entry is inserted. The Remote Monitor uses polling to detect unprocessed log entries, as depicted in Fig. 23. When the *Write Pointer* differs from the *Read Pointer* a log entry is examined and, based on the value of the Code field, data from the

Value Log, Address Log or both is analyzed. Yet, if desired, power efficiency can be enhanced by using interrupt-driven software instead of polling or by applying other optimizations. As new log entries are identified, a verification function - incorporating all data integrity rules - is executed. As new log entries are identified - for either buffer - a verification function, incorporating all data integrity rules, is executed:

```
bool verifyRules(int ID , uint32_t* ptr_add,void * ptr_val);
```

The first argument of the above function prototype is of uttermost significance. The `ID` identifies the log entry being analyzed, and triggers the execution of the individual rules. Subsequent arguments are solely pointers to the Address Log and Value Log with an offset according to the value of the *Read Pointer*. The results of the previous section allow to understand the bulk of this function. The contents of this function are automatically generated by the Rule Mapper (Fig. 19) from the abstract rules defined in Listing 5.15. The code is fully available in appendix A.

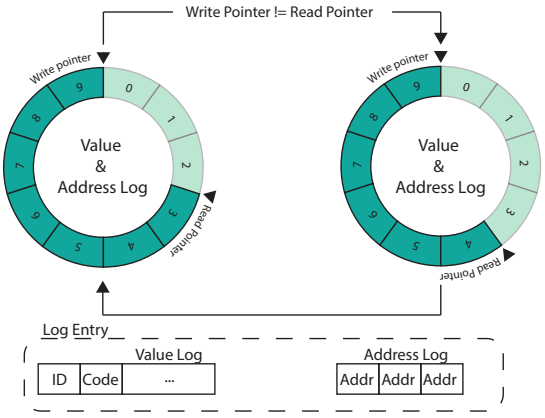


Fig. 23: Remote Monitor's polling process.

5.11 Preparing the Final Executable

The Remote Monitor and the VMM executables must be linked into a single binary image. (Fig. 19). A new section, *.myremote*, is included in the μRTZVisor's object file, using the GNU objcopy utility [122], at the address *0x03200000*. The Remote Monitor code begins at the previous address and can occupy as much as 1 MB. The platform's memory resources are large enough to include this new section, having a 512MB DDR3 memory. Presently, each VM occupies a maximum of 4 MB and, in a system configuration with two VMs, there is more than half a gigabyte available; thus, memory constraints are close to none. However, the section for the Remote Monitor can be shortened and/or reallocated with ease, by providing the desired size and address to the objcopy utility. The filled rectangles, in Fig. 24, depict MMU write-protected memory slots:

the logs and the Remote Monitor code. Logs are stored in the top 64 KB section of the OCM and write-protected with the MMU, which has a maximum granularity equal to the size of this section. Regarding TrustZone, only the Secure MMU interface (pleaser refer to Section 2.2.2) is configured to protect these memory regions, since an attacker must violate the VMM software to access this sensitive memory regions and the VMM is protected by the proposed Data Integrity security mechanism (complemented with a CFI implementation). This extra layer of protection tries to mitigate attacks to the μRTZVisor VMM targeting the security mechanism, providing isolation.

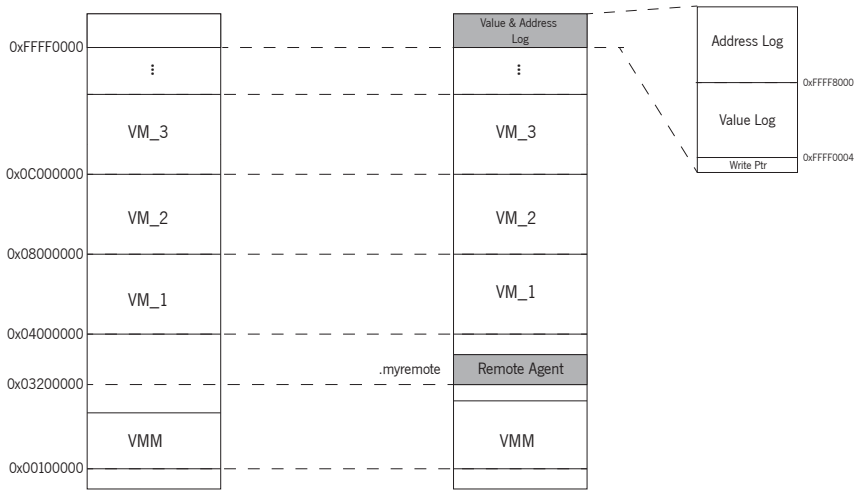


Fig. 24: Example of memory map for a hypothetical system. A memory map without the security mechanism is depicted on the left. On the right, the memory map is complemented with the code of the Remote Monitor and the logs.

The structure of the *Value & Address Log* section consists of one pointer, whose value gets updated when a new log entry is inserted, and the logs. Logs' read pointers are internal to the Remote Monitor, as the VMM does not check if the a log is full before writing. Until this point, there have been no problems with log fullness but that can change with bigger code and more critical variables to monitor. In that case, instrumentation or log sizes must be adjusted. The Value Log starts after the *Write Pointer* and ends at 32KB of the 64 KB available. The remaining memory belongs to the Address Log. Depending on requirements, log size and position can be easily adapted.

5.11.1 Starting the Remote Monitor

In the current application, the Remote Monitor runs in the second of a two core CPU. The VMM software runs on the primary core, which must bootstrap the second one. μRTZVisor code was edited for this purpose. This causes no security concerns when relying on secure boot, which is

being developed at the time of writing. As stated in the threat model, the secure boot is considered a trustworthy component. The Remote Monitor is also enabled only after all initializations are performed and before executing any guest OSs, therefore ahead of interactions with exterior entities. This process consists of clearing buffer pointers and using an interrupt-like instruction to wake up the second core and start the enforcement of the data integrity rules.

5.12 Completing Data Integrity with CFI

Data Integrity provides protection over critical data, including code-pointers. However, by itself, it is not sufficient to detect all control-hijacking attacks. At the present, every VMM indirect branch uses code-pointers already protected, by this Data Integrity method. Due to strict programming guidelines, C++ polymorphism is not allowed resulting in far less indirect branches. The stack is the principal attack surface for control-hijack attacks, as it is left unprotected. For this purpose, Data Integrity is complemented with the work carried out in another project, which aims to detect illegal control-flow transfers (i.e., enforce CFI). By detailing its implementation, the goal is to understand the close association and dynamics between these two approaches.

Alike Data Integrity, CFI uses compile-time instrumentation to trace control-flow transfers. Using a GCC plug-in, unique labels are inserted on function invocations, prologues and epilogues. At run-time, labels are logged in a circular buffer, when the processor traverses the instrumented code. Note that only indirect function calls are labeled and logged. Instrumentation before function's invocation grants coarse-grained protection against JOP [71]. Function's epilogue is instrumented to avoid ROP [72]. To ensure that the invoked function executes from the beginning, a label is inserted in the prologue. The different checkpoints where instrumentation is inserted are all depicted in Fig. 25. While Data Integrity uses the GIMPLE IL to detect memory writes to critical static variables, this CFI implementation detects sensitive control-transfers.

The CFI plugin also stores meta-data in an auxiliary file, mapping inserted labels to their position in the code. A given position is defined by the name of the function it belongs to and a number, creating an unique identifier. This identifier can correspond to function invocations, prologue or epilogue. The auxiliary file, resulting from compiling the code in Fig. 25, is depicted in Listing 5.21. Analyzing this file and *FuncA* instrumentation, a label with value 4 was inserted in the prologue, characterized by the `FuncA - 1` identifier (identifiers are highlighted in bold). In the same way, the `FuncA - 2` identifier states that the epilogue was instrumented with a label with value 5. *FunC* is an alternative function for the indirect branch invoking *FuncA*. The `main - 2` identifier corresponds to this indirect branch instruction present in the *main()* function.

The CFG for this code is available in Fig. 26, and in textual form in Listing 5.22. The CFG can be created before compilation as function's identifiers `<funtion_name> <integer>`

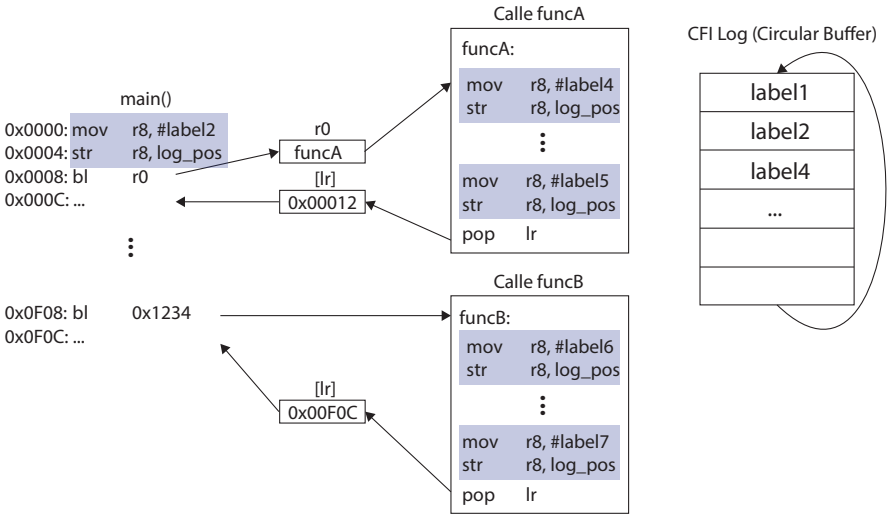


Fig. 25: Example code instrumented for CFI. Execution starts in main(). Then, it is transfered to funcA in 0x0008, via an indirect branch. After a few instructions, funcB is executed. Meanwhile, on the left, the CFI Log is being filled as the code executes.

Listing 5.21: CFI meta-data file for the scenario presented in Fig. 25.

```
1  main 1 – 1
2  main 2 – 2
3  main 3 – 3
4  FuncA 1 – 4
5  FuncA 2 – 5
6  FuncB 1 – 6
7  FuncB 1 – 7
8  FuncC 1 – 8
9  FuncC 2 – 9
```

follow a logic: the prologue has number 1, invocations follow incrementing the `<integer>` and the last number corresponds to the epilogue. For the function currently being instrumented, the `<function_name>` field in the identifier remains the same. The most important part of this CFG is line 2, where the targets, *FuncA* and *FuncC*, for the indirect branch instruction in *main()* are defined. The remaining of this CFG only states that function's prologue must precede their respective epilogue - ensuring that functions start executing from the very beginning - and that *FuncA* must precede *FuncB*, while the latter must execute before *Main*'s epilogue.

Listing 5.22: CFG, in textual form, for the code in Fig. 25.

```
1  main 1 – main 2
2  main 2 – funcA 1 – funcC 1
3  funcA 1 – funcA 2
4  funcA 2 – funcB 1
5  funcB 1 – funcB 2
6  funcB 2 – main 3
7  funcC 1 – funcC 2
8  funcC 2 – funcB 1
```

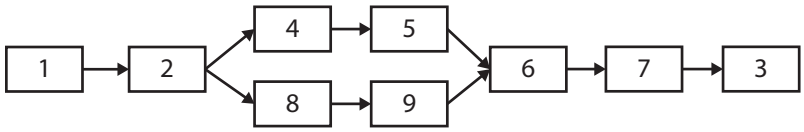


Fig. 26: CFG, in graphical form, for the code in Fig. 25.

CFI executes simultaneously and continuously with Data Integrity, in the second core. Using the same log mechanism, CFI labels are written to the same MMU protected read-only memory region, located in the same 64KB section where the Data Integrity logs are stored. Thus, the *Value & Address Log* memory region, depicted in Fig. 24, is updated as portrayed in Fig. 27. Using the MMU to restrict access policies, on log data, requires extra instrumentation to be inserted to disable and re-enable the MMU every time a log update occurs, alike the proposed Data Integrity method.

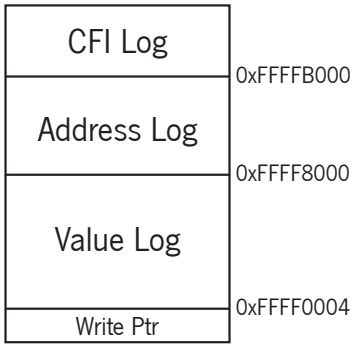


Fig. 27: Log address map updated with CFI Log.

Benefiting from the same protection mechanisms as Data Integrity, CFI postpones the control-flow integrity verification, using the logging mechanism. Deferring the verification is only possible due to the W$\oplus$E policy enforced upon the code of μRTZVisor. This policy ensures that all executable code is instrumented, disabling code-injection attacks. Isolated in the second core, CFI uses logged values to ensure that control-flow obeys the aforementioned CFG.

5.13 Limitations

While this section thoroughly describes the adopted technique, it is necessary to clarify some limitations and provide guidance on how to outweigh them. Indirect Writes are, logically, the most problematic type of access to monitor. Regarding Direct Writes, the only verification re-

quired is for the written value. For indirect writes, both addresses and value must be monitored 1) to detect the accessed memory location and 2) perform the data integrity check. The discussed indirect write types can be over or under-flowed, granting illegal access to prohibited memory regions. This is an acknowledged problem, which results from the weak C/C++ type system; however, these unsafe languages are still used, due to their benefits in terms of flexibility. Although, while this problem grants viability to the suggested approach, it can also be its downfall. Envision a non-monitored indirect access to a global negligible static variable, this could undermine all the monitoring efforts. For instance, an ”indirect attack” corrupting this variable to modify other critical monitored static variables could go unnoticed. To solve this question, two immediate solutions arise. The first is to monitor all target program's indirect accesses (partial and total) with instrumentation and perform bound checking, in the Remote Monitor. This is a non-scalable approach since, depending of the coding paradigm and code size, the target program could become bloated with instrumentation, creating significant performance overhead. The second solution is to move all critical static variables to a hardware write-protected memory section and insert instrumentation, allowing them only to be written by benign accesses. This solution also has a drawback, since it requires all indirect accesses - to critical static variables - to be instrumented for benign updates. Please note that when working with structures and classes, not all member fields have to be critical, despite being in the same static variable. Notwithstanding, it limits the amount of inserted instrumentation since only accesses that need to modify a critical static variable need to be instrumented. To understand the inherent problem associated with this solution, look at the following example. A static variable with type class A is monitored for writes to a critical member variable B but not to variable C. In this case, any indirect access to variable C would have to be monitored despite its non-criticality, since any memory corruption in that access could compromise the other critical variable B. This is due to the continuous memory layout of classes and structures in C/C++ and the lack of hardware to protect memory at the required level of granularity. To provide field-sensitive data integrity, the second approach was implemented.

Alike many hypervisors, μRTZVisor in partially implemented in assembly. This code cannot be instrumented using this mechanism. As of now, inline assembly code must be manually instrumented, especially if it can create a security hole.

5.14 Summary

This chapter presented a method to instrument μRTZVisor in order to log writes performed upon sensitive variables. The data is then used by the Remote Monitor which enforces data integrity rules translated from an abstract specification to C++. This is a complex process involving lots

of auxiliary files whose generation can be, to a certain degree, even more automated. The next chapter will focus on automation and modeling.

Despite being specifically designed for μRTZVisor, the proposed approach can be adopted by other software. This countermeasure was designed for a software heavily reliant on static data; thus, software using dynamic data might need to extent this protection scheme. Both the design of the target application and the rules devised can impact performance. As a consequence of the copy-based mechanism, data-intensive applications can be bottlenecked by this approach. Special attention must be provided to the devised data integrity rule set, as it should only contain the set of rules necessary to ensure target's data-plane integrity.

CHAPTER 6

Data Model and Code Generation

Another major part of this work is associated with ontologies and model-driven programming. In this chapter, information related to the Data Integrity security mechanism is translated to an ontological meta-model to uniform knowledge representation. Following, integration with the SeML infrastructure is discussed to present how its code generation capabilities are leveraged to automate the deployment of the security mechanism.

6.1 Introduction

Ontologies used as knowledge bases provide a standard for data representation and sharing, easing maintainability. Data used in the method introduced in the previous chapter operates upon data associated to μRTZVisor, mostly stored in implementation specific files using non-standard representations. This limits the usage of information to the developed security mechanism.

The main goal of this chapter is to create a meta-model using ontologies to represent the data plane of multiple programs, alongside information required for the Data Integrity security countermeasure, introduced in the previous chapter. The devised meta-model is closely related to GCC's since this work is partly associated with it. GCC's meta-model is complemented and adapted to meet model's requirements. Ultimately, conceived models, containing target program's data plane information, will be automatically translated to the text representations, required to add Data Integrity to μRTZVisor. The ontological meta-model is inserted into the SeML modeling framework for this purpose, generating code from ontological models. At the same time, ontology-based models are subject to semantic validations provided by meta-model axioms and applied by the reasoner. SeML IDE (Front End) is not used as this information must be provided by the developer for a specific implementation thus, it is non-modifiable by DSL users aiming to create systems based on μRTZVisor.

Firstly, this chapter will focus on the modeling efforts. Information will be provided following the structure depicted in Fig. 28. Parts of the meta-model (*M2*), and model (*M1*) of μRTZVisor

will be simultaneously scrutinized with some examples, starting with variable definition. Then, declaration of types will be considered, followed by references. Lastly, abstract rules are analyzed, which will be an essential addition to the meta-model. References refer to the usage of variables throughout the code and can be thought of, despite not being, as the last abstraction level, M0, being instances of a model containing variable and type definitions. The chapter ends discussing the integration with the SeML infrastructure and code generation through model-driven programming.

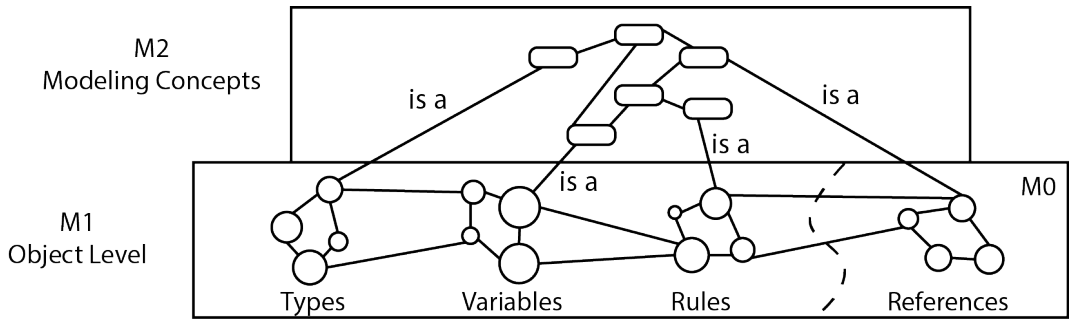


Fig. 28: Structure of the modeling efforts.

6.2 Variable Declaration

Variables declared in the source code can be translated into a model using the developed meta-model. Although not complete, Fig. 29 depicts some parts of a model and the meta-model, regarding variable definition. The *kindOf* and *is-a* relationships denote subsumption and instantiation, respectively, as explained in Section 2.5.1. For instance, "*hypercall_table* is a statically defined array with integer type and scope ::urtzvisor::Hypercalls" and "*instance* is an object of type *GuestManager*" is knowledge that can be intuitively inferred from this network. Code variables are translated to the model as instances of *StaticVariable*, which is defined as a specific type of *Variable*. *StaticVariable*s must have a *Type* and scope. The latter can be inherited from the *RecordType* a variable belongs to (i.e., when static variables are defined inside C++ classes), or directly from a *Namespace* if variables are defined there instead. *hypercall_table* is defined inside the *Hypercalls* class thus, inherits the scope of the class itself: the *urtzvisor* namespace. Type is given by the *staticHasType* relationship that connects *StaticVariable* individuals to *Type* individuals. The *Integer* concept obligatorily contains only one individual, being a C++ basic type alike float or double. Arrays are distinguished from other objects by the *arrayHasType* relationships and other distinct data properties.

Every static variable must have a type and scope. In Protégé, such restrictions are applied as depicted in Fig. 31. While the reasoner does not present any errors, the infrastructure where

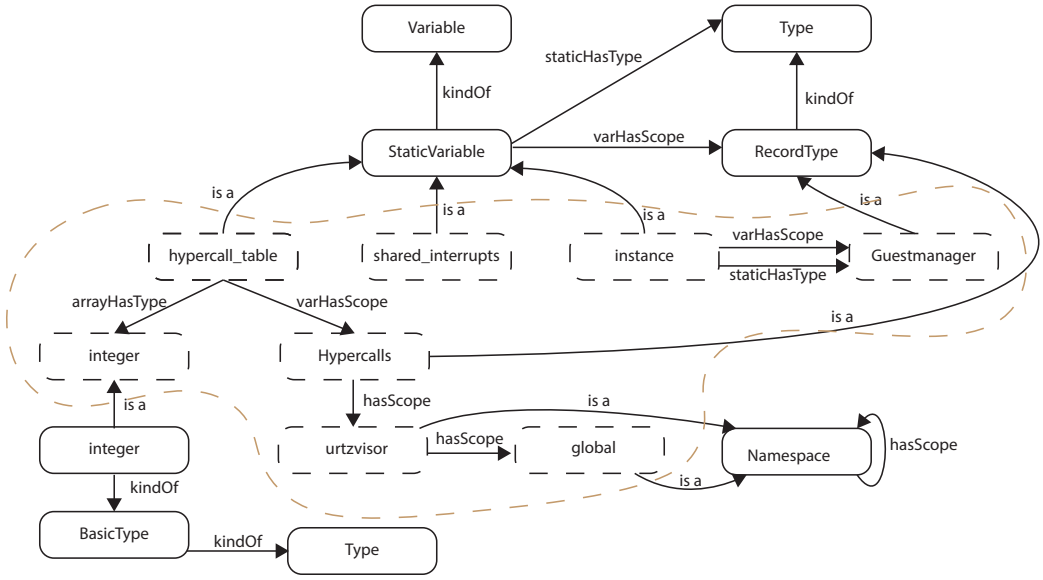


Fig. 29: Semantic network with a partial conceptualization of C++ variable declarations. The colored dashed line separates the model (individuals) from the meta-model (concepts).

this ontology is inserted does. Another semantic rule is that a variable identifier is unique and corresponds to a name plus scope. Unfortunately, this rule cannot be yet constructed due to the nature of this ontology for scope definition.

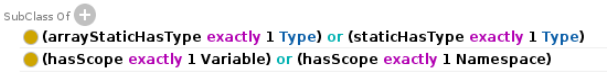


Fig. 30: Semantic rules created using Protégé.

6.3 Type Declaration

Types are subdivided into three subtypes: *BasicType*, *RecordType* and *TemplateInstantiatedType*, as depicted in Fig. 31. The first characterizes basic types from programming languages (e.g., string, float and double) while the second characterizes user-defined data types (e.g., classes and structures). Individuals of *TemplateInstantiatedType* are instantiations of C++ templates [123]. User-defined classes (i.e., individuals of *RecordType*) connect to their member fields through the *hasField* object property, a sub-property of *isComposedOf*. Therefore, *FieldDeclaration* aggregates every field belonging to classes and structures. Furthermore, *hasField* possesses an inverse property: *isFromStruct*. The *GuestManager* is a C++ class with, at least, *mCurrentGuest* and *guestList* as members. These are individuals of *FieldDeclaration*. *mCurrentGuest* is an integer and has an offset of 1016 bytes from the beginning of the *GuestManager* class, *guestList* has an

offset of zero (not depicted in Fig. 31) since it is the first field defined in the *GuestManager* class. *guestList* is an array member with two *Guest* type elements. The element number is provided by the *arraySize* data property and the type of the elements by the *arrayFieldHasType* object property. *shared_interrupts* variable's type is an instantiation of a C++ template: *SharedInterrupts*. Every static variable whose type is a template instantiation is provided a new type. This occurs as, usually, C++ templates modify the memory layout.

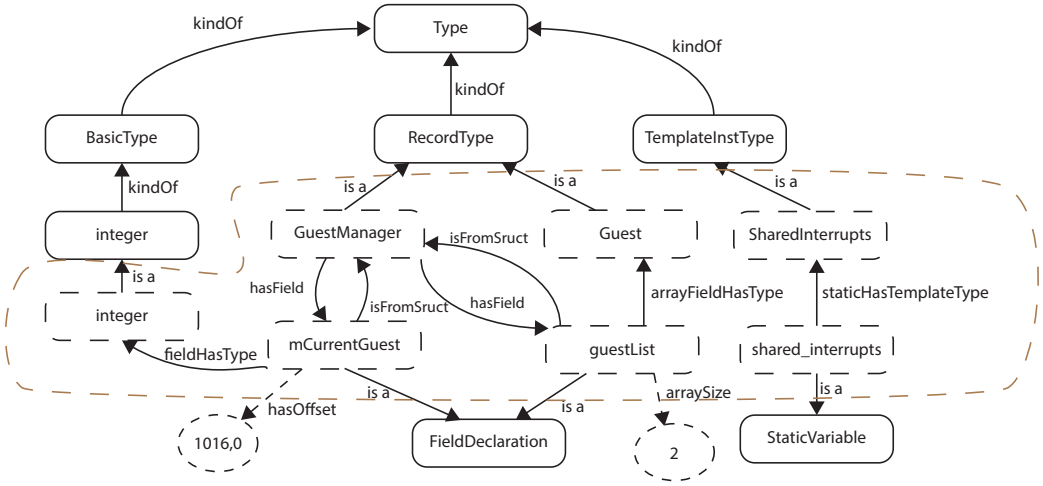


Fig. 31: Semantic network with a partial conceptualization of C++ type definitions.

In terms of semantics, individuals of *FieldDeclarations* must contain an offset as a data property and have to belong a *RecordType* or *TeamplateInstType*, through the *hasField* object property. *Type*'s individuals must always have a size and scope, similarly to a C++ program. Finally, *FieldDeclarations*, corresponding to arrays and internal to structures or classes, must point to a data property with the number of elements. At present, only static variables are considered so dynamic-sized arrays are not supported.

6.4 References

References are mentions to variables, either for the purpose of updating or reading them. As aforementioned, an alternative way to interpret references is as the conception of a new fictitious model, using the two introduced above (Variable and Type declaration) as meta-models, which is not the case. As of now, references are divided into component and array. The former comprises of accesses to *FieldDeclaration*s (i.e., class members) and *StaticVariables*. The latter consists of accesses to array class members. Beyond the conceptual distinction, references to arrays can possess a data property denoting an index. A special object property, *references*, is used by both

component and array references to connect them to a specific variable or class member. Furthermore, chains of multiple references can be created to portray complex accesses (e.g., accessing fields of classes inside classes, and so on). There is a sequential order associated with references, represented by the *preceding* object property. If a references precedes another, it comes first in the modeled reference.

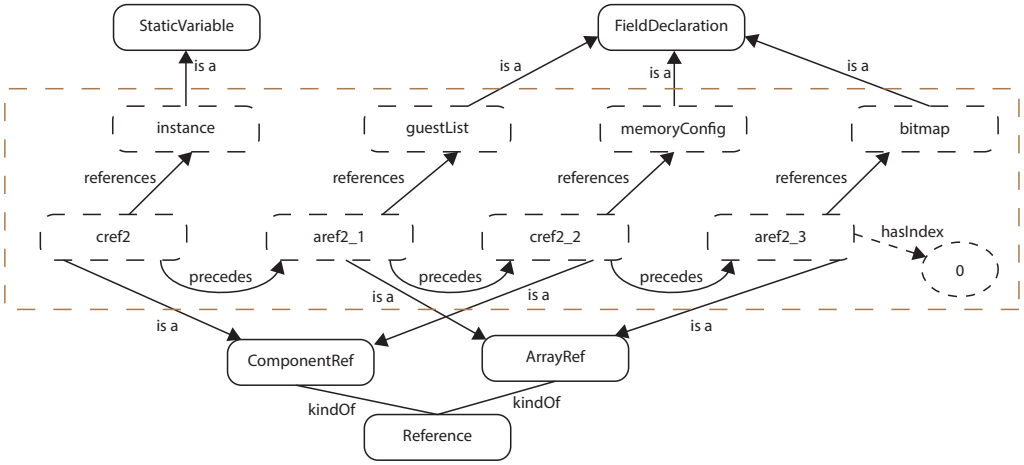


Fig. 32: Semantic network with a partial conceptualization of C++ references.

The reference of the *immutable_vec_rule* rule is translated into a model, as depicted in Fig. 32. *Instance* is associated with *cref2* via the *references* object property. The remaining references attach to fields of *instance* alike. This rule requires an array index in its reference, provided by the *hasIndex* data property. The meta-model provides sequentiality to references with the *precedes* object property.

References possess several semantic restrictions. Primarily, when a reference precedes another, the latter must reference a member/field of the previous reference's class/ structure. In other words, references can only be created for class members, similarly to variable accesses in C++. An SWRL rule was devised to enforce this restriction, as depicted in Listing 6.1. Firstly, two OWL superproperties are defined, namely, `referencesType` and `referencesFieldFrom`. The former connects a reference, on the LHS of a *precedes* object property, to the type of the individual it points to (i.e., the type of a *StaticVariable* or *FieldDeclaration*). The latter, gets the type of the class a field belongs to, on the RHS of a *precedes* object property. If both superproperties point to the same individual (i.e., the same *Type* individual), it means that the reference on the RHS of the *precedes* object property references to a field internal to the *Type* of the first reference. If the *Type* is not the same, an error, associated with the second reference, is generated in SeML with the following help text: "referencing a field which does not belong to a structure".

Listing 6.1: SWRL rule enforcing that only references to internal class members exist.

```
1 references(?r1, ?fd) ^ precedes(?r1, ?r2) ^ referencesFieldFrom(?r2, ?target2) ^ references(?r2
    , ?fd2) ^ referencesType(?r1, ?target) ^ differentFrom(?target, ?target2) -> autogen0:
    hasError(?r2, "referencing a field which doesn't belong to structure")
```

6.5 Abstract Rules

Abstract rules are elucidated in Section 5.9. Rule types are divided into specific concepts, in the meta-model, deriving from an abstract *Rule* concept. Listing 5.15 contains the textual representation of the set of rules enforced on μRTZVisor, which are partially translated to the model in Fig. 33. Logically, individuals derived from *Rule* (e.g., RangeInt) are associated with a constraint that must be enforced at run-time to grant data integrity to critical static variables. Wrapping up Fig. 32, rules are linked to references via specific object properties, derived from an abstract *hasReference* property, as depicted in Fig. 33. *Rule1*, instantiated from *RangeInt*, has two data properties denoting the maximum and minimum values an integer variable accepts. Similarly, the *regvalpattern1* rule, an instantiation of *RuleRegPattern*, has its own pattern, stored in the *hasPattern* data property.

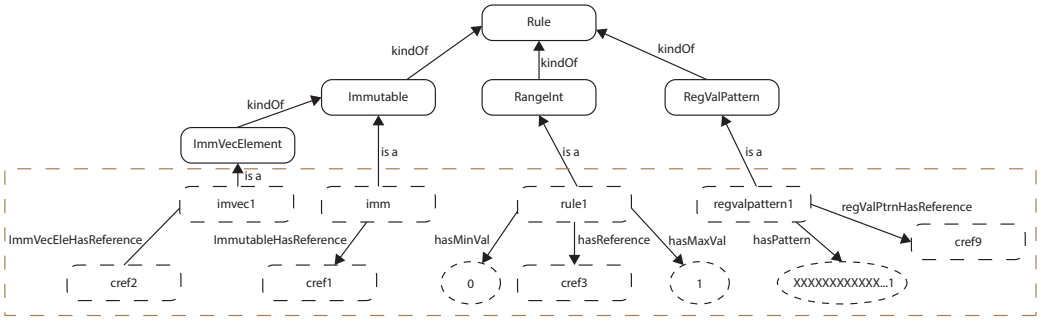


Fig. 33: Semantic network with the conceptualization for the abstract data integrity rules.

Semantically, rules cannot be applied to the same reference. Restricting rules to point to different references is a step towards having a rule per static variable or class member, as required by the security countermeasure. Nevertheless, this is not ideal since two different chains of references can point to the same spot. However, due to the meta-model organization and current framework limitations, the SWRL code in Listing 6.2 is the closest approximation. The *RangeInt* and *RegValPattern* rules possess data properties with attributes essential to them. In both cases, restrictions are enforced to guarantee the existence of such properties.

Listing 6.2: SWRL code to ensure a reference only belongs to an abstract rule.

```
1 RuleReferences(?d2, ?ref2) ^ sameAs(?ref1, ?ref2) ^ Rule(?d2) ^ RuleReferences(?d, ?ref1) ^
    Rule(?d) -> autogen0:hasError(?ref1, "one reference can only have one rule")
```

6.6 Code Generation

Integrating the above depicted modeling efforts into the SeML infrastructure further allows to leverage its code generation capabilities, translating ontological model elements to a textual representation or implementation artifacts. As previously stated, the formerly introduced security solution requires auxiliary files to perform its operations. Firstly, the Instrumentation Pass calls for a list of critical variables (Listing 5.9), alongside a memory layout of the VMM, containing both data structures and static variables (Section 5.8). A points-to file (Listing 5.10), containing information regarding indirect accesses, is also required. However, as of now, this file must be manually created, as the information it contains is not yet modeled. At last, to create the C++ rules, a file with a set of abstract rules (Section 5.9) and the same memory layout file are required.

The main contribution of modeling dwells in automating, to some extent, the deployment of the security countermeasure. Two separate external tools were developed to perform code generation. The procedure used to generate the code is detailed in Section 4.2.1. The first tool analyzes references and translates abstract rules from the model representation to text while automatically generating the input file to the instrumentation phase. Monitored critical variables can be inferred from the abstract rules. The second tool generates the memory layout file. The generated files from μRTZVisor's ontological model can be visualized, in full, in appendix B. Knowledge associated with the VMM is specified under the same language. At the moment, the model is only being used to generate implementation files for the security countermeasure. Nevertheless, using other external tools, different code can be generated. Constraining developers to a unique language diminishes engineering efforts, easing re-usability of information.

CHAPTER 7

Evaluation

A general approach to evaluate the effectiveness of a security countermeasure is to analyze how to subvert it, from an attacker's perspective. The first part of this chapter corresponds to a security analysis where the instrumentation, logs, and Remote Monitor are considered. Next, an artificial, expressive, vulnerability is used granting total control over the stack to the attacker. Alike other security techniques, there is a trade-off between performance and security. Apart from the security analysis, an evaluation on the impact of the mechanism - on the target platform - is performed, regarding used resources and effects on determinism. During this chapter, the devised security mechanism is complemented with a CFI implementation for a complete security evaluation.

7.1 Security Analysis

Performing a quantitative analysis of the security benefits provided by the Data Integrity and CFI mechanisms is maybe impossible. Above all, this security evaluation must be complete, in a manner that no attack vector can be disregarded. There are two security invariants that must be enforced:

1. Tamper proof – attackers shouldn't manipulate either the data or code of the Remote Monitor. This includes the logs - shared between the target program (μRTZVisor) and the Remote Monitor - and the instrumentation.
2. Non-bypassable – attackers cannot bypass the logging instrumentation.

μRTZVisor, using TrustZone technology, provides the first line of defense against attacks originated in VMs, either aiming at the VMM or the Remote Agent. So, an attacker must exploit a memory corruption vulnerability while the processor is in Monitor mode - VMM privilege level - to be able to breach the security invariants. If such an exploit occurs, the devised security

mechanism has memory protections in place that ensure the first invariant. Namely, the ARM MMU enforces access control policies on memory. Each core possesses its own MMU physical interface and configuration. The core executing the Remote Monitor application completely disregards the MMU, having full access to memory. This can be tolerated as the Remote Monitor code is an isolated entity, in the sense that all interactions with external entities are performed via the instrumented code and are statically known. However, the primary core, executing the VMM, is unable to write to any memory (code and data) used by the Remote Monitor application. Tampering with the Remote Monitor is only possible by disabling the MMU. To do so, it requires special instructions that are inserted with the instrumentation and are not available as a function. Furthermore, an attacker cannot disable the MMU without re-enabling it as the instrumentation inserted in critical writes is an atomic basic block (i.e., does not possess any branch instructions). Log data structures are read-only, also enforced by ARM MMU. Log write protections are disabled by the inserted instrumentation and re-enabled shortly after the log update. While attackers can target the page table, their efforts would be fruitless, as the physical addresses of the page table are not mapped in the page table itself. Changing the base address for the page table also requires specific assembly instructions.

By adding all instrumentation after the write operation and due to the aforementioned atomicity, invariant 2) is enforced. An attacker cannot perform a write and avoid instrumentation as branch instructions are not present.

Although extremely unlikely, interrupt handlers can be used to subvert this Data Integrity scheme. Having disabled the MMU, non-instrumented indirect accesses could be used to write to illegal memory without ever being detected. Due to the unlikeliness of this event and attending that interruptions are designed to be short and deterministic, interrupts are not disabled while a log update occurs. Nevertheless, they could be disabled and re-enabled for extra security.

7.1.1 Facing State of the Art Attacks

Complementing Data Integrity with a CFI implementation allows to thwart attacks either to data- or control-planes of the VMM. Using both approaches separately would augment the attack surface of μRTZVisor. The proposed approach aims at mitigating non-control-data attacks. Considering code pointers as critical variables that must be protected, it also provides control-flow protection to a certain extent. However, the stack is completely disregarded as it is not considered a source of non-control-data attacks. CFI arises as a complement to further secure the control-plane of μRTZVisor.

By exploiting a memory corruption vulnerability such as an integer overflow, format string or buffer stack overflow, an attacker is mostly unable to stealthily update a critical static variable with

spurious values. By only allowing access to critical static variables to specific indirect accesses, all other indirect accesses are useless when trying to modify such variables.

Non-control-data attacks such as DOP [105] are mitigated in their expressiveness as an attacker is not able to arbitrarily change a critical static variable. However, due to the granularity of the proposed approach these attacks are still possible. For example, the stack can be corrupted to perform operations on spurious data without diverging from the legal control-flow. Although that would go undetected, once an indirect access tried to access critical variables without permission it would most likely be stopped by the MMU. Furthermore, attacks aiming at data attributed to critical variables are eliminated as that data is further analyzed by the Remote Monitor.

Additionally, the CFI implementation allows to detect further control-flow related attacks. Firstly, the ARM MMU ensures that memory is either writable or executable, thwarting code-injection attacks. CFI also detects more sophisticated attacks such as ROP and JOP. Essentially, both attacks consist on identifying and chaining gadgets terminating on pop or branch instructions, respectively. By labeling critical control transfers, in the code, CFI can detect such attacks. However, due to the CFI's granularity, some very specific attacks can go undetected, as depicted in Listing 7.1. As labels are only used in function's prologue, epilogue and invocation, it is possible to return from function `B()` to any code existing between function `B()` and `C()` or directly to the invocation of function `C()`. This is possible since there are no labels between the end of `B()` and the beginning of `C()`.

Listing 7.1: Example denoting the insuficiency of this CFI scheme.

```
1 void A(){
2         //Variable initialization and code
3         B();
4         //Critical code
5         C();
6 }
```

A special characteristic of this log-based protection scheme is the detection of transient attacks. In this type of attacks, the adversary may cause harm and then hide its traces. With this scheme, this kind of attacks are not possible, as every write to critical variables and variation in the control-flow are registered for further analysis. Periodic integrity verification tools can only detect permanent integrity damage. For example, methods using introspection techniques [14, 15, 99] are invoked periodically to analyze the memory of a monitored entity.

7.2 Performance and Code Size

It is extremely difficult to perform a quantitative evaluation on performance for this tailor-made security solution. Performance overhead is directly related to the number of rules specified and

the respective instrumented writes, as well as their invocation at runtime. Furthermore, with higher interactions between guests and the hypervisor, performance is generally worse as more instrumented code is executed. Unfortunately, the hypervisor is still in development and, while using an intermediary version sufficed to test the implementation, it is not adequate for a performance benchmark. This is especially true since the hypercall API is yet to be implemented. Any performance test would be deceiving as no guest-hypervisor interactions occur, which is the main source of performance overhead.

However, code size can be measured for this intermediate implementation of μRTZVisor, for the existing rule set (Listing 5.15). An executable with the hypervisor and the Data Integrity mechanism, compiled with all optimization levels, adds approximately a thousand lines of assembly code to μRTZVisor. CFI inserts approximately eight times more since all functions' prologues, epilogues and indirect invocations are instrumented. In the case of Data Integrity, only critical writes to static variables are instrumented.

ASM Lines of Code	-o0	-o1	-o2	-o3	
VMM Only	89322	86495	86979	87866	
DI Only	1032	962	922	991	(approx. 1%)
CFI Only	8715	8599	8386	7628	(approx. 9%)
Total	99069	96056	96287	96485	

Fig. 34: Comparison between the number of lines of code inserted by the Data Integrity and CFI mechanisms.

Despite no benchmarks results being provided, Fig. 35 provides an overview of the inserted instrumentation. On the right, assembly code corresponds to line 6 of the C++ code on the left. This example logs a value calculated from `slot_id` into the Value Log - as that value is used as a new value for `guestptr->memoryConfig(.Bitmap)` - and the written memory address in the Address Log. In Fig. 35, both inserted logging procedures are gray colored. The specific parts where the log is actually updated are highlighted blue, as their size can vary in the case of the Address Log. The green highlighted code can belong to either the Value Log or the Address Log procedure. In case of a direct memory write, the Address Log code is not inserted as there are no indirections. In that case, the green code is inserted in the Value Log code to update the Write Pointer. Similarly, if a constant is written into a critical variable, also the Value Log code would be removed, as constants do not need to be logged.

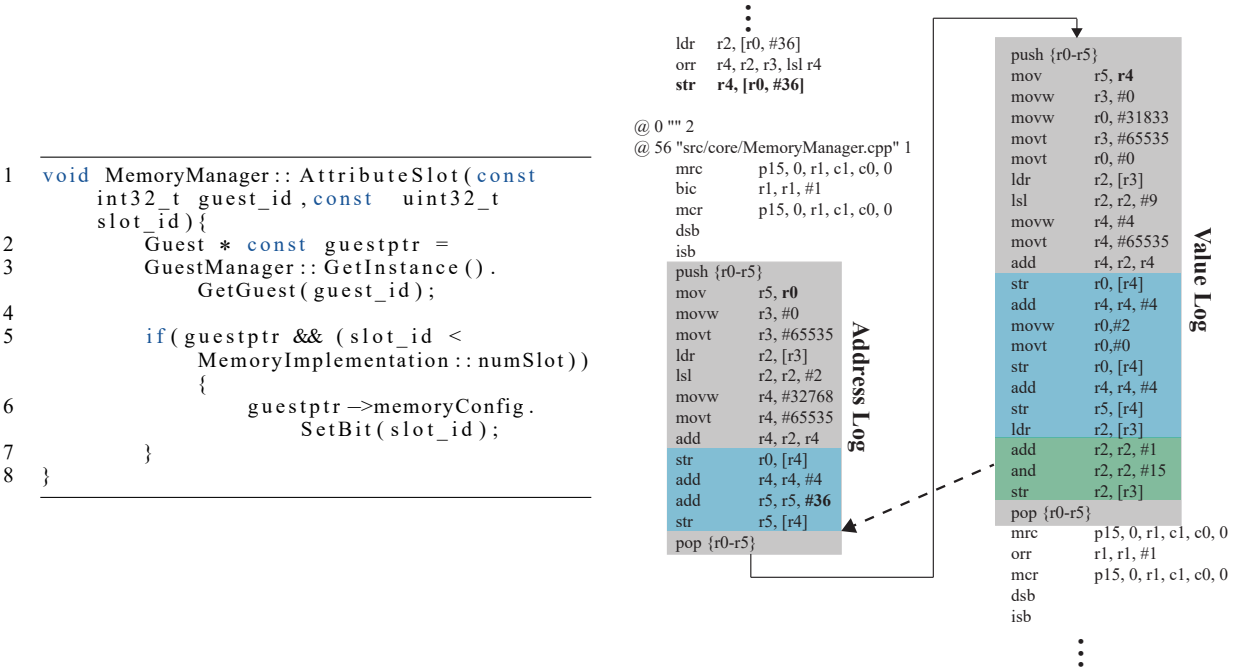


Fig. 35: Instrumented code and its equivalent in assembly. Line 6 is translated to the code on the right. The instrumented code disables the MMU, logs the written value and address and re-enables the MMU.

7.3 Use Case Scenario

To jointly validate the Data Integrity and CFI approach, an experiment was conducted to evaluate the effectiveness of both mechanisms against several synthetic attacks. The set of attacks is contextualized to a virtualized environment and aims at both data and control planes. Specifically, μRTZVisor's hypercall API is leveraged to implement vulnerable code.

The code in Listings 7.2 and 7.3 acts like a generic hypercall by providing services to guest OSs, with small but significant differences. Particularly, it allows a guest to request a shared memory slot with a driver/IO server, which is a higher privileged VM. To connect to a specific IO server, a guest provides a command to the hypercall with the name of the server in the input `InputData` structure. Then, the hypercall parses the server name (`removeSpaces`) and opens the same memory slot to the IO server, via `OpenServerSlot`. Finally, the `AttributeSlot` method is executed to change the memory permissions of the requester guest OS, to consider the new shared memory slot. Listing 7.2 differs from 7.3 since the latter has 1) sanity check in line 7 and 2) a memory corruption vulnerability in line 11.

Listing 7.2 presents a case where input data is trusted to be correct, a somewhat common error. The `AttributeSlot` function is instrumented to disable writes to the first element of an integer array - of three elements - corresponding to the memory access permissions of the DDR memory.

DDR memory permissions cannot be changed after initialization as only OCM memory can be shared (the other two array elements) between guests. In this case, when the attacker specifies a slot corresponding to the first array element, the Remote Monitor logs the operation detecting an attack to the integrity of the hypervisor.

Listing 7.2: Non-vulnerable code.

```
1  void hypercall(InputData* buf){
2  int slot; int requestType;
3  slot = ((int*)buf->data)[0];
4  requestType = buf->type;
5
6  if(requestType == IOREQ_SERVER){
7      char serverName[16];
8
9      for(uint32_t i=0;i<16;i++){
10     serverName[i]=((char*)buf->data)[i];
11     }
12     removeSpaces(serverName);
13
14     OpenServerSlot(slot,serverName);
15     MemoryManager::GetInstance().
           AttributeSlot(
16     GuestManager::GetInstance().
           mCurrentGuest,slot);
17     } else { //Update response
18     }
19 } //else ...
20 }
```

Listing 7.3: Vulnerable code.

```
1  void hypercall(InputData* buf){
2  int slot; int requestType;
3  slot = ((int*)buf->data)[0];
4  requestType = buf->type;
5
6  if(requestType == IOREQ_SERVER){
7    if(slot >= 32 && slot <=95) {
8      char serverName[16];
9
10     for(uint32_t i=0;i<buf->len;i++){
11     serverName[i]=((char*)buf->data)[i];
12     }
13     removeSpaces(serverName);
14
15     OpenServerSlot(slot,serverName);
16     MemoryManager::GetInstance().
           AttributeSlot(
17     GuestManager::GetInstance().
           mCurrentGuest,slot);
18     } else { //Update response
19     }
20 } //else ...
21 }
```

In the second case, a memory corruption vulnerability occurs in line 11 allowing to overflow the buffer containing the server name (`buf->data`) by providing a buffer length bigger than the `serverName` local variable. Sanity check on the slot variable is insufficient in this situation as the memory vulnerability occurs afterwards. Alike the previous case, when the `AttributeSlot` method tries to modify the guest memory permissions, an attack will be detected when trying to give access to other DDR partitions, to a specific guest VM. In both cases, control-flow is not altered but hypervisor's integrity is attacked. For test purposes, the Remote Monitor logs all its operations through a serial port for debugging purposes, as depicted in Fig. 36.

7.3.1 Performing a Control Flow Attack

The same code, with a slight variation, is used to perform a control-flow hijack attack. The memory vulnerability is removed from the hypercall and inserted into the `ioreqServerParse` function. This allows to change the return address on the stack when `ioreqServerParse` is invoked, using the vulnerability. In turn, execution can change to any of the already existing code. Code injection attacks are disabled by the W⊕E protection. The `PrivateTimer::Disable` method is executed stopping the hardware timer used for scheduling, as depicted in Fig. 37.

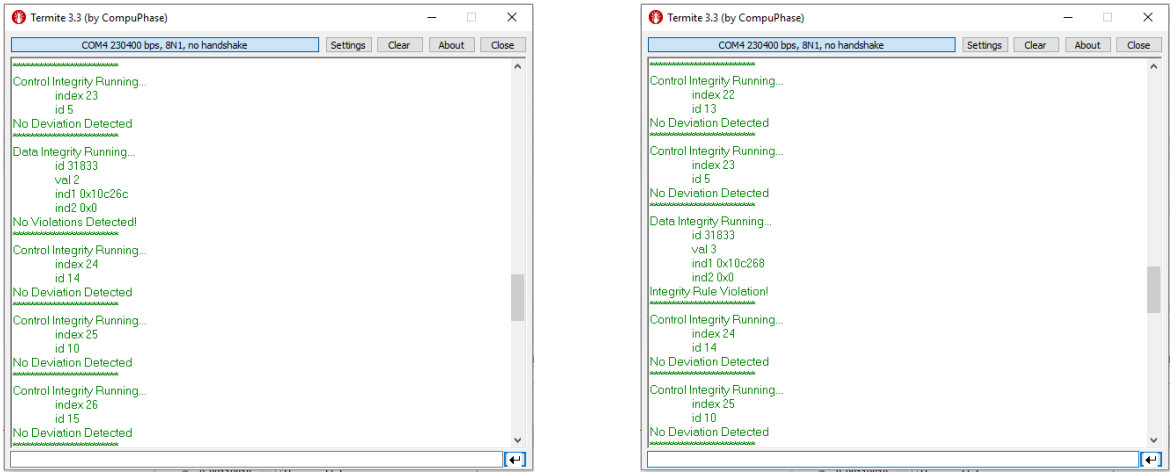


Fig. 36: Remote Monitor output for the code in Listing 7.2. On the left, no data integrity violation is detected in the accessed performed at address 0x10c26c. However, on the right, when trying to access the first array element at 0x10c268, the Remote Monitor emits an alert enforcing the first rule in B.1. Note that no CFI violations are detected.

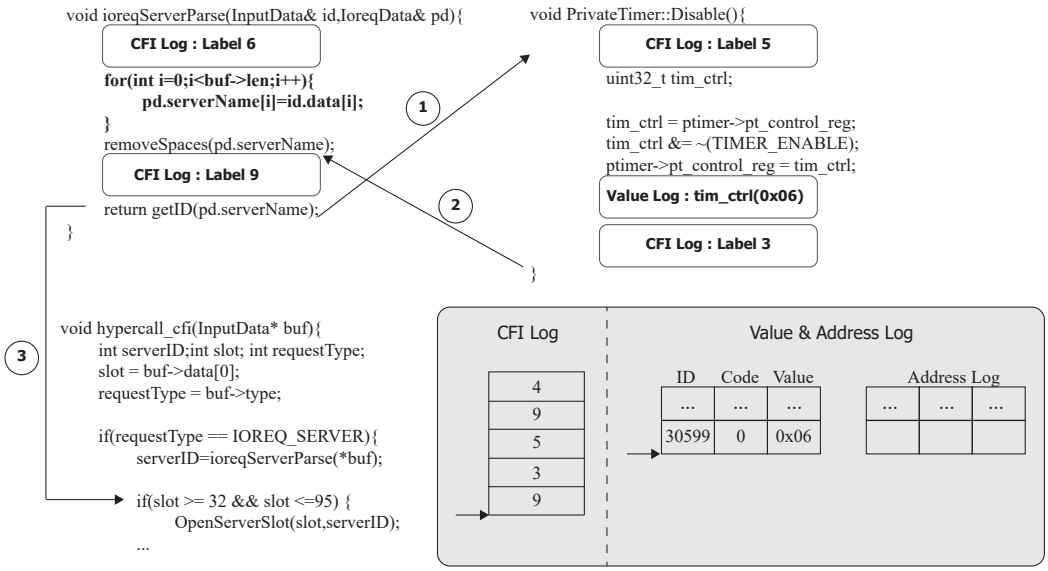


Fig. 37: Memory vulnerability being exploited to perform a control-hijack attack.

Beginning in the `hypercall_cfi` function, execution is legally transfered to `ioreqServer Parse`. Following, the memory corruption vulnerability - highlighted in bold - corrupts the stack's return address, invoking the `PrivateTimer::Disable` method instead of returning to `hypercall_cfi`, in the epilogue. Meanwhile, a new entry is added to the CFI Log. In `Private Timer::Disable`, the value written to the register of the hardware timer is stored in the Value Log and, before returning, another label is stored in the CFI log. Execution then returns to

`ioreqServerParse` and to the next instruction to be executed in the `hypercall_cfi` function. The logged contents are available in the bottom right of the figure.

This example creates two alerts in the Remote Monitor since 1) the execution flow diverted from the legal CFG and 2) the control register for the hardware timer took an illegal value according to the data integrity rules. In this very specific case, the control-flow attack created a non-control-data attack, resulting in two warnings being emitted from both Reference Monitors (i.e., Data Integrity and CFI), as depicted in Fig. 38. For further understanding, compare the logged values in Fig. 37 with the output provided below.

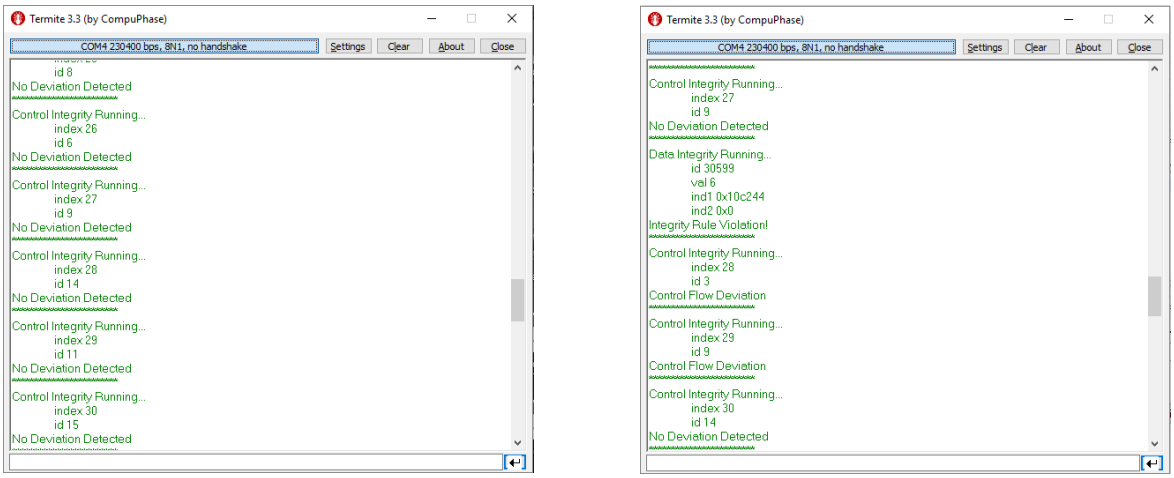


Fig. 38: Remote Monitor output for the code in Fig. 37. On the left, no control-flow integrity violation is detected. However, on the right, both CFI and data integrity violations are caught.

CHAPTER 8

Conclusion and Future Research Directions

8.1 Summary

One of the most popular attack vectors to compromise computer systems is the exploitation of memory corruption vulnerabilities. In this thesis, we described the implementation of the Data Integrity security countermeasure, tailor-made for μRTZVisor. Moreover, an ontology-based metamodel was devised to automate the deployment of the security mechanism. Recognizing future developments in μRTZVisor, efforts went on providing a future-proof security solution.

Firstly, the Data Integrity security solution was uncovered to enforce correctness of μRTZVisor's data-plane, even in the present of state of the art attacks (i.e, non-control-data attacks). The Zynq-7000 SoC was leveraged to isolate a remote integrity monitor from the hypervisor, in a separate core. Through compile-time instrumentation, an execution trace is collected by recording updates to critical static variables on μRTZVisor. The monitor audits this trace searching for violations of developer-defined data integrity rules, concurrently to hypervisor's execution. The peculiarities of μRTZVisor grant feasibility to the proposed scheme, especially when considering its disregard for dynamic data. μRTZVisor is designed with a security rationale, based on strict security requirements, such as compliance with security-related coding guidelines, focus on static data, heap non-use, amongst others.

Non-control-data attacks came into play as defenses against control-data attacks began being widely adopted. The proposed solution does not suffice to effectively mitigate all attack vectors aiming to divert program's execution from its legal control-flow. Several rules were designed by μRTZVisor developers to protect critical code-pointers. However, those rules demonstrated insufficiency against stack-oriented attacks. The stack is completely disregarded as it is not a source for non-control-data attacks, on critical static variables. The next logical step was to integrate the Data Integrity mechanism with a CFI implementation, to reduce μRTZVisor's attack surface.

The proposed combined solution of Data Integrity and CFI is coarse-grained. The focus of the former is on writes to critical static variables. The latter focuses only on function invocations.

This allows to reduce performance overhead over classical approaches such as DFI [90]. A comprehensive security evaluation was performed on both security mechanisms and to examine their resilience facing state of the art attacks. Firstly, it was concluded that the security mechanism itself is strongly protected against direct attacks, via the ARM MMU hardware. Secondly, the combined scheme seems able to mitigate the most recent non-control and control-data attacks with a relatively low performance overhead.

Integrating this work into the SeML modeling framework allowed to leverage 1) its generative programming capabilities and 2) uniformize knowledge representation. The devised ontological meta-model allows to automatically fit the security mechanism to future μRTZVisor releases, reducing the demand for a domain expert. Meta-model's capabilities were demonstrated by modeling knowledge associated with the current μRTZVisor implementation. Using an uniformized knowledge representation (i.e., ontologies) eases model's maintainability.

8.2 Future Work

The proposed Data Integrity solution, complemented with CFI and $W \oplus X$ protection, can detect control-flow hijack attacks combining stack and code-pointer protection, while making promising progress protecting against non-control-data-oriented attacks. By not striving for complete memory protection, attacks are still possible but very limited in their expressiveness. For example, DOP attacks are possible as long as they do not change critical static variables.

Yet, logging instrumentation can be enhanced for performance. Inserting instrumentation via inline assembly code may not be the best route to achieve the lowest performance overhead. Inline assembly also locks the security application to certain platforms which can become a problem in the near future.

A second avenue of work are the response mechanisms. With an isolated and independent monitor with full access to memory, system's integrity can be restored when constraints or rules are violated. A response mechanism could log suspicious occurrences, shutdown the system or employ advanced system's recovery strategies. Having policies for integrity (i.e., rules or constraints) and for response mechanisms would increase system's fault tolerance, improving both security and safety.

Points-to analysis [124] is also being manually performed from source code analysis. This process could be automated to a certain extent by leveraging the compiler [125]. Likewise, the integrated CFI approach could automate CFG generation, due to the lack of indirect branch instructions in μRTZVisor. Having a good set of rules is paramount to the success of this security solution. Another research direction could also delve into automatic rule inference to achieve complete protection of critical static variables, alike KENALI [94].

Presently, dynamic variables are completely disregarded as μRTZVisor exclusively relies on static data. Going forward, VMM developments might require dynamic data for increased flexibility. This dynamic data could enjoy a similar protection scheme as the one proposed for static data. Furthermore, considering dynamic data in this approach could ease its adoption by programs reliant on dynamic data. As the hypervisor is being designed using security coding guidelines (MISRAC++ [125]), an interesting research would be to analyze how adequate these dynamic remote attestation mechanisms are when such code is present.

Modeling efforts are not yet fully integrated in SeML. The constructed meta-model must inter-operate with a VMM meta-model and others. This will allow to transparently create integrity rules based on user-defined system configurations. For example, an user could define a system with three or four VMs. Model inter-operation would allow to capture those application's semantics, which could then be used to automatically create particular integrity rules for a specific system implementation.

8.3 Closing Remarks

This thesis documented the everlasting arms race between countermeasure design and attack advancement. Firstly, the key challenges to defend against data- and control-oriented attacks were examined. Then, defenses to thwart novel attacks were proposed. Overall, this thesis is highly specialized to the μRTZVisor VMM. But several improvements and adaptations, as stated above, could be implemented to create a more generalized security mechanism, evolving state of the art security countermeasure design. The special characteristics of μRTZVisor, such as its extremely low code size and focus on static variables, required a specialized security solution that would be both deterministic and with low performance overhead while effective in detecting violations to its integrity. This is exactly what was proposed in this thesis.

Bibliography

[1] ARM, "Arm security technology - building a secure system using trustzone technology," *Tech. Rep*, 2009. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/CACJBHJA.html

[2] D. Novillo, "Gcc an architectural overview, current status, and future directions," in *Proceedings of the Linux Symposium*, vol. 2, 2006.

[3] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[4] G. Suh, J. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI*, pp. 85–96, 2004, cited By 317.

[5] T. Donghai, J. Xiaoqi, C. Junhua, and H. Changzhen, "A concurrent security monitoring method for virtualization environments," *China Communications*, vol. 13, no. 1, pp. 113–123, 2016.

[6] A. Arabo and B. Pranggono, "Mobile malware and smart device security: Trends, challenges and solutions," *Proceedings - 19th International Conference on Control Systems and Computer Science, CSCS 2013*, pp. 526–531, 2013.

[7] "Common criteria portal," accessed: 14-11-2017. [Online]. Available: http://www.commoncriteriaportal.org/

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[9] S. Tverdyshev, "Security by design: Introduction to mils," 2017.

[10] I. Furgel and V. Saftig, "Euro-mils: Secure european virtualization for trustworthy applications in critical domain," http://www.euromils.eu/downloads/EURO-MILS-Protection-Profile-V2.03.pdf, 2016, accessed: 25-11-2017.

[11] H. Löhr, A.-R. Sadeghi, C. Stüble, M. Weber, and M. Winandy, "Modeling trusted computing support in a protection profile for high assurance security kernels." in *Trusted Computing: Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings*. Springer, 2009, pp. 45–62.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." vol. 14, 2005.

[13] H. Hu, S. Shinde, S. Adrian, Z. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 969–986, 2016.

[14] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[15] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, vol. 3, no. 2003, 2003, pp. 191–206.

[16] A. Lanzi, M. I. Sharif, and W. Lee, "K-tracer: A system for extracting kernel malware behavior." in *NDSS*, 2009.

[17] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[18] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 335–350.

[19] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[20] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi, "Security as a new dimension in embedded system design," *Proceedings - Design Automation Conference*, pp. 753–760, 2004.

[21] M. Vai, D. J. Whelihan, B. R. Nahill, D. M. Utin, S. R. O'Melia, and R. I. Khazan, "Secure embedded systems," *LINCOLN LABORATORY JOURNAL*, vol. 22, no. 1, 2016.

[22] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "*μ*rtzvisor: A secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, p. 93, 2017.

[23] M. S. Simpson and R. K. Barua, "Memsafe: ensuring the spatial and temporal memory safety of c at runtime," *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, 2013.

[24] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 128–139.

[25] V. Silva, A. Tavares, A. Gardel, S. Ivanov, F. Giunchiglia, M. Shalchian, and S. Pinto, "Collaborative Design Automation for IoT Edge and Fog Devices : a perspective paper."

[26] K. S. Hoo, A. W. Sudbury, and A. R. Jaquith, "Tangible roi through secure software engineering," *Secure Business Quarterly*, vol. 1, no. 2, p. Q4, 2001.

[27] A. Jaquith, "The security of applications: Not all are created equal," *At Stake Research. http://www. atstake. com/research/reports/acrobat/atstake_app_unequal. pdf*, 2002.

[28] V. Suma, B. Shubhamangala, and L. M. Rao, "Impact analysis of volatility and security on requirement defects during software development process," 2012.

[29] E. B. Fernandez, "A methodology for secure software design." in *Software Engineering Research and Practice*, 2004, pp. 130–136.

[30] P. H. Meland and J. Jensen, "Secure software design in practice," in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. IEEE, 2008, pp. 1164–1171.

[31] T. Stahl and M. Volter, *Model-driven software development: technology, engineering, management*. J. Wiley & Sons, 2006.

[32] S. Beydeda, M. Book, V. Gruhn *et al.*, *Model-driven software development*. Springer, 2005, vol. 15.

[33] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[34] G. Heiser, "Virtualizing embedded systems: why bother?" in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 901–905.

[35] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*. IEEE, 2014, pp. 189–198.

[36] S. Gansel, S. Schnitzer, F. Dürr, K. Rothermel, and C. Maihöfer, "Towards virtualization concepts for novel automotive hmi systems," in *International Embedded Systems Symposium*. Springer, 2013, pp. 193–204.

[37] N. Navet, B. Delord, M. Baumeister *et al.*, "Virtualization in automotive embedded systems: an outlook," in *Seminar at RTS Embedded Systems*, 2010.

[38] T. Gomes, P. Lopes, J. Alves, P. Mestre, J. Cabral, J. Monteiro, and A. Tavares, "A Modeling Domain-Specific Language for IoT-enabled Operating Systems," 2017.

[39] S. Pinto, A. Tavares, and S. Montenegro, "Hypervisor for Real Time Space Applications," 2016.

[40] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "Ltzvisor: Trustzone is the key," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[41] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE, 2010, pp. 222–226.

[42] S. Suresh and M. Kannan, "A study on system virtualization techniques," *International Journal of Advanced Research in Computer Science and Technology*, vol. 2, no. 1, pp. 134–139, 2014.

[43] J. Liedtke, "On micro-kernel construction," *15th ACM Symposium on Operating System Principles*, p. 237–250, 1995.

[44] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, "Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, 2004, p. 22.

[45] "Arm® architecture reference manual." [Online]. Available: http://bits-please.blogspot.pt/2015/08/android-linux-kernel-privilege.html

[46] "Processor modes." [Online]. Available: http://flint.cs.yale.edu/feng/research/BIOS/procModes.htm

[47] ARM, "Arm® cortex™-a series: Programmer's guide," *Tech. Rep*, 2011.

[48] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[49] L. Hendren, C. Donawa, M. Emami, G. Gao, B. Sridharan *et al.*, "Designing the mccat compiler based on a family of structured intermediate representations," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1992, pp. 406–420.

[50] "Gcc internals document." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.5.4/gccint.pdf

[51] J. Ye, G. Stevenson, and S. Dobson, "A top-level ontology for smart environments," *Pervasive and Mobile Computing*, vol. 7, no. 3, pp. 359–378, 2011.

[52] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International journal of human-computer studies*, vol. 43, no. 5-6, pp. 907–928, 1995.

[53] J. Domingue, D. Fensel, and J. A. Hendler, *Handbook of Semantic Web Technologies*, 1st ed., ser. Springer Reference. Springer-Verlag Berlin Heidelberg, 2011.

[54] J. F. Sowa, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1991.

[55] S. B. Lippman, *Inside the C++ Object Model*. Addison-Wesley Professional, 1996.

[56] J. H. Dean Allemang, *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan-Kaufmann, 2004.

[57] "Owl web ontology language reference," 2004. [Online]. Available: https://www.w3.org/TR/owl-ref/

[58] F. Baader, I. Horrocks, C. Lutz, and U. Sattler, *An Introduction to Description Logic*, 2007.

[59] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semantics: science, services and agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[60] M. Horrgide, "A practical guide to building owl ontologies using protégé 4 and co-ode tools." [Online]. Available: http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf

[61] "Owl 2 web ontology language structural specification and functional-style syntax (second edition)." [Online]. Available: https://www.w3.org/2007/OWL/wiki/Syntax

[62] "Owl 2 web ontology language direct semantics (second edition)." [Online]. Available: https://www.w3.org/TR/owl-direct-semantics/

[63] I. Horrocks, P. PAtel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," *W3C Member Submission*, 2014. [Online]. Available: https://www.w3.org/Submission/2004/SUBM-SWRL-20040521/#1

[64] "Swrl built-ins." [Online]. Available: http://www.daml.org/swrl/proposal/builtins.html

[65] V. Van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future," *Research in Attacks, Intrusions, and Defenses*, pp. 86–106, 2012.

[66] P. Zatko, "How to write buffer overflows," 1995.

[67] BugTraq, r00t, and Underground, "Smashing the stack for fun and profit." [Online]. Available: http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[68] S. Designer. (1997) Linux kernel patch to remove stack exec permission. [Online]. Available: http://seclists.org/bugtraq/1997/Apr/31

[69] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[70] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[71] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[72] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[73] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, 2003, pp. 243–255.

[74] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux," in *In-depth security conference, DeepSec, November*, 2014.

[75] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[76] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[77] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: an efficient pointer arithmetic checker for c programs," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 145–156.

[78] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[79] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[80] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 213–223.

[81] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, no. 001. Linköping University Electronic Press, 1997, pp. 13–26.

[82] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector." in *NDSS*, vol. 2004, 2004, pp. 159–169.

[83] N. Brown, "Control-flow integrity for real-time embedded systems," Ph.D. dissertation, WORCESTER POLYTECHNIC INSTITUTE, 2017.

[84] T.-c. Chiueh and F.-H. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Distributed Computing Systems, 2001. 21st International Conference on.* IEEE, 2001, pp. 409–417.

[85] M. Frantzen and M. Shuey, "Stackghost: Hardware facilitated stack protection." in *USENIX Security Symposium*, vol. 112, 2001.

[86] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams." in *USENIX Security Symposium*, 2002, pp. 61–79.

[87] D. Nebenzahl, M. Sagiv, and A. Wool, "Install-time vaccination of windows executables to defend against stack smashing attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 1, pp. 78–90, 2006.

[88] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks." in *USENIX Annual Technical Conference, General Track*, 2003, pp. 211–224.

[89] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[90] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.

[91] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 263–277, 2008.

[92] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2001, pp. 54–61.

[93] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 254–263.

[94] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[95] I. Díez-Franco and I. Santos, "Data is flowing in the wind: A review of data-flow integrity methods to overcome non-control-data attacks," in *International Conference on EUropean Transnational Education*. Springer, 2016, pp. 536–544.

[96] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*. IEEE, 1994, pp. 134–144.

[97] C. Ko, M. Ruschitzka, and K. Levitt, "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 175–187.

[98] R. Sekar and P. Uppuluri, "Synthesizingfastintru sion prevention/detectionsystemsfrom high-levelspecifications," in *Proceedings of USENIX*, vol. 99, 1999.

[99] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data." in *USENIX Security Symposium*, 2006.

[100] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX Security Symposium*. San Diego, USA, 2004, pp. 179–194.

[101] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[102] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[103] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 156–168.

[104] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[105] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *USENIX Security Symposium*, 2015, pp. 177–192.

[106] "Gcc melt website," http://gcc-melt.org/, accessed: 25-09-2017.

[107] B. Starynkevitch. Customizing your gcc compiler with melt extensions. [Online]. Available: http://gcc-melt.org/gcc-melt-sheet.pdf

[108] D. Rosenberg, “Qsee trustzone kernel integer over flow vulnerability,” in *Black Hat conference*, 2014.

[109] G. Beniamini, “etting arbitrary code execution in trustzone’s kernel from any context.” [Online]. Available: http://bits-please.blogspot.pt/2015/03/getting-arbitrary-code-execution-in.html

[110] ——, “Exploring qualcomm’s trustzone implementation.” [Online]. Available: http://bits-please.blogspot.pt/2015/08/exploring-qualcomms-trustzone.html

[111] ——, “Full trustzone exploit for msm8974.” [Online]. Available: http://bits-please.blogspot.pt/2015/08/full-trustzone-exploit-for-msm8974.html

[112] ——, “Android linux kernel privilege escalation vulnerability and exploit (cve-2014-4322).” [Online]. Available: http://bits-please.blogspot.pt/2015/08/android-linux-kernel-privilege.html

[113] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, “Design patterns: Elements of reusable object-oriented software,” *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[114] “Reflection (computer programming),” https://en.wikipedia.org/wiki/Reflection_(computer_programming), accessed: 25-09-2017.

[115] “Zybo™ fpga board reference manual.” [Online]. Available: https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf

[116] “Zynq-7000 all programmable soc technical reference manual.” [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[117] M. Vogt, G. Hempel, J. Castrillon, and C. Hochberger, “Gcc-plugin for automated accelerator generation and integration on hybrid fpga-socs,” *arXiv preprint arXiv:1509.00025*, 2015.

[118] “C++ reference: The ”this” pointer.” [Online]. Available: http://en.cppreference.com/w/cpp/language/this

[119] M. Shapiro and S. Horwitz, “Fast and accurate flow-insensitive points-to analysis,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997, pp. 1–14.

[120] O. Lhotak and L. Hendren, “Scaling java points-to analysis using s park,” in *Compiler Construction*. Springer, 2003, pp. 153–169.

[121] "Gnu nm utility." [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_4.html

[122] "Gnu objcopy utility." [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_5.html

[123] "C++ templates," http://en.cppreference.com/w/cpp/language/templates, accessed: 25-09-2017.

[124] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *ACM SIGPLAN Notices*, vol. 29, no. 6. ACM, 1994, pp. 242–256.

[125] "Alias analysis by gcc." [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Alias-analysis.html

Appendices

CHAPTER A

Rule Enforcement in C++

The body of the verifyIndirect function (chapter 5.10.2) is available in Listing A.1. This code was automatically generated from the Rule Mapper and it is associated with a single μRTZVisor compilation, due to the static addresses used.

Listing A.1: verifyIndirect function body.

```
1  bool verifyIndirect(int ID , uint32_t* ptr_add,void * ptr_val){
2  int index1 = *(++add_ptr), index2 = *(add_ptr+2),index3 = *(add_ptr+3);
3  int val_val = *((int*)ptr_val);
4
5  if( ID == 4888 && index1 >= 0 && index1 <= 96 ) {
6                  return false;
7  }
8
9  if( ID == 6102 && index1 >= 0 && index1 <= 12 ) {
10                 return false;
11 }
12
13 if( ID == 23009 && index1 >= 0 && index1 <= 96 ) {
14                 return false;
15 }
16
17 if( ID == 6103 && index1 >= 0 && index1 <= 12 ) {
18                 return false;
19 }
20
21 if( ID == 29109 && index1 >= 0 && index1 <= 2 ) {
22         if(index2 >= 0 && index2 <= 3 ) {
23                 return true;
24         }
25 }
26
27 if( ID == 18007 && index1 >= 0 && index1 <= 2 ) {
28         if(index2 >= 0 && index2 <= 3 ) {
29                 return true;
30         }
31 }
32
33 if( ID == 28345 && index1 >= 0 && index1 <= 32 ) {
34                 return false;
35 }
36
37 if(ID ==29204 && *((int*)ptr_val) >= 0 && *((int*)ptr_val) <= 1) {
38         return true;
39 }
40
41 if(ID ==29807 && ( *((int*)ptr_val) & 0x1) == 0x1 && ( ~*((int*)ptr_val) & 0x0) == 0x0 ) {
42         return true;
43 }
44
45 if(ID ==30599 && ( *((int*)ptr_val) & 0x1) == 0x1 && ( ~*((int*)ptr_val) & 0x0) == 0x0 ) {
46         return true;
47 }
```

```
48
49  if(ID == 31833 && index1 >= 0x1077e4  && index1 <= 0x107be0){
50          if(index1 >= 0x1077e4 && index1<= 0x1079e0){
51                  int addr = 0x1077e4;
52                  if(index1 >= addr + 0x30 && index1<= addr + 0x34 ){
53                          if(ID ==31833) {
54                                  return false;
55                          }
56                  }
57                  if(index1 >= addr + 0x34 && index1<= addr + 0x38 ){
58                          if(ID ==31833) {
59                                  return false;
60                          }
61                  }
62                  if(index1 >= addr + 0x38 && index1<= addr + 0x3c ){
63                          if(ID ==31833) {
64                                  return false;
65                          }
66                  }
67          }
68
69          if(index1 >= 0x1079e0 && index1<= 0x107bdc){
70                  int addr = 0x1079e0;
71                  if(index1 >= addr + 0x30 && index1<= addr + 0x34 ){
72                          if(ID ==31833) {
73                                  return false;
74                          }
75                  }
76                  if(index1 >= addr + 0x34 && index1<= addr + 0x38 ){
77                          if(ID ==31833) {
78                                  return false;
79                          }
80                  }
81                  if(index1 >= addr + 0x38 && index1<= addr + 0x3c ){
82                          if(ID ==31833) {
83                                  return false;
84                          }
85                  }
86          }
87  }
88
89  if(ID == 31833 && index1 >= 0x1077e4  && index1 <= 0x107be0){
90          if(index1 >= 0x1077e4 && index1<= 0x1079e0){
91                  int addr = 0x1077e4;
92                  if(index1 >= addr + 0x24 && index1<= addr + 0x28 ){
93                          return false;
94                  }
95                  if(index1 >= addr + 0x28 && index1<= addr + 0x2c ){
96                          return true;
97                  }
98                  if(index1 >= addr + 0x2c && index1<= addr + 0x30 ){
99                          return true;
100                 }
101         }
102         if(index1 >= 0x1079e0 && index1<= 0x107bdc){
103                 int addr = 0x1079e0;
104                 if(index1 >= addr + 0x24 && index1<= addr + 0x28 ){
105                         return false;
106                 }
107                 if(index1 >= addr + 0x28 && index1<= addr + 0x2c ){
108                         return true;
109                 }
110                 if(index1 >= addr + 0x2c && index1<= addr + 0x30 ){
111                         return true;
112                 }
113         }
114 }
115
116 if(ID == 21246 && index1 >= 0x1077e4  && index1 <= 0x107be0){
117         if(index1 >= 0x1079d4 && index1<= 0x1079d8){
118                 int addr = 0x1079d4;
119                 return true;
120         }
121         if(index1 >= 0x1079d8 && index1<= 0x1079dc){
```

```
122                     int addr = 0x1079d8;
123                     return true;
124             }
125             if(index1 >= 0x1079dc && index1<= 0x1079e0){
126                     int addr = 0x1079dc;
127                     return true;
128             }
129     }
130
131     if(ID == 31795 && index1 >= 0x1077e4  && index1 <= 0x107be0){
132             if(index1 >= 0x1079d4 && index1<= 0x1079d8){
133                     int addr = 0x1079d4;
134                     return true;
135             }
136             if(index1 >= 0x1079d8 && index1<= 0x1079dc){
137                     int addr = 0x1079d8;
138                     return true;
139             }
140             if(index1 >= 0x1079dc && index1<= 0x1079e0){
141                     int addr = 0x1079dc;
142                     return true;
143             }
144     }
145
146     if(ID == 24051 && index1 >= 0x1077e4  && index1 <= 0x107be0){
147             if(index1 >= 0x1079d4 && index1<= 0x1079d8){
148                     int addr = 0x1079d4;
149                     return true;
150             }
151             if(index1 >= 0x1079d8 && index1<= 0x1079dc){
152                     int addr = 0x1079d8;
153                     return true;
154             }
155             if(index1 >= 0x1079dc && index1<= 0x1079e0){
156                     int addr = 0x1079dc;
157                     return true;
158             }
159     }
160
161     if(ID == 21306 && index1 >= 0x1077e4  && index1 <= 0x107be0){
162             if*((int*)ptr_val) >= 0 && *((int*)ptr_val) <= 1) {
163                     return true;
164             }
165     }
166
167     if(ID == 9941 && index1 >= 0x1077e4  && index1 <= 0x107be0){
168             if(index1 >= 0x1077e4 && index1<= 0x1079e0){
169                     int addr = 0x1077e4;
170                     return true;
171             }
172             if(index1 >= 0x1079e0 && index1<= 0x107bdc){
173                     int addr = 0x1079e0;
174                     return true;
175             }
176     }
177
178     if(ID == 3474 && index1 >= 0x1077e4  && index1 <= 0x107be0){
179             if(index2 >= 0 && index2 <= 2 ) {
180                     return true;
181             }
182     }
183     return false;
184     }
```

CHAPTER B

Generated files from VMM's Ontological Model

The implementation files, automatically generated from the devised μRTZVisor ontological model, for the Data Integrity security countermeasure, are depicted below.

B.1 Astract Rules File

Listing B.1: Automatically generated file with the abstract data integrity rules for μRTZVisor.

```
1  immutable_vec_element ::urtzvisor::GuestManager::instance.guestList.memoryConfig.bitmap
       [0]
2  immutable ::urtzvisor::InterruptManager::private_interrupt_attribution
3  immutable ::urtzvisor::InterruptManager::shared_interrupts.mBitmap
4  immutable ::urtzvisor::Hypercalls::hypercall_table
5  immutable ::urtzvisor::InterruptManager::handler_table
6  immutable ::urtzvisor::InterruptManager::private_interrupts.mBitmap
7  immutable ::urtzvisor::GuestManager::instance.guestList.interruptConfig.bitmap
8  register_val_pattern ANONYMOUS.pt_control_reg XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX1
9  range_int ::urtzvisor::GuestManager::instance.mCurrentGuest 0 1
```

B.2 Instrumentation Input File

Listing B.2: Automatically generated file used as input for the Instrumentation Pass.

```
1  ::urtzvisor::GuestManager::instance.guestList.memoryConfig.bitmap
2  ::urtzvisor::InterruptManager::private_interrupt_attribution
3  ::urtzvisor::InterruptManager::shared_interrupts.mBitmap
4  ::urtzvisor::Hypercalls::hypercall_table
5  ::urtzvisor::InterruptManager::handler_table
6  ::urtzvisor::InterruptManager::private_interrupts.mBitmap
7  ::urtzvisor::GuestManager::instance.guestList.interruptConfig.bitmap
8  ANONYMOUS.pt_control_reg
9  ::urtzvisor::GuestManager::instance.mCurrentGuest
10 from F8F00000 to F9F00000
```

B.3 Memory Layout File

Listing B.3: Automatically generated file for *μ*RTZVisor memory layout.

```
struct Vcp15 size 80 [bytes]
Field> integer_type int PRRR
Bytes: 48                Bits: 0
Field> integer_type int NMRR
Bytes: 48                Bits: 32
Field> integer_type int PAR
Bytes: 40                Bits: 32
Field> integer_type int TTBR0
Bytes: 8                 Bits: 32
Field> integer_type int TTBR1
Bytes: 16                Bits: 0
Field> integer_type int CSSELR
Bytes: 0                 Bits: 0
Field> integer_type int 56,0
Bytes: 56                Bits: 0
Field> integer_type int SCTLR
Bytes: 0                 Bits: 32
Field> integer_type int DFSR
Bytes: 24                Bits: 32
Field> integer_type int IFSR
Bytes: 32                Bits: 0
Field> integer_type int FCSEIDR
Bytes: 56                Bits: 32
Field> integer_type int CONTEXTIDR
Bytes: 64                Bits: 0
Field> integer_type int TPIDRURO
Bytes: 72                Bits: 0
Field> integer_type int DACR
Bytes: 24                Bits: 0
Field> integer_type int IFAR
Bytes: 40                Bits: 0
Field> integer_type int TTBCR
Bytes: 16                Bits: 32
Field> integer_type int TPIDRPRW
Bytes: 72                Bits: 32
Field> integer_type int ACTLR
Bytes: 8                 Bits: 0
Field> integer_type int DFAR
Bytes: 32                Bits: 32
Field> integer_type int TPIDRURW
Bytes: 64                Bits: 32
struct CoreRegs size 116 [bytes]
Field> integer_type int sp_abt
Bytes: 80                Bits: 0
Field> integer_type int lr_abt
Bytes: 80                Bits: 32
Field> integer_type int spsr_abt
Bytes: 88                Bits: 0
Field> integer_type int sp_irq
Bytes: 104               Bits: 0
Field> integer_type int spsr_mon
Bytes: 48                Bits: 32
Field> integer_type int sp_sys
Bytes: 72                Bits: 0
Field> integer_type int lr_undef
Bytes: 96                Bits: 0
Field> integer_type int sp_svc
Bytes: 56                Bits: 32
Field> array_type int* regs size 13
Bytes: 0                 Bits: 0
Field> integer_type int spsr_irq
Bytes: 112               Bits: 0
Field> integer_type int lr_svc
Bytes: 64                Bits: 0
Field> integer_type int lr_mon
Bytes: 56                Bits: 0
Field> integer_type int lr_sys
```

```
68  Bytes: 72                Bits: 32
69  Field> integer_type int lr_irq
70  Bytes: 104               Bits: 32
71  Field> integer_type int spsr_undef
72  Bytes: 96                Bits: 32
73  Field> integer_type int sp_undef
74  Bytes: 88                Bits: 32
75  Field> integer_type int spsr_svc
76  Bytes: 64                Bits: 32
77  struct MemoryManager size 1 [bytes]
78  struct Gic size 1 [bytes]
79  struct Hypercalls size 1 [bytes]
80  struct MemoryConfiguration size 0 [bytes]
81  Field> array_type int* bitmap size 3
82  Bytes: 0                 Bits: 0
83  struct VirtualMachine size 448 [bytes]
84  Field> record_type Vcpu mVcpu
85  Bytes: 0                 Bits: 0
86  Field> record_type Vgic mVgic
87  Bytes: 192               Bits: 32
88  struct Vcpu size 196 [bytes]
89  Field> record_type CoreRegs mVcore
90  Bytes: 0                 Bits: 0
91  Field> record_type Vcp15 mVcp15
92  Bytes: 112               Bits: 32
93  struct InterruptDistributor size 4096 [bytes]
94  Field> array_type int* reserved9 size 232
95  Bytes: 2144              Bits: 0
96  Field> array_type int* reserved8 size 232
97  Bytes: 1120              Bits: 0
98  Field> integer_type int ICDIIDR
99  Bytes: 8                 Bits: 32
100 Field> array_type int* reserved7 size 61
101 Bytes: 776               Bits: 32
102 Field> array_type int* reserved6 size 29
103 Bytes: 648               Bits: 32
104 Field> array_type int* ICPIDR_4_7 size 4
105 Bytes: 4048              Bits: 0
106 Field> array_type int* reserved1 size 29
107 Bytes: 8                 Bits: 32
108 Field> array_type int* reserved5 size 29
109 Bytes: 520               Bits: 32
110 Field> array_type int* reserved4 size 29
111 Bytes: 392               Bits: 32
112 Field> array_type int* reserved3 size 29
113 Bytes: 264               Bits: 32
114 Field> array_type int* SPI_STATUS size 2
115 Bytes: 3328              Bits: 32
116 Field> array_type int* reserved2 size 29
117 Bytes: 136               Bits: 32
118 Field> array_type int* ICPIDR_0_3 size 4
119 Bytes: 4064              Bits: 0
120 Field> integer_type int ICDDCR
121 Bytes: 0                 Bits: 0
122 Field> array_type int* reserved11 size 125
123 Bytes: 3336              Bits: 32
124 Field> array_type int* reserved12 size 51
125 Bytes: 3840              Bits: 32
126 Field> array_type int* reserved10 size 58
127 Bytes: 3096              Bits: 0
128 Field> array_type int* ICDIPTRx size 24
129 Bytes: 2048              Bits: 0
130 Field> array_type int* ICCIDRx size 4
131 Bytes: 4080              Bits: 0
132 Field> array_type int* ICDIPRx size 24
133 Bytes: 1024              Bits: 0
134 Field> array_type int* ICDABRx size 3
135 Bytes: 768               Bits: 0
136 Field> array_type int* ICDISRx size 3
137 Bytes: 128               Bits: 0
138 Field> integer_type int ICDICTR
139 Bytes: 0                 Bits: 32
140 Field> array_type int* ICDICPRx size 3
141 Bytes: 640               Bits: 0
```

```
142 Field> integer_type int PPI_STATUS
143 Bytes: 3328              Bits: 0
144 Field> array_type int* ICDISERx size 0
145 Bytes: 256               Bits: 0
146 Field> integer_type int ICDSGIR
147 Bytes: 3840              Bits: 0
148 Field> array_type int* ICDICERx size 0
149 Bytes: 384               Bits: 0
150 Field> array_type int* ICDICFRx size 6
151 Bytes: 3072              Bits: 0
152 Field> array_type int* ICDISPRx size 3
153 Bytes: 512               Bits: 0
154 struct MemorySlot size 8 [bytes]
155 Field> integer_type int address
156 Bytes: 0                 Bits: 0
157 Field> integer_type int size
158 Bytes: 0                 Bits: 32
159 struct Guest size 508 [bytes]
160 Field> integer_type int id
161 Bytes: 0                 Bits: 0
162 Field> record_type VirtualMachine vm
163 Bytes: 56                Bits: 32
164 Field> record_type InterruptConfiguration interruptConfig
165 Bytes: 48                Bits: 0
166 Field> array_type int* name size 32
167 Bytes: 0                 Bits: 32
168 Field> record_type MemoryConfiguration memoryConfig
169 Bytes: 32                Bits: 32
170 struct Vgic size 256 [bytes]
171 Field> integer_type int ICCICR
172 Bytes: 0                 Bits: 0
173 Field> array_type int* vICDISERx size 3
174 Bytes: 224               Bits: 32
175 Field> integer_type int ICCPMR
176 Bytes: 0                 Bits: 32
177 Field> array_type int* ICDICFRx size 6
178 Bytes: 200               Bits: 32
179 Field> integer_type int ICCBPR
180 Bytes: 8                 Bits: 0
181 Field> array_type int* ICDIPRx size 24
182 Bytes: 8                 Bits: 32
183 Field> array_type int* vICDISPRx size 3
184 Bytes: 240               Bits: 0
185 Field> array_type int* ICDIPTRx size 24
186 Bytes: 104               Bits: 32
187 struct PrivateTimer size 1 [bytes]
188 struct MemoryImplementation size 1 [bytes]
189 struct GuestManager size 1020 [bytes]
190 Field> array_type Guest* guestList size 2
191 Bytes: 0                 Bits: 0
192 Field> integer_type int mCurrentGuest
193 Bytes: 1016              Bits: 0
194 struct InterruptManager size 1 [bytes]
195 struct CpuInterface size 256 [bytes]
196 Field> integer_type int ICCIDR
197 Bytes: 248               Bits: 32
198 Field> integer_type int ICCHPIR
199 Bytes: 24                Bits: 0
200 Field> integer_type int ICCICR
201 Bytes: 0                 Bits: 0
202 Field> integer_type int ICCIAR
203 Bytes: 8                 Bits: 32
204 Field> integer_type int ICCEOIR
205 Bytes: 16                Bits: 0
206 Field> integer_type int ICCABPR
207 Bytes: 24                Bits: 32
208 Field> array_type int* reserved1 size 55
209 Bytes: 32                Bits: 0
210 Field> integer_type int ICCBPR
211 Bytes: 8                 Bits: 0
212 Field> integer_type int ICCRPR
213 Bytes: 16                Bits: 32
214 Field> integer_type int ICCPMR
215 Bytes: 0                 Bits: 32
```

```
struct ::urtzvisor::InterruptManager::private_interrupts size 12 [bytes]
Field> array_type int* mBitmap size 3
Bytes: 0                Bits: 0
struct GuestConfig size 16 [bytes]
Field> integer_type int num_slots
Bytes: 8                Bits: 0
Field> integer_type int entry_point
Bytes: 0                Bits: 32
struct InterruptConfiguration size 12 [bytes]
Field> array_type int* bitmap size 3
Bytes: 0                Bits: 0
struct PrivateTimerRegs size 16 [bytes]
Field> integer_type int pt_control_reg
Bytes: 8                Bits: 0
Field> integer_type int pt_interrupt_status_reg
Bytes: 8                Bits: 32
Field> integer_type int pt_load_reg
Bytes: 0                Bits: 0
Field> integer_type int pt_counter_reg
Bytes: 0                Bits: 32
struct ::urtzvisor::InterruptManager::shared_interrupts size 12 [bytes]
Field> array_type int* mBitmap size 3
Bytes: 0                Bits: 0
static ::urtzvisor::InterruptManager::shared_interrupts record_type TEMPLATE
static ::urtzvisor::InterruptManager::private_interrupts record_type TEMPLATE
static ::urtzvisor::Hypercalls::hypercall_table array_type integer_type
elements 32
static ::urtzvisor::InterruptManager::handler_table array_type integer_type
elements 96
static ::urtzvisor::InterruptManager::private_interrupt_attribution array_type integer_type
elements 96
static ::urtzvisor::GuestManager::instance record_type GuestManager
static ANONYMOUS record_type PrivateTimerRegs
At       f8f00600
```