Universidad Politécnica de Valencia

Departamento de Informática de Sistemas y Computadores

Metodología para hipervisores seguros utilizando técnicas de validación formal

Tesis Doctoral Presentada Por:

Salvador Peiró Frasquet

Dirigida por:

Dr. Alfons Crespo i Lorente

Dr. Miguel Masmano Tello

Dr. José Simó Ten

Valencia, December 22, 2015

This page is intentionally left blank.

Acknowledgements

This words are dedicated to those that have provided his support to perform this work, specially, to my supervisors, colleagues, friends and family.

This page is intentionally left blank.

Abstract

The availability of new processors with more processing power for embedded systems has raised the development of applications that tackle problems of greater complexity. Currently, the embedded applications have more features, and as a consequence, more complexity. For this reason, there exists a growing interest in allowing the secure execution of multiple applications that share a single processor and memory. In this context, partitioned system architectures based on hypervisors have evolved as an adequate solution to build secure systems.
One of the main challenges in the construction of secure partitioned systems is the verification of the correct operation of the hypervisor, since, the hypervisor is the critical component on which rests the security of the partitioned system. Traditional approaches for Validation and Verification (V&V), such as testing, inspection and analysis, present limitations for the exhaustive validation and verification of the system operation, due to the fact that the input space to validate grows exponentially with respect to the number of inputs to validate. Given this limitations, verification techniques based in formal methods arise as an alternative to complement the traditional validation techniques.

This dissertation focuses on the application of formal methods to validate the correctness of the partitioned system, with a special focus on the XtratuM hypervisor. The proposed methodology is evaluated through its application to the hypervisor validation. To this end, we propose a formal model of the hypervisor based in Finite State Machines (FSM), this model enables the definition of the correctness properties that the hypervisor design must fulfill. In addition, this dissertation studies how to ensure the functional correctness of the hypervisor implementation by means of deductive code verification techniques.

Last, we study the vulnerabilities that result of the loss of confidentiality (CWE-200 [CWE08b]) of the information managed by the partitioned system. In this context, the vulnerabilities (infoleaks) are modeled, static code analysis techniques are applied to the detection of the vulnerabilities, and last the proposed techniques are validated by means of a practical case study on the Linux kernel that is a component of the partitioned system.

This page is intentionally left blank.

Resumen

La disponibilidad de nuevos procesadores más potentes para aplicaciones empotradas ha permitido el desarrollo de aplicaciones que abordan problemas de mayor complejidad. Debido a esto, las aplicaciones empotradas actualmente tienen más funciones y prestaciones, y como consecuencia de esto, una mayor complejidad. Por este motivo, existe un interés creciente en permitir la ejecución de múltiples aplicaciones de forma segura y sin interferencias en un mismo procesador y memoria. En este marco surgen las arquitecturas de sistemas particionados basados en hipervisores como una solución apropiada para construir sistemas seguros.
Uno de los principales retos en la construcción de sistemas particionados, es la verificación del correcto funcionamiento del hipervisor, dado que es el componente crítico sobre el que descansa la seguridad de todo el sistema particionado. Las técnicas tradicionales de V&V, como testing, inspección y análisis, presentan limitaciones para la verificación exhaustiva del comportamiento del sistema, debido a que el espacio de entradas a verificar crece de forma exponencial con respecto al número de entradas a verificar. Ante estas limitaciones las técnicas de verificación basadas en métodos formales surgen como una alternativa para completar las técnicas de validación tradicional.

Esta disertación se centra en la aplicación de métodos formales para validar la corrección del sistema particionado, en especial del hipervisor XtratuM. La validación de la metodología se realiza aplicando las técnicas propuestas a la validación del hipervisor. Para ello, se propone un modelo formal del hipervisor basado en máquinas de autómatas finitos, este modelo formal permite la definición de las propiedades que el diseño hipervisor debe cumplir para asegurar su corrección. Adicionalmente, esta disertación analiza cómo asegurar la corrección funcional de la implementación del hipervisor por medio de técnicas de verificación deductiva de código.

Por último, se estudian las vulnerabilidades de tipo *information leak* (CWE-200 [CWE08b]) debidas a la perdida de la confidencialidad de la información manejada en el sistema particionado. En este ámbito se modelan las vulnerabilidades, se aplican técnicas de análisis de código para la detección de vulnerabilidades en base al modelo definido y por último se valida la técnica propuesta por medio de un caso práctico sobre el núcleo del sistema operativo Linux que forma parte del sistema particionado.

This page is intentionally left blank.

Resum

La disponibilitat de nous processadors amb major potencia de còmput per a aplicacions empotrades ha permès el desenvolupament de aplicacions que aborden problemes de major complexitat. Degut a açò, les aplicacions empotrades actualment tenen més funcions i prestacions, i com a conseqüència, una major complexitat. Per aquest motiu, existeix un interès creixent en per permetre la execució de múltiples aplicacions de forma segura i sense interferències en un mateix processador i memòria. En aquest marc sorgeixen les arquitectures de sistemes particionats basats en hipervisors com una solució apropiada per a la construcció de sistemes segurs
Un dels principals reptes en la construcció de sistemes particionats, es la verificació del correcte funcionament del hipervisor, donat que aquest es el component crític sobre el que descansa la seguretat del sistema particionat complet. Les tècniques tradicionals de V&V, com són el testing, inspecció i anàlisi, presenten limitacions que fan impracticable la seva aplicació per a la verificació exhaustiva del comportament del sistema, degut a que el espai de entrades a verificar creix de forma exponencial amb el nombre de entrades a verificar. Front a aquestes limitacions les tècniques de verificació basades en mètodes formals sorgeixen com una alternativa per a completar les tècniques de validació tradicional.

Aquesta dissertació es centra en la aplicació de mètodes formals per a validar la correcció del sistema particionat, en especial d del hipervisor XtratuM. La validació de la metodología es realitza aplicant les tècniques proposades a la validació del hipervisor. Per a aquest fi, es proposa un model formal del hipervisor basat en màquines de estats finits (FSM), aquest model formal permet la definició de les propietats que el disseny del hipervisor deu de complir per assegurar la seva correcció. Addicionalment, aquesta dissertació analitza com assegurar la correcció funcional de la implementació del hipervisor mitjançant tècniques de verificació deductiva de codi.

Per últim, s'estudien les vulnerabilitats de tipus *information leak* (CWE-200 [CWE08b]) degudes a la pèrdua de la confidencialitat de la informació gestionada per el sistema particionat. En aquest àmbit, es modelen les vulnerabilitats, s'apliquen tècniques de anàlisis de codi per a la detecció de les vulnerabilitats en base al model definit, per últim es valida la tècnica proposada mitjançant un cas pràctic sobre el nucli del sistema operatiu Linux que forma part de l'arquitectura particionada.

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

List of Listings

This page is intentionally left blank.

Chapter 1
Introduction

This chapter introduces the main topics addressed in this dissertation. First, the section 1.1 presents the topics of secure hypervisors construction and validation. Next, the section 1.2 details the main motivation and objectives of the work followed by our contributions section 1.3. Last, the section 1.4 defines the outline, and section 1.5 sets the research context where this work has been performed.

1.1 Secure partitioned systems

The increasing complexity of current software systems complicates its exhaustive V&V by means of traditional validation techniques, such as testing, analysis and inspection [D+09]. This is caused by the fact that the size of the input space to validate grows exponentially with the number of inputs, making impracticable the exhaustive validation of the input space as the number of inputs to test increases. In this context, the V&V techniques based on formal methods arise as an alternative that complements traditional V&V techniques, and, enable to reduce the input space, by means of abstract execution and state-space reduction techniques.

The study of the security vulnerability history of current operating systems [CMW+11] provides insight about the defects that affect the security and quality of current operating systems. These defects can lead to: (1) Failures in critical systems, that affect the safety of humans. (2) Security breaches in systems used by companies and government institutions. (3) Security issues in the widespread consumer services and devices, such as: mobile phones, laptops, servers, routers, ...

In the above scenario, this thesis focuses on the analysis of the secure hypervisors (subsection 1.1.1), and, the application of formal validation methods (subsection 1.1.2) to improve the hypervisors security.

1.1.1 Secure hypervisors

The availability of new processors for embedded applications has raised new possibilities for these applications. Now, the embedded applications have more functionalities and, as consequence, more complexity. There exist a growing interest in enabling multiple applications to share a single processor and memory. To facilitate such a model the execution time and memory space of each application must be protected from other applications in the system.

Partitioned software architectures represent the future of secure systems. They have evolved to fulfill security and avionics requirements where predictability is extremely important [fAR12].

The separation kernel proposed by Rushby et al. [Rus81a] established a combination of hardware and software to allow multiple functions to be performed on a common set of physical resources without interference. The MILS initiative is a joint research effort between academia, industry, and government to develop and implement a high-assurance, real-time architecture for embedded systems. The technical foundation adopted for the so-called MILS architecture is a separation kernel. Also, the ARINC-653 standard [Ai96] uses these principles to define a baseline operating environment for application software used within IMA, based on a partitioned architecture.

The idea behind a partitioned system is the virtualization. This idea is present in current operating systems: processor and memory are multiplexed to processes. A physical computer is partitioned into several logical partitions, each of which looks like a real computer. Each of these partitions can have an operating system installed on it, and function as if it were a completely separate machine. Virtual machine technology can be considered a secure and efficient way to build partitioned systems. A virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. **Hypervisor** (also known as VMM [Gol74]) is a small layer of software (or a combination of software/hardware) that enables to run several independent execution environments or partitions in a single computer. The key difference between hypervisor technology and other kinds of virtualisation (such as Java virtual machine or software emulation) is the performance.

The low overhead and the reduced size of the hypervisor can be considered as an appropriated solution to achieve secure systems if it is designed following strict design criteria to meet security requirements. Its correctness can be sufficient to ensure the security of the system as a whole or, at least, the security of a set of trusted partitions. In a partitioned system, the partitions can accommodate different kinds of applications: real-time, trusted, non trusted, etc.

The concept of partitioned software architectures [Rus01] was developed to address security and safety issues. The central design criteria behind this concept consists in isolating modules of the system in *partitions*. The temporal and spatial isolation properties of the partitioned software architectures are the key aspects in partitioned systems. Therefore, the validation and verification of the hypervisor properties are fundamental, and, are introduced in the next subsection 1.1.2.

1.1.2 Validation and verification of secure hypervisors

The V-Model software development cycle schedules phases [BF+14, §7] for the construction and verification of the software product. The verification phases are counterparts of the construction phases that verify that the outputs of each phase are correctly built using an incremental approach, these are depicted in blue in Figure 1.1. In the V-Model, the traditional V&V phases are performed by means of testing, inspection and analysis activities [BF+14, §10] to achieve the defined testing objectives:

- *Traditional System verification activities.*

  System testing activities consists in the specification and development of a testsuite that checks the system specification (system requirements).

- *Traditional Integration verification activities.*

  Integration testing activities consist in the specification and development of a testsuite that checks the components specification (component contracts).

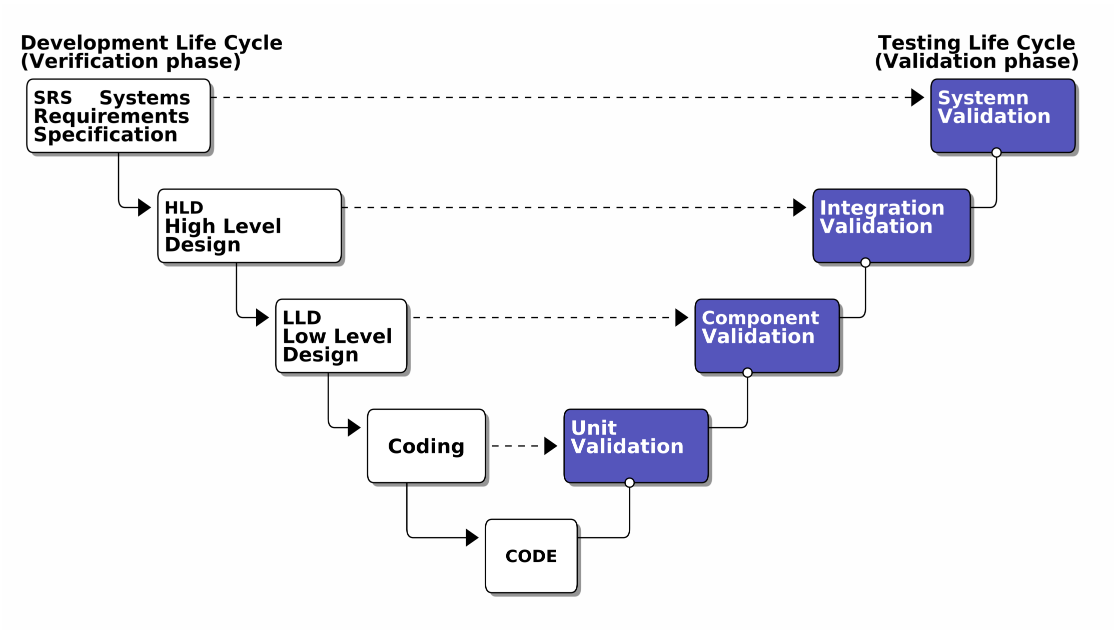

Figure 1.1: V&V phases on V Model Software Development Cycle

- *Traditional Unit verification activities.*

  Unit testing activities consist in the specification and development of a testsuite that check the units specification (function contracts).

For each testing activities, the testing objectives are defined based on risk, criticality and prioritization analysis. Each of the testing objectives are defined in terms of: (1) Testing the whole functionality offered by the component/unit, and, (2) Ensuring that the execution of testsuite reaches a high structural code coverage value (typically above 90%). However, these objectives alone are not considered sufficient to ensure high confidence on the robustness of the software product being built [IH14].

In addition to the traditional V&V activities presented above, formal methods [BF$^+$14, §4] arise as an alternative to complement traditional V&V activities to address the limitations and challenges of V&V (subsection 1.1.3). Formal methods can be applied along the V&V phases of the V-Model, depicted in Figure 1.1:

- *Formal System verification activities.*

  System verification activities consists in the use of formal methods to check the system specification (system requirements) against a formal model.

- *Formal Integration verification activities.*

  Integration verification activities consist in the use of formal methods to check the implementation of software components against a specification of the component contracts [Mey92].

- *Formal Unit verification activities.*

  Unit verification activities consist in the use of formal methods to check the implementation of software units against a specification of the unit (functions) contracts [Mey92].

This work addresses the use of formal methods for the Validation and Verification.

1.1.3 Challenges on the validation of secure hypervisors

This section analyses the main challenges and causes that difficult the validation of secure hypervisors. Next, we discuss each of the identified challenges in a separate paragraph.

The Complexity of Secure Hypervisors

The Complexity of current software systems is a key challenge to overcome in the construction of secure hypervisor systems [D+09], since complexity of software systems directly relates to the validation complexity. The analysis of software complexity identifies two types of complexity that contribute to the increase of the complexity [D+09, §A, p. 32]:

(1) *Essential complexity*: Complexity essential to the problem to solve.
(2) *Incidental complexity*: Complexity non-essential introduced by the approach selected to solve the problem.

The analysis of incidental complexity [D+09, on COTS] notes that the use of COTS can contribute to the complexity of software products. COTS components are used as a way to reduce costs and risks by utilizing well-tested components. This is the trend of current hypervisors [Xen, Vxworks, pikeos, . . .] which are designed as COTS components. However, for COTS components to remain widely applicable need to remain highly adaptable and configurable to different application contexts, to achieve this goal COTS feature multiple components. We can identify three main state spaces that contribute to complexity depicted in Figure 1.2:

- *Input state space*, that comprises all the combinations of calls of the offered services, e.g.: `call service(parameters)`.
- *Configuration space*, that comprises all the combinations of build time configuration parameters.
- *Environment space*, that comprises all the combinations of environments where the system runs.

Each of the above state spaces contribute to the Equation 1.1 of the complexity (C) of the system depicted in Figure 1.2:

$$C(product) = C(application) \times C(code \times code\_cfg) \times C(hardware \times hardware\_cfg) \quad (1.1)$$

The difficulties of exhaustive testing for validation purposes

Based on the above complexity analysis and performing some back-of-the-envelope calculations of the size of the state space under test:

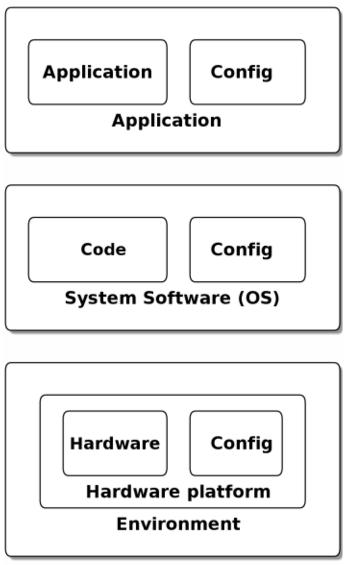

Figure 1.2: COTS product components

- Input API state space: Is the state space that consists of all the possible API call input parameters combinations. For a typical 32-bit based CPU with an small API of 30-50 services each one receiving between 1 an 5 input parameters (Linux sports 300 system calls). The total number of combinations to test exhaustively would yield a total of $50 * (2^{32})^5$ tests to be performed. Then assuming a rate of 1000 tests/second requires $8 * 10^{46}$ seconds, or $2 * 10^{39}$ years, that is not feasible.
- System Configuration state space: Is the state space that consists of all the possible configuration parameters that determine the behaviour of software, this configuration parameters, refer mainly configuration parameters that are fixed at build time.
- Environment state space: Is the state space that consists of combinations of environments where the system runs, that is, hardware platforms, etc.

Notice that we are not considering each of the state spaces separately, but the complete state space is the Cartesian product of the state spaces: $input \times config \times environment$ that results in a greater state space.

The main conclusion drawn is that due to the great size of the state space [1] traditional validation techniques alone can only test a subset of the state space, therefore, leaving untested areas of the state space, that contribute to untested and potentially faulty behaviour in the program space [And86]. Additionally, when considering traditional V&V testing is important to consider that: *"testing alone can only be used to show the presence of defects, however, it can never be used to show the complete absence of defects"* [DDH72].

[1] There exist testing techniques that enable to reduce the size of the input space, such as equivalence partitioning [BF+14, §3.2.1], by considering only a subset of the input space to test. But this reduction comes at the cost of assuming equivalence of behaviour for similar inputs as a justification to not test those inputs.

1.1.4 Terminology

The main topic of this work is the validation of secure hypervisors by means of formal methods. Therefore, a common ground for terms used in this work is required. Where possible we have employed definitions coming from standard bodies and institutions as the Institute of Electrical and Electronics Engineers (IEEE), otherwise, we provide references to the research works that define the terms we used. Through the rest of this work we use: (1) The dependability and security definitions from by Avizienis taxonomy et al. [ALRL04]. (2) The software engineering, specially validation and verification terminology refers to the IEEE Software Engineering Book (SWEBOOK) [BF+14, oEE90]. Last, a glossary of the terms and acronyms used through this work is available at section 6.3.

1.2 Motivation and main goals

The main challenges in the validation of secure hypervisors are identified in subsection 1.1.3, and are the main motivation for our work, these are summarised below:

1. The increasing complexity of secure software systems subsection 1.1.3.
2. The difficulties of traditional validation techniques for the exhaustive validation of complex software systems subsection 1.1.3.
3. The inevitability of failures in complex operating systems [LSM+98].

Main goals

The above challenges define the main objectives of the thesis, and, have encouraged us to propose solutions that address them: (1) The use of formal models to define the correctness properties of the hypervisor design. (2) The use of source code analysis techniques to verify the functional correctness of the hypervisor implementation. (3) The definition of formal models to detect information disclosure vulnerabilities (infoleaks) that compromise the confidentiality of the information managed on a secure system.

1.3 Contributions of this thesis

This thesis proposes formal methods for hypervisors verification that complement traditional V&V approaches. The contributions are presented in the order in which these apply through the software development cycle (subsection 1.1.2) as depicted in Figure 1.3. The main contributions of this thesis are:

1. At the integration validation (HLD).

   The first contribution is the proposal of a formal model of the secure hypervisor, and the definition of the properties that ensure its correctness in chapter 3.

2. At the component validation (LLD).

   The second contribution is the proposal of an approach for the verification of the functional correctness of the hypervisor components implementation, by applying deductive analysis techniques to the components source-code in chapter 4.

3. At the unit validation (Coding)

   The third contribution is the security analysis, characterization and detection of the vulnerabilities that affect the confidentiality of the information managed by secure hypervisors in chapter 5.

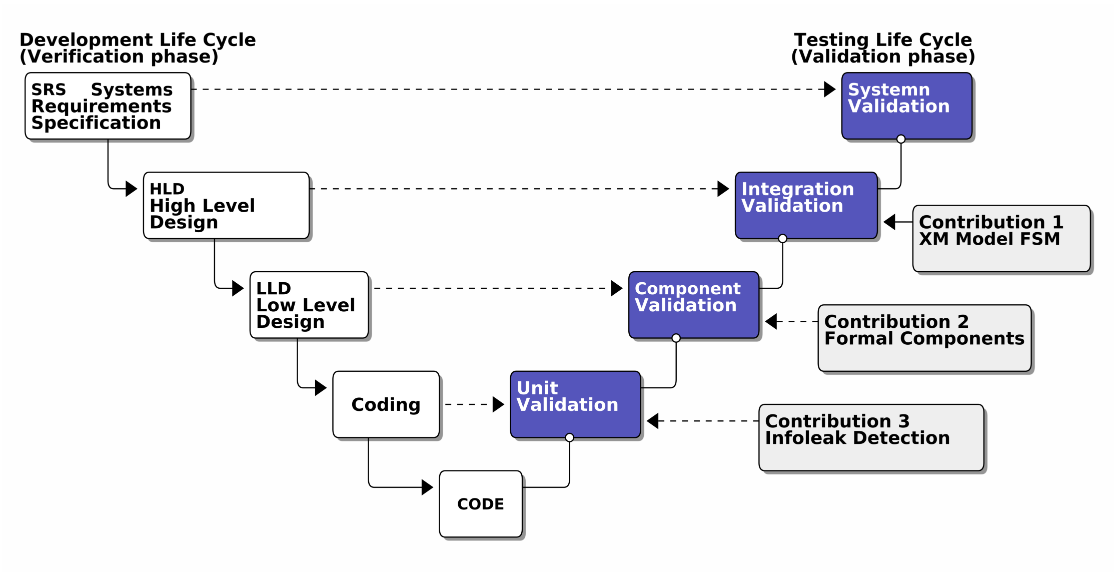

Figure 1.3: Contributions along the Hypervisor Software Development Life Cycle

1.4 Outline of this thesis

The contributions are presented in the order in which these apply through the V&V phases of the software development cycle as depicted in Figure 1.3. With the above structure in mind, the remainder of the thesis is organised as follows:

First, the chapter 2 surveys the current state of art in the V&V of secure hypervisors using formal methods. Then, the chapter 3 proposes an approach to ensure the correctness of the hypervisor based in the formal modelling of the hypervisor using FSM and the definition of the properties that the secure hypervisor must meet.
Next, the chapter 4 applies deductive verification techniques to the hypervisor components source code to ensure the functional correctness.
The chapter 5 analyses the security impact of the confidentiality vulnerabilities in the security of the partitioned system, then proposes and evaluates source code static analysis techniques to the detection of vulnerabilities that affect the confidentiality (*information leaks*) of the information managed by the components of the partitioned system.

Last, the chapter 6 summarises the main contributions of this thesis, presents the final conclusions and the open research lines that derive from this dissertation.

1.5 Research Context

This thesis has been developed in the context of the following research projects focused on the industrialization of secure hypervisor technologies.

- *DREAMS: Distributed REal-time Architecture for Mixed Criticality Systems.*

  The main goal is to develop a cross domain architecture and design tools for networked complex systems where application subsystems of different criticality, executing on networked multi core chips.

- *MultiPARTES: Multi-cores Partitioning for Trusted Embedded Systems.*

  The main goal is the development tools and solutions for building trusted embedded systems with mixed criticality components on multi core platforms.

- *OVERSEE: Open Vehicular Secure Platform*

  The overall goal of OVERSEE is to contribute to the efficiency and safety of road transport by developing the OVERSEE platform, which will provide a secure, standardized and generic communication and application platform for vehicles.

- *HI-PartES: High Integrity Partitioned Embedded Systems*

  The main goal of HI-PartES is the development of a execution platform providing virtualisation, and the development of tools oriented to model the development of high integrity partitioned systems.

In the above research context, the techniques presented in this dissertation have contributed to improve the quality of the developed software products. In addition, the results of this thesis have been presented in national and international publications, that are listed in chapter 6.

Chapter 2

Secure Hypervisor Verification

This chapter present the current state of the art in the construction of secure hypervisors using formal verification techniques. First, the section 2.1 presents the approaches that are applied currently to the formal verification of hypervisors. Next, the section 2.2 presents the application of formal methods in safety critical standards. Last, the section 2.3 summarises the conclusions of the chapter, that motivate the main contributions of this dissertation.

2.1 Formal methods in secure hypervisors

Formal validation of operating systems has been an on going research topic on operating systems [Rus81b] to achieve the security and safety critical objectives mandated by the safety standards [fAR92, Cri12, IEC10] to reach the assurance levels required for the industrial qualification of the system. The exhaustive verification of software systems by means of formal methods, presents various advantages over traditional validation techniques: (1) The formal verification enables the exhaustive verification of software in an automated way. (2) The formal verification reduces the number of defects found during after deployment. (3) The above two advantages lead to reduce the development costs of validation and testing phases in software projects.

The wide adoption of the virtualisation technologies, and its implantation in critical systems as solutions to provide fault isolation, has led to the application of formal methods for the validation of hypervisors. Next, we review the use of formal methods for the validation of secure hypervisors on the open literature. These are grouped in three main groups according to the formal verification theories applied (Figure 2.1):

- subsection 2.1.1 Deductive verification
- subsection 2.1.2 Theorem provers
- subsection 2.1.3 Static code analysis

2.1.1 Deductive verification

Baumann et al. [BBBB10] perform the formal verification of the PikeOS microkernel to reach Common Criteria (CC) qualification [Cri12]. The objective is the application of the source code deductive techniques to verify the functional correctness of the PikeOS microkernel. The authors achieve their objective by using the *Verifying C compiler (VCC)* [CDH+09] that enables the

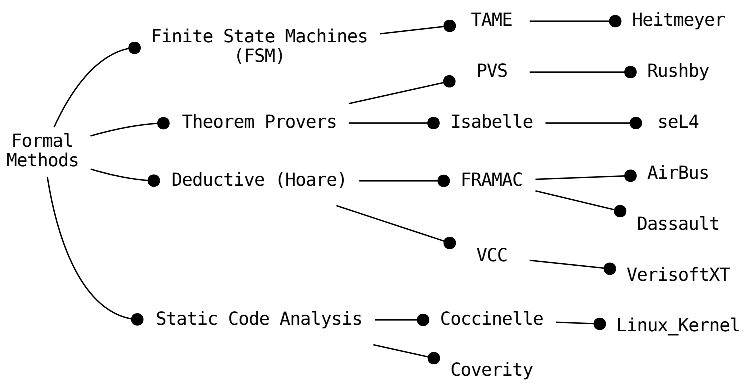

Figure 2.1: Formal methods applied to hypervisor verification.

deductive verification of the microkernel code annotated with function contracts to ensure the functional correctness.

Souyris et al. [SWDD09] present a methodology for the integration of formal verification techniques in the development cycle of the avionics software products developed at AirBus. The objective of the work is qualifying the software products for DO-178B [fAR92]. To achieve the objective the authors apply source code deductive verification techniques [CKK+12] during the software V&V phases of the Software Development Life Cycle (SDLC).

D. Pariente et al. [PL10] present the results and lessons learn on the application of source code verification techniques [CKK+12] for the verification of Industrial C code developed at Dassault Aviation.

Frama-C [CKK+12] is an open-source static analysis tool that targets ANSI C programs, constructed with a plugin architecture. That allows one to connect different kinds of analysis tools together such that they can cooperate and provide precise results. Frama-C is based on the work of Hoare [Hoa69].

2.1.2 Theorem provers

Klein et al. [KEH+09] perform the formal verification of the seL4 microkernel. The objective of the work is to prove functional correctness of the hypervisor implementation with respect to an abstract hypervisor specification. The target of the verification is a simplified version of the seL4 hypervisor for the ARM architecture, that is comprised of 8700 C Source Lines of Code (SLOC) and 600 assembly SLOC. The verification is performed using the Isabelle [Pau94] theorem prover, to prove the functional correctness required to reach the Evaluation Assurance Level 7 (EAL7) of the CC standard [Cri12]. The presented approach makes the following assumptions: (1) correctness of compiler, assembly code and hardware.

Heitmeyer et al. [HALM08] present the formal verification of security properties on the ED (Embedded Device). The objective is to prove the security properties of the ED to reach CC certification [Cri12]. To achieve the objective the authors present an innovative approach that reduces the verification costs by means of code partitioning techniques based of the criticality of the code to verify. This has the main benefit of reducing the amount of critical code to verify to the 10% of the total. Additionally, the approach presented enables the use of the theorem prover PVS [ORS92] based on annotations of the security properties to verify on the source code.

2.1.3 Static code analysis

J. L. Lawall et al. developed Coccinelle [LBP+09] a tool that performs control-flow based program searches and transformations in C code. Coccinelle is actively used to perform API evolutions and identify defects on the Linux kernel [LBP+09].

Coverity Prevent [Cov02] is a source code static analysis tool developed by Coverity. The tool was originated at Stanford META/xgcc project to find defects in source-code, the authors evaluate the tool effectiveness on finding defects on open-source projects ranging from OpenSSL to the Linux Kernel [CEH02].

2.2 Formal methods in safety standards

This section surveys and analyzes the application of formal methods on safety-critical standards. The survey targets the field of software development of safety-critical systems [ALRL04] and is based on the work of J. Bowen [Bow93]. Due to the nature of safety-critical systems, the majority of standards are safety related, however, security related standards as CC [Cri12] have also been considered. The selection, which is summarized in Table 2.5, although incomplete (as there are more than 170 safety standards [Bow93]), considers the current developments in the areas of certification of secure hypervisors:

(1) Avionics: Requirements and Technical Concepts for Aviation (DO-178B).
(2) Space industry: European Cooperation for Space Standardization (ECSS).
(3) Industrial: International Electro-technical Commission (IEC).
(4) Information Technologies (IT) security: CC.

2.2.1 RTCA Standards

The US Radio Technical Commission for Aeronautics (RTCA) produced a guideline on Software Considerations in Airborne Systems and Equipment Certification (DO-178A) [fAR85] and defines five Safety Levels ranging from Level E (lowest) to Level A (highest) (see Table 2.1). Initially DO-178A did not explicitly recognise formal methods as part of accepted practice. However, the DO-178B [fAR92] guideline was updated in 1992 and completely rewritten to include a very brief reference to formal methods in [fAR92, subsection 12.3.1] This gives a general introduction to formal methods and mentions three levels of rigour: (1) formal specification with no proofs, (2) manual proofs and (3) automatically checked or generated proofs. It is now possible for a manufacturer following the DO-178B guideline to make use of formal methods in the context of aircraft certification, although it is incumbent on the manufacturer to justify its use.

In the latest update to the DO-178C guideline the RTCA has also issued separate guideline that specifically cover the use of formal methods DO-333 [fAR11]. Additionally, this update considers other techniques for achieving certification, as the robust partitioning techniques Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations (DO-297) in [fAR05].

	Failure Condition	Description
A	Catastrophic	Failure may cause multiple fatalities, usually with loss of the airplane (Extremely improbable) .
B	Hazardous	Failure has a large negative impact on safety or performance, causing serious or fatal injuries (Extremely remote).
C	Major	Failure significantly reduces the safety margin or significantly increases crew workload (Remote).
D	Minor	Failure slightly reduces the safety margin or slightly increases crew workload (Probable).
E	No effect	Failure has no impact on safety, aircraft operation, or crew workload (Not applicable).

Failure Condition	**Description**

Table 2.1: DO-178B/ED-12B Criticality Levels Overview

2.2.2 IEC-61508 Standards

The IEC has issued a standard Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (IEC-61508) [IEC10]. This is an international standard focusing on the functional security of electrical/electronic/programmable devices. This is a generic international standard (umbrella standard) which is designed to be applied to several industrial sectors, some instantiations of the IEC-61508 are: (1) Automotive (ISO 26262), (2) Railway (EN/CELENEC 50128) and (3) Process Industry (IEC 61511). These standard was originally issued in 1989, but have subsequently been updated and reissued. Is concerned with the functional safety of programmable electronic systems in general.

The [IEC10] standard is organized in several parts covering the Functional Safety of Safety-Related Systems, amongst them: (1) Generic requirements; (2) Requirements for electronic/electronic/programmable systems; and (3) Software requirements. The formal methods CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM and Z are specifically mentioned in [IEC10] as possible techniques to be applied in the development of safety-critical systems in an extensive section [IEC10, B.30, pp. B-14 B-18].

SIL	Failure prob. per hour (systems active>once per year)	Failure prob. per demand (systems active<once per year)
SIL1	$10^{-6} - 10^{-5}$	$10^{-1} - 10^{-2}$
SIL2	$10^{-7} - 10^{-6}$	$10^{-2} - 10^{-3}$
SIL3	$10^{-8} - 10^{-7}$	$10^{-3} - 10^{-4}$
SIL4	$10^{-9} - 10^{-8}$	$10^{-4} - 10^{-5}$

Table 2.2: IEC 61508 SIL.

2.2.3 ECSS Standards

The European Space Agency (ESA) has issued guidelines for software engineering standards: ECSS-E-40 [ECS09a] is the ECSS standard for software engineering. ECSS-Q-80 [ECS09b] is the ECSS standard for software Product Assurance. They are both based on ISO/IEC 12207.

These suggests that formal methods such as Z or VDM should be considered for the specification of safety-critical systems in the Software Requirement Document . A short section on formal proof suggests that proof of the correctness of the software should be attempted if practicable. Because of the possibility of human error, proofs should be checked independently. Methods such as formal proof should always be tried before testing is undertaken. This the use of formal methods is strongly recommended, but not mandated by the document.

2.2.4 Common Criteria Framework

The CC [Cri12] is a framework that allows the rigorous specification of security and assurance requirements, that are implemented by vendors in their products, and evaluated by testing laboratories to determine if the products meet their requirement claims. The CC defines seven Evaluation Assurance Level (EAL), which range from EAL1 (lowest) to EAL7 (highest) assurance level (see Table 2.4). The Table 2.4 presents the formal methods requirements at each assurance level, where formal methods are mandated to achieve the certification of the highest levels: EAL6 and EAL7.

EAL	**Description**
EAL1	Functionally tested, security threats not serious
EAL2	Structurally tested, low to moderate assurance
EAL3	Methodically tested and checked, maximum assurance without infringing sound
EAL4	Methodically designed, tested and reviewed, maximum assurance compatible with good commercial practise
EAL5	Semi-formally designed and tested, maximum assurance with moderate security engineering
EAL6	Semi-formally verified design and tested, protect high value assets against significant risk
EAL7	Formally verified design and tested, extremely high risk situations and/or high assets values

Table 2.3: Common Criteria Evaluation Assurance Level

CC	**Requirement**	**Specification**	**Design**	**Implementation**
EAL1	Informal	Informal	Informal	Informal
EAL2	Informal	Informal	Informal	Informal
EAL3	Informal	Informal	Informal	Informal
EAL4	Informal	Informal	Informal	Informal
EAL5	**Formal**	*Semi-formal*	*Semi-formal*	Informal
EAL6	**Formal**	*Semi-formal*	*Semi-formal*	Informal
EAL7	**Formal**	**Formal**	**Formal**	Informal

Table 2.4: Common Criteria Assurance levels.

2.3 Summary

This section analyses the information of the survey to conclude the position of the standards regarding formal methods, which is summarized in table 2.5.

Standard	**Formal Methods**	**Organization**	**Sector**	**Country**	**Year**
DO-178C	Recommended	RTCA	Avionics	USA	2012
IEC-61508	Recommended	IEC	Industrial	Europe	2010
ECSS-E-40	Recommended	ESA	Space	Europe	2009
CCV3.1R4	Recommended	ISO/IEC	IT	International	2012

Table 2.5: Safety-critical standards summary.

- **DO-178C**: Formal methods are explicitly considered as a complement to dynamic testing, that can be presented as evidences for certification. Additionally, formal methods (DO-333) and robust partitioning techniques (DO-297) are specifically addressed on separate guidelines.
- **IEC 61508**: Formal methods are “highly recommended” to achieve SIL4, while are “recommended” to achieve the lower SIL2 and SIL3 safety levels [IEC10, part 3]
  - **EN 50128 (railway)**: Formal methods are “highly recommended” to achieve SIL3 and SIL4, while they’re “recommended” to achieve SIL1 and SIL2.
  - **ISO 26262 (automotive)**: Formal methods are “recommended” to achieve ASILB, ASILC and ASILD.
- **ECSS-E-40**: Formal methods are “recommended” as supplementary or replacement of the existing practices defined in [ECS09a].
- **CCV3.1R4**: Formal methods are “mandated” for EAL7 and semi-formal methods are “mandated” for Evaluation Assurance Level 6 (EAL).

The main conclusion to be drawn is that the surveyed standards recommend the use of formal methods to achieve the highest safety levels, and for highly critical applications formal methods are mandatory.

This page is intentionally left blank.

Chapter 3
XtratuM foundations: A formalisation approach.

Partitioned software architectures represent the future of secure systems. They have evolved to fulfill security and avionics requirements where predictability is extremely important.

The idea behind a partitioned system is the virtualization. A virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. A Hypervisor (also known as virtual machine monitor) is a layer of software (or a combination of software/hardware) that enables to run several independent execution environments or partitions in a single computer. Hypervisor is a new and promising technology which can be designed to meet safety and security properties. In order to achieve these properties, the hypervisor has to follow strict design criteria and be modeled using a formal approach.

In this chapter first, the XtratuM hypervisor is presented, next, we present our contribution: an attempt to extend the trusted environment from the hardware level to the hypervisor level on the basis of preserving the temporal and spatial isolation properties.

3.1 Introduction

The availability of new processors for embedded applications has raised new possibilities for these applications. Now, the embedded applications have more functionalities and, as consequence, more complexity. There exist a growing interest in enabling multiple applications to share a single processor and memory. To facilitate such a model the execution time and memory space of each application must be protected from other applications in the system.

Partitioned software architectures represent the future of secure systems. They have evolved to fulfill security and avionics requirements where predictability is extremely important. The separation kernel proposed in [Rus81a] established a combination of hardware and software to allow multiple functions to be performed on a common set of physical resources without interference. The MILS (Multiple Independent Levels of Security and Safety) initiative is a joint research effort between academia, industry, and government to develop and implement a high-assurance, real-time architecture for embedded systems. The technical foundation adopted for the so-called MILS architecture is a separation kernel. Also, the ARINC-653 [Ai96] standard uses these principles to define a baseline operating environment for application software used within Integrated Modular Avionics (IMA), based on a partitioned architecture.

Virtual machine technology can be considered the most secure and efficient way to build partitioned systems. A virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. **Hypervisor** (also known as virtual machine monitor

VMM [Gol74]) is a small layer of software (or a combination of software/hardware) that enables to run several independent execution environments or partitions in a single computer. The key difference between hypervisor technology and other kind of virtualisation (such as java virtual machine or software emulation) is the performance. In bare-machine hypervisors the overhead can be very low maintaining the throughput of the virtual machines very close to the native hardware. Hypervisor is a new and promising technology, but has to be adapted and customized to the requirements of the target application.

The low overhead and the reduced size of the hypervisor can be considered as an appropriated solution to achieve secure systems if it is designed following strict design criteria to meet security requirements. Its correctness can be sufficient to ensure the security of the system as a whole or, at least, the security of a set of trusted partitions. In a partitioned system, the partitions can accommodate different kinds of applications: real-time, trusted, non trusted, etc. As consequence, the partition's operating system can be customised to provide the specific services to its applications.

In this chapter we present a solution for partitioned based on a bare-metal hypervisor called XtratuM. The XtratuM hypervisor has been designed specifically for critical real-time systems following a set of requirements for secure space applications and a set of services to build applications based on the ARINC-653 standard

In the next section, we present a review of the virtualisation techniques with special emphasis in the real time characteristics. Section 3.2 presents the main design criteria. Also, we analyse the processor dependencies and the virtualised services to the partitions. Section 3.2.1 describes the hypervisor architecture and the services provided to the partitions. It also provides a model for interrupt and fault management to the partitions. Finally some conclusions are enumerated.

3.2 XtratuM Overview

XtratuM [MRC05] has been designed to achieve temporal and spatial requirements of safety critical systems. It is being used as TSP-based solution for payload on-board software, highly generic and reusable, in space applications [AMGC09] by CNES using the LEON2 [Res05] processor. TSP (Time and Space Partitioning) based architecture has been identified as the best solution to ease and secure reuse, enabling a strong decoupling of the generic features to be developed, validated and maintained in mission specific data processing [AM08].

LEON2 processor is a 32-bit processor core based on the SPARC V8 architecture suitable for system-on-a-chip (SOC) designs, which can be synthesized in an FPGA. It is used by the European Space Agency and has successfully been used in various commercial and research endeavors.

3.2.1 XtratuM Architecture

Figure 3.1 depicts the complete system architecture. The main components of this architecture are:

XtratuM is in charge of virtualisation services to partitions. It is executed in supervisor processor mode and virtualises the cpu, memory, interrupts and some specific peripherals. The internal XtratuM architecture includes: memory management, scheduling (fixed cyclic scheduling), interrupt management, clock and timers management, partition communication management

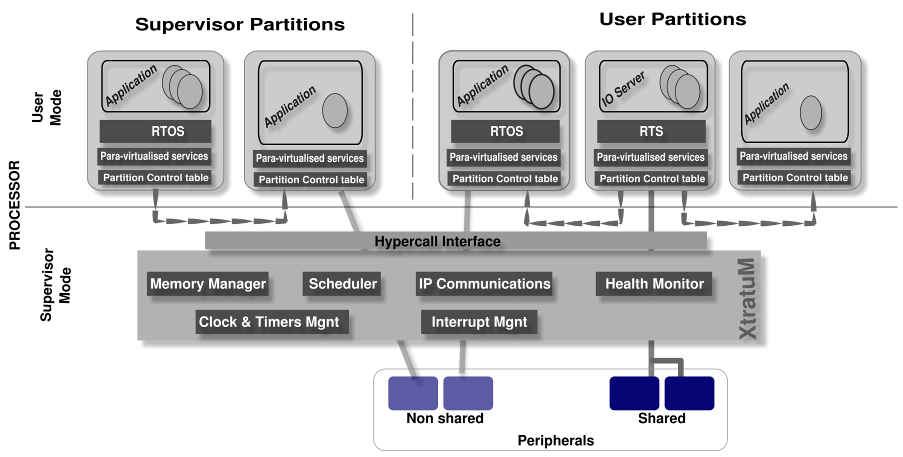

Figure 3.1: XtratuM architecture.

(ARINC-653 communication model), health monitoring and tracing facilities. Three layers can be identified:

- Hardware-dependent layer: It implements the set of drivers required to manage the strictly necessary hardware: processor, interrupts, hardware clocks, hardware timers, paging, etc. This layer is isolated from the rest through the Hardware Abstraction Layer (HAL). Thus, the HAL hides the complexity of the underlying hardware by offering a high-level abstraction of it (for instance, a ktimer is the common abstraction to all the hardware timers).
- Internal-service layer: Those services are not available to the partitions. This layer includes a minimal C library which provides the strictly required set of standard C functions (e.g. strcpy, memcpy, sprintf) and a bundle of data structures. The system boot is also part of the internal services.
- Virtualization-service layer: It provides the services required to support the para-virtualisation services, which are provided via the hypercall mechanism to partitions. Some of these services are also used from other XtratuM modules.

3.3 Trustability enforcement

In this section we provide a set of principles that permit to achieve a high secure hypervisor.

- Strong spatial isolation: Hypervisor has to be executed in privilege processor mode whereas partitions are executed in user processor mode. Partitions are allocated in independent physical memory addresses. Partitions can not access to other partition memory addresses.
- Strong temporal isolation: Hypervisor enforces the temporal isolation by using the appropriated scheduling policies to execute partitions. The policies can be cyclic scheduling or fixed priority scheduling based on the server concept.

- Partition management: Partitions are executed in user mode, thus guaranteeing that they have not access to processor control registers. Any partition access to a processor device is detected and handled by the hypervisor.
- Supervisor partitions: Some partitions can use *special* services provided by the hypervisor. These services include: partition management, access to system logs, etc.
- Robust communication mechanisms: Partitions can communicate with other partitions using the specific services provided by the hypervisor. The basic mechanism provided to the partitions is the port based communication. The hypervisor implements the link (channel) between partitions. Two types of ports are provided: sampling a queuing as defined in [Ai96].
- Interrupt Model: The hypervisor provides an interrupt model to the partitions. Partitions can not interact with native traps. All the interrupts are detected handled by the hypervisor and propagated to the partitions according to the interrupt model.
- Fault management model: Faults are detected and handled by the hypervisor. The detection of a fault can be the occurrence of a system trap or the occurrence of an event generated by the hypervisor code. The hypervisor code includes a set of assertions to verify the properties of the system. All hypervisor services have a set of pre- and post- conditions that assert the system properties. A Health Monitor module in the hypervisor implements the fault management model. Actions associated to the Health Monitor depend on the partition or hypervisor fault generator.
- Non-preemptible: In order to reduce the design complexity and increase the reliability of the implementation, the hypervisor is designed as a monolithic kernel to be non-preemptible.
- Resource allocation: Fine grain hardware resource allocation is specified in the system configuration file. This configuration permits to assign system resources (memory, I/O registers, devices, memory, etc.) to the partitions.
- Minimal entry points: the hypervisor has to clearly identify the execution paths and the entry points.
- Small: The level of difficulty and complexity of validation and formal verification increases in an exponential manner with the number of analyzed lines of code. The hypervisor code shall provide the minimum services in order to be as minimal as possible. XtratuM has around 5 MLOCs.
- Deterministic hypercalls: All services (hypercalls) shall be deterministic and fast.

The hardware protection mechanisms imposes strong limitations in the hypervisor design:

- Partition allocation restrictions (1) Each partition is allocated in one non overlapping memory region. (2) A partition has to be multiple of 64 Kbytes.
- On-chip peripherals are handled exclusively by the hypervisor and virtualised to the partitions.
- IO ports are handled exclusively by the hypervisor due to the protection mechanisms (1 bit). Partitions have to use specific services (hypercalls) to access to IO ports.
- Window registers are handled by the hypervisor providing a flat stack model to the partitions.

3.3.1 Interrupt Model

Different manufacturers use terms like exceptions, faults, aborts, traps, and interrupts to describe the processor mechanism to receive a signal indicating the need for attention. Also, different authors adopt different terms to their own use. In order to define the interrupt model, we provide the definition of the terms used in this work.

A *trap* is the mechanism provided by the processor to implement the asynchronous transfer of control. When a trap occurs, the processor switches to supervisor mode and unconditionally jumps into a predefined handler.

A *software trap* is raised by a processor instruction and it is commonly used to implement the system call mechanism in the operating systems.

An *exception* is an automatically generated interrupt that occurs in response to some exceptional condition violation. It is raised by the processor to inform about a condition that prevents the continuation of the normal execution sequence. There are basically two kind of exceptions: those caused by the normal operation of the processor (like register window under/overflow in Sparc architecture), and those caused by an abnormal situation (like an memory error).

A *hardware interrupt* is trap raised due to an external hardware event (external to the CPU). These interrupts generally have nothing at all to do with the instructions currently executing and informs the CPU that a device needs some attention.

In a partitioned system, the hypervisor handles these interrupts (*native interrupts*) and generates the appropriated *virtual interrupts* to the partitions.

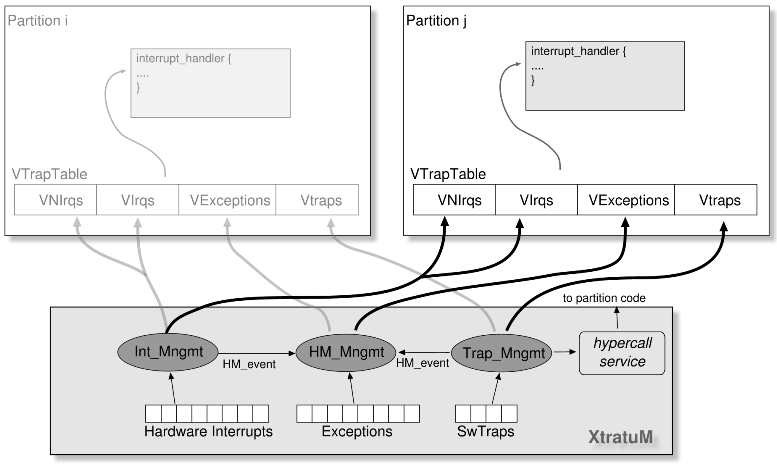

Figure 3.2: Interrupt model.

Figure 3.2 depicts the interrupt model. A partition have to deal with the following *virtual traps*:

- *virtual traps* are generated by the hypervisor to the partitions as consequence of a *native trap* occurrence.
- *virtual exceptions* are the exceptions propagated by the hypervisor to the partitions as consequence of a *native exception* occurrence. Not all the *native exceptions* are propagated

to the partition. For instance, a memory access error that is generated as consequence of a space isolation violation is handled by the hypervisor which can perform a halt partition action or can generate another different virtual exception (like memory isolation fault). On the other hand, a numeric error is propagated directly to the partition. *Virtual exceptions* are a superset of the *native exceptions* which include additional exceptions generated by the hypervisor (virtual processor). Some of them are: memory isolation error, IO isolation error and temporal isolation error.

- *virtual hardware interrupts* are directly generated by the real or the virtual hardware. The real hardware corresponds to external devices (dedicated devices technique) or peripherals and the virtual hardware includes the different virtual devices associated to the virtualisation. Some of these virtual devices are:
  - Virtual hardware and global and local clocks and timers
  - New message arrival. The communication mechanism (channel) implemented by XtratuM is seen as a hardware device.
  - Partition slot execution. In a partitioned system the partition is aware of the partition scheduling, this interrupt informs to the partition that a new slot has been scheduled.

Only *virtual hardware interrupts* can be enabled or disabled by partitions.

We will use in the next paragraphs the terms *trap* and *vtrap* as the main mechanism to deal with any kind of interrupt at native or virtual level. Four strategies have been used to prevent partitions to jeopardise temporal isolation:

- Partitions have no access to the trap table. Thus, partitions are unable to install their own trap handlers. All traps are directly handled by XtratuM and, when required, propagated to partitions which define its own *virtual trap table.*
- Partitions cannot interact with native traps. Partitions are executed in user mode, thus guaranteeing that they have not access to control registers.
- A partition can not enable/unmask those *virtual hardware interrupts* not allocated to the partition.
- When a partition is scheduled, the *hardware interrupts* associated to other partitions are disabled. When a partition context switch occurs, the hypervisor detects the hardware interrupts pending for the next partition to be executed and raise them depending on the partition interrupt mask.

3.3.2 Fault Management Model

The Health Monitor (HM) is the part of XtratuM that detects and reacts to anomalous events or states. The purpose of the HM is to discover the errors at an early stage and try to solve or confine the faulting subsystem in order to avoid or reduce the possible consequences.

HM is executed as result of a HM_event occurrence. Next scenarios can raise a HM_event:

- An exception has been raised by the CPU. The exception handler generates the associated HM_event.

- A native interrupt has been received and the temporal or spatial properties are not validated.
- A trap has been received and the temporal or spatial properties are not validated.
- A partition detects an abnormal internal situation and raises a HM_event. For instance, the operating system inside of a partition detects that the application is corrupted.
- When the partition request a hypervisor service (hypercall), the spatial or temporal properties are verified as pre- and post-conditions. If these validations fail, a HM_event is generated.

Previous cases cover all entry points to the hypervisor. As result of enforcing the isolation of the partitions, XtratuM performs a check of the temporal and spatial properties each time that it is invoked.

The HM_event occurrence is the manifestation of an error. XtratuM reacts to the error providing a simple set of predefined actions to be done when it is detected.

XtratuM HM subsystem is composed by four logical components:

- HM configuration: to bind the occurrence of each HM event with the appropriate HM action. This bind is specified in the configuration file.
- HM event detection: to detect abnormal states, using logical assertions in the XtratuM code.
- HM actions: a set of predefined actions to recover the fault or confine the error.
- HM notification: to report the occurrence of the HM events.

Once a HM event is raised, XtratuM performs an action that is specified in the configuration file.

XtratuM provides a mechanism to store and retrieve the traces generated by partitions and XtratuM itself. Traces can be used for debugging, during the development phase of the application, but also to log relevant events or states during the production phase. In order to enforce resource isolation, each partition (as well as XtratuM) has a dedicated trace log stream to store its own trace messages. Trace streams are stored in buffers (RAM or FLASH). Only supervisor partitions can read from a trace stream.

3.3.3 System specification

Deploying a partitioned system presents many challenges related to the system specification, configuration, resource allocation and validation. This configuration process involves two different type of roles: the system integrator and the partition developers. The integrator, jointly with the partition developers, have to define the resources allocated to each partition. Figure 3.3 depicts the deployment process.

In XtratuM, the system specification is detailed in a XML file which contains the five main elements:.

- XMHypervisor: Specifies the board resources and the hypervisor health monitoring table. It includes:

  - PhysicalMemoryAreas: List of memory regions allocated to XtratuM.
  - HwDescription: Defines the main characteristics of the processor and the target board (memory layout).
  - Processor: Specifies the processor frequency and the scheduling plan
  - ResidentSoftware: This is an optional element which describes (provides information to XtratuM) about the resident software.

- PartitionTable: This is a container element which holds all the partition elements. A partition elements specifies:

  - PartitionId: Defines the partition identifier.
  - MemoryAreas: List of memory allocated to the partition.
  - HwResources: Contains the list of interrupts and IO ports of the partition.
  - PortTable: Contains the list of communication ports (queuing and sampling ports) of the partition.

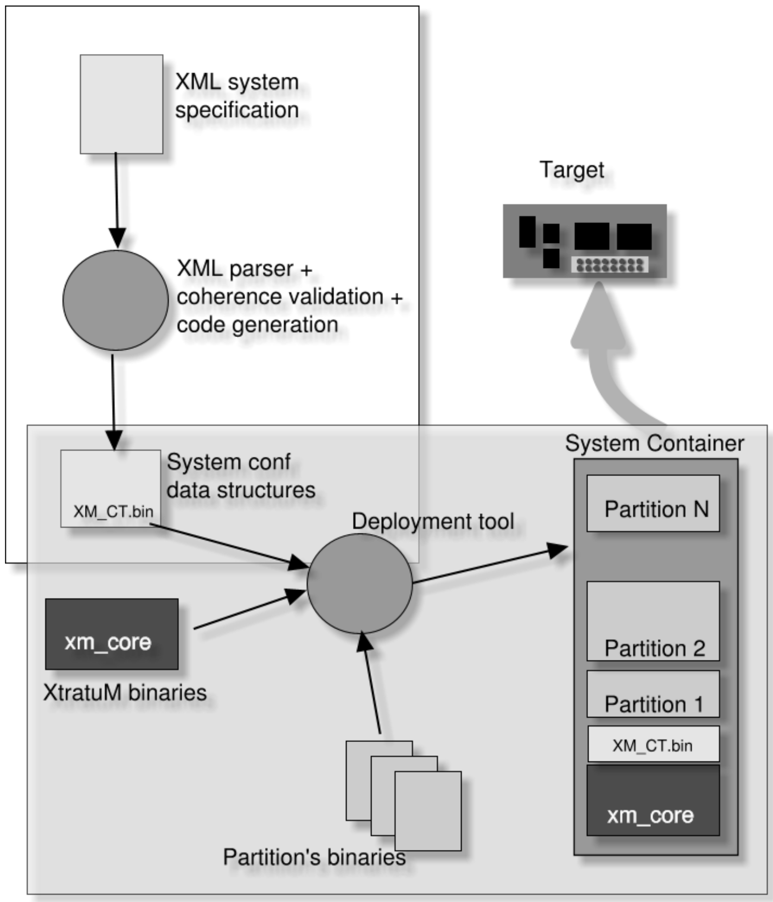

Figure 3.3: XtratuM deployment.

  - Trace: Configuration of the trace facility of the partition.
  - Device: List of the devices used by this partition.
  - TemporalRequirements: Contains the period and the capacity of the partition.
  - Flags: Boolean flags that permit specify if a partition will be booted by XtratuM, is a supervisor partition, uses the floating point unit, etc.

- Channels: A list of channels which define the port connections. For each channel, the following information is specified: channel identifier, type, input and output ports, maximum message size, maximum number of messages (queuing channels).
- Devices: Contain the configuration of virtualised devices.

The XML file is parsed and validated against the main system properties of a partitioned system. The result of this validation process is a set of automatically generated data structures (XM_CT) that will be compiled in the deployment phase, jointly with the binaries of the partitions and the hypervisor, to generate the system container to be loaded in the target.

3.4 Hypervisor model

In order to model the hypervisor, some aspects have to be initially considered:

- It is non preemptible. When any of the entry-point is invoked, it is executed with disabled interrupts returning the control to a partition.
- It has three entry points and one return point to partition.
- The system configuration is specified using a UML model and compiled generating a set of static data structures (XM_CT) used by the execution environment.
- Only the hypervisor can access to the processor registers and virtualised services.
- Internal code of partitions is not relevant from the hypervisor point of view

Additionally, we assume that the underlaying hardware is trusted. It means that the internal processor registers will work properly if they are used in the correct way. For the temporal and spatial isolation purposes, we assume:

- The access to the processor registers is only allowed when the processor is in privileged mode. The processor mode can set/unset by accessing the control processor status (PMS).
- The processor memory protection registers (WPR) will raise an exception when a instruction tries to write in the protected area.
- A specific timer is exclusively used by XtratuM to control the slot duration.
- The IOP bit in the control processor status permits to enable/disable the access to the IO ports. This bit is unset when any partition is in execution.
- The interrupt vector is handled exclusively by XtratuM. Its access/modification can be done only when the processor is in privileged mode.

Based on the hardware mechanisms (PRegs) and its used by XtratuM, we try to extend the trusted environment to the hypervisor level.

A partitioned system can be described in terms of a tuple ($\Sigma = (H, HM, \Delta, \Omega, \Pi)$), where Θ is the hypervisor, Π a set of partitions, Δ is a set of devices and Ω is the set of channels to inter-partition communications.

The hypervisor H is defined by $H = (\Lambda, \rho)$. Λ is, in general, a set of memory area regions which are defined in terms of *initial address* and *size*. These memory areas specify the memory allocations of the hypervisor. In this work, we assume that Λ is restricted to one memory region due to the absence of the MMU unit and the constraints of the memory protection mechanisms.

ρ is the cyclic scheduling plan to be executed by the hypervisor.

HM is the set of health monitor events. Each HM_{event} has associated a default action depending on the execution context.

Δ is a set of devices (non virtualised) available in the system. Each device is defined as a IO-memory area region.

Ω is the set of channels for partition communication. Each channel (γ_i) is defined as the channel type (sampling or queuing), an input and output port, the number of messages (queuing channels) and the maximum message size.

Π is a set of partitions. Each partition π_i is described as: $\pi_i = (\Lambda, HMA, Dev, Ports)$, where Λ is the memory region where the partition is allocated,

HMA is the set of the specific HM actions to be executed when a *HM_event* occurs for this partition, *Dev* is a sub-set of Δ (set of devices) used by this partition and *Ports* is the set of visible ports for this partition.

XtratuM can be defined as a finite state machine [AHU74] given by $(\Omega, \alpha, S, S_0, \theta, F)$ where:

- Ω is the system configuration. It is automatically generated from the system specification (XM_CT).
- α is the input alphabet described by the set of events that accept the hypervisor (hardware interrupts, exceptions and traps).
- S is a finite, non-empty set of states. These finite states are result of the scheduling plan. Each state corresponds to the execution of a partition in the scheduling plan and has associated a relative initial and final time with respect to the origin of the MAF. From the temporal and spatial isolation properties only three events are significant: *next_slot*, an exception that can perform a system halt (HM action), and a trap (system halt hypercall).
- S_0 is the initial state which correspond to the hypervisor state after booting and partition loading.
- θ is the state-transition function given by $\delta : S \times \Sigma \rightarrow S$. A function $f^{\Omega}(S_i)$ extracts the set of hardware parameters associated to the S_i state.
- F is the final states which correspond to a system halt.

Figure 3.4 shows the set of states of the hypervisor generated from the static scheduling plan defined in the specification. Each state S_i models the status of the hypervisor when a partition P_i

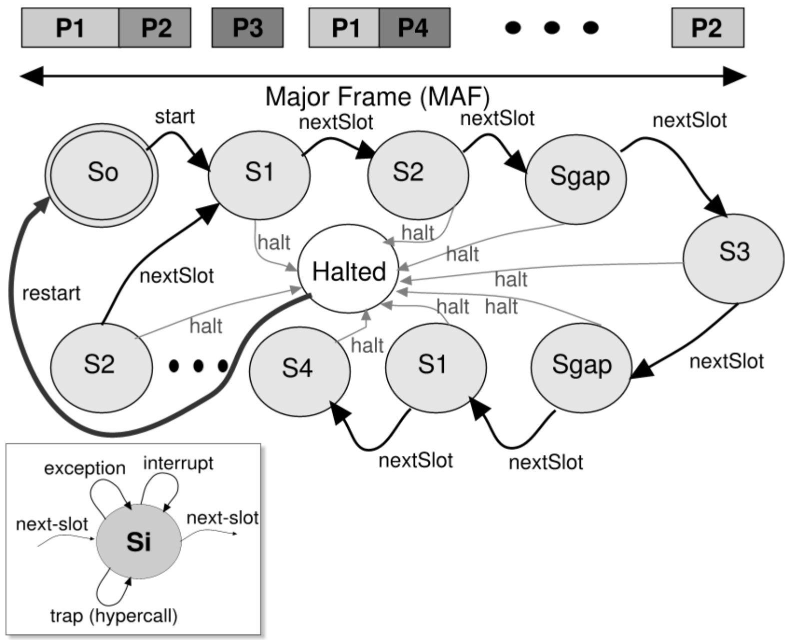

Figure 3.4: Finite set of states.

is under execution. The transition from one state to the next one, is consequence of the *next_slot* occurrence which is the slot duration defined in the scheduling plan. This event is the hardware interrupt associated to the slot duration timer. After the arrival of this event (interrupt), the hypervisor, stores the context of the previous partition, selects the next partition to be executed, extracts from the XM_CT table the hardware context to guarantee the spatial and temporal isolation, and then the control is transfered to this partition if the partition status is in ready state.

During the execution of a partition, the hypervisor can be invoked as consequence of an interrupt, exception or hypercall (trap). However, the hypervisor invocation have not effect from the temporal and spatial isolation point view because of the hypervisor state remains in the same state until the arrival of the next_slot event. Anyway, a pre- and post condition validation of the temporal and spatial properties is performed each time the hypervisor is invoked. A violation of these properties will raise the appropriated HM_event.

Figure 3.5 depicts a view of the hypervisor model and the state variable used to extend the trustability.

3.4.1 Hypervisor state variables

The hypervisor state is defined by a set of variables that specify:

- The partition identifier ($p \in \Pi$)
- The absolute time of current MAF origin (AT_{MAF})
- The current clock value ($current\_clock$).

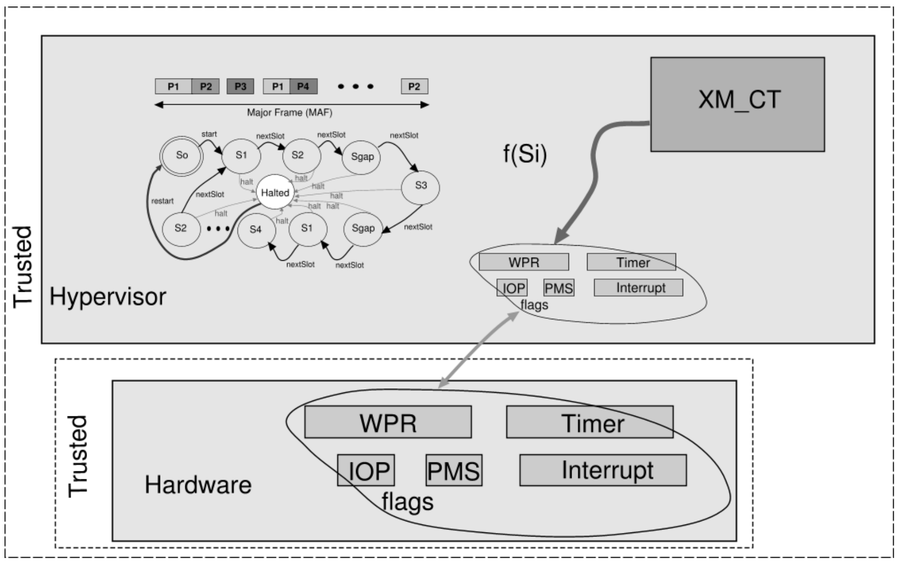

Figure 3.5: State variables.

- the processor registers: memory protection registers (WPR), interrupt vector (IV), mode processor status (MPS), IO protection bit (IOP).

3.4.2 General properties

General properties are applicable at any system execution time and refers to basic properties that have to guarantee.

Property 1 *Partitions are executed in user processor mode whereas hypervisor is executed in privilege mode.*

3.4.3 Spatial isolation properties

The basic concern of spatial isolation is to detect and avoid the possibility that a partition can access to another partition for reading or writing. Hardware provides some basic mechanisms to guard against violations of spatial isolation.

Spatial isolation properties are conditions that permit to guarantee that the hardware mechanisms are set to the appropriated values when a partition is under execution. These conditions have to compare the memory area region of the current partition against the processor memory protection registers in the LEON2 architecture.

The spatial isolation property states that data processing in any partition i can not access to any memory address outside of the its address memory region. This is granted if the hardware mechanisms are used according the principles announced previously.

Property 2 *Suppose the hypervisor is in state S_i and the next state will be S_j as consequence of the event $next\_slot \in \alpha$ occurrence. In this situation, at the entry of the hypervisor: $PRegs = f^{\Omega}(S_i)$, and at the output of the hypervisor $PRegs \leftarrow f^{\Omega}(S_j)$).*

Where $f^{\Omega}(S_i)$ the function that extracts the $PRegs$ values from a XM_CT and $PRegs$ are the hardware registers.

For $\forall e \neq next\_slot \in \alpha$ the state is not changed.

Also, the spatial isolation refers to the IO memory access. In the LEON2 processor, the IO memory access protection is a global mechanism. To guarantee the isolation, IO memory access is forbidden to partition and provided through specific hypercalls that control the IOmemory addresses to each partition. The hypervisor has to validate that the hardware mechanism is enabled.

Property 3 *Independently of the initial and final states, when an event $e \in \alpha$ occurs, the IO memory protection ($IOP \in PRegs$) bit is set to 1 at the entry of the hypervisor $IOP = 1$ and again $IOP = 1$ when the control is transferred to the partition. During the execution of the hypervisor $IOP = 0$.*

3.4.4 Temporal isolation properties

Temporal isolation refers to the system ability to execute several executable partitions guaranteeing:

- the timing constraints of the partition tasks
- the execution of each partition does not depend on the temporal behaviour of other partitions.

The temporal isolation enforcement is achieved designing a schedulable plan and guaranteeing that it is executed as specified. The hypervisor scheduling is responsible of the correct execution of the plan. XtratuM implements a static (cyclic) scheduling following the ARINC 653 specification [Ai96] which defines a cyclic scheduling for the global scheduler and a preemptive fixed priority policy for the local scheduler (partition level).

The concern of temporal isolation is to guarantee that a partition is executed only in the intervals specified in the scheduling plan. Moreover, interrupts allocated to other partitions shall not impact on the partition execution.

If the execution plan is guaranteed, there is not possibility for any partition to monopolise the CPU or crash the system. Other scenarios that can cause a partition to fail to relinquish the CPU on time include simple schedule overruns are detected and handled properly.

Slot duration is controlled by a processor register (timer) that is used exclusively by the hypervisor and can not be influenced by the partitions timers which are attached to a second timer which is virtualised to the partitions.

The temporal isolation properties can be defined as:

Property 4 *At any instant of the state S_i, the clock value is in the interval specified by the slot interval. $current\_clock \in f^{\Omega}_{clk}(S_i)$*

where $f^{\Omega}_{clk}(S_i)$ is the function that return the interval of the slot associated to S_i).

Property 5 *In a S_i state executing a partition i, this state will not accept hardware interrupts from devices that are not managed by partition i. At the entry of the hypervisor the interrupt vector has to be bitmap associated to this partition. At the output of the hypervisor, the mask has to be set to the value of the next or same partition depending on final state. At the input: $IV = \pi^{iv}_i$. At the output: $IV \leftarrow \pi^{iv}_i IV = \pi^{iv}_i$.*

3.4.5 Hypervisor state management

The hypervisor has the following behavior.

State transition: $S_i \to S_j$

Entry action: Performed when entering the state. Raises a *HM_event* if some pre-conditions are not validated.

> **Pre-conditions** General properties; Spatial properties for S_i; Temporal properties for S_i;

Service action Actions executed if the pre-conditions are validated.

Exit action: Performed when exiting the state. Raises a *HM_event* if some post-conditions are not validated.

> **Post-conditions** General properties; Spatial properties for S_j; Temporal properties for S_j;

Service actions perform the appropriated actions:

- Hardware interrupts:
  - Generated by slot timer: perform the partition context switch.
  - Generated by virtualised devices: handle and propagate if enabled and unmasked to all partitions.
  - Generated by dedicated devices: propagate the interrupt to the partition if enabled and unmasked.
- Exceptions: raises the appropriated *HM_event*.
- Trap: Validate the hypercall parameters and execute the hypercall service. If the hypercall parameters are not validated an error is returned to the partition.

3.4.6 Hypervisor pre- and post-conditions

Each time the hypervisor is invoked a pre- and post-conditions are evaluated. These conditions validate the correct status of the hardware mechanism (temporal and spatial isolation properties) If any of these conditions are not validates a HM_event is raised.

3.5 Conclusion

Complexity of embedded systems within satellites is growing dramatically. Payload software in that context have evolved during the past 10 years from simple data processing, mainly formatting and transferring to ground, into complex data processing and autonomous treatments. In this context, TSP (Time and Space Partitioning) based architecture has been identified as the best solution to ease and secure reuse, enabling a strong decoupling of the generic features to be developed, validated and maintained, XtratuM is used as enabling technology for TSP developments.

In this chapter we have presented a hypervisor specifically designed for safety critical applications. XtratuM has been designed following strict criteria to guarantee the temporal and spatial isolation properties as defined in the ARINC 653 standard and the MILS approach. XtratuM defines a virtual machine very close to the native where the main resources are virtualised. The executable entities (partitions) are executed on top of a virtual machine. The virtual machine defines an interrupt model to the partitions which is a superset of the system interrupts. The hypervisor defines its own virtual interrupts which are delivered to the partitions. Also, there is a Fault Management model which is directly related to the health monitor included in the internal architecture. This health monitor is in charge of the fault detection and error isolation through a set of action that are directly related to the partition under execution.

Finally, it has been presented an extension of the trusted environment from the hardware platform to the hypervisor limits. The presented model is based in the hardware mechanisms provided by the specific processor (LEON2) and an exclusive use of these mechanisms by the hypervisor. This model is based on finite state machine as formalism.

The hypervisor solutions are efficient solutions for partitioned systems. For space reasons, we have not included performance evaluation of XtratuM. In [MRC+09] can be found an evaluation of this proposal. In this evaluation the overhead measured is lower then 1% for slot duration higher than 1 millisecond.

This page is intentionally left blank.

Chapter 4
Formal Validation of XtratuM Components

This work presents the experiences on the application of formal methods for the validation and verification of the selected components of the XtratuM hypervisor. The goal is to use deductive verification techniques to verify the functional correctness of the XtratuM XM2 hypervisor. XtratuM is a "Type 1" hypervisor specially designed for real-time embedded systems. In order to achieve a highly dependable and secure hypervisor, the use of formal methods is increasingly being mandated by the highest assurance levels of regulatory standards.

The presented approach begins with (a) First, The analysis of the XtratuM hypervisor code base to identify the core components and their interfaces. (b) Next, the components are validated using deductive verification techniques using the Frama-C software analysis framework. Shorty, Frama-C provides support for deductive verification of function ACSL contracts annotated in the source code. (c) Last, the results obtained by the presented method are reviewed.

4.1 Introduction

This section sets the context of this work and justifies the interest in the deductive program verification applied to the XtratuM hypervisor from the standpoint of product certification.

Secure and dependable software systems are required by regulatory authorities to achieve acceptable failure rates, dependent on the criticality of the software systems. Acceptable failure rates are accomplished by adhering during the whole software process to regulatory standards [ALRL04] such as DO-178B [fAR92] and IEC-61508 [IEC10] for safety resp. Common Criteria [Cri06] for security. Safety standards define complete validation and verification plans that cover all the phases of the software development life-cycle of the software product.

The V Life Cycle model [Pre01] defines several development phases, each of them having its validation and verification counterpart. These phases include the unitary component validation phase that consists in the validation of each of the interfaces provided by each component. This phase is typically achieved by unitary testing. However, besides unit testing, safety standards increasingly recognise and mandate the use of formal methods as evidence for validation purposes.

Formal methods are defined by Rushby et al. [Rus00] as: *"The idea behind formal methods is to construct a mathematical model of a software or system design so that calculations based on the model can be used to predict properties of the actual system, where the 'calculation' is performed by so-called 'formal deduction'"*.

Among the plethora of existing formal methods, recently the deductive program verification has become an active research area, that is providing static analysis solutions for industrial use, to cite some of the solutions: VCC [CDH+09], Frama-C [CKK+12], Polyspace [Mat09], Coverity [BBC+10]. Amid these solutions, there is Frama-C, which is precisely the framework selected for the deductive verification of the XtratuM hypervisor core.

4.1.1 XtratuM Hypervisor core

XtratuM [MRCM09] is a “Type 1” hypervisor specially designed for real-time embedded systems. XtratuM provides a framework to run several operating systems (or runtime environments) in a robust partitioned environment ensuring strong temporal and spatial isolation properties.

Figure 4.1 shows the complete system architecture. The main components of this architecture are: the XtratuM hypervisor core and the partitions executing on top of the hypervisor.

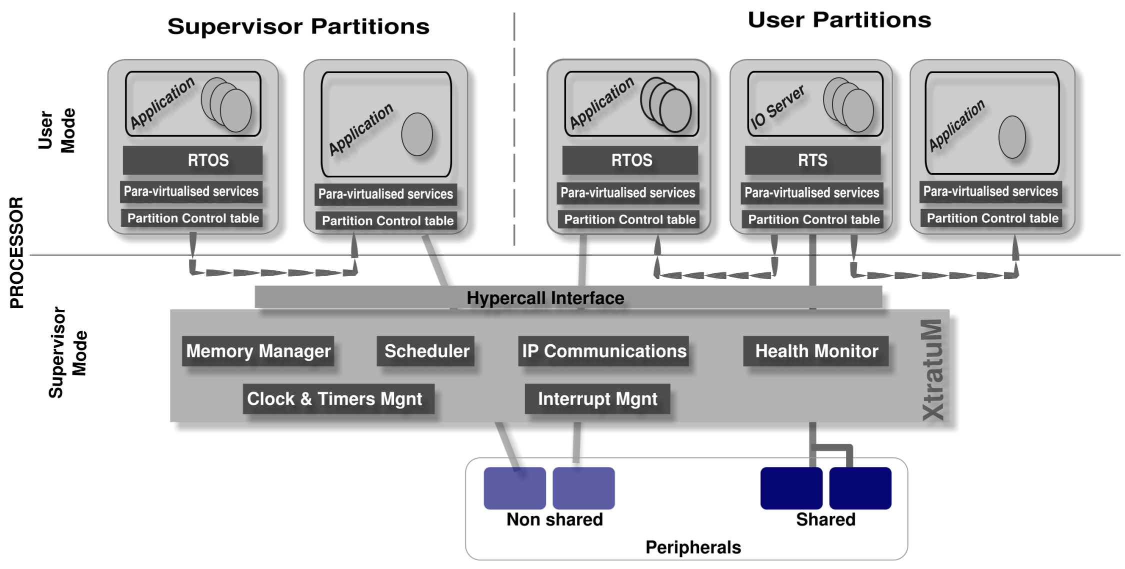

Figure 4.1: XtratuM architecture.

The XtratuM hypervisor core offering the minimum services to enforce TSP isolation, The core is comprised of 7K SLOC (Source Lines of Code) of C code, and 1K SLOC of assembly code. The core can be decomposed in components to be validated separately and then composed (Unit/Integration proofs)

4.2 Deductive Formal Methods

Formal validation of operating systems has been an on going topic for secure operating systems (Rushby) [Rus81b] in order to achieve the assurance levels required by critical systems. With the increasing adoption of virtualisation technologies, formal validation of virtualisation technologies has also been performed.

Klein et al. [KEH+09] achieved the formal verification of the seL4 microkernel, which comprises 8,700 lines of C code and 600 lines of assembler. The verification was performed using the Isabelle theorem prover to ensure functional correctness required to obtain EAL7 level of the Common Criteria.

Baumann et al. [BBBB10] performed the formal verification of the PikeOS microkernel. The verification was performed using the VCC (the Verifying C compiler) which perform deductive program verification based on function contracts to ensure the functional correctness required by the Common Criteria framework.

Heitmeyer [HALM08] presented an innovative approach which reduces the cost of the verification by partitioning the code to be verified in categories, reducing the amount of verified code to the 10%. And allowing the use automated theorem provers (PVS) based on annotating the source code with the properties to be verified.

The work performed by Souyris [SWDD09] at Airbus, presents the integration of the Frama-C formal verification techniques into the development on of the avionics software products in order to comply with DO-178B.

This short survey serves to show the increasing application of formal methods, specifically of deductive program verification techniques to ensure functional correctness.

4.3 Proposed Approach

The proposed approach is to annotate the code of the XtratuM core with function contracts specified in ACSL [BCF+11] that can be verified by automated theorem provers (ATP).

The use of the Frama-C static analyzers together with ACSL contracts annotated on the source code restricts us to the validation of the C code. For the validation of code, the code must be strictly C99 compliant, this means that the code has to be simple and free of complex code constructs, such as: function pointers, complex casts, dynamic memory allocation, etc. For the same reason, assembler code present must be verified by other means as it is not covered by Frama-C/ACSL, in this case correctness of assembler code can be proved using unitary tests. This restrictions require the XM 2 core code to be first adapted and restructured.

4.3.1 Method

The method for annotating the XM 2 source code with ACSL contracts follows a bottom-up approach, that is, start annotating the leaf functions and proceeds from the bottom to the intermediate functions calling the leaf functions.

This method begins with the preparation of a call graph of the code (DOT diagram) showing the dependences between the functions of the XM 2 code base. From here a DSM (Dependency Structure Matrix) is built that shows all functions dependences grouped by component. This allows to identify three types of functions:

1. The leaf functions that do not depend on other functions are identified and annotated with contracts.
2. The unused functions that are not called by other functions are identified and removed.
3. The intermediate functions, that is functions that call other functions are annotated once after annotating their callee.

In order to ensure that the completeness of method covers all the functions, a Frama-C plugin is written in OCaml to obtain the coverage of which functions are annotated and the results of the proof verifications.

4.3.2 Contract Specification

The contracts [Dij75] are written using a machine readable format named ACSL which stands for *"ANSI/ISO C Specification Language"* [BCF+11], in order to generate proof predicates that are processed by a Automated Theorem Prover to discharge the predicates and prove the correctness of the contracts.

An example of a contract can be: $P\,\{statement\}\,Q$, where:

- $\{statement\}$: source code to verify.
- P: pre-conditions that are required to hold before *statement*.
- Q: post-conditions that are ensured to hold after *statement*.

The conditions P and Q are annotated directly on the source code using ACSL.

4.3.3 The Frama-C Framework

Frama-C [CKK+12] is an open-source static analysis tool that targets ANSI C programs, constructed with a plugin architecture. This allows one to connect different kinds of analysis tools together such that they can cooperate and provide precise results. Currently, the following plugins are provided with Frama-C:

- RTE (Run Time Errors): this is the core module that computes for a given function, let *main()*, an abstract interpretation of the code and returns a set of alarms. Each alarm is a potential error and relates to a given location within the code and a set of local and global variables.

- Value-analysis: For each variable and each location, the value-analysis provides an over-approximation of the set of values taken by this variable at the indicated location. The domain of values computed is guaranteed to be correct, i.e. contains the real set of values taken by the variable during any execution. Over-approximations might therefore lead to false alarms.

- Jessie/WP: this module implements a deductive verification tool, based on Hoare Logic [Hoa69]. Each C function must be annotated by extra predicates (pre-, post-conditions, loop and data-types invariants, assertions, etc.) written in the ACSL [BCF+11]) and that builds up its specifications. The Jessie module proves that the code is correct w.r.t this specification. To reach this goal, some verification conditions (VC) are computed using the WP calculus and are handed over to some automatic or semi-automatic theorem provers. The code is correct iff all VC are satisfied.

4.4 Approach Evaluation

The above verification method has been already applied to a small component of the XM2 code base. Namely the `klibc` library, which is a reduced `libc` providing the *memset, memcpy, strcpy, strlen, printf, . . .* functions, consisting of 1K SLOC, extensively used by all the core services of the hypervisor.

This allows the application of static analysis and formal methods and yet be representative subset which allows to extract conclusions on the results obtained to serve as ground for more thorough following verification approaches.

In order to present the process of verifying code function, here we show an example of the steps performed to verify the *memset* function.

4.4.1 Code Refactor

As mentioned above, before attempting the deductive verification of the *klibc* component first the *klibc* has been simplified, and remove all complex code constructs not supported by the Frama-C static analyser, such as function pointers. This has further reduced the *klibc* code size to be verified to 0.5K SLOC.

This refactor has the further benefit of reducing the functionality to the minimum required, and hence reducing as well the verification efforts.

4.4.2 Contract Annotation

The functions to verify are annotated using ACSL. This includes pre-conditions (**requires** annotations) and post-conditions (**ensures** annotations) as well as code assertions such as **asserts** and **loop invariants**. Listing 4.1 depicts the source code of the memset function together with its contract annotated in ACSL.

```
/*@
    requires count >= 0;
    requires \valid_range((char*)dst, 0, count-1);
    assigns ((char*)dst)[0 .. count-1];
    ensures \forall integer k; 0 <= k < count && ( ((char*)dst)[k] == s );
@*/
void *memset(void *dst, xm_s32_t s, xm_u32_t count) {
    register xm_s8_t *a=dst;
    count++;
    /*@
        loop invariant count >= 0;
        loop invariant \forall integer k; 0 <= k < (\at(count, Pre)-count) && ((char
    *)dst)[k] == s;
    @*/
    while (--count)
        *a++ = (xm_s8_t)s;
    return dst;
}
```

Listing 4.1: memset function annotated for verification.

4.4.3 Proof Verification

Run the static analysis tools on the selected code base. In this case the WP (Weakest Precondition) plugin of the Frama-C analyser is run which generates Proof Obligations (PO) in a language suitable for the selected theorem prover from the **\ensures** annotations. Then the ATP uses the **\required** annotations to obtain the weakest pre-conditions that satisfy the post-conditions (WP) and concludes if the proof obligations can be proved assuming the pre-conditions.

4.4.4 Proof Results

Review of static analysis results obtained. The listing 4.2 shows the summary report of the automated theorem prover, which shows the attempts to prove each of the proof obligations for the `memset` function, for each proof obligation the status of the proof attempt is shown on the left, with: `Valid`, if the ATP could automatically prove the proof obligation or `Unknown` otherwise.

```
Properties for Function memset:
[  Valid  ] memset ensures \forall integer k;
            ((0 <= k) && (k < \at(count,Old))) &&
            (((char *)\at(dst,Old))[k] == \at(s,Old))
[  Valid  ] memset requires (count >= 0)
[ Partial ] memset requires \valid_range((char *)dst, 0, count-1)
[ Partial ] memset assigns ((char *)dst)[0..count-1];
[ Partial ] memset loop invariant count >= 0;
[ Partial ] memset loop invariant \forall integer k; ((0 <= k) && (k < \at(count,Pre)-count)
      ) && ((char *)dst)[k] == s;
```

Listing 4.2: XtratuM code properties verification

4.4.5 Results

The Table 4.1 and Table 4.2 present a summary of results of the *klibc* verification, including the coverage of the functions verified along with the results of the proof verification.

Contracts per file	Annotated	Missing	Percentage%
klibc/string.c	16/19	3/19	84%
klibc/stdio.c	10/17	7/17	58%
klibc/mpool.c	10/54	44/54	81%
klibc/stdlib.c	5/10	5/10	50%
klibc/sparcv8/string.c	11/12	1/12	91%

Table 4.1: Coverage of contracts.

The Table 4.1 shows in the column **Annotated** the number of functions that have been *annotated* with function contracts over the total of functions defined in the source code file, while the **Percentage** column provides the latter number as a percentage ratio.

Proofs per file	Proved	Missing	Percentage%
klibc/string.c	83/116	33/116	71%
klibc/stdio.c	89/095	6/95	93%
klibc/mpool.c	85/087	2/87	97%
klibc/stdlib.c	68/069	1/69	98%
klibc/sparcv8/string.c	64/067	3/67	95%

Table 4.2: Proof verification results.

The Table 4.2 shows in the column **Proved** the number of proof obligations have been *proved* by the ATP over the total of proof obligations generated for the source code file, while the **Percentage** column provides the latter number as a percentage ratio.

4.5 Conclusions

This work has presented the formal verification of a small component of the XtratuM hypervisor core, namely the *klibc* in order to prove its functional correctness. This method can be used to provide evidences to achieve the higher assurance levels of certification standards such as the Common Criteria [Cri06] and DO-178B [fAR92] standards.

The above listed benefits come at a expense of an additional effort during the coding phase to produce simple code that can be verified, as well as the efforts during the unit validation phase for the contracts annotation and verification.

This page is intentionally left blank.

Chapter 5
Analysing the Impact and Detection of Kernel Stack Infoleaks

The Linux kernel has become a fundamental component of mainstream computing solutions, now being used in a wide range of applications ranging from consumer electronics to cloud and server solutions. Being expected to continue its growth, especially in the mission-critical workloads.
Parallel to the Linux adoption has increased its misuse by attackers and malicious users. This has increased attention paid to kernel security through the deployment of kernel protection mechanisms. Kernel based attacks require reliability, where kernel attack reliability is achieved through the information gathering stage, where the attacker is able to gather enough information about the target to succeed. The taxonomy of kernel vulnerabilities includes information leaks (CWE-200), that are a class of vulnerabilities that permit access to the kernel memory layout and contents. Information leaks can improve the attack reliability enabling the attacker to read sensitive kernel data to bypass kernel based protections.
In this work we aim at the analysis and detection of stack based information leaks to harden the security of the kernel. First, we analyse the problem of kernel infoleaks in [Section 5.3], next, we examine the impact of infoleaks attacks on the security of the kernel in [Section 5.4]. Then, we present a technique for detecting kernel based infoleaks through static analysis [Section 5.5]. Next, we evaluate our technique by applying it to the Linux kernel [Section 5.6]. Last, we discuss the applications and limitations of our work [Section 5.6.3] and finally we drawn our concluding remarks.
Keywords
- Confidentiality, Information Security, Information Disclosure (Infoleak), Operating System, Kernel

5.1 Introduction

The Linux kernel has become a fundamental component of mainstream computing solutions, now being used in a wide range of applications ranging from consumer electronics to cloud and server solutions. And is expected to continue its growth, especially in the mission-critical workloads. Parallel to this growth the Linux kernel has become an interesting target for attackers. Recent advances in hardening userland with protection mechanisms as ASLR [Tea01], StackGuard [CPM+98] and DEP [Cor07, 5.13] have increased the difficulty of userland based attacks, moving the attacker focus to the kernel. The main difference between userland and kernel based attacks is the consequence of the attack failure that leads to system panic/halt on kernel attacks, while on userland attacks, failure is more benign as implies a process crash/restart. Therefore, the reliability of the attack is critical when targeting the kernel, kernel attack reliability is achieved through the information gathering stage where the attacker is able to gather enough information about the target to succeed in its purposes.

Among the taxonomy of kernel vulnerabilities [CMW+11] are information leaks vulnerabilities (infoleaks), identified as CWE-200 by MITRE [CWE08b], that allow to access kernel data from malicious user process. Information leaks are often underestimated as they can improve the attack efficiency allowing the attacker to access sensitive kernel data to bypass kernel based protection mechanisms.
In this work we aim at the detection of stack based information leaks to harden the security of the kernel. First, we analyse the problem of kernel infoleaks in [Section 5.3]. Next, we examine the impact of infoleaks attacks on the security of the kernel in [Section 5.4]. Then, we present a technique for detecting kernel based infoleaks through static analysis [Section 5.5]. Next, we evaluate our technique by applying it to the Linux kernel [Section 5.6]. Last, we discuss the applications and limitations of our work [Section 5.6.3] and drawn our concluding remarks in [section 6.1].
Motivation for our work. We analyse the security of current kernel protection mechanisms [CPM+98, Tea01]. Discuss how these protection mechanisms can be circumvented by leveraging on information disclosure vulnerabilities [CWE08b] to access sensitive data of the protection mechanisms. This motivates our work on the detection stack based kernel information leaks.
Contributions. The overall contribution is a systematic approach for the detection of stack based infoleak vulnerabilities, in more detail, we make the following contributions:
- **Analysis of kernel information leak vulnerabilities**, focusing on its impact on the security of kernel protection mechanisms [Section 5.2.1 and 5.4].
- **Classification of kernel information leaks**, based on our analysis we perform a classification of information leaks vulnerabilities [Section 5.3].
- **Detection of kernel stack based information leaks** present and evaluate a technique for the detection of stack based information leaks [Sections 5.5, 5.6].

5.2 Related Work

Two main topics are addressed here:

- The current security protection mechanisms implemented by the Linux Kernel.
- The current techniques to address security vulnerabilities on the Linux Kernel.

5.2.1 Protection Mechanisms

We start reviewing the kernel protection mechanisms against memory corruption vulnerabilities.
StackGuard. StackGuard [CPM+98] is a compiler technique that thwarts buffer overflows [CWE08a] vulnerabilities by placing a “canary” word next to the return address on the stack. If the canary is altered when the function returns a smashing attack has been attempted, and the program responds by emitting an intruder alert.
Address Space Layout Randomization. The goal of Address Space Layout Randomization (ASLR) [Tea01] is to introduce randomness into addresses used by a given task. This will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task.
Data Execution Prevention Data Execution Prevention, also known as No Execute (NX), is a capability of x86 processors to prevent data pages from being used by malicious software to execute code [Cor07, Part 1, Sec 5.13].

- NX features: Non executable and writable pages present W^X, known as NX Intel technology. Carefully controlling the protection bits of the memory pages the OS can reduce the number of pages that are RWX, since these can be used to write code into a writable page and jump to the written code (execute). By setting the protection bits of the kernel memory pages to WˆX, write or execute, but never both, hence mitigating this kind of attack. Still the kernel allows the user to mmap() user pages with RWX protection, therefore, enabling the user to define a exploitable mapping, and redirecting the execution to the user mmap()ed memory pages. A common kind of attack is mmap()ing the NULL page, and causing NULL page dereference from the kernel.

- Supervisor Mode Execution Prevention (SMEP). SMEP introduces security checks to prevent the CPU to fetch user memory pages while the CPU is in CPL 0. This technique blocks the NULL page dereference attacks used to gain code execution. SMEP goes together with SMAP that controls when the CPU can access userland data pages:
  - SMEP detects when kernel is fetching instructions from userland
  - SMAP detects when kernel accessing data from userland.

Still writes from kernel to user still must be enabled at controlled points for the kernel to provide information to the userland and vice-versa (eg. ioctl interfaces), therefore, faults at these controlled points is still possible and its the sole responsibility of the kernel to prevent them, in case the kernel fails to prevent them, we can have two kinds of vulnerabilities:

- Arbitrary kernel memory write (kernel write), affecting the integrity of the kernel
- Arbitrary kernel memory read (infoleaks), affecting the confidentiality of the kernel.

The effectiveness of these protection mechanisms relies on the protection secrets remaining unknown to the attacker, i.e., canary value in the case of StackGuard and the randomized base address to load executable code in case of ASLR. Otherwise, revealing these secrets leads to circumvent these protection mechanisms [ea13b, ea09]. The confidentiality property of the operating system is required for the protection mechanisms to remain effective, confidentiality is achieved through the hardware processors paging and memory management units [Cor07]. However, in the last stage is the task of the OS to ensure the confidentiality of its memory.

5.2.2 Protection Techniques

The detection of software vulnerabilities is a classic topic of computer security, various techniques have been applied to the vulnerability detection. Next, we review and compare related approaches for the detection of infoleaks.
Static and Data flow Analysis Sparse [Tor06] is a Semantic Checker/Parser for C used for kernel code checking and static analysis. The C programming language has some unsafe/unspecified/compiler-dependent behaviours. To solve this limitation Sparse provides C -> AST and then allows the developer to write code analyses to verify the kernel security properties, such as: kernel/user pointers, locks, integer over/under flows, endianness. Smatch [Dan13] is a rewrite of the Standford Checker (MC) using sparse, to provide the static analysis checks at the Linux kernel. Coccinelle [LBP+09] is a tool for performing control-flow based program searches and transformations in C code. Coccinelle is actively used to perform API evolutions of the Linux kernel [LBP+09] as well as finding defects in Linux and open source projects [Stu08]. **Type Inference** Taint Analysis and Type Inference [JW04] as a variants of static analysis have also been performed on kernel. **Fuzzing** Kernel API fuzzing [Jon13] is actively used to test the kernel API for unexpected vulnerabilities. **Real-time detection**: Real-time Intrusion detection techniques (IDS) are prevalent as attack prevention technique [ea13a]. **Operating system**

level techniques PAX security features [Tea01] provide GCC compiler *stackleak* plugin that zeroes all automatic user structs that are allocated on the stack, providing an effective protection against infoleaks. **Coding standards** The safety critical MISRA-C standard [MIS13] mandates the initialization of all automatic variables: *"Rule 9.1: All automatic variables shall have been assigned a value before being used."* However, checking of the mandatory rules requires code reviews to enforce the coding standard, however this can be effective but time consuming and non exhaustive. **Hardware protection techniques** Hardware techniques such as DEP/SMAP (subsection 5.2.1) provide partial protection against infoleaks.

5.3 Classification of Information Disclosure Vulnerabilities

Information disclosure vulnerabilities [CWE08b] are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program. Infoleaks are relevant as they allow for the undesired disclosure of information that circumvents the confidentiality enforced by the operating system [Tan07, Security Threats 9.1.1]. The failure to protect confidentiality can be used by an attacker to increase the attack efficiency, an example of the latter are *stackjack attacks* [RO11] where infoleak vulnerabilities are employed to selectively control the stack values/contents disclosed in order to build a kernel read primitive. The kernel data read primitive is used afterwards to gain knowledge about the kernel protection mechanisms in place, for example as stack pointers, canary values and ASLR base addresses that lead to effective exploits [ea13b, ea09].
In this section we analyse the problem of infoleaks to understand it as the first step towards its solution, we begin our discussion with a real-world infoleak example [Section 5.3.1], next a classification of the different types of infoleaks [Section 5.3.2]. We summarize the classification of infoleak vulnerabilities in figure 5.1, where the type of infoleaks we target appear grayed out.

5.3.1 The Anatomy Of An Infoleak

To focus our discussion we start off with the discussion of a real world infoleak vulnerability CVE-2013-2147 [MIT13]. The CVE-2013-2147 [MIT13] is a kernel stack infoleak that enables an attacker to read 2 bytes of uninitialised memory after field `->buf_size` of the `IOCTL_Command_Struct` where the memory contents are leaked from the kernel stack of the process. The relevant code displaying the vulnerability is depicted in listing 5.1, along with an explanation of the vulnerability details.

```
1 static int cciss_ioctl32_passthru(
2     struct block_device *bdev, fmode_t mode, unsigned cmd, unsigned long arg) {
3     IOCTL_Command_struct arg64;
4     IOCTL_Command_struct __user *p = compat_alloc_user_space(sizeof(arg64));
5     int err;
6     err |= copy_to_user(p, &arg64, sizeof(arg64));
7     if (err)
8         return -EFAULT;
```

Listing 5.1: Example of infoleak code from CVE-2013-2147 (edited to fit)

The listing 5.1 contains an excerpt of function `cciss_ioctl32_passthru()` where the `arg64` local variable is declared at line 3 without explicit initialisation. At the compiler level the effect is that memory from the kernel stack is reserved for the `arg64` variable, however, the `arg64` memory is left uninitialised containing the data already present on the stack. This memory is afterwards copied to user space through the `copy_to_user()` at line 6 that allows an attacker to read the memory contents of the kernel stack.

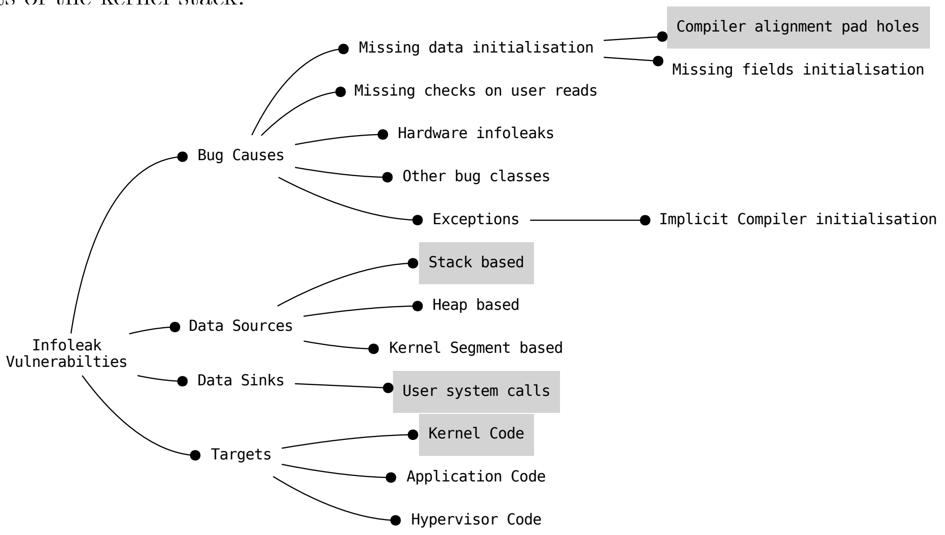

Figure 5.1: Identification and Classification of the Infoleak vulnerabilities.

5.3.2 Targets of Infoleaks

The previous example introduced kernel based infoleaks, however, infoleaks are also present in systems ranging from application to hypervisor level. The following examples give an idea of the targets of infoleaks:

- **Application infoleaks**: A common case of application infoleak is the disclosure of sensitive data by a server process to a remote client CVE-2012-0053 [MIT12].
- **Kernel infoleaks**: These disclose kernel memory as in CVE-2013-2147 [MIT13].
- **Hypervisor infoleaks**: Disclose hypervisor data to guest CVE-2010-4525 [MIT10].

In the case of kernel code, infoleaks have a high impact as disclose sensitive kernel data to user processes breaking the data confidentiality property enforced by the OS [Tan07, 9.1.1]. The above reasoning motivates us to focus on kernel infoleaks as we consider these the most critical case.

5.3.3 Infoleaks Bug Causes

As defined in [Section 5.3], infoleaks are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program. This section analyses the causes that lead to information disclosure.

- **Compiler padding holes.** Compilers align data structures for performance reasons, this leads the compiler to introduce padding holes between structure fields in order to improve their memory access times [Cor07]. Therefore when copying data to the userland the uninitialised struct holes leak kernel information. Padding holes in structures allow data to pass between user-kernel land without explicit checks, this can happen in both directions. Depending on the direction of the information flow, we can identify two situations:
  - **Writes from kernel to user**: Results in an infoleak to userland as depicted in sub-figures 5.2A and 5.2B, and is the case we target in our study.

  - **Writes from user to kernel**: Results in a kernel write from userland as depicted 5.2C. This can be regarded as a critical vulnerability, as represents a kernel write from user land, giving an attacker the ability to alter the contents of kernel memory. However, in this case the contents of the padding holes are usually discarded by the kernel and, are out of the scope of this work.

- **Missing memory initialisation.** When a local variable is declared on a kernel function without explicit initialization, according to the C99 Standard [ISO99, Sect. 6.7.8/10] its contents are indeterminate. In practice, however, the variable gets allocated on the stack, and its value is determined by the memory contents already present on the stack, that remain uncleared for performance reasons. When the variable is copied afterwards to userland it leads to an information leak of kernel memory, as depicted in sub-figure 5.2B.

- **Missing checks on user reads.** Missing or incorrect checks on buffer bounds (start, size) when copying data to user enable the user to read memory contents outside of the buffer. That kind of vulnerability named *buffer overreads* [ea09] allow to read data that was not intended to be accessed.

- **Hardware based infoleaks.** Hardware infoleaks belong to the kind of infoleak provided by the environment, this type of infoleaks is provided by sensitive instructions [PG74], i.e. instructions that modify or reveal CPU machine state to the user, break the confidentiality, and, enable to detect if a system is running under VM/VMM (pills). Examples of sensitive instructions on the x86 architecture are: `sidt/sgdt/sldt/smsw/str` [RI00], that enable to read supervisor related information, such as, the physical address of kernel interrupt descriptor tables.

- **Other bug classes leading to infoleaks.** Other sources of infoleaks not explored in this work, are those related to information already available in the environment, say for example the kernel pointer addresses provided by the `/proc/`, `/sys/` and `/boot/` file-systems that are secured in Linux Kernel by the `kptr_restrict` mechanism [Ros10]. A broader source of information disclosure flaws are covert and side channels, such as cache and TLB timing attacks [ea13b] that exploit the shared nature of these hardware resources to infer information regarding memory addresses.

- **Exceptions.** There are exceptions to the infoleak bug causes discussed above, for example, variable declarations followed by a partial initialization, e.g. with `var = {0}` all fields get initialized with zeros. The behaviour mandated by the C99 Standard [ISO99, Sect. 6.7.8/19] is implemented by the compiler we have used during our analysis GCC [Pro14] The GCC performs the implicit variable zeroing preventing the occurrence of infoleaks, even in the above cases of padding holes or missing initialisation.

5.3.4 Infoleaks Data Sources

Information leaks disclose kernel memory contents, therefore, depending on the memory section affected, a leak can disclose different kinds of information. We focus on the three main sources from where kernel memory is allocated [Gor04].

- **Data segment**. The kernel data segment is the area that contains global kernel variables fixed during compilation time. A data segment leak can disclose the contents of static kernel symbols such as configuration variables.

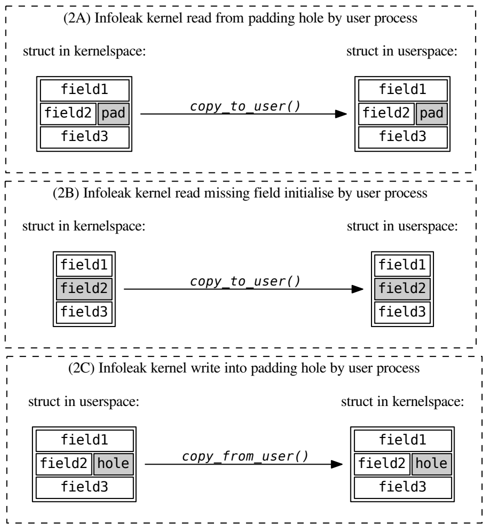

Figure 5.2: Directions of data flow in kernel information leaks and writes.

- **Stack section**. The kernel stack is allocated at runtime and its operation is defined by the kernel C procedure call convention (ABI). Stack content leaks contain valuable information, as they can reveal return addresses, stack pointer, and other data contained in the stack; such as function call parameters, passed on through stack on x86-32 architecture. Other data that is kept on the stack are kernel protection mechanism secrets, such as canary values for StackGuard [CPM$^+$98] protection. In addition, with non-randomized kernel process stacks, the stack layout remains unchanged and provides a predictable stack layout when the same kernel path is called repeatedly [RO11].
- **Heap section**. The kernel heap is managed by memory allocators employed by kernel subsystems when dynamically allocated memory is required. Due to the nature of kernel allocators, heap leaks can disclose memory around the object being allocated and its nearby objects, this can include leaks of object the type and contents, i.e., the values of its fields.

5.4 Analysis on the Impact of Stack Infoleaks

To understand why infoleak vulnerabilities are important to kernel security, we analyse the role played by infoleaks in the attack process, we start with the steps that compose a kernel attack [subsection 5.4.1], then, analyse the contents of the stack in [subsection 5.4.2] and last infoleak based attacks [subsection 5.4.3].

5.4.1 The Anatomy of An Attack

Kernel attacks have the goal of increasing the privilege level of the code run by the attacker. To do so, the attacker must gain controlled code execution capabilities to increase its privilege level, code execution is achieved by exploiting kernel vulnerabilities. An attack scenario with current kernel protection mechanisms [subsection 5.2.1] in place involves the following steps to break kernel security:

1) Perform identification of the system.

2) Bypass the kernel protection mechanisms: Find canary, return address.
   2.1) Find a kernel memory read vulnerability (infoleak).
3) Transfer kernel execution control to injected attacker code: Write canary, return address.
   3.1) Find an arbitrary kernel memory write vulnerability.

An attacker targeting the kernel performs the above steps, among them, the step (2.1) is required to gain enough information to ensure attack success, otherwise, the result of a failed attack is panic/halt the system. To succeed the attacker must find a way learn the canary value and return address. In user land attacks this is achieved using brute force to guess the secrets, where failed probe causes the server process to be restarted without major interference to the system. However, this no longer happens in kernel land, where a failed canary overwrite leads to a kernel panic.

5.4.2 The Contents of the Kernel Stack

The target of this work are kernel stack based infoleaks, to analyse the impact of infoleaks we examine the contents of the kernel stack when a kernel system call is performed in order to identify what kinds of information can be obtained by an attacker. The contents of the kernel stack on a function call are defined by the C API function call procedure which in our case is defined in the Intel x86-32 architecture [Cor07], and implemented by the C compiler GCC x86-32 [Pro14]. The Table 5.1 details the contents of the kernel process stack when a kernel system call is invoked, three main groups of data can be identified, starting from the top:

- The callee arguments to the function call: *callee(param1, param2, param3, ...)*
- The saved CPU registers stored by the caller to restore on return: *EIP, EBP, canary.*
- The local variables of the callee: *local1, local2, local3, ...*

Address	Value	Description
$ebp + 8$	params	Function parameters
$ebp + 4$	SEIP	Saved EIP
$ebp + 0$	SEBP	Saved EBP *(optional)*
$ebp - 4$	PAD	GCC Stack padding
$ebp - 8$	CAN	Saved Stack Canary
$ebp - 12$	locals	Local variables

Table 5.1: Stack Layout on function call() relative to %ebp (x86-32)

All kernel memory dumps are a security issue, however, the kernel stack dumps are more relevant due to the nature of stack, and the C99 language [ISO99] calling conventions, the following contents can be found:

- params: The argument/parameters passed to the function.
- SEIP: The return addresses to kernel code, enable to find the kernel load address.

  - Kernel address reveal the load address of the kernel image.
  - Module address reveal the load address of the module address.

- SEBP: The saved stack base pointer *(optional)*.
- CAN: The saved canary (per kernel process thread).
- locals: The local variables allocated by the function.

5.4.3 Infoleak Based Attacks

After an analysis of the contents of the stack in subsection 5.4.2, we outline the possible uses of the information obtained by an infoleak.

- **Precise system identification**

  Perform the identification of the system. Being able to leak kernel addresses, enables effective system identification of the exact kernel version running on the system, by fingerprinting the kernel function addresses, and building a table of tuples $(address, kernel\_version)$. This fact should not be overlooked since effective identification of the running kernel target, is the first step towards effective attacks.

- **StackGuard protection bypass**

  Obtain the canary values to bypass the kernel StackGuard [CPM+98] protection. As seen in Table 5.1, the canary value (CAN) resides on the stack, therefore, an info leak reveals the canary value of the current kernel thread process allowing to bypass the *StackGuard*.

- **KASLR protection bypass**

  Obtain the kernel text return addresses to bypass the kernel KASLR [Coo13] protection. As seen in Table 5.1, the return address value (SEIP) resides on the stack, therefore, an info leak reveals the SEIP ($aslr\_ktext\_addr$) value of the current kernel thread process allowing to bypass *KASLR*. Since both $aslr\_ktext\_addr$ and $ktext\_addr$ are known, the randomised kernel text section load offset introduced by the *KASLR* can be computed as: $aslr\_ktext\_offset = aslr\_ktext\_addr - ktext\_addr$ [Byp09].

- **Stack trace**

  Full stack-trace of the call leading to info leak.

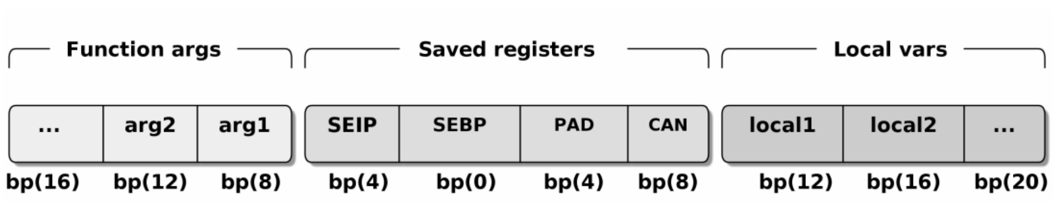

Figure 5.3: Stack Layout on function call relative to ebp (x86-32).

5.4.3.1 Infoleak Attacks Variations

The stack infoleaks disclose only the uninitialised part of the current stack frame, therefore, acting as a window that enables to look at the contents of the stack. The infoleak window is described as $W = (offset, size, contents)$:

- $offset$: The $offset$ is measured from the top of the stack and its value fixed by the function call stack frame [1].
- $size$: The $size$ is fixed and defined by the size of uninitialised stack frame section disclosed.
- $contents$: The contents of the stack revealed by the infoleak window. These can be influenced by the attacker, since calling different system calls, leaves the contents of the $syscall()$ stack frame, that can be later retrieved by triggering the infoleak.

Figure 5.4 depicts this scenario where the contents of the kernel process stack are shown after performing different syscalls: In Figure 5.4(a) after $syscall1()$ the kernel stack contains $W.contents = 0x11111111$. In Figure 5.4(b) after $syscall2()$ the kernel stack contains $W.contents = 0x22222222$. The scenario depicts how different $W.contents$ can be disclosed depending on the $syscall$ invoked by the attacker. This case is important for system calls that leave security critical information on the kernel stack, as well as, for small infoleaks.

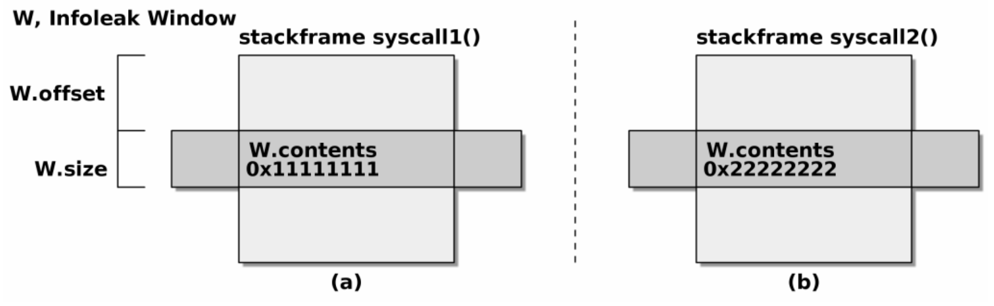

Figure 5.4: Kernel stackframe infoleak window.

5.5 The Detection of Information Leak vulnerabilities

In this section we present our technique, the main steps of the technique are outlined here: (5.5.1) Analysis of the attack and vulnerability model. (5.5.2) Design a semantic patch of the vulnerability model. (5.5.3) Filter and rank the code matches to remove false positives. (5.5.4) Review and correct the vulnerabilities detected.

5.5.1 Infoleak Vulnerability Model

We analyse infoleaks vulnerabilities in order to model them as a first step towards the detection of infoleaks. In our model of stack based kernel infoleak vulnerabilities we adopt the notions of taint analysis [DD77]. We focus on infoleaks of privileged kernel memory to userland as depicted in figure 5.2, and start with the identification of data sources, data sinks, and taint property:

- **Data Sources**: The interesting *data sources* for our analysis are the uninitialised kernel stack memory contents. As discussed in [Section 5.3.1] on source kernel data are uninitialised local variables declared on kernel functions.

- **Data Sinks**: The type of *data sinks* we are interested are those reachable from userland, these are part of the kernel API exposed through the system call interface. Examples of these are file-system *read()* operations these are interesting sinks for our analysis as they allow data to flow from kernel to user, here we focus on the `copy_to_user()` calls data sinks.

[1]The analysis of attacks that enable to control $offset$ is on going.

- **Taint Property**: The taint property we are interested in is the flow of uninitialised data from the identified kernel space *sources* to user space *sinks*.

5.5.2 Semantic Patch Preparation

Based on the vulnerability model developed in our previous analysis, we prepare a semantic patch [LBP+09] to perform control-flow program static analysis to detect vulnerable code sites matching the vulnerability model. To this end we select Coccinelle [LBP+09] that is an open-source developer-friendly static analysis tool widely used in open source projects to perform automated API evolutions.

```
1 handler(...) {
2 <...
3     T ID;
4     ... when != memset(&ID, 0, ...)
5         when != ID = ...
6 *   copy_to_user(EV, &ID, EN)
7 ...> }
```

Listing 5.2: Semantic patch (SmPL) for stack based infoleak detection (edited to fit)

For our analysis we develop a Coccinelle semantic patch depicted at listing 5.2 that matches the infoleak vulnerability model discussed above.

- **Data Source**: The local variable `ID` of `handler()` declared at line 3.
- **Data Sink**: The local variable `ID` is copied to the user pointer `EV` at line 6.
- **Taint Property**: The property we want to ensure is that memory contents of `ID` remain uninitialised, therefore, we restrict to the situations where no `memset()` or initialisation operations occur at line 4.

5.5.3 Filter and Rank of matches

The results of the execution of the semantic patch discussed at the step 5.5.2 contain the potential vulnerabilities ranked according to its likelihood of being a real vulnerability. The ranking is performed to reduce the amount of required manual work during code audits of the infoleak detection results to increase the vulnerability detection rate. For each code location matched by the semantic patch, the following fields are extracted from the match to identify each vulnerability $vuln = (function, variable, struct)$. The filtering function in equation 5.5.3 calculates the size of the infoleak in bytes as the size of the padding holes in the struct. The equation 5.5.3 determines the relevance of the infoleak and allows to order the results giving a higher relevance to those leaking more bytes.

$$leaksize(struct) = sizeof(struct) - \sum_{f \in struct} sizeof(struct.f) = \begin{cases} = 0 & \text{No leak.} \\ > 0 & \text{Leak.} \end{cases}$$

5.5.4 Infoleak Code Review and Correction

The last step is to review the detected vulnerabilities, to triage the real bugs out of the potential vulnerabilities. This is the only step requiring manual intervention, but, can be partially automated by zeroing all the detected local variable declarations thus preventing the detected infoleaks, however, requires a compromise between performance impact and security implications.

5.6 Experimental Evaluation of the Detection Technique

To evaluate our approach we select the Linux kernel sources as the target for the detection potential infoleaks. First, we evaluate our approach using an experiment aimed at detecting already known vulnerabilities [Section 5.6.1]. Last, we study how our approach applies to discovery of new vulnerabilities [Section 5.6.2].

5.6.1 Existing Infoleak Detection

To evaluate the performance of our detection technique, we prepare an experiment targeting known infoleak vulnerabilities present in the Linux kernel v3.0 series. For this we first review the MITRE Vulnerabilities CVE database, and select several stack based infoleak vulnerabilities in Linux kernel, such as CVE-2013-2147 [MIT13]. With this set of infoleak vulnerabilities we prepare a kernel source tree containing the unpatched vulnerabilities, then target our detection approach towards it to verify the approach detects the infoleak vulnerabilities introduced. With this we can evaluate the detection performance of our technique.

Measure/Kernel ver	*v2.6*	*v3.0*	*v3.2*	*v3.4*	*v3.8*	*v3.14*
Vulns Detected/Present	13/8	14/8	12/6	12/6	11/4	9/4
True Positive (TPR%)	100.0	100.0	100.0	100.0	100.0	50.0
True Negative (SPC%)	99.2	99.2	99.3	99.3	99.4	99.5
Positive Pred (PPV%)	61.5	57.1	50.0	50.0	36.3	22.2
False Positive (FPR%)	0.8	0.8	0.7	0.7	0.6	0.5

Table 5.2: Statistical performance of stack infoleak detection per kernel version.

The table 5.2 shows the statistical performance measures of the infoleak detection for stack based kernel infoleaks with $leaksize(struct) > 0$, i.e., those where the bug cause are compiler padding holes. The detection performance presents a high sensitivity (TPR) and high specificity (SPC) both close to 100%, while the false positive rate (FPR) is close to zero. This enables analysts to perform security code audits to verify and correct the vulnerabilities detected.

5.6.2 Discovery of Vulnerabilities

We have applied our detection technique to the Linux kernel v3.12, as a result five new infoleak vulnerabilities have been uncovered disclosing between two and four bytes of the kernel stack contents to userland. The affected device driver files are: `net/wan/{farsync.c,wanxl.c}`, `tty/{synclink .c,synclink_gt.c}`, and `net/hamradio/yam.c`. After preparation of the corresponding patches that correct the infoleaks have been applied to the kernel development tree [Pei14].
During 2015 we performed a second infoleak analysis to the Linux kernel v4.0, applying our detection technique that uncovered three new infoleak vulnerabilities that disclose between 16 and 200 bytes of kernel stack contents to userland, the corrections have been applied to the kernel development tree [Pei15]. Among the vulnerabilities we found [Pei15, CVE-2014-1739] that caused 200 bytes of kernel stack to be read by a local user on devices using the Linux media subsystem, and affected desktop and Android devices using Linux video devices and cameras

(V4L).

5.6.3 Applications and Limitations of our Approach

We believe that our approach improves kernel security, we base our discussion on the Linux kernel, however, the approach is applicable to systems presenting a similar vulnerability model. The main application of our technique is on conducting security audits at different stages of the development cycle: **(a) At Release stage** to ensure that less bugs get into the product release. **(b) At Development stage** to avoid introducing errors in early development stage. **(c) At Regression stage** to ensure a known bug is not re-introduced.
Similar to other methods for the discovery of security flaws, our approach cannot overcome the inherent limitations of vulnerability identification, i.e., vulnerability detection is an undecidable problem that stems from Rice's theorem [Hop08]. Our technique aims at finding known vulnerabilities at the source code level, therefore unknown flaws not matching the vulnerable model remain undetected, for example our model only considers infoleaks at a single function level, therefore,infoleaks involving multiple functions are not covered by our approach. The result derives from the limitations of black-listing as a security measure [SS75, Fail-safe defaults], where blacklist detects only a subset of unallowed patterns. Therefore, a better approach is to enforce a white-list to detect all unallowed patterns.

5.7 Conclusions and Further Work

In this work, we presented an analysis and classification of information leaks causes and their impact on the security of the kernel. Then, we proposed a technique for the detection of the class of stack based kernel information leaks. Last, we evaluated our technique applying it to the Linux kernel, our evaluation results showed that the detection technique is effective to improve operating system security. We focused on infoleaks at operating system level, however, infoleaks are present in hypervisors as well, where a malicious guest virtual machine can use infoleaks to compromise the security of the remaining guests. Further work covers detection of infoleaks in hypervisor to improve the confidentiality of the guest virtual machines [SPM+12] and overcome our approach limitations [Section 5.6.3].

This page is intentionally left blank.

Chapter 6
Conclusions and open research lines

This chapter summarizes the main conclusions of this dissertation, the section 6.1 summarizes the main contributions, the section 6.2 presents the research lines open by this study. Last, the section 6.3 lists the national and international publications in conferences and research journals that relate to this work.

6.1 Conclusions

The availability of new processors with more processing power for embedded systems has raised the development of applications the tackle problems of greater complexity [D+09]. Currently, the embedded applications have more features, and as a consequence, more complexity. For this reason, there exists a growing interest in allowing the secure execution of multiple applications that share a single processor and memory. In this context, partitioned system architectures based on hypervisors have evolved as an adequate solution to build secure systems.
One of the main challenges in the construction of secure partitioned systems is the verification of the correct operation of the hypervisor, since, the hypervisor is the critical component on which rests the security of the partitioned system. Traditional approaches for V&V, such as testing, inspection and analysis, exhibit limitations that make impractical its application for the exhaustive verification of the system operation, due to the fact that the input space to validate grows exponentially with respect to the number of inputs to validate. Given this limitations, verification techniques based in formal methods arise as an alternative to complement the traditional validation techniques.
This dissertation has analysed the current state of art in the verification and validation of secure hypervisors, focusing in the application of formal methods for the hypervisor validation and verification. The analysis of the state of art has identified the open problems to solve, we proposed the following solutions to the open problems that result in the main contributions of this work:

- The formal model of the hypervisor and the definition of the properties that ensure the correctness of the hypervisor operation.
- The verification of the functional correctness of the hypervisor components source code.
- The analysis and detection of the vulnerabilities that affect the confidentiality of the information managed by the hypervisor and the rest of the components of the partitioned system.

As concluding remarks, the main contributions of this thesis contribute to a bigger goal, that is, the application of formal methods as a tool to unambiguously and precisely define the exact behaviour of a secure software system. This application enables to bridge the gap between natural language based requirements specification and the implementation. Thereby, providing a mathematical specification of the behaviour of the software systems to use as basis for the design, implementation and validation phases. The main benefits of this application being the identification of security issues, design and implementation flaws in early stages of the development.

6.2 Research Lines

The work performed in this dissertation opens various research lines. Some of these research lines are briefly detailed next:

- This dissertation deals exclusively with the confidentiality of the information managed by the hypervisor. This work can be extended by considering the remaining aspects that compose the information security, namely: the information integrity and availability.
- The wide adoption of multi-core CPU architectures in consumer electronics [PGPC15, TCAP14] present new challenges to the V&V of multi-core systems. The key point to address is the exponential growth of the program state space as the number of concurrent CPUs executing increases. Formal methods can be applied to model the concurrent aspect of multi-core systems, in order to verify security properties, such as: absence of deadlocks, or absence of race conditions.
- Another research line is the improvement of the compilers used to build secure systems to overcome the limitations of the C99 standard [ISO99], such as, undefined and unspecified behaviours present in the C99 standard, that have been analysed in this dissertation (chapter 5).

6.3 Publications related to this thesis

The Table 6.1 classifies the main publications related with this dissertation, that are detailed below according to the publication type:

- Identifier: The identifier of the publication.
- Type: The type of publication (Conference/Journal).
- Ranking: The ranking of the publication based on the following rankings:
  - CORE: Conference publication according to the Computing Research Education (CORE) ranking.
  - JCR: Journal Classification according to the Journal Citation Reports (JCR) ranking.

Identifier	Type	Ranking	Scope
[PMC15]	Journal	Factor: 0.52	International
[PGPC15]	Journal	Factor: 1.35	International
[PMMC14]	Conference	Core B	International
[CMC$^+$14]	Conference	Core C	International
[SPM$^+$12]	Conference	_	International
[SP13]	Conference	_	National

Identifier	Type	Ranking	Scope
[CEPP12]	Conference	Core B	International
[MPS+12]	Conference	_	International
[MPS+11]	Conference	_	National
[MRCP10]	Conference	_	International
[RCM+10]	Conference	_	International
[CRMP10]	Conference	Core C	International

Table 6.1: Listing of main publications of this dissertation

Bibliography

[Ai96] Airlines Electronic Engineering Committee . *Avionics Application Software Standard Interface (ARINC-653)*, March 1996. Airlines Electronic Eng. Committee. 22, 37, 40, 49

[AFOTH06] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The mils architecture for high-assurance embedded systems. *IJES*, 2(3/4):239–247, 2006. 89

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. 46

[ALRL04] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004. 26, 32, 53

[AM08] P. Arberet and J. Miro. IMA for space : status and considerations. In *ERTS 2008. Embedded Real-Time Software.*, Jannuary. Toulouse. France 2008. 38

[AMGC09] P. Arberet, J.-J. Metge, O. Gras, and A. Crespo. TSP-based generic payload on-board software. In *DASIA 2009. DAta Systems In Aerospace.*, May. Istanbul 2009. 38

[And72] Anderson, James P. Computer Security Technology Planning Study. Volume 2. Technical report, Electronic Systems Division (ESD) of the United States Air Force, 1972. 89

[And86] Stephen J Andriole. *Software validation, verification, testing and documentation: a source book*. Petrocelli Books, Inc., 1986. 25

[BBBB10] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Ingredients of Operating System Correctness: Lessons Learned in the Formal Verification of PikeOS. In *Emb. World Conf., Nuremberg, Germany*, 2010. 29, 55

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010. 54

[BCF$^+$11] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliatre, Claude Marche, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language Version 1.5. *none*, 2011. 55, 56

[BF$^+$14] Pierre Bourque, Richard E Fairley, et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014. 22, 23, 25, 26

[Bow93] J. Bowen. Formal methods in safety-critical standards. In *Software Engineering Standards Symposium, 1993. Proceedings., 1993*, pages 168–177. IEEE, 1993. 32

[Byp09] Bypassing PaX ASLR protection. Tyler Durden, 2009. http://www.phrack.com/issues.html?issue=5&id=9. 69

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009. 29, 54

[CEH02] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in metal. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 51–60, New York, NY, USA, 2002. ACM. 31

[CEPP12] Hakan Cankaya, Thomas Enderle, Salva Peiro, and Andreas Platschek. OVERSEE: Investigation of Requirements and Analysis of Solutions for an In-Vehicle Open and Secure Platform. In *19th ITS World Congress, Vienna, Austria, 22/26 October 2012*, 2012. 77

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. 30, 54, 56

[CMC+14] Alfons Crespo, Miguel Masmano, Javier O. Coronel, Salvador Peiró, Patricia Balbastre, and Jose Simo. Multicore partitioned systems based on hypervisor . In *19th IFAC World Congress to be held in Cape Town, Africa, August 2014.* . Instituto de Automática e Informática Industrial, Universitat Politècnica de València, Spain, 2014. 76

[CMW+11] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011. 21, 62

[Coo13] Kees Cook. Linux Kernel ASLR (KASLR). In *Linux Security Summit*, October 2013. 69

[Cor07] Intel Corp. *IA-32 Architecture Software Developer's Manual - Volume 3A*, 2007. 61, 62, 63, 65, 68

[Cov02] Coverity. Prevent, 2002. Online: http://www.coverity.com. 31

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998. 61, 62, 67, 69

[Cri06] Common Criteria. Common Criteria for Information Technology Security Evaluation v3.1. Release 3, 2006. 53, 59

[Cri12] Common Criteria. Common Criteria for Information Technology Security Evaluation v3.1. Release 4, September 2012. CMB-2012-09-001, CMB-2012-09-002, CMB-2012-09-003. 29, 30, 32, 34

[CRMP10] A. Crespo, I. Ripoll, M. Masmano, and S. Peiro. Partitioned Embedded Architecture based on Hypervisor: the XtratuM approach. In *8th European Dependable Computing Conference*, 2010. 77

[CWE08a] MITRE. Common Weakness Enumeration. CWE-121: Stack-based Buffer Overflow., 2008. http://cwe.mitre.org/data/definitions/121.html. 62

[CWE08b] MITRE. Common Weakness Enumeration. CWE-200: Information Exposure., 2008. http://cwe.mitre.org/data/definitions/200.html. 5, 7, 9, 62, 64

[D$^+$09] Daniel Dvorak et al. Nasa study on flight software complexity. *NASA office of chief engineer*, 2009. 21, 24, 75

[Dan13] Dan Carpenter. Smatch, Static analysis for C, 2013. 63

[DD77] Dorothy E Denning and Peter J Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977. 70

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972. 25

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975. 56

[ea09] R. Strackx et al. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM. 63, 64, 66

[ea13a] A. Herrero et al. RT-MOVICAB-IDS: Addressing real-time intrusion detection. *FGCS '13*, 29(1):250–261, 2013. 63

[ea13b] R. Hund et al. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE SSP*, 2013. 63, 64, 66

[ECS09a] ECSS: European Cooperation for Space Standardization. *ECSS-E-ST-40C Space Engineering - Software*. ESA-ESTEC, Requirements & Standards Division, March 2009. 33, 35

[ECS09b] ECSS: European Cooperation for Space Standardization. *ECSS-Q-ST-80C - Software Product Assurance*. ESA-ESTEC, Requirements & Standards Division, March 2009. 33

[fAR85] US Radio Technical Commission for Aeronautics (RTCA). *RTCA DO-178: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, January, 1985. 32

[fAR92] US Radio Technical Commission for Aeronautics (RTCA). *RTCA DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, December 1, 1992. 29, 30, 32, 53, 59 [fAR05] US Radio Technical Commission for Aeronautics (RTCA). *RTCA DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. RTCA, 2005. 32, 89

[fAR11] US Radio Technical Commission for Aeronautics (RTCA). *RTCA DO-333: Formal Methods Supplement to DO-178C and DO-278A*. RTCA, December 13 2011. 32

[fAR12] US Radio Technical Commission for Aeronautics (RTCA). *RTCA DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, January 5, 2012. 21

[Gol74] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974. 22, 38, 89

[Gor04] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004. 66

[HALM08] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Applying Formal Methods to a Certifiably Secure Software System. *IEEE Trans. Software Eng.*, 34(1):82–98, 2008. 30, 55

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969. 30, 56

[Hop08] J. E Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2008. 73

[IEC10] IEC. IEC-61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 2010. 29, 33, 35, 53

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, 2014. 23

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. 66, 68, 76

[Jon13] D. Jones. The Trinity system call fuzzer, Linux Kernel, 2013. 63

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004. 63

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009. 30, 54

[LBP$^+$09] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 43–52. DSN'09, IEEE, 2009. 31, 63, 71

[LSM$^+$98] Peter A Loscocco, Stephen D Smalley, Patrick A Muckelbauer, Ruth C Taylor, S Jeff Turner, and John F Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, volume 10, pages 303–314, 1998. 26

[Mat09] MathWorks. The Polyspace verification tool , 2009. 54

[Mey92] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25:40–51, 1992. 23, 24

[MIS13] MISRA. MISRA C:2012. Guidelines for the use of the C language in critical systems, March 2013. 64

[MIT10] MITRE. CVE-2010-4525. kvm: x86: zero kvm_vcpu_events->interrupt.pad infoleak, 2010. CVE-2010-4525. 65

[MIT12] MITRE. CVE-2012-0053: Apache information disclosure on response to Bad HTTP Request, 2012. CVE-2012-0053. 65

[MIT13] MITRE. CVE-2013-2147. fix info leak in cciss_ioctl32_passthru()., 2013. https://git.kernel.org. 64, 65, 72

[MPS$^+$11] M. Masmano, S. Peiro, J. Sanchez, J. Simo, and A. Crespo. Device Virtualization in a Partitioned System: the OVERSEE Approach. In *Jornadas de Tiempo Real (JTR-2011)*. Instituto de Automática e Informática Industrial, Universitat Politècnica de València, Spain, 2011. 77

[MPS$^+$12] M. Masmano, S. Peiro, J. Sanchez, J. Simo, and A. Crespo. IO Virtualisation in a Partitioned System . In *ERTS2012: Embedded Real Time Software and Systems 2012*. Instituto de Automática e Informática Industrial, Universitat Politècnica de València, Spain, 2012. 77

[MRC05] M. Masmano, I. Ripoll, and A. Crespo. Introduction to XtratuM. 2005. 38

[MRC$^+$09] M. Masmano, I. Ripoll, A. Crespo, J.J. Metge, and P. Arberet. Xtratum: An open source hypervisor for TSP embedded systems in aerospace. In *DASIA 2009. DAta Systems In Aerospace.*, May. Istanbul 2009. 51

[MRCM09] M. Masmano, I. Ripoll, A. Crespo, and J.J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *Eleventh Real-Time Linux Workshop*, Dresden (Germany), 28-30 September 2009. 54

[MRCP10] M. Masmano, I. Ripoll, A. Crespo, and S. Peiro. XtratuM for LEON3: an OpenSource Hypervisor for High-Integrity Systems . In *Embedded Real Time Software and Systems ERTS2010*. Toulouse, France, May 2010. 77

[oEE90] Institute of Electronical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology (IEEE-610.12-1990). *Office*, 121990(1):1, 1990. 26, 89

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated Deduction—CADE-11*, pages 748–752. Springer, 1992. 31

[Pau94] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994. 30

[Pei14] S. Peiró. CVE request: Assorted Kernel infoleak security fixes, 2014. CVE-2014-1444. 72

[Pei15] S. Peiró. CVE Request: Linux Kernel ioctl infoleaks fixes, 2015. CVE-2015-7884 and CVE-2014-1739. 72

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974. 66

[PGPC15] Héctor Pérez, Javier Gutiérrez, Salvador Peiró, and Alfons Crespo. Distributed architecture for developing mixed-criticality systems in multi-core platforms. In *Journal of Systems and Software (JSS)* , 2015. (pending publication). 76

[PL10] D. Pariente and E. Ledinot. *Formal Verification of Object-Oriented Software*. Karlsruhe Institute of Technology, Faculty of Informatics, Paris, France, June 28-30 2010. 30

[PMC15] S. Peiró, M. Muñoz, and A. Crespo. Analysing the Impact and Detection of Kernel Stack Infoleaks . In *Logic Journal of the IGPL*. Springer, 2015. (pending publication). 76

[PMMC14] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. Detecting Stack based kernel Information leaks . In *CISIS'14*. Springer, 2014. 76

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. 53

[Pro14] The GNU Project. The GNU C Compiler Collection (GCC) gcc-4.7, 2014. http://gcc.gnu.org/gcc-4.7/. 66, 68

[RCM$^+$10] I. Ripoll, A. Crespo, M. Masmano, V. Brocal, P. Balbastre, S. Peiró, P. Arberet, and J.J. Metge. Configuration and Scheduling tools for TSP systems based on XtratuM. In *DASIA 2010. DAta Systems In Aerospace.*, May. Budapest 2010., 2010. Instituto de Automatica e Informatica Industrial, Universitat Politècnica de València, España. http://www.fentiss.com/documents/dasia2010.pdf. 77

[Res05] Gaisler Research. Leon2 processor user's manual, version 1.0.30. http://www.gaisler.com, 2005. 38

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 3–4, 2000. 66

[RO11] D. Rosenberg and J. Oberheide. Stackjacking: A PaX exploit framework, 2011. https://github.com/jonoberheide/stackjacking/. 64, 67

[Ros10] Dan Rosenberg. kptr_restrict for hiding kernel pointers, 2010. http://lwn.net/Articles/420403/. 66

[Rus81a] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM. 22, 37

[Rus81b] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP 81, pages 12–21, New York, NY, USA, 1981. ACM. 29, 54

[Rus00] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000. 53, 89

[Rus01] John Rushby. Security requirements specifications: How and what? In *Symposium on Requirements Engineering for Information Security (SREIS)*, Indianapolis, IN, mar 2001. 22

[SP13] A. Crespo S. Peiró, M. Masmano. Formal Validation of XtratuM Components . In *Jornadas de Tiempo Real (JTR-2013)*. Instituto de Automática e Informática Industrial, Universitat Politècnica de València, Spain, 2013. 76

[SPM$^+$12] J. Sánchez, S. Peiró, M. Masmano, J. Simó, and P. Balbastre. Linux porting to the XtratuM Hypervisor for x86 processors. In *14th Real Time Linux Workshop*, October 2012. 73, 76

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 73

[Stu08] Henrik Stuart. *Hunting Bugs with Coccinelle*. PhD thesis, Diku, 2008. 63

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin / Heidelberg, 2009. 30, 55

[Tan07] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2007. 64, 65

[TCAP14] Salvador Trujillo, Alfons Crespo, Alejandro Alonso, and Jon Pérez. Multipartes: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. *Microprocessors and Microsystems*, 38(8):921–932, 2014. 76

[Tea01] PAX Team. PaX address space layout randomization (ASLR), 2001. http://pax.grsecurity.net/docs/aslr.txt. 61, 62, 64

[Tor06] Linus Torvalds. Sparse: A semantic parser for C, 2006. http://sparse.wiki.kernel.org. 63

This page is intentionally left blank.

Acronyms

API Application Programming Interface. 25, 31, *Glossary:* API

ARINC-653 ARINC-653. 22

ARM Acorn RISC Machines. 30

CC Common Criteria. 29, 30, 32, 34

CORE Computing Research Education. 76

COTS Commercial-Off-The-Shelf. 15, 24, 25

DO-178A Software Considerations in Airborne Systems and Equipment Certification. 32

DO-178B Requirements and Technical Concepts for Aviation. 32

DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. 32

DO-333 Formal Methods Supplement to DO-178C and DO-278A. 32

EAL Evaluation Assurance Level 6. 35

EAL Evaluation Assurance Level. 34

EAL7 Evaluation Assurance Level 7. 30, 34, 35

ECSS European Cooperation for Space Standardization. 32, 33

ESA European Space Agency. 33, 35

FSM Finite State Machines. 5, 9, 27

IEC International Electro-technical Commission. 32, 33, 35

IEC-61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. 33

IEEE Institute of Electrical and Electronics Engineers. 26

IMA Integrated Modular Avionics. 22, 32, 87, *Glossary:* IMA

ISO/IEC ISO/IEC. 35

IT Information Technologies. 32, 35

JCR Journal Citation Reports. 76

MILS Multiple Independent Levels of Security. 22, *Glossary:* MILS

Partitioning Robust partitioning. *Glossary:* Partitioning

Reference monitor Reference Monitor. *Glossary:* Reference monitor

RISC Reduced Instruction Set Computers. 87

RTCA US Radio Technical Commission for Aeronautics. 32, 35

SDLC Software Development Life Cycle. 30

Separation kernel Separation kernel. *Glossary:* Separation kernel

SIL Safety Integrity Level. 17, 33

SLOC Source Lines of Code. 30

SWEBOOK Software Engineering Book. 26

TCB Trusted Computing Base. 75, *Glossary:* TCB

V&V Validation and Verification. 5, 7, 9, 15, 21–27, 30, 75, 76

VM virtual machine. 22

VMM Virtual Machine Monitor. 22, *Glossary:* VMM

Glossary

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 87

Complexity The measure of how difficult software is to understand, and thus to analyse, test, and maintain [oEE90]. . 24

IMA "Integrated Modular Avionics is defined as a shared set of flexible reusable, and interoperable hardware and software resources that create a platform that provides services, designed and verified to a defined set of safety and performance requirements, host applications performing aircraft related functions" [fAR05]. 87

MILS Multiple Independent Levels of Security, see Rushby et al. [AFOTH06]. 88, 89

Partitioning Refers to Rushby "Gold standard" for robust partitioning [Rus00].

> "A partitioned system should provide fault containment equal to an idealized system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines.
> "

- *Spatial partitioning* must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions.
- *Temporal partitioning* must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it.. 88

Reference monitor Defined by Anderson et al. in [And72].. 88

Separation kernel See MILS.. 88

TCB Trusted Computing Base, see Rushby et al. [AFOTH06]. 88

VMM Virtual Machine Monitor, see Goldberg et al. [Gol74]. 88