Cross-platform **O**pen **S**ecurity **S**tack for **Co**nnected Device

D2.4 CROSSCON Formal Framework - Final

Document Identification			
Status	Final	**Due Date**	31/03/2025
Version	1.0	**Submission Date**	31/03/2025

Related WP	WP2	**Document Reference**	D2.4
Related Deliverable(s)	D2.2, D2.3	**Dissemination Level(*)**	PU
Lead Participant	UNITN	**Lead Author**	Marco Roveri
Contributors	BEYOND	**Reviewers**	Ziga Putrle (BEYOND), Huimin Li (TUD)

Keywords
CROSSCON Trusted Execution Environment, CROSSCON Hypervisor, CROSSCON System on Chip, CROSSCON Formal Specification

This document is issued within the frame and for the purpose of the CROSSCON project. This project has received funding from the European Union's Horizon Europe Programme under Grant Agreement No.101070537. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level: (PU) Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page). (SEN) Sensitive, limited under the conditions of the Grant Agreement. (Classified EU-R) EU RESTRICTED under the Commission Decision No2015/444. (Classified EU-C) EU CONFIDENTIAL under the Commission Decision No2015/444. (Classified EU-S) EU SECRET under the Commission Decision No2015/444.

Document Information

List of contributors	
Name	**Partner**
M. Roveri	UNITN
J. Mihelič	BEYOND
A. Tacchella	UNITN
Z. Putrle	BEYOND

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	14/11/2024	M. Roveri (UNITN)	Initial draft structure.
0.2	20/11/2024	M. Roveri (UNITN)	Copied material from D2.2.
0.3	04/12/2024	M. Roveri (UNITN)	Revised sections 2, 3, and 4.
0.4	15/12/2024	J. Mihelič (BEYOND)	Revised sections 2 to introduce the initial version of the dynamic creation of VMs.
0.5	20/12/2024	A. Tacchella (UNITN)	Revised formalization of the dynamic creation of VMs.
0.6	07/01/2024	A. Tacchella (UNITN)	Initial version of the formalization of the verification of the CROSSCON configuration file.
0.7	15/01/2025	M. Roveri (UNITN)	Revised the formalization of the formalization of the verification of the CROSSCON configuration file.
0.8	20/01/2025	Z. Putrle (BEYOND)	Added initial structure of experience chapter on the use of open source verification tools.
0.9	20/01/2025	Z. Putrle (BEYOND)	Some work on the chapter on the experience on the use of open source verification tools.
0.10	24/01/2025	M. Roveri (UNITN)	Global revision of what is in the document so far.
0.11	10/02/2025	A. Tacchella (UNITN)	Initial draft of the chapter on the verification of the dynamic VM creation.
0.12	17/02/2025	M. Roveri (UNITN)	Revised the chapter on the verification of the dynamic VM creation.
0.13	20/02/2025	Z. Putrle (BEYOND)	Some work on the chapter on the experience on the use of open source verification tools.
0.14	25/02/2025	A. Tacchella (UNITN)	Initial draft of the chapter on the certification manifest.
0.15	05/03/2025	Z. Putrle (BEYOND)	Some further work on the chapter on the experience on the use of open source verification tools.
0.16	06/03/2025	M. Roveri (UNITN)	Some global revision.

0.17	06/03/2025	Z. Putrle (BEYOND)	Some further work on the chapter on the experience on the use of open source verification tools.
0.18	13/03/2025	Z. Putrle (BEYOND)	Finalized the chapter on the experience on the use of open source verification tools.
0.20	14/03/2025	M. Roveri (UniTN)	Revised whole document to circulate for internal feedback.
0.21	24/03/2025	A. Tacchella, M. Roveri (UniTN)	Addressed feedback.
0.22	26/03/2025	J. Mihelič, Z. Putrle (BEYOND)	Addressed feedback.
0.23	27/03/2025	M. Roveri (UNITN)	Final version and approved, ready for quality control.
0.24	28/03/2025	Juan Andrés Alonso (ATOS)	Final version based on quality control.
1.0	31/03/2025	H. Koshutanski (ATOS)	Final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Marco Roveri	27/03/2025
Quality Manager	Juan Andrés Alonso (ATOS)	28/03/2025
Project Coordinator	H. Koshutanski (ATOS)	31/03/2025

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation / acronym	Description
CHERIoT	Capability Hardware Extension to RISC-V for Internet of Things
GWV	Greve, Wilding and Vanfleet
HW	Hardware
IoT	Internet of Things
IPC	Interrupt Process Control
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LIA	Linear Integer Arithmetic
MCU	Micro Controller Units
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
MPU	Microcontroller Processing Units
REE	Rich Execution Environment
RTOS	Real-Time Operating System
SMT	Satisfiability Modulo Theory
TCM	Trusted Computing Module
TEE	Trusted Execution Environment
VM	Virtual Machine

Executive Summary

This document is the final version of [1]. It provides the basis for a formal specification framework that can be used to specify the design and deployment of trusted Internet of Things (IoT) application. We present the final version of the definition of the concepts of safety and assurance for IoT applications and the results of some verification efforts for addressing some specific problems.

In this document, we take advantage of temporal logic specifications complemented with security concepts to prove some of the defined security properties. We rely on existing frameworks and verification tools (possibly adapted) to address the identified verification problems. We also present the final version of the CROSSCON security properties and the final formalization of the CROSSCON separation kernel that considers the dynamic creation of sub-domains as well as shared memory among domains. Moreover, we present manual proofs of some of the properties. Furthermore, we discuss and present formalization and automatic proofs for the case where we use TEE and virtualization-less architecture and how to prove the correctness of the CROSSCON configuration file. We also present the final version of the CROSSCON certification manifest and some effort in leveraging open source verification tools to prove the correctness of some components.

1 Introduction

This document provides the final results related to formalizing the CROSSCON stack. It summarises the results of Tasks 2.2 and 2.3, which aim to provide a comprehensive formal picture of the CROSSCON architecture.

In this document, we discuss in detail the security properties that we aim for and their formalization in the scope of the CROSSCON separation kernel. Moreover, we present results for the formalization and verification tasks and discuss the experience gained using open source verification tools for verifying digital hardware designs written in Verilog.

This document will be used as the basis for the formal technical work of the project.

1.1 Relation to Other Project Work

The document is closely related to the initial version of the CROSSCON Open Specification preliminary discussed in D2.1 [2] and finalized in D2.3 [3], the result of Task 2.1. In turn, it relates to the definitions of use cases in D1.1 [4] and the technical specification of the corresponding general requirements as documented in D1.2 [5].

The formal specification laid out in this document will benefit the work related to the CROSSCON stack, performed in WP3, and domain-specific hardware extensions, performed in WP4.

1.2 The CROSSCON high-level architecture

As thoroughly discussed in D2.1 [2] and finalised in D2.3 [3], the current abstract architecture at the highest level of CROSSCON can be represented graphically as shown in Figure 1.

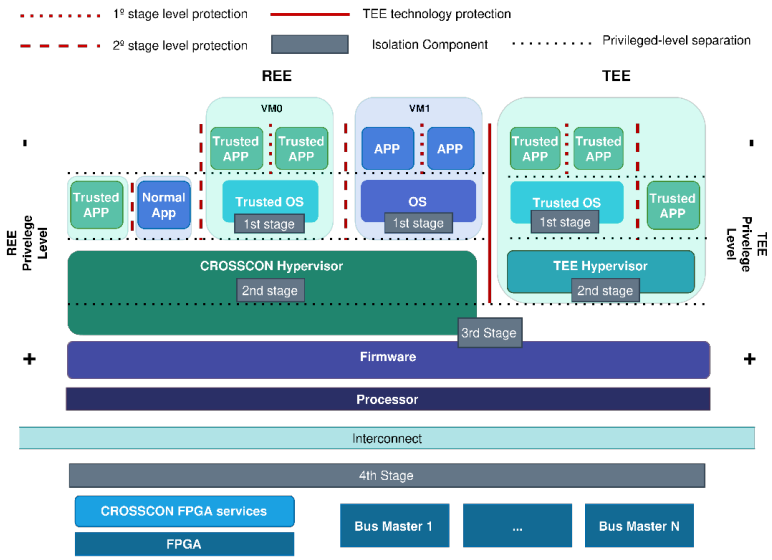

Figure 1: The CROSSCON high-level architecture.

As discussed in D2.1 [2] and in D2.3 [3], we distinguish two high-level cases: deployment in the case of i) a Rich Execution Environment (REE) and ii) a Trusted Execution Environment (TEE), where both REE and

TEE support several different privilege levels. In both cases, at the lower levels of the CROSSCON stack, we have an Instruction Set Architecture (ISA) that varies depending on the CPU architecture families (e.g., RISC-V, Arm with all their variants) considered, each equipped with different capabilities (e.g., the presence of a TEE, privilege levels). On top of the ISA, we consider a firmware layer, which might not be needed, depending on the CPU architecture and the needs of the upper layers of the stack.

1.3 Structure of the Document

This document is structured as follows. In Chapter 2, we discuss related works and present the final version of the security properties. In Chapter 3, we discuss the final version of the formalization of the CROSSCON separation kernel together with the proof of some security properties. In Chapter 4, we discuss a detailed formalization of the CROSSCON design for the TEE and virtualization-less case together with an automated proof of correctness. In Chapter 5, we discuss the formalization of the checks on the CROSSCON hypervisor configuration file to ensure that the properties and separations discussed in Chapter 3 are satisfied. Chapter 6 discusses the formalization and verification of the finite-state machine governing the dynamic creation of virtual machines. Chapter 7 discusses the experience of using open source formal verification tools to verify digital hardware designs written in Verilog. In Chapter 8, we discuss the structure of the certification manifest, in which we propose to include certifications on properties of the artifact to which the manifest will be associated. Finally, in Chapter 9, we conclude and outline future work.

1.3.1 Major changes w.r.t. D2.2

As mentioned above, this is the final version of the deliverable 2.2 [1]. We take several sections from the first version with minor modifications (this is the case for chapters 2, 3, 4).

The major changes correspond to new sections and/or chapters. In particular, we added Section 3.7, and Chapters 5, 6, 7, and 8.

2 CROSSCON Security properties

As the first step towards the formalization of the CROSSCON stack, we analyzed different specifications available in the literature to assess both the definition of isolation and the security properties they considered. In Section 2.1 we briefly summarize each of them, while in Section 2.2 we summarize what we are currently envisaging to adopt (we may revise them in the remaining time of the project).

2.1 Related works

The most related approaches to formalize isolation and to define security properties are:

- seL4 – The world's most highly assured operating system kernel,
- CHERIot – Capability Hardware Extension to RISC-V for Internet of Things (CHERIoT),
- GWV – Security model formalization by Greve, Wilding, Vanfleet,
- Noninterference security guarantee.

In the remainder of this section, we summarize their main characteristics and their definition of security properties.

2.1.1 Security properties of seL4

The seL4 is a general purpose operating system micro-kernel whose specification has been subject to machine-checked formal verification [6], and its implementation has proven to be functionally correct according to the specification. Among other things, it guarantees the three classical security properties: *confidentiality*, *integrity*, and *availability* which are roughly defined as follows:

Confidentiality: The seL4 will not allow an entity to read data without having been explicitly given read access to the data.

Integrity: The seL4 will not allow an entity to modify data without having been explicitly given write access to the data.

Availability: The seL4 will not allow an entity to prevent the authorized use of resources by another entity.

From a thorough reading of the documentation, we notice that the proofs for guaranteeing the above security properties do not consider temporal aspects. Thus, the above security properties are static and not associated with time. Moreover, all the proofs are based on the following assumptions [7]:

- The hardware operates as intended, meaning that we assume that it functions correctly by adhering to its specifications. In practical terms, this assumption implies that the hardware has not been tampered with or is operating within its designated conditions.
- The theorem prover is correct, i.e., we can trust the solver used to prove the properties, which is bug-free, and the proof rules used to prove the properties are correct.

There are also other assumptions, and we refer the reader to [7] for additional details.

2.1.2 CHERIoT security properties

Capability Hardware Extension to RISC-V for Internet of Things (CHERIoT) [8] is an Instruction Set Architecture and software model built on top of CHERI [9][1] and RISC-V to provide spatial memory safety, deterministic use-after-free protection, and lightweight compartmentalization exposed directly to the C/C++ language model.

CHERIoT can run existing embedded software components on a clean-slate Real-Time Operating System that scales up to large numbers of isolated (yet securely communicating) compartments. Thus, CHERIoT considers notions of memory safety. In its settings, a system is said to be memory safe if its references to memory are:

Unforgeable: A reference to memory (in particular, the authority to access memory) can be constructed only from other references.

Monotonic: A constructed reference will have no more authority than its progenitor reference (and may have less).

Spatially Safe: References to memory authorize access to a set of memory locations determined when the reference is constructed.

Temporally Safe: References to a region of memory will not remain usable across memory reuse for a different allocation.

CHERIoT leverages the notion of compartment: a collection of code, data, and capabilities that serves as a callable security context. Given these concepts, it defines two global security properties:

- No compartment should be able to access another compartment's data, except where explicitly shared.
- No thread should be able to access the data of another thread except where explicitly shared.

We remark that, although CHERI was also applied to other architectures, like for example MIPS and ARM, CHERIoT is tightly coupled to the RISC-V architecture, and its extension to other families is not trivial. Moreover, CHERIoT targets embedded devices that do not support virtualization of the memory in the classical sense; thus, CHERIoT devices do not support classical operating systems (e.g., Linux), while CHERI, differently from CHERIoT, considers virtualization of the memory. In the case of the CROSSCON stack, we aim to cover both cases (with classical memory virtualization and without).

2.1.3 GWV security properties

Greve, Wilding and Vanfleet (GWV) [10] proposed a security policy that defines a model of a separation kernel, which enforces partitioning between applications running on a single processor system. To this extent, GWV defines notions of partitions and segments and proposes three basic separation properties:

Non-exfiltration: This property indicates that an executing partition will not influence memory segments outside of its permitted set of segments.

Non-infiltration This property indicates that the execution partition can only use information from its permitted set of segments to affect its execution behavior.

Non-mediation: This property indicates that when a partition executes, the effect on a segment does not depend on anything other than the segment's original value and the values of the current partition.

[1] The RISC-V open source project https://riscv.org/.

GWV security properties are very general and different from the other two previously considered cases; they encompass time (execution).

2.1.4 Noninterference security property

Non-interference (sometimes also called non-inference) is a model of multiple levels of security proposed by Goguen and Meseguer [11]. It models a system composed of a number of users, inputs and outputs, and actions with a state-transition machine. The main motivation is a security policy represented by a relation between users specifying which information flows are permissible (usually from low-marked inputs to high-marked outputs) and which are not (from high-marked inputs to low-marked outputs). Therefore, non-interference is a property that restricts the information flow through the system. In this setting, the main verification effort is to show that high-marked inputs cannot interfere with low-marked outputs.

Non-interference: X is non-interfering with Y in a system M if X's input to M does not affect M's output to Y.

Confidentiality: An immediate consequence of non-interference is that the observations of Y are entirely independent of the actions of X. Therefore, the non-interference property also expresses the confidentiality guarantee of X': X cannot reveal any secrets to Y through M.

Integrity: Similarly, the consequence is that no information flows from X to Y through M. Therefore, it also expresses the guarantee of the integrity of Y': Y cannot be corrupted by X through M.

It seems intuitive that the information flow relation must be transitive; that is, if the flow from A is allowed and the flow from B to C is allowed, then the flow from A to C must also be allowed. However, this transitivity property then makes the ability of trusted users to downgrade the information (for example, from high to low mark) irrelevant, as transitivity implies that this always happens. As reasoning about what a user can do (e.g., downgrade) is useful, non-interference without the transitive property was also studied under the name of *intransitive non-interference* [12].

2.2 The CROSSCON proposal

In CROSSCON, we adopt a slight variant of the classical seL4 security properties: confidentiality, integrity, and availability, as these properties are generic enough to avoid being RISC-V specific (as in the case of CHERIoT) and at the same time are not too abstract (as in the case of GWV). In addition, confidentiality and integrity are both part of the CIA triad, which is an established model designed to guide policies for information security. This allows us to properly capture the architecture-independent spirit of the CROSSCON stack together with the necessary hardware features. Thus, similarly to the seL4 case, we will consider the following security properties.

Confidentiality: The CROSSCON stack will not allow an entity to read data without first giving the entity explicit permission to do so.

Integrity: The CROSSCON stack will not allow an entity to modify (write) data without explicitly having been given the entity write access to the data.

Availability: The CROSSCON stack will not allow an entity to prevent an authorized use of resources by another entity. This is similar to the case of seL4, which means that the different CROSSCON elements will eventually get the resources needed if authorized to get them.

Furthermore, we also adopt the non-interference property since it is suitable for establishing the security property of memory isolation and separation of domains. Informally, we state this property as follows.

Non-interference: The CROSSCON stack will provide domain separation in the sense that changes in the state of one domain do not affect the state of any other domain.

In the formalization (next chapter), we provide formal counterparts to the above properties. Similarly to the other approaches considered, we assume the following points.

- Unless strictly necessary, we keep temporal aspects out of the formal specification, thus considering static properties.
- We assume that the Hardware (HW) behaves correctly.
- We assume that there are no side-channel attacks. We may then reconsider this in future revisions of the formalization.
- We assume that the verification engine is correct, i.e., it does not allow one to conclude false results.

For the formalization, we will follow the architecture decomposition of the CROSSCON stack. For the verification of the security properties, we aim to adapt existing verification techniques such as model checking [13], theorem proving, for instance by means of Satisfiability Modulo Theory (SMT) solvers [14], and compositional reasoning [15], along the lines followed in the preliminary work on limited HW architectures [16].

3 The CROSSCON separation kernel formalization

In this chapter, we present our approach towards the formalization of a *CROSSSCON separation kernel*, which serves as a model for the CROSSCON hypervisor. We define fundamental notions and specify necessary assumptions about the separation kernel model. Within our formal setting, we also characterize and prove several security properties that are often found in the related literature.

3.1 Separation kernel

Conceptually, a *separation kernel* is the core component of a system that divides the system's resources into distinct *domains*, often also called *partitions* or *worlds*. Its main goal is to enforce separation between these domains, akin to the level of separation found in physically distributed systems.

In practice, a separation kernel comprises various hardware and/or software modules that are integrated into a hypervisor (i.e., virtual machine monitor) or a supervisor operating system implemented for a compatible computer architecture. Assuming that the architecture provides appropriate protection and isolation mechanisms, such as support for at least two processor privilege levels and a memory protection unit or memory management unit, we devise a separation kernel model to support the implementation of the separation kernel based on the hardware and software co-designed approach. However, completely software-based approaches based on code instrumentation and instruction emulation are also possible.

We conceptualize the separation kernel as a cohesive entity that delivers specific security guarantees. Our formalization of the separation kernel includes a model of a machine (e.g., a processor and memory management unit) that provides separated execution environments as well as assumptions about the behavior and operation of the model. Based on these, we also reason about the model's security-related properties for which we provide several guarantees.

The execution environment provided by the separation kernels consists of one or more *domains* isolated via *isolation and protection mechanisms*. We offer (and later in this document, we also formalize) the following two views on the separation provided by the separation kernel model:

- separation of the separation kernel from the domains, and
- separation of the domains from each other.

3.2 Machine state

We start our formalization with a definition of the state of a computer system, simply called a machine, comprising at least a main memory and a microprocessor. Modern microprocessors in a single integrated circuit often provide several separate processing units, called cores. Hence, we first define a *single-core state* representing microprocessors having one processing unit. Afterward, we expand this representation to *multiple-core state* to include processors with two or more processing units that share the main memory. Finally, we consider various modes and assumptions regarding how cores can be shared within security domains.

3.2.1 Single-core state

Consider a computing system consisting of a main memory and a single processing unit containing a set of general-purpose registers and several special-purpose ones. We use σ to denote the state of the machine and represent it with a quintuple

$$\sigma = (mem, reg, pc, pl, ad)$$

where the components of the state σ are as follows:

- A mapping $mem : A_M \rightarrow V_M$ representing a memory, for example, a full physical memory address space that a processor can address. Here, A_M is a set of all memory addresses, and V_M is a set of values that a memory location can contain.
- A mapping $reg : A_R \rightarrow V_R$ representing a set of registers. Here, A_R is a set of register labels, and V_R is a set of values that a register can hold.
- A variable $pc : A_M$, which represents the program counter register usually incorporated within a processor.
- A variable $pl : \{\mathrm{S}, \mathrm{U}\}$ that identifies the current privilege level where S indicates the privileged level (i.e., used for a separation kernel or operating system) and U indicates the unprivileged level (i.e., used for operating system or user applications).
- A variable $ad : \{1, \ldots, N\}$ that identifies the currently active domain from the set of domains. We consider that there are N domains.

Practical computer architectures may be very complex in organizing their memory and registers. Without delving into the details of these practicalities, we consider a simplified and unified view of memory and registers. In particular, we assume that the model uses W-bit memory addresses that may hold W-bit values, as well as processor registers containing W-bit words. We assume that there are L registers, i.e., $|A_R| = L$. Furthermore, we also assume that the sets $A_M, V_M, A_R, V_R \subseteq \mathbb{N}$ contain consecutive numbers starting from 0. For example, if we set $W = 32$ and $L = 16$, we obtain $A_M = V_M = V_R = \{0, \ldots, 2^{32} - 1\}$ and $A_R = \{0, \ldots, 15\}$. We formalize these conditions in the following assumptions.

Assumption 1. *The main memory consists of 2^W cells, where each cell contains a W-bit value, i.e.,*

$$A_M = V_M = \{0, \ldots, 2^W - 1\}.$$

Assumption 2. *The processor supports L general purpose registers, where each register contains a W-bit value, i.e.,*

$$A_R = \{0, \ldots, L - 1\} \quad \textit{and} \quad V_R = \{0, \ldots, 2^W - 1\}.$$

Hence, we have

Lemma 1. $A_M = V_M = V_R$.

For the simplicity of definitions and specifications of the model behavior and semantics of machine instructions, we also assume that W bits are enough for encoding machine instructions and representing their operands. In particular, all instructions (and their operands) are encoded with exactly W bits. As a consequence, incrementing the program counter register (which points to the current machine instruction) changes it to point to the succeeding instruction.

Assumption 3. *Machine instructions and corresponding operands are uniformly encoded and represented with exactly W bits.*

Dynamic core assignment. The component ad of the state σ, i.e., $\sigma.ad$, specifies the active domain. Depending on this setting, the separation kernel must configure the corresponding isolation and protection mechanisms. For a single-core machine running a separation kernel providing multiple domains,

a domain-switching procedure must also be implemented. As discussed later in this document, this can be modeled with special instructions that perform the re-configuration and set the active domain component ad to the proper value.

3.2.2 Multi-core state

Practical multi-core systems share the main memory between instruction processing units, but they have their own set of registers. It is straightforward to expand the single-core state to model the multi-core processors by duplicating certain parts of the original state.

Consider a processor with $P \in \mathbb{N}$, where $P \geq 1$, cores, then the multi-core state σ_P is defined as

$$\sigma_P = (mem, \sigma'_1, \ldots, \sigma'_P) \quad \text{where} \quad \sigma'_i = (reg_i, pc_i, pl_i, ad_i).$$

Here, σ_i is a sub-state of the i-th core while the components are defined accordingly, as in the case of the single-core representation. The sub-state σ_i represents the i-th core registers (e.g., general-purpose and specific registers) but excludes the representation of the main memory, which is shared among all the cores.

Observe that any code is always executed in the context of a particular core. Hence, besides the shared main memory, the executed instruction can only alter the core-dependent sub-state. For this purpose, we define the i-th projection f_i of the multi-core state σ_P into the i-th single-core state σ_i, i.e.,

$$\sigma_i = f_i(\sigma_P) = (mem, \sigma'_i).$$

Thus, the projected state includes only the components that are relevant to the execution of the instruction.

Static and dynamic core assignment. Now, consider a multi-core machine where the component ad of the i-th sub-state σ_i, i.e., $\sigma_i.ad$, specifies the active domain for the i-th core. The inverse information of that represented by $\sigma_i.ad$, is to obtain all the cores assigned to a given domain $d \in \{1, \ldots, N\}$. For this purpose, we define a function

$$\text{cores} : \{1, \ldots, N\} \to 2^{\{1,\ldots,P\}} \quad \text{as} \quad \text{cores}(d) = \{i \mid \sigma_i.ad = d\}.$$

Our first assumption related to core assignment states that no two domains share the same core. Notice that a domain can use several cores.

Assumption 4.

$$\text{cores}(i) \cap \text{cores}(j) = \emptyset \quad \text{for all} \quad 1 \leq i, j \leq N, i \neq j.$$

Static core assignment refers to the assumption that the assignment of cores does not change with time.

Assumption 5. *In static core assignment, the component $\sigma_i.ad$ is fixed throughout execution for all $i \in \{1, \ldots, N\}$.*

The dynamic core assignment allows for a change of the active domain if the no-sharing assumption remains satisfied.

Assumption 6. *In the dynamic core assignment, the component $\sigma_i.ad$ can change throughout execution for all $i \in \{1, \ldots, N\}$ due to the constraint that Assumption 4 remains satisfied.*

3.2.3 State notation

Observe that there is intentionally not much difference in the definition of the single-core state σ and particular multi-core sub-states σ_i, where $1 \leq i \leq P$. As many of the definitions and results discussed

in this document apply equally well to all of them, we use the notation σ to denote any of them in the rest of the text. Moreover, we often omit σ as well and only use the state component names.

In particular, we write only mem to denote $\sigma.mem$, reg to denote $\sigma.reg$, pc to denote $\sigma.pc$, pl to denote $\sigma.pl$, and ad to denote $\sigma.ad$.

3.2.4 Configuration

In the previous deliverable D2.2 [1], we have assumed that the configuration of the separation kernel is implicitly contained within a state. In particular, memory configuration and partitioning were described with additional assumptions considering the static setting, where the configuration cannot change with time (i.e., during the execution of the separation kernel).

Here, we extend the state formalization with the corresponding formalization of the configuration. We define it using a function

$$\Sigma : \{1, \ldots, N\} \mapsto \mathcal{P}(A_M) \times \mathcal{P}(A_M) \times \mathcal{P}(A_M) \times \mathcal{P}(A_M)$$

Using the notation from the previous section, we are also able to define it as a function

$$\Sigma(d) = (A_d, C_d, D_d, E_d).$$

Here, the components specify address regions for the domain identified with d. In particular, for a domain d, the sets A_d, C_d, D_d, E_d are the full address space, the code memory segment, the data memory segment, and the memory-mapped devices segment, respectively.

Not every configuration is a valid one. The main configuration constraints related to the domain memory protection are defined later with Assumption 7.

3.3 Memory protection

In this section, we discuss the assumptions and constraints we envisage for the CROSSCON separation kernel by defining the memory layout partitioning, the memory-mapped I/O sub-regions, and the concept of an effective domain.

3.3.1 Memory partitioning

Each domain, as well as the separation kernel, must have its own memory region to function properly. Let A_d represent the addresses occupied by the memory region assigned to the domain d, where $1 \leq d \leq N$, and let A_S represent the addresses occupied by the memory region assigned to the separation kernel. We often denote $A_0 = A_S$ to simplify notation in formulas.

Each address region A_d, where $0 \leq d \leq N$, consists of the following two sub-regions:

- a *code* sub-region $C_d \subseteq A_d$ containing the instructions that can be executed;
- a *data* sub-region $D_d \subseteq A_d$ containing data that can be read or written to.

Assumption 7. *Separation of memory regions:*

- $A_d \subseteq A_M$ *for all* $0 \leq d \leq N$ *(the main regions).*
- $A_i \cap A_j = \emptyset$ *for all* $0 \leq i < j \leq N$ *(the main regions are disjoint).*
- $C_d \subseteq A_d$ *and* $D_d \subseteq A_d$ *for all* $0 \leq d \leq N$ *(code and data regions are sub-regions).*
- $C_d \cup D_d = A_d$ *for all* $0 \leq d \leq N$ *(the code and data sub-regions fully cover the domain sub-region).*

- $C_d \cap D_d = \emptyset$ *for all* $0 \leq d \leq N$ *(code and data sub-regions are disjoint).*

An example memory map that satisfies the above assumptions is depicted in Figure 2.

Figure 2: An example of a memory map.

From these assumptions follow the observations contained in the next lemma regarding the disjointness of the specific sub-regions.

Lemma 2. *For all* $0 \leq i \neq j \leq N$*, we have*

- $C_i \cap C_j = \emptyset$
- $D_i \cap D_j = \emptyset$
- $C_i \cap D_j = \emptyset$

Proof. For example, to see $C_i \cap C_j = \emptyset$, observe that $C_i \subseteq A_i$ and $D_j \subseteq A_j$. Then the disjointness follows from $A_i \cap A_j = \emptyset$. □

3.3.2 Memory mapped I/O sub-region

When a domain requires access to an I/O device, we employ a memory-mapped I/O approach, where device registers are mapped to specific locations of the main memory. In our model, these locations are represented by a sub-region $E_d \subseteq A_d$. In particular, we also assume that it is part of the data sub-region, i.e., $E_d \subseteq D_d$. The rationale for this is that the hypervisor under consideration only supports pass-through access to I/O devices with static assignment of the I/O regions.

Assumption 8. *Statically assigned memory-mapped I/O, that is,*

$$E_d \subseteq D_d \quad \textit{for all domains } d \textit{ where } \quad 1 \leq d \leq N.$$

In order to obtain security-related guarantees, we have several options related to the E_d regions:

- $E_d = \emptyset$ for all $0 \leq d \leq N$.
- We trust the I/O system and statically assign I/O regions (as explained above).
- We related to WP4 and the I/O protection mechanisms developed there.

In this document, we mostly reason based on the first option. However, our results also apply to the other two options if there is no information flow between different I/O regions. The non-interference property is useful here. In particular, for $1 \leq i, j \leq N$, we say that E_i is non-interfering with E_j if an alteration of locations A_i does not affect any location A_j.

3.3.3 Effective domain

The separation kernel provides domain separation by controlling memory access by proper configuration of the memory management unit. Exactly one domain is active at a time and only its memory and the memory of the kernel can be accessed. Let $d = ad$ be the active domain identifier. Then, only the memory region A_0 of the separation kernel and the memory region A_d of the domain identified by d are accessible. Furthermore, access rules for these two regions depend on the current privilege level stored in pl.

In the case of a violation of memory protection, that is, when a location outside these two regions is accessed, the computation halts. In practice, an exception in the processor may be triggered. However, there are also other conditions that may cause the computation to halt. We describe the details of the memory protection in the following text.

To specify instructions that our model is able to provide, we consider three kinds of memory accesses:

- R – read access: an instruction reads data from the memory,
- W – write access: an instruction writes data to the memory, and
- X – fetch access: the instruction pointed to by the $\sigma.pc$ register is read from the memory.

Whether a particular memory access is successful also depends, in addition to the kind of access, on the current privilege level, i.e., $\sigma.pl$. At any moment, only the memory regions of the separation kernel and the active domain identified with $\sigma.ad$ are accessible according to the following access rules (see also Figure 3):

Unprivileged level: Applicable when $pl = \mathsf{U}$. Consider $d = ad$.

- R – read: Data may be loaded from regions C_d or D_d.
- W – write: Data may be stored in the region D_d.
- X – execute: Instructions may be fetched from the region C_d.

Privileged level: Applicable when $pl = \mathsf{S}$.

- R – read: Data may be loaded from regions C_S or D_S.
- W – write: Data may be stored in the region D_S.
- X – execute: Instructions may be fetched from the region C_S.

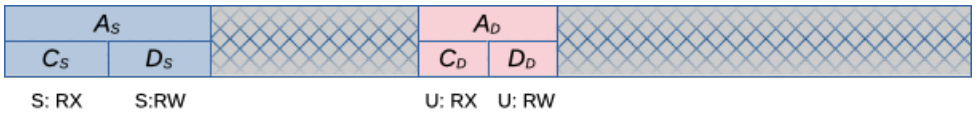

Figure 3: The protection provided by separation kernel.

However, notice that in some systems, the privileged level, that is, $pl = \mathsf{S}$, may also allow access to the corresponding regions of the active domain. However, we assume that the CROSSCON Hypervisor never allows or performs such access.

Assumption 9. *When $pl = \mathsf{S}$, the memory access outside of the A_S region is never performed or halts the computation.*

To formalize the above access rules, we first introduce three auxiliary functions: canr(a), canw(a), and canx(a) for read, write, and fetch access, respectively, to an address a, i.e.,

- $\text{canr}(a) \equiv pl = \mathsf{S} \wedge a \in A_S \vee pl = \mathsf{U} \wedge a \in A_{ad}$,
- $\text{canw}(a) \equiv pl = \mathsf{S} \wedge a \in D_S \vee pl = \mathsf{U} \wedge a \in D_{ad}$,
- $\text{canx}(a) \equiv pl = \mathsf{S} \wedge a \in C_S \vee pl = \mathsf{U} \wedge a \in C_{ad}$.

Observe, that σ argument is implicit for all three functions.

Assumption 10. *Protection mechanism guarantees. An instruction*

- *can read the data stored at the address $a \in A_M$, if and only if* canr(a) *is true,*
- *can write the data to the address $a \in A_M$, if and only if* canw(a) *is true,*
- *can be fetched/executed from the address $a \in A_M$, if and only if* canx(a) *is true.*

When a violation of these rules is detected, an exception (i.e., protection interrupt) is thrown.

3.4 Unprivileged instruction semantics

In this section, we consider several exemplary instructions representing arithmetic and logic operations, load and store operations, and control-flow operations. We use $\mathsf{I}(a)$ to indicate an instruction that requires access to the address a (for reading or writing data or fetching the instruction). These instructions are sufficient to model any possible complex instruction. In our discussion, we follow the RISC (reduced instruction set computer) computer architecture paradigm.

Note that in all our definitions of instructions, we assume that at the end of the instruction execution, the program counter register is also incremented, i.e., $pc \leftarrow pc + 1.$, unless pc is explicitly set for that part of the execution (relevant only for definitions of control-flow operations).

3.4.1 Memory access obligations

Here, we present a framework for defining the semantics of machine instructions. To do this, we specify a function $[\![\cdot]\!] : \sigma \rightarrow \sigma$, which maps a state to a state (that is, it specifies how the instruction modifies the state to which it is applied). In definitions, we omit σ as a function argument (which is implicitly used). We also assume $r \in A_R$ and $a \in A_M$. However, we do not explicitly give all the details for all the instructions, but we propose only guidelines and obligations for the definitions.

For the semantics to be aligned with the separation kernel protection mechanism discussed above, several obligations must be respected in the definition of the instructions. In this sense, the following assumption is a consequence of Assumption 10.

Assumption 11. *The semantics of each instruction* $\mathsf{I}(a)$ *must satisfy the following obligations when the memory address* $a \in A_M$ *is accessed:*

- *Read obligation: Any read from the address* a*, i.e., use of the value* $mem[a]$*, is bound to the* $\mathsf{canr}(a)$ *predicate:*
$$\neg\mathsf{canr}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$
- *Write obligation: Any write to the address* a*, i.e., writing to the value* $mem[a]$*, is bound to the* $\mathsf{canw}(a)$ *predicate:*
$$\neg\mathsf{canw}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$
- *Fetch obligation: A fetch of the instruction* $\mathsf{I}(a)$ *from the address* a*, i.e., fetching from the* $mem[a]$*, is bound to the* $\mathsf{canx}(a)$ *predicate:*
$$\neg\mathsf{canx}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$

3.4.2 Arithmetic and logic operations

We present several examples of unary and binary arithmetic and logic instructions following the RISC (reduced instruction set computer) computer architecture paradigm. We start with an example of unary instruction.

$$[\![\mathsf{NOT}\ r]\!] \equiv \begin{cases} reg[r] \leftarrow \sim reg[r] & \mathsf{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

Note that, $\sim$ denotes the operation of taking a binary representation of the operand and negating its bits (one by one).

We continue with an example of binary instruction

$$[\![\text{ADD } r_d, r_s]\!] \equiv \begin{cases} reg[r_d] \leftarrow reg[r_d] + reg[r_s] & \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

3.4.3 Load/store operations

Now, we consider LOAD r, a, which loads into the register r the content of the memory at address a, and is defined as follows.

$$[\![\text{LOAD } r, a]\!] \equiv \begin{cases} reg[r] \leftarrow mem[a] & \text{canx}(pc) \wedge \text{canr}(a) \\ \bot & \text{otherwise} \end{cases}$$

It is easy to check that LOAD r, a satisfies all three memory access obligations (Assumption 11):

- It is fetched from the program counter address. Therefore, canx(pc) appears in the condition. If canx(pc) is false, the semantic function evaluates to $\bot$.
- It reads from the memory address a. Therefore, canr(a) protects read access, and if canr(a) is false, the semantic function evaluates to $\bot$.
- In does not write to the memory. Therefore, canw(a) is not used.

We also consider STORE r, a, which writes in memory at address a the content of the register r, and is defined as follows.

$$[\![\text{STORE } r, a]\!] \equiv \begin{cases} mem[a] \leftarrow reg[r] & \text{canx}(pc) \wedge \text{canw}(a) \\ \bot & \text{otherwise} \end{cases}$$

Other more complex instructions can be modeled similarly by considering simpler instructions. Their semantics must be defined to satisfy the Assumption 11. For an example, consider an instruction LOAD.REL r_d, r_s, a that loads into the register r_d the content of the memory at the location specified by $reg[r_s] + a$, where r_s is a register and a is a memory address.

$$[\![\text{LOAD.REL } r_d, r_s, a]\!] \equiv \begin{cases} reg[r_d] \leftarrow mem[reg[r_s] + a] & \text{canx}(pc) \wedge \text{canr}(reg[r_s] + a) \\ \bot & \text{otherwise} \end{cases}$$

3.4.4 Control-flow operations

Here, we consider control-flow operations. We start with JUMP a, which modifies the program counter in order to continue the execution at the value specified at address a[1].

$$[\![\text{JUMP } a]\!] \equiv \begin{cases} pc \leftarrow a & \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

We continue with the BRANCH.ZERO r, a, which conditionally sets the program counter to the address a if the content of a given register r equals zero, otherwise the execution continues regularly with the next instruction.

$$[\![\text{BRANCH.ZERO } r, a]\!] \equiv \begin{cases} pc \leftarrow a & \text{canx}(pc) \wedge reg[r] = 0 \\ \text{no operation} & \text{canx}(pc) \wedge reg[r] \neq 0 \\ \bot & \text{otherwise} \end{cases}$$

[1] Without loss of generality, a can be either a static address, and in this case, we use the value directly, or a register, and in this case, we use the value stored in the register.

3.5 Memory security

In this section, we analyze the proposed separation kernel model in relation to various security properties of accessing the memory.

3.5.1 Validity of access

Consider an instruction $\mathsf{I}(a)$ that accesses the memory at an address $a \in A_M$. The type of access can be any of read/write/fetch.

3.5.1.1 Invalid access

If $a \in A_M$ falls outside the allowed memory regions (i.e., the separation kernel or domain memory region), then due to protection, the execution of the instruction will cause an exception. We state this formally with the following lemma.

Lemma 3. *Consider an instruction* $\mathsf{I}(a)$ *that accesses (read/write/fetch) the memory at the location* $a \in A_M$. *If* $a \notin A_S \cup A_{ad}$, *then* $[\![\mathsf{I}(a)]\!] = \bot$.

Proof. For read, write, and fetch access $\mathsf{canr}(a)$, $\mathsf{canw}(a)$, and $\mathsf{canx}(a)$ must hold, respectively. Consider the former. By definition of $\mathsf{canr}(a)$, we have that $pl = \mathsf{S} \wedge a \in A_S \vee pl = \mathsf{U} \wedge a \in A_{ad}$, which is definitely not true if $a \notin A_S \cup A_{ad}$. Similarly, we can conclude the same result for write and fetch access. □

3.5.1.2 Valid access

Since the addresses $a \notin A_S \cup A_{ad}$ are invalid, let us focus on the addresses $a \in A_S \cup A_{ad}$ and see if they are all valid. We consider $a \in C_S \cup D_S \cup C_{ad} \cup D_{ad}$, thus having a more granular view of the memory regions with respect to the data/code section and the separation kernel/domain region.

An overview of the results is shown in Table 1, where a particular cell shows a privilege level when an instruction successfully accesses (read/write/execute access) the memory at address a. If the cell contains $\bot$, then the access is invalid, and $[\![\mathsf{I}(a)]\!] = \bot$. The main two rows, denoted by SK and DOM, contain the rules for the regions of the separation kernel and the effective domain, respectively.

Table 1: Valid memory accesses.

			read	write	fetch	
	$a \in$		canr(a)	canw(a)	canx(a)	$pc \in$
SK	A_S	C_S	$pl = \mathsf{S}$	$\bot$	$pl = \mathsf{S}$	C_S
		D_S	$pl = \mathsf{S}$	$pl = \mathsf{S}$	$\bot$	
DOM	A_{ad}	C_{ad}	$pl = \mathsf{U}$	$\bot$	$pl = \mathsf{U}$	C_{ad}
		D_{ad}	$pl = \mathsf{U}$	$pl = \mathsf{U}$	$\bot$	

For example, if an instruction makes a write access to the main memory at address $a \in D_S \subseteq A_S$, i.e., the separation kernel data region, we check the corresponding cell and see that $pl = \mathsf{S}$ must hold for the access to be allowed. Otherwise, the computation would halt. In the following sections, we explain the details of Table 1 and prove the particular security results.

3.5.2 Access type

Consider a state σ and an instruction $\mathsf{I}(a)$ that is successfully executed, that is, $[\![\mathsf{I}(a)]\!] \neq \bot$, in the context of the state σ. In what follows, we examine such instructions with regard to read, write, and fetch memory access where the address is part of the separation kernel or active domain memory region, that is, $a \in A_S \cup A_{ad}$.

3.5.2.1 Read access

Lemma 4. *When an instruction* $\mathsf{I}(a)$ *executes successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \bot$*, and reads from the memory at address* $a \in A_S \cup A_{ad}$*, we have*

$$a \in A_S \iff pl = \mathsf{S} \quad \text{and} \quad a \in A_{ad} \iff pl = \mathsf{U}.$$

Proof. From $[\![\mathsf{I}(a)]\!] \neq \bot$ and the read obligation, i.e., $\neg\mathrm{canr}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot$ we get that $\mathrm{canr}(a)$ holds. From $a \in A_S \cup A_{ad}$ and $A_S \cap A_{ad} = \emptyset$ (disjointness), we have two cases: either $a \in A_S$ or $a \in A_{ad}$. By the definition of $\mathrm{canr}(a)$ and disjointness, either the left or right term of

$$pl = \mathsf{U} \wedge a \in A_{ad} \quad \vee \quad pl = \mathsf{S} \wedge a \in A_S$$

holds.

Now, if $a \in A_S$, then $a \notin A_{ad}$, and hence $pl = \mathsf{S}$. And vice versa, if $pl = \mathsf{S}$ then $pl \neq \mathsf{U}$, and hence $a \in A_S$. Similarly, if $a \in A_{ad}$ then $a \notin A_S$, and hence $pl = \mathsf{U}$. And vice versa, if $pl = \mathsf{U}$ then $pl \neq \mathsf{S}$, and hence $a \in A_{ad}$. □

3.5.2.2 Write access

Lemma 5. *When an instruction* $\mathsf{I}(a)$ *executes successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \bot$*, and writes to the memory at address* $a \in A_S \cup A_{ad}$*, we have*

$$a \in D_S \iff pl = \mathsf{S} \quad \text{and} \quad a \in D_{ad} \iff pl = \mathsf{U}.$$

Proof. From $[\![\mathsf{I}(a)]\!] \neq \bot$ and the write obligation, i.e., $\neg\mathrm{canw}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot$ we get that $\mathrm{canw}(a)$ holds. From $a \in D_S \cup D_{ad}$ and $D_S \cap D_{ad} = \emptyset$ (disjointness), we have two cases: either $a \in D_S$ or $a \in D_{ad}$. By the definition of $\mathrm{canw}(a)$ and disjointness, either the left or the right term of

$$pl = \mathsf{U} \wedge a \in D_{ad} \vee pl = \mathsf{S} \wedge a \in D_S$$

holds.

Now, if $a \in D_S$ then $a \notin D_{ad}$, and hence $pl = \mathsf{S}$. And vice versa, if $pl = \mathsf{S}$ then $pl \neq \mathsf{U}$, and hence $a \in D_S$. Similarly, if $a \in D_{ad}$, then $a \notin D_S$, and hence $pl = \mathsf{U}$. And vice versa, if $pl = \mathsf{U}$ then $pl \neq \mathsf{S}$, and hence $a \in D_{ad}$. □

3.5.2.3 Fetch access

Lemma 6. *When an instruction* $\mathsf{I}(a)$ *is successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \bot$*, fetched from the memory at address* $a \in A_S \cup A_{ad}$*, we have*

$$a \in C_S \iff pl = \mathsf{S} \quad \text{and} \quad a \in C_{ad} \iff pl = \mathsf{U}.$$

Proof. From $[\![\mathsf{I}(a)]\!] \neq \bot$ and the fetch obligation, i.e., $\neg\mathrm{canx}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot$ we get that $\mathrm{canx}(a)$ holds. From $a \in C_S \cup C_{ad}$ and $C_S \cap C_{ad} = \emptyset$ (disjointness), we have two cases: either $a \in C_S$ or $a \in C_{ad}$. By the definition of $\mathrm{canx}(a)$ and disjointness, the left or right term of

$$pl = \mathsf{U} \wedge a \in C_{ad} \vee pl = \mathsf{S} \wedge a \in C_S$$

holds.

Now, if $a \in C_S$, then $a \notin C_{ad}$, and hence $pl = \mathsf{S}$. And vice versa, if $pl = \mathsf{S}$ then $pl \neq \mathsf{U}$, and hence $a \in C_S$. Similarly, if $a \in C_{ad}$ then $a \notin C_S$, and hence $pl = \mathsf{U}$. And vice versa, if $pl = \mathsf{U}$ then $pl \neq \mathsf{S}$, and hence $a \in C_{ad}$. □

Corollary 7. *If $a = pc$ then we have*

$$pc \in C_S \iff pl = \mathsf{S} \quad \text{and} \quad pc \in C_{ad} \iff pl = \mathsf{U}.$$

3.5.3 Security properties

In general, an access policy specifies subjects that are authorized to perform specific operations on particular objects. Concerning the memory access policy, our model specifies the following refinements:

- The subjects are represented by instructions executed in one of the two privilege levels, i.e., $pl \in \{\mathsf{U}, \mathsf{S}\}$.
- The operations are of three types, that is, read, write, and fetch operations.
- The objects are represented by the data and code regions of the separation kernel and domains.

3.5.3.1 Integrity

Integrity often relates to a security guarantee of access to objects while also allowing them to mutate, that is, *an object cannot be altered by non-authorized subjects*, or, in other words, *a subject must be authorized to alter an object*.

Taking into account our model, the access that mutates the memory is monitored through the write operation and, respectively, the canw(a) function. Therefore, the access policy related to the integrity guarantee can be verified by observing the corresponding column in Table 1. In particular, the guarantees are as follows.

Separation kernel data integrity: Separation kernel data memory can only be written by instructions executed at the privileged level. To see this, observe the corresponding cell in Table 1, which contains $pl = \mathsf{S}$ for the address $a \in D_S$.

Domain data integrity: Domain data memory can only be written by instructions executed at the unprivileged level. Observe the corresponding cell in Table 1 that contains $pl = \mathsf{U}$ for $a \in D_{ad}$.

Separation kernel code integrity: Separation kernel code cannot be altered. The corresponding cell in Table 1 contains $\perp$ for $a \in C_S$, which means that canw(a) cannot be satisfied.

Domain code integrity: The domain code cannot be altered. The corresponding cell in Table 1 contains $\perp$ for $a \in C_{ad}$, which means that canw(a) cannot be satisfied.

Inaccessible region integrity: All other write accesses result in an exception. For all other addresses $a \in A_M$, by Lemma 3, any instruction semantics equals $\perp$.

3.5.3.2 Confidentiality

Confidentiality often refers to a security guarantee of access to objects in order to learn the data, i.e., *an object cannot be read by non-authorized subjects*, or, in other words, *a subject must be authorized to read an object*.

Taking into account our model, the read access to the memory is monitored through read and fetch operations, respectively, the functions canr(a) and canx(a). Confidentiality restrictions are specified in the following access policies.

- The Separation Kernel (Data and Code) Memory Region can be read by instructions executed at the privileged level.
- Domain (data and code) memory region can be read by instructions executed at the unprivileged level.
- An instruction fetch from the separation kernel code memory region is allowed at the privileged level.
- An instruction fetch from the domain code memory region is allowed at the unprivileged level.
- All other read/write/fetch accesses result in an exception.

To argue for confidentiality, we can now observe the read and fetch columns of Table 1.

Separation kernel data confidentiality: Separation kernel data memory can be read from at the privileged level. The row $a \in D_S$ contains $pl = \mathsf{S}$ in the read column.

Domain data confidentiality: Domain data memory can be read from at the privileged level. The row for $a \in D_S$ contains $pl = \mathsf{U}$ in the read column.

Separation kernel code confidentiality: Separation kernel code memory can be read and fetched from the privileged level. The row for $a \in C_S$ contains $pl = \mathsf{S}$ in the read and fetch columns.

Domain code confidentiality: Domain code memory can be read and fetched from the unprivileged level. The row for $a \in C_{ad}$ contains $pl = \mathsf{U}$ in the read and fetch column.

From Table 1 (and from the definitions of canr(), canw() and canx()) we can also observe that in our model the notion of confidentiality is more permissive in relation to integrity. In other words, the predicates canw() and also canx() represent a stricter form of access rights than canr(). We state this more formally with the following corollary.
Corollary 8. *For all $a \in A_M$ we have*

$$\mathrm{canw}(a) \Rightarrow \mathrm{canr}(a) \quad \text{and} \quad \mathrm{canx}(a) \Rightarrow \mathrm{canr}(a).$$

3.5.3.3 Separation

First, we consider the separation kernel and show that its memory region is always accessed by a code running at the privileged level.
Corollary 9. *Consider an instruction* I(a) *that is fetched from the pc address and that performs read or write access to the address $a \in A_S$. We have*

$$a \in A_S \iff pl = \mathsf{S} \iff pc \in C_S\,.$$

Proof. Use Lemmas 4 and 5 together with Corollary 7. □

Next, we consider the active domain and show the same property in a similar way.
Corollary 10. *Consider an instruction* I(a) *that is fetched from the pc address and that performs read or write access to the address $a \in A_{ad}$. We have*

$$a \in A_{ad} \iff pl = \mathsf{U} \iff pc \in C_{ad}.$$

Finally, consider the memory region, which is neither part of the separation kernel region nor the active domain region.

Corollary 11. *Consider an instruction* $\mathsf{I}(a)$ *that is fetched from the* pc *address and that performs read, write, or fetch access to the address* $a \in A_i$*, where* $i \neq 0$ *and* $i \neq ad$*, then* $\mathsf{I}(a) = \bot$.

Proof. Consider a proof for read access. Since $i \neq 0$ and $i \neq ad$, $\mathsf{canr}(a)$ does not hold. Hence, by the read obligation, $\mathsf{I}(a) = \bot$. Similar reasoning can be used to prove the property for the write or fetch instruction. □

3.5.3.4 Availability

Availability is achieved by construction in the defined CROSSCON separation kernel. Indeed, if the configuration of the memory layout is performed correctly (i.e., satisfying the considered assumptions), each domain has access to its own resources (e.g., memory, I/O), so it turns out to be impossible for a domain to prevent another domain to get its own resources.

3.6 Context switching

In this section, we discuss how to formalize context switching within the defined CROSSCON separation kernel. In particular, we analyze the safe storage of domain execution contexts and how to deal with interrupt handling.

3.6.1 Safe domain execution context storage

First, let us introduce the concept of safe storage, which is used to store and retrieve the execution context of domains. The storage is safe in the sense that ordinary unprivileged instructions cannot access it, but it can be manipulated only with specially designated instructions. The storage may be part of the state σ, but it is hidden and can be manipulated only through the auxiliary functions defined below. Every domain has its own storage with the capacity to store at least one execution context.

For manipulating the execution context, we consider two auxiliary functions with functionality:

$$\mathsf{pushContext} : \sigma \to \sigma \quad \text{and} \quad \mathsf{popContext} : \sigma \to \sigma.$$

Here, the former stores the execution context of the active domain in its respective safe storage, while the latter retrieves it. Both functions operate according to the LIFO (last-in, first-out) principle, as it is commonly found in the stack abstract data structure.

Assumption 12. *Invariance of the push/pop functions. Let* X *be a property of state* σ*, defined as a component* reg*,* pc*,* pl*, or* ad*, but not* mem*. We assume*

$$\mathsf{popContext}(\mathsf{pushContext}(\sigma)).X = \sigma.X$$

3.6.2 Interrupt handling

In the following, we analyse the assumptions and the semantics of interrupts and interrupt handling in the defined CROSSCON separation kernel. We consider a scenario of vector interrupts in which several numbered interrupts are possible.

Consider an interrupt i, where $i \in \mathbf{N}$ (e.g., $0 \leq i \leq 255$), which is handled by the i-th interrupt handler (i.e., interrupt service routine). Let $intentry_i \in A_M$ be the entry point (i.e., the memory address) of the i-th interrupt handler.

In real systems, an interrupt may arrive anytime and is usually handled immediately after the current instruction finishes. In order to formalize interrupt handling, we consider two new instructions. The first

is the IREQ i instruction, which starts with handling of the i-th interrupt and executing the corresponding interrupt handler at $intentry_i$ address. Its semantics is defined as follows.

$$[\![\text{IREQ } i]\!] \equiv \begin{cases} \text{pushContext}(\sigma) \\ pl \leftarrow \text{S} \\ pc \leftarrow intentry_i \end{cases}$$

It safely stores the current domain execution context, then it switches to the privileged level and transfers control to the i-th interrupt handler.

The second is the IRET instruction, which ends the current interrupt handler and returns control to the original code that triggered the interrupt.

$$[\![\text{IRET}]\!] \equiv \begin{cases} \text{popContext}(\sigma) & pl = \text{S} \wedge \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

The instruction IREQ i is virtual in the sense that it can be triggered (i.e., arbitrarily inserted) after any other instruction (with some exceptions stated below). The instruction IRET is coded explicitly; every interrupt handler contains at least one.

Assumption 13. *Correctness of interrupt handlers.*

- *Interrupt handlers are part of the separation kernel code:* $intentry_i \in C_S$.
- *The interrupt handler finishes with and returns via the explicit* IRET instruction.

For the simplicity of separation kernel implementation, one may also assume non-reentrant interrupts.

Assumption 14. *Non-reentrant interrupt handlers.*

- *During the execution of the interrupt handler code, the interrupts are disabled. We assume no* IREQ i *is inserted within the corresponding code.*
- *In practice, interrupts are usually disabled and enabled via hardware support. One may assume that* IREQ i *disables interrupts and* IRET enables them.

3.6.3 Software interrupts

In some systems, interrupts may also be triggered through code. We model this by the following instruction.

$$[\![\text{INT } i]\!] \equiv \begin{cases} [\![\text{IREQ } i]\!] & \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

3.6.4 Separation kernel calls

Here, we consider two instructions. Namely, SCALL to call the separation kernel and SRET to return from the separation kernel. We also introduce a new register called sl (separation kernel link register).

Let $sentry \in A_M$ be the address of an entry point for the separation kernel that is, the address of the routine that handles calls to the separation kernel.

Assumption 15. *Correctness of kernel handlers.*

- *Kernel handlers are part of the separation kernel code:* $intentry_i \in C_S$.
- *The kernel handler finishes with and returns via the explicit* SRET instruction.

To enter the separation kernel one can use:

$$[\![\text{SCALL}]\!] \equiv \begin{cases} \left.\begin{array}{l} sl \leftarrow pc \\ pl \leftarrow S \\ pc \leftarrow sentry \end{array}\right\} & pl = U \wedge \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

To exit the separation kernel and return to the caller, one can use SRET.

$$[\![\text{SRET}]\!] \equiv \begin{cases} \left.\begin{array}{l} pl \leftarrow U \\ pc \leftarrow sl \end{array}\right\} & pl = S \wedge \text{canx}(pc) \\ \bot & \text{otherwise} \end{cases}$$

The corresponding handler may be non-reentrant (see also Assumption 14). It can only be called from the unprivileged mode and cannot be called again from the privileged mode. Thus, we must ensure that the privileged code does not execute SCALL (maybe by inspecting the kernel separation code or by a trap mechanism provided by a particular architecture). We must also ensure that SRET triggers an exception if called from an unprivileged code.

3.6.5 Domain switching

Making a transition from one domain to another requires privileged instruction. In our model, this is simply performed by setting the active domain component of the state, and storing/restoring the context (of the active domain). The instruction is as follows.

$$[\![\text{DOMSET } d]\!] \equiv \begin{cases} \left.\begin{array}{l} \text{pushContext}(\sigma) \\ ad \leftarrow d \\ \text{popContext}(\sigma) \end{array}\right\} & pl = \text{S} \wedge \text{canx}(pc) \wedge d \in \{1, \ldots, N\} \\ \bot & \text{otherwise} \end{cases}$$

In practical processors, such an instruction does not exist. However, the domain switch is usually performed by a specific procedure consisting of many instructions. During this procedure, several special-purpose registers may be set or the memory-protection unit may be reconfigured, etc.

3.7 Dynamic domains

In this section, we discuss possible extensions to our previous definitions in order to support the creation of new domains during the execution of the separation kernel.

3.7.1 Passage of time

We start our discussion by introducing the notion of time. In particular, we divide the execution time of the separation kernel into several periods where in each period, a particular number of domains are managed by the separation kernel. We denote the time period with $\tau \in \mathbf{N}_0$[2] and the number of managed domains in the period τ with N^τ[3]. The separation kernel is initialized in the first period $\tau = 0$ with N^0 domains.

Domains can only be created, i.e., no destruction of domains is currently supported by our model. Hence, the N^τ monotonically increases with time τ. Furthermore, we also assume that the division

[2] $\mathbf{N}_0$ denotes natural numbers including zero.

[3] The superscript notation is used to represent the time period (not the usual mathematical exponentiation).

into time periods is minimal in the sense that no redundant periods are present, i.e., where the number of domains in the two consecutive periods would be the same. We state this with the following assumption.

Assumption 16. *The number of domains may grow through time, i.e.,*

$$N^{\tau} < N^{\tau+1}.$$

3.7.2 Dynamic state

Now, let us introduce time into the state definition, where we start from the existing definition of state σ introduced in Section 3.2. We call such a state a dynamic state and denote it with σ^{τ}, where $\tau \in \mathbf{N}_0$ represents the time period, to contrast it with the static state σ. Most of the existing components of the static state σ remain unchanged, except the last component, the variable ad that represents the currently active domain. The goal of this change is to mathematically properly encapsulate the (mathematical) domain of the variable ad, since it must reflect the number of managed (secure) domains in a particular time period. Hence, we define the dynamic state as

$$\sigma^{\tau} = (mem, reg, pc, pl, ad^{\tau})$$

where $\tau \geq 0$ denotes the time period in which the state σ^{τ} and respectively ad^{τ} are effective.

Furthermore and most importantly, the (function) domain of the variable ad^{τ} is

$$ad^{\tau} : \{1, \ldots, N^{\tau}\},$$

where N^{τ} is the number of (secure) domains in the period τ. Notice that, it is N^{τ} which changes with a transition from the time period τ to $\tau + 1$. But this is not enough to fully mathematically capture new domain creation, as a new domain is created one must also take care of the new memory configuration to be valid.

3.7.3 Dynamic configuration

On a new domain creation, besides the dynamic state, we also have to consider the new memory configuration of the separation kernel. Here, the update mechanism is even more fundamental. To model dynamic configuration, we first define a function Σ^{τ} with a functionality

$$\Sigma^{\tau} : \{1, \ldots, N^{\tau}\} \mapsto \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m).$$

Here, $\mathcal{P}(A)$ represents the power set of A, i.e., all subsets of set A.

Now, to obtain the particular configuration for the domain identified with $d \in N^{\tau}$ in the time period $\tau \in \mathbf{N}_0$, we define $\Sigma^{\tau}(d)$ as

$$\Sigma^{\tau}(d) = (A_d^{\tau}, C_d^{\tau}, D_d^{\tau}, E_d^{\tau}),$$

where the sets $A_d^{\tau}, C_d^{\tau}, D_d^{\tau}, E_d^{\tau}$ are the full address space, the code memory region, the data memory region, and the memory-mapped I/O devices region in the time period τ, respectively

3.7.4 State and configuration evolution

The machine is initialized and starts execution in the first state σ^0 and configuration Σ^0. In the CROSSCON Hypervisor, these are initialized from the config.h file. See also Section 5 for a discussion on the verification of the configuration file.

The dynamic state σ^{τ} changes through time, i.e., is updated with every executed machine instruction. For example, the following sequence represents a possible state evolution

$$\sigma^0, \ldots, \sigma^0, \sigma^1, \ldots, \sigma^1, \sigma^2, \ldots, \sigma^2, \ldots,$$

for which we also have a corresponding configuration sequence

$$\Sigma^0, \ldots, \Sigma^0, \Sigma^1, \ldots, \Sigma^1, \Sigma^2, \ldots, \Sigma^2, \ldots$$

The purpose of the dynamic state and configuration as well as the parameter τ is, thus, to track the time period when a particular configuration is active.

With each executed instruction the state changes and its components are updated while the configuration remains unchanged. The exception to this is a domain creation instruction (or, in practice, a corresponding procedure). Notice that even if domain creation is performed in a procedure consisting of many instructions, we conceptually consider it as an atomic transaction.

3.7.5 Domain creation

When a new domain is created the machine goes from the time period τ to $\tau + 1$. This transitions also reflects in the transition of state from σ^τ to $\sigma^{\tau+1}$ and configuration from Σ^τ to $\Sigma^{\tau+1}$.

To summarize, the domain creation is represented with the transition from the time period τ represented with the state

$$\sigma^\tau = (mem, reg, pc, pl, ad^\tau) \quad \text{with} \quad ad^\tau : \{1, \ldots, N^\tau\},$$

and configuration

$$\Sigma^\tau : \{1, \ldots, N^\tau\} \mapsto \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m) \quad \text{, where} \quad \Sigma^\tau(d) = (A_d^\tau, D_d^\tau, C_d^\tau, E_d^\tau),$$

to the state in the period $\tau + 1$ represented with the state

$$\sigma^{\tau+1} = (M, R, pc, pl, ad^{\tau+1}) \quad \text{with} \quad ad^{\tau+1} : \{1, \ldots, N^{\tau+1}\},$$

and configuration

$$\Sigma^{\tau+1} : \{1, \ldots, N^{\tau+1}\} \mapsto \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m) \times \mathcal{P}(A_m), \text{ where } \Sigma^{\tau+1}(d) = (A_d^{\tau+1}, D_d^{\tau+1}, C_d^{\tau+1}, E_d^{\tau+1}).$$

In general, the new number of domains $N^{\tau+1}$ must be greater than N^τ (see Assumption 16). However, without loss of generality, we also assume that the number of domains is always incremented (increased by one). We have the following assumption.

Assumption 17.

$$N^{\tau+1} = N^\tau + 1.$$

To achieve the effect of creating multiple domains, one only needs to repeat several single-domain creation procedures.

3.7.6 Memory protection

Now, we have prepared the mathematical notions to support dynamic domains. However, domain creation must be performed carefully so as not to violate memory protection. We rewrite Assumption 7 from Section 3.3 to take into account the possibility of dynamic domain creation.

Assumption 18. *Separation of memory regions. For each $\tau \geq 0$ we have*

- $A_d^\tau \subseteq A_M$ *for all* $0 \leq d \leq N^\tau$ *(the main regions of each domain).*
- $A_i^\tau \cap A_j^\tau = \emptyset$ *for all* $0 \leq i < j \leq N^k$ *(the main regions are disjoint, with no shared memory).*
- $C_d^\tau \subseteq A_d^\tau$ *and* $D_d^\tau \subseteq A_d^\tau$ *for all* $0 \leq d \leq N^\tau$ *(code and data regions are sub-regions of the corresponding main region).*
- $C_d^\tau \cup D_d^\tau = A_d^\tau$ *for all* $0 \leq d \leq N^\tau$ *(the code and data sub-regions fully cover the corresponding main region).*
- $C_d^\tau \cap D_d^\tau = \emptyset$ *for all* $0 \leq d \leq N^\tau$ *(the code and data sub-regions are disjoint).*

3.7.7 Domain splitting

A general mechanism for domain creation may be an arbitrary one that complies with memory protection constraints. Nevertheless, the CROSSCON hypervisors support the splitting of an existing domain into subdomains where the subdomains get the address space of the original domain.

Consider a time period $\tau \geq 0$ and a domain $p \in D^\tau$. The domain p is split into two:

- the parent domain, denoted with p (i.e., its domain identifier remains the same),
- the child domain, denoted with $c = N^\tau + 1$ (i.e., a domain with a new identifier).

The new domains identified with p and c in the time period $\tau + 1$ are called subdomains of the original domain p in the time period τ. The creation of domains admits, besides memory protection constraints, also to the following assumptions.

Assumption 19. *Consider a time period τ and a domain p to be split into subdomains p and $c = N^\tau + 1$. We have*

- $A_p^{\tau+1} \subseteq A_p^\tau$ *(the parent domain memory region is a subregion of the original domain region),*
- $A_c^{\tau+1} \subseteq A_p^\tau$ *(the child domain memory region is a subregion of the original domain region),*
- $A_p^{\tau+1} \cap A_c^{\tau+1} = \emptyset$ *(the new regions of parent and child are disjoint),*
- $A_p^{\tau+1} \cup A_c^{\tau+1} = A_p^\tau$ *(the union of the new regions of parent and child is the original domain region).*

4 Formalization of a TEE for Hypervisor-less hardware

As highlighted in [2], numerous low-cost, small-size, and low-power consumption hardware devices emerged as possible hardware to deploy secure applications. These devices often do not have basic hardware-based memory safety features, such as Micro Controller Units (MCU), and this makes them more vulnerable to attacks. Thus, as discussed in [2], within CROSSCON we adopt a software-based methodology to ensure the isolation between normal applications and trusted applications. As part of this methodology we adopted PISTIS [16], a framework that allows for memory isolation, remote attestation, and secure code update services, all fortified by robust security assurances. Figure 4 shows the architecture of such a setup.

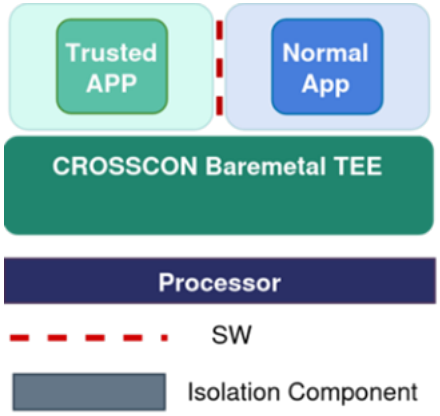

Figure 4: TEE and virtualization less architecture.

Figure 5 is a refinement of the memory map of Figure 2 for the case of a TEE in a virtualization less architecture.

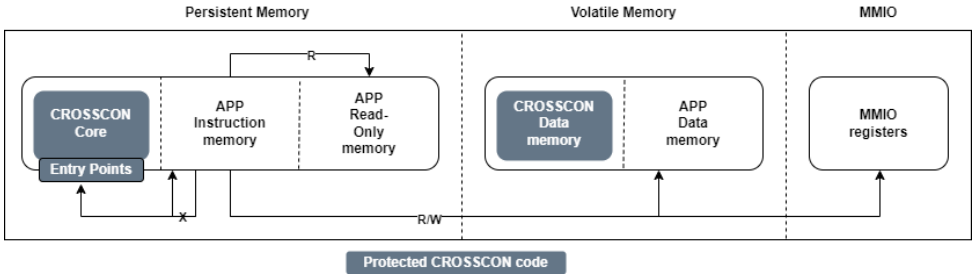

Figure 5: An example of a refined memory map for the TEE and virtualization less case.

To formalize the CROSSCON design and the layout of the memory map described in Figure 5, and prove that it preserves memory isolation, we first need to introduce some basic concepts. For simplicity, assume that we have a memory M that contains the code where data can also be stored and that there is a finite number of registers which include the program counter PC, the stack pointer SP, and for intermediate results (e.g. R_j for some j). Moreover, let us also assume that there is an interrupt vector table (IVT) that contains the addresses of the first instruction of the Interrupt Service Routine (ISR).

Any application can be thought as a sequence of the following *primitive instructions*:
Definition 1 (Primitive (unsafe) instructions)**.** *The set of* primitive (unsafe) instructions *is the following (we assume that the address i is a valid address for the memory M):*

- $Read(M, i)$ *that reads the content of memory M at address i: it denotes $M[i]$;*

- *Write(M,i,V) that writes V in memory M at address i: it denotes $M[i] = V$;*
- *Goto(M,i) that modifies the program counter PC to contain the address i of the given memory M: it denotes PC = i;*
- *End that terminates the execution of the entire application;*
- *Primitive operations (e.g. and, add, comparison) operating on registers/constants, and assignment (=) to a register.*

An application *P is a sequence and combination of the above primitive instructions, together with the IVT table specifying the addresses of the interrupt service routines.*

Without loss of generality, we can assume that all the instructions of a typical MCU can be written in terms of these basic primitives. Let us consider, for instance, some instructions taken from the MSP430 MCU's family:

- `CALL funcname`: the execution of this instruction stores in the stack (in the memory) the current `PC`, modifying the `SP` to create space for storing the `PC`, and then modifies the `PC` to point to the address corresponding to `funcname` in the memory. Thus, it corresponds to `SP = SP-1`, to create space in the stack, followed by `Write (M,SP,PC)`, to write `PC` in the stack, followed by `Goto (M,funcname)` to update `PC` and continue execution from address `funcname`.
- `RET`: corresponds to `R = Read(M,SP)` followed by `SP = SP+1`, and finally `Goto(M,R)`, using the value stored in the register `R` .

The primitive instructions also allow the modeling of more complex instructions, like `PUSH` and `POP` in a very similar fashion, as well as different addressing modes (e.g. indexed, symbolic, indirect, absolute, as, for instance, supported by the MSP430 MCU's family). For example,

- `MOV 0x22(R3), 0x10(R4)` corresponds to `Write(M,0x10 + R4, Read(M, 0x22 + R3));`
- `ADD 0x22(R3), 0x10(R4)` corresponds to `Write(M, 0x10 + R4, Read(M, 0x22 + R3)+ Read(M, 0x10 + R4));`
- `MOV @R4, &0x3000` corresponds to `Write(M, 0x3000, Read(M, R4));`
- `MOV 0x22, &0x3000` corresponds to `Write(M, 0x3000, Read(M, PC + 0x22))`.

Given the above basic concepts and letting `word [N]` denote a bit vector of size `N`, to proceed with the formalization of the CROSSCON design, the layout of the memory map described in Figure 5, and the access policy AP (for reading, writing, and jumping), we need to: (i) refine the memory M distinguishing between the persistent memory (M_P), the volatile memory (M_V), and the memory mapped IO (M_{MIO})[1]; (ii) explicitly introduces the finite set of Entry Point addresses for the CROSSCON core memory $EnPo$ = { `i : word[N]` } of addresses for the CROSSCON core instruction area as specified by the access policy; and finally (iii) introduce V_{IVT} : `array [K] of word[N]`, the interrupt vector `IVT` of size K. Moreover, we also need:

- utc_b, utc_e - start and end addresses of the CROSSCON core;
- aim_b, aim_e - start and end addresses of the App Instruction Memory;
- $arom_b$, $arom_e$ - start and end addresses of the App Read-Only Memory;
- $utdm_b$, $utdm_e$ - start and end addresses of the CROSSCON Data Memory;
- adm_b, adm_e - start and end addresses of the App Data memory;
- $mmio_b$, $mmio_e$ - start and end addresses of the MMIO Registers.

The following constraint formalizes the memory layout and the non-overlapping of the different memory areas.

$$0_N < utc_b < utc_e < aim_b < aim_e < arom_b < arom_e \leq 2_N^{N-1}$$
$$0_N \leq utdm_b < utdm_e < adm_b < adm_e \leq 2_N^{N-1}$$
$$0_N < mmio_b < mmio_e \leq 2_N^{N-1}$$

[1]Without loss of generality, we consider each memory as an array of size 2^N of bit vectors of size `N` (i.e. `array word[N] of word[N]`), although only a small part might be used. The addresses are bit vectors of size `N` (i.e. `word[N]`. We use 0_N to represent the unsigned word of size N corresponding to value 0, similarly, 2_N^{N-1} to represent the unsigned word of size N corresponding to value 2^{N-1}.

To model the constraint that the execution of CROSSCON core instructions only happens through predefined Entry Points, we need a predicate $EP(i)$ which is true iff the address i is $utc_b \leq i \leq utc_e$ and i is an Entry Point address $EP_j \in EnPo$ for the CROSSCON core memory.

$$EP(i) \leftrightarrow ((utc_b \leq i \leq utc_e) \wedge i \in EnPo)$$

Then, to formalize the read/write/jump policies as those enforced in the Figure 5, we define the following terms:

$$AP_R(i, M) \leftrightarrow \begin{pmatrix} (M = M_P \rightarrow (arom_b \leq i \leq arom_e)) & \wedge \\ (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}$$

$$AP_W(i, M) \leftrightarrow \begin{pmatrix} (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \wedge \\ (M = M_P) \rightarrow \bot) & \end{pmatrix}$$

$$AP_X(i, M) \leftrightarrow \Big((M = M_P) \wedge (EP(i) \vee (aim_b \leq i \leq aim_e))\Big)$$

Moreover, we also need to ensure that each element of the `IVT` table is a valid address w.r.t. the access policy, so that:

$$AP_{IVT}(M) \leftrightarrow \forall i. AP_X(V_{IVT}[i], M)$$

We denote by AP the access policy resulting from the memory layout, from the read/write/jump constraints, and from the fact that each $i \in EnPo$ is such that $utc_b \leq i \leq utc_e$. Given the above formalizations, we can formally define when an application preserves memory isolation w.r.t. a given access policy AP.

Definition 2. *Given a memory layout and an access policy AP, an application P* preserves memory isolation w.r.t. the access policy AP *iff any of its read/write/jump instructions in all possible executions is such that the addresses used in such instructions satisfy the given access policy AP.*

We remark that the primitive instructions in Def. 1 do not enforce particular restrictions on the addresses where to read/write or jump (it suffices for them to be valid addresses). As thoroughly discussed, if the addresses of the application are constant, then checking whether the application preserves memory isolation is trivial. It suffices to check whether each address in the application satisfies AP. However, as noted, in many cases, such addresses are the results of the execution of the application itself. Thus, the check can only be performed during the execution of the application.

To enforce memory isolation, for a given access policy AP, we can define safe variants of the read-/write/jump instructions that will guarantee at runtime that no violation of the access policy occurs.

Definition 3 (Safe primitive instructions)**.** *Given an access policy AP, the* safe *read/write/jump* primitive instructions w.r.t. AP *are:*

- $Read_{sf}(M, i)$ *that reads the content of memory M at address i if address i is such that read policy $AP_R(i, M)$ holds, otherwise it ends execution;*
- $Write_{sf}(M, i, V)$ *that writes V in memory M at address i if address i is such that write policy $AP_W(i, M)$ holds, otherwise it ends execution;*
- $Goto_{sf}(M, i)$ *that modifies the program counter PC to contain the address i if address i is such that branch access policy $AP_X(i, M)$ holds, otherwise it ends execution.*

The following theorem holds for the *safe primitive instructions* defined above.

Theorem 12. *Given an access policy AP, the* safe primitive instructions $Read_{sf}$, $Write_{sf}$, *and* $Goto_{sf}$ *w.r.t. such AP preserve memory isolation and do not allow us to violate AP.*

The proof directly follows from the definition of the *safe primitive instructions* w.r.t. an AP (see Section 4.1 for the proof). If AP is violated, then each instruction results in ending the execution of the entire application, thus preventing access to memory areas prohibited by AP. On the other hand, if the address satisfies AP, instruction-specific access to the specified memory location is allowed.

Given this, we can prove that any application P such that i) $AP_{IVT}(M)$ holds (i.e. each address $i \in V_{IVT}$ satisfies $AP_X(M, i)$), and ii) uses only the safe primitive instructions, preserves memory isolation.
Theorem 13. *Let AP be an access policy and P be an application specified with the set of (unsafe) primitive instructions. Let P_{sf} be the application obtained from P by replacing each of the unsafe primitive instructions with the corresponding safe primitive instructions. If $AP_{IVT}(M)$ holds, then P_{sf} preserves memory isolation, preventing accessing memory addresses or executing code that violates AP.*

The proof is by induction on the structure of the application P_{sf} using Theorem 12 (see Section 4.1 for the proof). We note that enforcing each element of `IVT` to satisfy AP also ensures that the interrupt routines are located in memory locations allowed by AP. This requirement can be relaxed by not only rewriting the unsafe read/write/jump instructions with the corresponding safe ones, but also adding for each interrupt routine a new wrapping interrupt routine and modifying the `IVT` to point to the respective wrapping interrupt routine. Each wrapping interrupt routine first checks that the target address is safe; if so, it jumps to the original address in the non-modified `IVT` copy. Otherwise, it ends the execution.

We remark that for the case of TEE and virtualization-less architecture, we leverage a modified toolchain to transparently rewrite each potentially unsafe dynamic instruction and replace it with a safe virtualized equivalent that can be accessed via a call to a subroutine stored in the protected Trusted Computing Module (TCM) memory area. The target address of the corresponding instruction is verified at runtime when the subroutine is invoked. The execution continues normally if it is valid. Otherwise, a MCU reset is triggered.

4.1 Formal Proofs of Theorems 12 and 13

Proof of Theorem 12. In order to prove Theorem 12, we formalized the three operations in NuXmv [17] (a state-of-the-art symbolic model checker), and for each of the three formalizations we considered some Linear Temporal Logic (LTL) [18] properties aiming at proving the correctness of the operations. In this approach we codify the three different operations as a sequential program encoded in NuXmv in the form of Single Static Assignment [19] (as is typically done in compilers and software model checking), specifying for each value of the program counter: `s0` before the implicit condition checking the respective access policy, i.e. $AP_R(i, M)$, $AP_W(i, M)$ or $AP_X(i, M)$); `s1` if the access point condition holds; `s2` right after the real operation on the memory for reading, writing, or modifying the program counter if correct; `end` representing the location to jump to if the access policy is violated. The variable `i` is a free variable that models the address we aim to read from, write to and jump to, respectively. Then we have the three memories `PM` (Primary Memory), `VM` (Volatile Memory), and `MMIO`. `v` is the value to write in the memory in the case of $\texttt{Write}_{\texttt{sf}}$.

The Figure 7 contains the NuXmv code for $\texttt{Read}_{\texttt{sf}}$. Here we model the $\texttt{Read}_{\texttt{sf}}$ with `DEFINE ReadSV` that is a word of size `N+1` where the `N+1` bit is set to `0` if the access policy $AP_R(i, M)$ holds, and to `1` otherwise. The model is then complemented with three LTL properties. The first states that if the $AP_R(i, M)$ is always true, then the `state` can never take value `end`. The second property states that if the $AP_R(i, N)$ is always true, then the `N+1` is always `0` if the `state` is different from `s0` (i.e., the state before a $\texttt{Read}_{\texttt{sf}}$). Finally, the last LTL property states that if it is possible to reach a state where the violation of $AP_R(i, N)$ holds, then it is possible to reach the state `end` (the state where the read has violated the access policy). In this model, the `i` variable can range over any possible value (there is thus an implicit universal quantification (∀)).

Figure 6 reports the output of running NUXMV (taken from https://nuxmv.fbk.eu/) on this file. The execution was done on a Linux laptop. NUXMV was able to prove (or disprove) the three LTL properties in a few seconds.

```
computer_shell > nuXmv −int −dcx pistis_read.smv
*** This is nuXmv 2.0.0 (compiled on Oct 14 2019)
....
nuXmv > go_msat
nuXmv > check_ltlspec_ic3 −i
....
-- LTL specification ( G APr -> G state != end) is true
-- LTL specification ( G APr -> G (state != s0 -> ReadSV[32 : 32] = 0ud1_0)) is true
-- LTL specification ( F !APr -> F state = end) is false
-- (trace generation was suppressed)
nuXmv > quit
```

Figure 6: Run of NUXMV on the `pistis_read.smv` file to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$ instruction.

These results show that the first two properties hold, while the last one is violated (as expected) to indicate that the only possible way to reach the `end` state is to violate $AP_R(i, M)$ in a $\texttt{Read}_{\texttt{sf}}$. In this case, NUXMV can generate a trace showing how to reach that state (in the run, for the sake of presentation we disabled the extraction of the counterexample, option `-dcx` at the command line).

Similar considerations hold for the other two instructions. Figure 8 is the NUXMV code for proving the correctness of $\texttt{Goto}_{\texttt{sf}}$, while the one for the instruction $\texttt{Write}_{\texttt{sf}}$ can be found in Figure 9.

Proof of Theorem 13. The proof of Theorem 13, proceeds by induction on the structure of the application P_{sf} leveraging on Theorem 12.

Base case: There are four base cases, each constituted respectively by: i) the `NOP` no-op instruction that does not perform any access to the memory neither for writing nor reading nor executing; ii) the $\texttt{Read}_{\texttt{sf}}$; iii) the $\texttt{Write}_{\texttt{sf}}$; iv) the $\texttt{Goto}_{\texttt{sf}}$. The last three instructions preserve memory isolation, as proved in Theorem 12. The `NOP` preserves memory isolation since it does not access memory to write, read, or execute. Thus, the single-instruction program preserves memory isolation.

Step case: Let us assume that a program P preserves memory isolation. Let $P' = P; \texttt{inst}$ be a program obtained by adding an instruction `inst` immediately after the last instruction of P. The possible instructions `inst` are: `NOP`, $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$. These instructions preserve memory isolation (given the base case and Theorem 12). Thus, given that P preserves memory isolation and the fact that the possible extension of the program P (that is, the program P') also preserves memory isolation, we can conclude that the theorem holds.

```
-- To run it:
-- shell > nuXmv -int pistis_read.smv
--
-- At the nuXmv prompt issue following commands:
-- go_msat; check_ltlspec_ic3 -i; quit
--
MODULE main
  VAR PM : array word[32] of word[32]; -- Persistent memory
  VAR VM : array word[32] of word[32]; -- Volatile memory
  VAR MMIO : array word[32] of word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM}; -- kind of memory
  VAR state : {s0, s1, s2, end}; -- Possible values of the PC
  VAR PC : word[32]; -- The program counter;
  VAR v : word[32]; -- Value to write
  VAR i : word[32]; -- Address to read from/write to

  -- Memory layout as mandated by the policy
  VAR EPadd : array word[3] of word[32]; -- List of entry points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;

  INVAR
    0h32_00000000 < utc_b & utc_b < utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b & arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b < utdm_e &
    utdm_e < adm_b & adm_b < adm_e & adm_e <
          0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b < mmio_e & mmio_e
          < 0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
   (((mem_k = _PM ) -> ((arom_b <= i) & (i <= arom_e))) &
    ((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APw :=
   (((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e))) &
   ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APx :=
   ((mem_k = _PM ) & (EP | ((aim_b <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i <= utc_e)) &
     ((i = READ(EPadd, 0d3_0)) | (i = READ(EPadd, 0d3_1)) |
      (i = READ(EPadd, 0d3_2)) | (i = READ(EPadd, 0d3_3)) |
      (i = READ(EPadd, 0d3_4)) | (i = READ(EPadd, 0d3_5)) |
      (i = READ(EPadd, 0d3_6)) | (i = READ(EPadd, 0d3_7))));

  -- Read_sf(M, i) := if (APr(i,M)) return M[i] else goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APr : s1;
                     state = s1 & APr : s2;
                     state = s2 : s2;
                     TRUE : end;
                   esac;

   DEFINE ReadSV := case
     state = s0 & APr : 0d33_0;
     state = s1 & APr : case
                  mem_k = _PM : 0d1_0 :: READ(PM, i);
                  mem_k = _VM : 0d1_0 :: READ(VM, i);
                  mem_k = _MMIO : 0d1_0 :: READ(MMIO, i);
                  TRUE : 0h33_FFFFFFFF;
                esac;
     state = s2 : 0d1_0 :: 0h32_FFFFFFFF;
     state = end : 0d33_0;
     TRUE : 0d33_0;
   esac;

  -- If the APr is always true, then there is not a possibility to
  -- reach the end state.
   LTLSPEC
    G(APr) -> G(state != end)

  -- If the APr is always true, and we are in any state different
        from
  -- s0, then the error flag bit is always 0
  LTLSPEC
      G(APr) -> G(state != s0 -> ReadSV[32:32] = 0d1_0)

  -- If APr is violated, then the state eventually become end,
  -- i.e. end is only reachable if APr is violated
  LTLSPEC
      F(!APr) -> F (state = end)
```

Figure 7: The encoding to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$

```
-- To run it:
-- shell > nuXmv -int pistis_goto.smv
--
-- At the nuXmv prompt issue following commands:
-- go_msat; check_ltlspec_ic3 -i; quit
--
MODULE main
  VAR PM : array word[32] of word[32]; -- Persistent memory
  VAR VM : array word[32] of word[32]; -- Volatile memory
  VAR MMIO : array word[32] of word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM}; -- kind of memory
  VAR state : {s0, s1, s2, end}; -- Possible values of the PC
  VAR PC : word[32]; -- The program counter;

  VAR v : word[32]; -- Value to write

  VAR i : word[32]; -- Address to read from/write to

  -- Memory layout as mandated by the policy
  VAR EPadd : array word[3] of word[32]; -- List of entry points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;
  DEFINE END := 0h32_10000000;

  INVAR
   0h32_00000000 < utc_b & utc_b < utc_e & utc_e < aim_b &
   aim_b < aim_e & aim_e < arom_b & arom_b < arom_e &
   arom_e < 0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < utdm_b & utdm_b < utdm_e &
   utdm_e < adm_b & adm_b < adm_e & adm_e <
          0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < mmio_b & mmio_b < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b <= i) & (i <= arom_e))) &
     ((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e ))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APx :=
    ((mem_k = _PM ) & (EP | ((aim_b <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i = READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i = READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i = READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i = READ(EPadd, 0d3_7))));

  -- goto_sf(M, i, v) := if (APx(i,M)) PC = i else goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APx : s1;
                     state = s1 & APx : s2;
                     state = s2 : s2;
                     TRUE : end;
                   esac;
  INIT
   PC != END;
  ASSIGN
   next(PC) := case
                 state = s0 & APx : PC;
                 state = s1 & APx : i;
                 state = s2 : PC;
                 TRUE : END;
               esac;

  -- If the APx is always true, then there is not a possibility to
       reach the end state.
  LTLSPEC
    G(APx) -> G(state != end)

  -- If the APx is always true, then the PC never assumes value
       END
  LTLSPEC
     G(APx) -> G (state != s0 -> PC != END)

  -- If APx is violated, then the state eventually become end,
  -- i.e. end is only reachable if APx is violated
  LTLSPEC
     F(!APx) -> F (state = end)
```

Figure 8: The encoding to prove the correctness of the $\texttt{Goto}_{sf}$

```
-- To run it:
-- shell > nuXmv -int pistis_write.smv
--
-- At the nuXmv prompt issue following commands:
-- go_msat; check_ltlspec_ic3 -i; quit
--
MODULE main
  VAR PM : array word[32] of word[32]; -- Persistent memory
  VAR VM : array word[32] of word[32]; -- Volatile memory
  VAR MMIO : array word[32] of word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM}; -- kind of memory
  VAR state : {s0, s1, s2, end}; -- Possible values of the PC
  VAR PC : word[32]; -- The program counter;
  VAR v : word[32]; -- Value to write
  VAR i : word[32]; -- Address to read from/write to

  -- Memory layout as mandated by the policy
  VAR EPadd : array word[3] of word[32]; -- List of entry points
  DEFINE utc_b := 0h32_00000010;
  DEFINE utc_e := 0h32_00000100;
  DEFINE aim_b := 0h32_00001000;
  DEFINE aim_e := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b := 0h32_00001000;
  DEFINE adm_e := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;

  INVAR
   0h32_00000000 < utc_b & utc_b < utc_e & utc_e < aim_b &
   aim_b < aim_e & aim_e < arom_b & arom_b < arom_e &
   arom_e < 0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < utdm_b & utdm_b < utdm_e &
   utdm_e < adm_b & adm_b < adm_e & adm_e <
          0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < mmio_b & mmio_b < mmio_e & mmio_e <
          0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM ) -> ((arom_b <= i) & (i <= arom_e))) &
     ((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e ))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM ) -> ((adm_b <= i) & (i <= adm_e))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APx :=
    ((mem_k = _PM ) & (EP | ((aim_b <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i = READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i = READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i = READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i = READ(EPadd, 0d3_7))));

-- write_sf(M, i, v) := if (APw(i,M)) M[i] = v else goto end;
ASSIGN
   init(state) := s0;
   next(state) := case
                        state = s0 & APw : s1;
                        state = s1 & APw : s2;
                        state = s2 : s2;
                        TRUE : end;
                      esac;

TRANS
   case
    state = s0 & APw : next(PM) = PM & next(VM) = VM &
          next(MMIO) = MMIO;
    state = s1 & APw : case
          mem_k = _PM : next(PM) = WRITE(PM, i, v) &
                        next(VM) = VM & -- VM not touched
                        next(MMIO) = MMIO; -- MMIO not
                              touched
          mem_k = _VM : next(VM) = WRITE(VM, i, v) &
                        next(PM) = PM & -- PM not touched
                        next(MMIO) = MMIO; -- MMIO not
                              touched
          mem_k = _MMIO : next(MMIO) = WRITE(MMIO, i, v) &
                        next(VM) = VM & -- VM not touched
                        next(PM) = PM; -- PM not touched
          TRUE : next(PM) = PM & next(VM) = VM & next(MMIO)
                = MMIO;
       esac;
    state = s2 : next(PM) = PM & next(VM) = VM & next(MMIO)
          = MMIO;
    state = end : next(PM) = PM & next(VM) = VM &
          next(MMIO) = MMIO;
    TRUE : next(PM) = PM & next(VM) = VM & next(MMIO) =
          MMIO;
   esac;

-- If the APw is always true, then there is not a possibility to
-- reach the end state.
LTLSPEC G(APw) -> G(state != end)

-- If APw is violated, then the state eventually become end,
-- i.e. end is only reachable if APw is violated
LTLSPEC
    F(!APw) -> F (state = end)

-- If the APw is violated, then the memory is not modified
LTLSPEC
    G(!APw) -> G (next(PM) = PM & next(VM) = VM &
          next(MMIO) = MMIO)
```

Figure 9: The encoding to prove the correctness of the $\texttt{Write}_{sf}$

5 Formalization of the CROSSCON Hypervisor configuration

This chapter aims to provide a formal correctness specification for the CROSSCON Hypervisor configuration data, extracted from the informal descriptions contained in the project documentation and in Section 5.2 of the deliverable D2.3 [3]. The specification is sometimes stricter than what the documentation allows; these points are explicitly noted in what follows.

The correctness predicate will be expressed as a set of formulae expressed in the formal theory of Linear Integer Arithmetic (LIA). The correctness of a configuration can then be checked by building the conjunction of all these formulae and checking the validity of the resulting formula by using any off-the-shelf SMT solver supporting that theory.

5.1 Summary of the configuration data

The C types used in the relevant code are the following:

- `bool`, `size_t` (standard);
- `paddr_t`, `vaddr_t` (defined as `uintptr_t`);
- `cpumap_t` (defined as `unsigned long int`);
- `irqid_t`, `deviceid_t` (defined as `unsigned int`).

Note that the size of these types is platform-dependent. In practice, the two intended targets are the ILP32 and the LP64 ABIs.

The configuration data are contained in a C struct with the following contents:

- A header, whose fields are out of the scope of this specification.
- A possibly empty list `shmemlist` of configured shared memory regions, of length `shmemlist_size`. The ID of a shared memory region is its position in this list.

  Each entry of the list is a struct of type `struct shmem`. The fields of this struct that are relevant to the present specification are listed below.

  - `base`: `vaddr_t` (mandatory for Microcontroller Processing Units (MPU) systems) — the base virtual address of the shared memory region.
  - `size`: `size_t` (mandatory) — the size (in bytes) of the shared memory region.
  - `place_phys`: `bool` (optional, default false) — if true, the virtual address corresponds to the physical address. Only meaningful for Memory Management Unit (MMU) systems.
  - `phys`: `paddr_t` (mandatory if `place_phys` is true) — the physical address where the memory region should be mapped.

  The fields `base` and `phys` are members of a nameless union, whose content is interpreted as a physical address for MPU systems, and if `place_phys` is true also for MMU systems.
- A nonempty list `vmlist` of configured VMs, of length `vmlist_size`.

  Each entry is a struct of type `struct vm_config`. The fields of this struct that are relevant to the present specification are listed below.

  - `image` (mandatory): a struct of type `struct vm_image` that contains information regarding the guest image. The fields relevant for the present specification are:

- `base_addr`: `vaddr_t` (mandatory) — the image load address in the guest physical address space.
- `load_addr`: `paddr_t` (mandatory) — the image load address in the physical address space.
- `size`: `size_t` (mandatory) — the image size in bytes.

– `entry`: `vaddr_t` (mandatory) — the entry point in the guest physical address space.

– `cpu_affinity`: `cpumap_t` (optional, default 0) — a bitmap signaling the preferred physical CPUs assigned to the Virtual Machine (VM). If the CPU affinities are mutually exclusive for all configured VMs, the physical CPUs assigned to each VM will follow the specified bitmaps; otherwise, the hypervisor will choose how to resolve the conflicts.

– `platform` (mandatory): a struct of type `struct vm_platform` describing the resources allocated to the VM. The fields relevant to the present specification are:

- `cpu_num`: `size_t` — the number of assigned CPUs.
- `regions`: a non-empty list of assigned memory regions, of length `region_num`. Each region is a struct of type `struct vm_mem_region`, with the following fields:
  - `base`: `vaddr_t` (mandatory) — the base guest physical address of the region.
  - `size`: `size_t` (mandatory) — the size in bytes of the region.
  - `place_phys`: `bool` (optional, default false) — if true, the guest physical address corresponds to the physical address.
  - `phys`: `paddr_t` (mandatory if `place_phys` is true) — the physical address where the memory region should be mapped.

  The fields `base` and `phys` are mutually exclusive (they are members of a nameless union).
- `ipcs`: a possibly empty list of Interrupt Process Control (IPC) objects of length `ipc_num`. Each IPC object is a struct of type `struct ipc`, with the following fields:
  - `base`: `vaddr_t` (mandatory) — the base guest physical address of the region. Must be equal to the shared memory object address.
  - `size`: `size_t` (mandatory) — the size in bytes of region. Must be less than or equal to the size declared in the shared memory list.
  - `shmem_id`: `size_t` (mandatory) — the identifier of the associated shared memory region.
  - `interrupts`: a possibly empty list of length `interrupt_num` whose elements (of type `irqid_t`) specify interrupt numbers.
- `devs`: a possibly empty list of devices objects, of length `dev_num`. Each device object is a struct of type `struct vm_dev_region`, with the following fields:
  - `pa`: `paddr_t` (mandatory) — the base physical address of the Memory Mapped Input/Output (MMIO) region of the device.
  - `va`: `vaddr_t` (mandatory) — the base guest physical address of the device MMIO region. Must be equal to `pa` for MPU systems.
  - `size`: `size_t` (mandatory) — the size in bytes of device MMIO region.
  - `interrupts`: a possibly empty list of length `interrupt_num` whose elements (of type `irqid_t`) specify the interrupt numbers.

5.2 Formal specification

We shall need to parameterize our formulae on a flag `Pmmu`, which is true if and only if the configuration is intended for an MMU-based system. We denote by $\ell(\mathtt{x})$ the length of the list `x`. We use the binary operator $\otimes$ to denote the *bitwise and* operation between machine integers.

The first condition to be satisfied by the configuration data is a simple consistency check on the length of the top-level lists:

$$(\ell(\mathtt{shmemlist}) = \mathtt{shmemlist\_size}) \wedge (\ell(\mathtt{vmlist}) = \mathtt{vmlist\_size}) \wedge (\mathtt{vmlist\_size} > 0) \tag{5.1}$$

5.2.1 Checks on shared memory definition

In this subsection, we list the conditions that define shared memory areas.

We operate under the assumption of physical addresses:

$$\psi_{\mathrm{shp}} := \forall i\ (\mathtt{shmemlist}[i].\mathtt{place\_phys} = \mathrm{true})$$

We define a family of predicates σ_i on an (unsigned) integer variable x expressing the fact that x belongs to the i-th shared memory region:

$$\sigma_i(x) := (\mathtt{shmemlist}[i].\mathtt{phys} \le x < \mathtt{shmemlist}[i].\mathtt{phys} + \mathtt{shmemlist}[i].\mathtt{size})$$

The conjunction of the following conditions then gives the correctness predicate for the shared memory definition:

- no region is empty:

$$\forall i\ \mathtt{shmemlist}[i].\mathtt{size} > 0 \tag{5.2}$$

- each region is correctly aligned:

$$\mathtt{Pmmu} \rightarrow \forall i\ (\mathtt{shmemlist}[i].\mathtt{phys} \bmod 4096 = 0) \wedge (\mathtt{shmemlist}[i].\mathtt{size} \bmod 4096 = 0) \tag{5.3}$$

$$\neg\mathtt{Pmmu} \rightarrow \forall i\ (\mathtt{shmemlist}[i].\mathtt{phys} \bmod 64 = 0) \wedge (\mathtt{shmemlist}[i].\mathtt{size} \bmod 64 = 0) \tag{5.4}$$

- regions do not overlap:

$$\forall i, i'\ (i < i' \rightarrow \forall x\ \neg(\sigma_i(x) \wedge \sigma_{i'}(x))) \tag{5.5}$$

5.2.2 Checks on single VM definitions

Let i be an index that runs on the list of VMs defined by the configuration. In this subsection, we formulate the conditions that must be satisfied for each value of i. For the sake of brevity, in each variable reference that follows a prefix $\mathtt{vmlist}[i]$ is understood (that is, we use `x` as a shorthand for $\mathtt{vmlist}[i].\mathtt{x}$).

We operate under the assumption of physical addresses:

$$\psi_{\mathrm{vmp},i} := \forall j\ (\mathtt{platform.regions}[j].\mathtt{place\_phys} = \mathrm{true})$$

We define a family of predicates $\mu_{i,j}$ on an (unsigned) integer variable x expressing the fact that x belongs to the j-th memory region defined for the i-th VM:

$$\begin{aligned}\mu_{i,j}(x) := \quad & (\mathtt{platform.regions}[j].\mathtt{phys} \le x \wedge \\ & x < \mathtt{platform.regions}[j].\mathtt{phys} + \mathtt{platform.regions}[j].\mathtt{size})\end{aligned}$$

and corresponding predicates μ_i expressing the fact that x belongs to the union of all the memory regions defined for the i-th VM:

$$\mu_i(x) \coloneqq \bigvee_j \mu_{i,j}(x)$$

We also define analogous predicates for MMIO regions:

$$\nu_{i,j}(x) \coloneqq (\texttt{platform.devs}[j].\texttt{pa} \leq x < \texttt{platform.devs}[j].\texttt{pa} + \texttt{platform.devs}[j].\texttt{size})$$

$$\nu_i(x) \coloneqq \bigvee_j \nu_{i,j}(x)$$

The correctness predicate for the definition of VM number i is then given by the conjunction of the following conditions:

- consistency checks on lists:

$$\begin{aligned}
&(\ell(\texttt{platform.regions}) = \texttt{platform.region_num}) \wedge (\texttt{region_num} > 0) \wedge \\
&(\ell(\texttt{platform.ipcs}) = \texttt{platform.ipc_num}) \wedge \\
&\forall j\ (\ell(\texttt{platform.ipcs}[j].\texttt{interrupts}) = \texttt{platform.ipcs}[j].\texttt{interrupt_num}) \wedge \\
&(\ell(\texttt{platform.devs}) = \texttt{platform.dev_num}) \wedge \\
&\forall j\ (\ell(\texttt{platform.devs}[j].\texttt{interrupts}) = \texttt{platform.devs}[j].\texttt{interrupt_num})
\end{aligned} \tag{5.6}$$

- checks on `image`: binary image is contained in μ_i:

$$\forall x\ (\texttt{image.base\_addr} \leq x < \texttt{image.base\_addr} + \texttt{image.size}) \rightarrow \mu_i(x) \tag{5.7}$$

- checks on `entry`: entry point belongs to μ_i:

$$\mu_i(\texttt{entry}) \tag{5.8}$$

- checks on `platform`: at least one CPU:

$$\texttt{platform.cpu\_num} > 0 \tag{5.9}$$

  at least one region:

$$\texttt{platform.region\_num} > 0 \tag{5.10}$$

  no region is empty:

$$\forall j\ (\texttt{platform.regions}[j].\texttt{size} > 0) \tag{5.11}$$

  each region is correctly aligned:

$$\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.regions}[j].\texttt{phys} \bmod 4096 = 0) \wedge (\texttt{platform.regions}[j].\texttt{size} \bmod 4096 = 0)) \tag{5.12}$$

$$\neg\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.regions}[j].\texttt{phys} \bmod 64 = 0) \wedge (\texttt{platform.regions}[j].\texttt{size} \bmod 64 = 0)) \tag{5.13}$$

  regions do not overlap:

$$\forall j, j'\ (j < j' \rightarrow \forall x\ \neg(\mu_{i,j}(x) \wedge \mu_{i,j'}(x))) \tag{5.14}$$

  each IPC area is declared:

$$\forall j\ (0 \leq \texttt{platform.ipcs}[j].\texttt{shmem\_id} < \texttt{shmemlist\_size}) \tag{5.15}$$

IPC area parameters corresponds to the declared ones: if $k_j := \texttt{platform.ipcs}[j].\texttt{shmem\_id}$,

$$\forall j \left((\texttt{platform.ipcs}[j].\texttt{base} = \texttt{shmemlist}[k_j].\texttt{phys}) \wedge (\texttt{platform.ipcs}[j].\texttt{size} = \texttt{shmemlist}[k_j].\texttt{size})\right) \tag{5.16}$$

(here the documentation allows the weakest condition $\texttt{platform.ipcs}[j].\texttt{size} \leq \texttt{shmemlist}[k_j].\texttt{size}$)

each IPC area is correctly aligned:

$$\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.ipcs}[j].\texttt{base} \bmod 4096 = 0) \wedge (\texttt{platform.ipcs}[j].\texttt{size} \bmod 4096 = 0)) \tag{5.17}$$

$$\neg\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.ipcs}[j].\texttt{base} \bmod 64 = 0) \wedge (\texttt{platform.ipcs}[j].\texttt{size} \bmod 64 = 0)) \tag{5.18}$$

each device MMIO region is non-empty:

$$\forall j\ (\texttt{platform.devs}[j].\texttt{size} > 0) \tag{5.19}$$

each device MMIO region is correctly aligned:

$$\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.devs}[j].\texttt{pa} \bmod 4096 = 0) \wedge (\texttt{platform.devs}[j].\texttt{size} \bmod 4096 = 0)) \tag{5.20}$$

$$\neg\texttt{Pmmu} \rightarrow \forall j\ ((\texttt{platform.devs}[j].\texttt{pa} \bmod 64 = 0) \wedge (\texttt{platform.devs}[j].\texttt{size} \bmod 64 = 0)) \tag{5.21}$$

MMIO regions do not overlap:

$$\forall j, j'\ (j < j' \rightarrow \forall x\ \neg(v_{i,j}(x) \wedge v_{i,j'}(x))) \tag{5.22}$$

interrupt numbers are assigned uniquely:

$$\forall j, j'\ j < j' \rightarrow \forall n\ \neg(n \in \texttt{platform.ipcs}[j].\texttt{interrupts} \wedge n \in \texttt{platform.ipcs}[j'].\texttt{interrupts}) \tag{5.23}$$

$$\forall j, j'\ j < j' \rightarrow \forall n\ \neg(n \in \texttt{platform.devs}[j].\texttt{interrupts} \wedge n \in \texttt{platform.devs}[j'].\texttt{interrupts}) \tag{5.24}$$

$$\forall j, j'\ \forall n\ \neg(n \in \texttt{platform.ipcs}[j].\texttt{interrupts} \wedge n \in \texttt{platform.devs}[j'].\texttt{interrupts}) \tag{5.25}$$

5.2.3 Checks on whole-system definition

In this subsection we group the conditions that involve the complex of all defined VMs. Let us collect all the assumptions made so far in a single formula:

$$\psi_{\text{phys}} := \psi_{\text{shp}} \wedge \forall i\ \psi_{\text{vmp},i}$$

Recall also that μ_i denotes the predicate expressing the fact that physical address x belongs to the memory allocated to the VM with index i.

The correctness predicate for the whole-system definition is given by the conjunction of the following conditions:

- attribution of CPUs is disjoint:

$$\forall i, i' \ (i < i' \rightarrow \texttt{vmlist}[i].\texttt{cpu\_affinity} \otimes \texttt{vmlist}[i].\texttt{cpu\_affinity} = 0) \tag{5.26}$$

  (this rules out configurations that are legal according to the documentation, but is needed to ensure determinism of CPU allocation)

- no overlap between memory areas assigned to different VMs:

$$\forall i, i' \ (i < i' \rightarrow \forall x \ \neg(\mu_i(x) \wedge \mu_{i'}(x))) \tag{5.27}$$

- no overlap between MMIO regions assigned to different VMs:

$$\forall i, i' \ (i < i' \rightarrow \forall x \ \neg(\nu_i(x) \wedge \nu_{i'}(x))) \tag{5.28}$$

- no interrupt is shared between devices of different VMs: defining

$$\texttt{dev\_ints}_i := \bigcup_j \texttt{vmlist}[i].\texttt{platform.devs}[j].\texttt{interrupts}$$

  we can express this condition as

$$\forall i, i' \ (i < i' \rightarrow \forall n \ \neg(n \in \texttt{dev\_ints}_i \wedge n \in \texttt{dev\_ints}_{i'}) \tag{5.29}$$

  (note that IPC interrupts are purely virtual; therefore, different VMs can have the same interrupt number without any issue).

  Note: in some platforms, some interrupts are core-specific and will be duplicated for different VMs. For instance, on ARM platforms, interrupt numbers from 0 to 31 are local to each CPU core. In this case the definitions of $\texttt{dev\_ints}_i$ must be modified as follows:

$$\texttt{dev\_ints}_i := \bigcup_j (\texttt{vmlist}[i].\texttt{platform.devs}[j].\texttt{interrupts} \cap I)$$

  where $I \subseteq \mathbb{N}$ is the subset of non-core-specific interrupt numbers.

5.2.4 Checks on instantiability

For this set of checks, we assume that a suitable source of information on the target platform is available ahead of deployment. In particular, we assume the presence of the following data:

- The number of physical CPUs available, denoted by Ncpu;
- The set of valid physical memory addresses, expressed by a predicate $\texttt{mem\_valid}(x)$ that is true if and only if address x is valid;
- The set of available MMIO addresses, expressed by a predicate $\texttt{mmio\_valid}(x)$ that is true if and only if address x is valid;
- The set of available IRQ lines, expressed by a predicate $\texttt{irq\_valid}(n)$ that is true if and only if integer n is a valid IRQ line.

With these assumptions, the instantiability predicate can be formulated as the conjunction of the following conditions:

- the cumulative count of CPUs allocated across all VMs does not exceed the total number of CPUs available on the platform:

$$\sum_i \texttt{vmlist}[i].\texttt{platform.cpu\_num} \leq \texttt{Ncpu} \tag{5.30}$$

- (optionally) allocated CPUs are exactly equal to the available CPUs:

$$\bigotimes_i \texttt{vmlist}[i].\texttt{cpu\_affinity} = 2^{\texttt{Ncpu}+1} - 1 \tag{5.31}$$

- every specified memory region is contained in the physical available memory:

$$\forall i \; \forall x \; (\mu_i(x) \rightarrow \texttt{mem\_valid}(x)) \tag{5.32}$$

- (optionally) allocated memory is exactly equal to the physically available memory:

$$\forall x \; (\bigvee_i \mu_i(x) \leftrightarrow \texttt{mem\_valid}(x)) \tag{5.33}$$

- every device MMIO region is contained in some MMIO range:

$$\forall i \; \forall x \; (\nu_i(x) \rightarrow \texttt{mmio\_valid}(x)) \tag{5.34}$$

- every declared interrupt number is available:

$$\forall i \; \forall n \; (n \in \texttt{dev\_ints}_i \rightarrow \texttt{irq\_valid}(n)) \tag{5.35}$$

6 Formalization of dynamic VM instantiation

This chapter provides a formal model for the dynamic VM instantiation feature that will be introduced in the final version of the CROSSCON Hypervisor, as described in the deliverable D2.3 [3].

6.1 Interface

The interface for accessing the dynamic VM management feature consists of the following three primitive operations.

- The `vm_create` function takes as input a configuration c for the new VM to be created and (if c is correct) returns an opaque identifier h, called a *VM handle*, which can be used in subsequent calls to the API to refer to the new VM.
- The `vm_invoke` function takes as input the handle h of the child VM to which the parent VM wishes to transfer control. If h is a valid handle, the transfer of control is performed. When the child VM yields control back, the function returns successfully to the parent VM.
- The `vm_delete` function takes as input the handle h of the child VM to be deleted. If h is a valid handle, the corresponding child VM is terminated and all the resources allocated to it are freed.

In order to proceed with the formalization, we will also need the following objects.

- A predicate $valid(c)$, defined on the set of all possible VM configurations, that is true for configuration c if and only if c is valid, namely, if and only if it satisfies the correctness predicate which has been defined in Chapter 5.
- A finite set $\mathcal{H}$ that contains all the possible handles for a dynamically-created VM. In practice, if we denote by N_{dyn} the maximum number of dynamically-created VMs that the hypervisor is able to handle, we can simply take $\mathcal{H} = \{1, \ldots, N_{dyn}\}$, and treat the concrete representation of handles as an (irrelevant) implementation detail.

6.2 State transition diagram

We can now define a formal model for the dynamic management of VMs by the CROSSCON Hypervisor. We shall use the well-known formalism of *extended finite-state automata*, namely finite-state automata whose states are defined partly explicitly by a state transition diagram and partly implicitly via the possible values that a given set of variables may assume.

In the present setting the only variable needed, which we shall denote by H, is the set of *currently active handles*. This is a subset of $\mathcal{H}$ that records, at each point in time, the handles that have been assigned to a child VM. The time evolution of this variable is clear from the above interface specification:

- following a successful *create* call, the handle h returned by the `vm_create` function (which is assumed to be distinct from every handle already present in H) must be added to the set H;
- a call to the *invoke* primitive is successful if and only if the input handle h belongs to the set of active handles H, and does not modify the set H;
- a call to the *delete* primitive is successful if and only if the input handle h belongs to the set of active handles H, in which case handle h must be removed from the set H.

In Figure 10 we depict the state transition diagram for a finite state automaton $\mathcal{A}$ that models the dynamic VM instantiation mechanism according to the above discussion. Here, we are using a standard notation for transitions, where a label of the form c/e means that the transition is taken if and only if the boolean condition c is verified, and in that case, the variables are updated according to the effect e. When a transition does not update any variable, the second part is simply omitted.

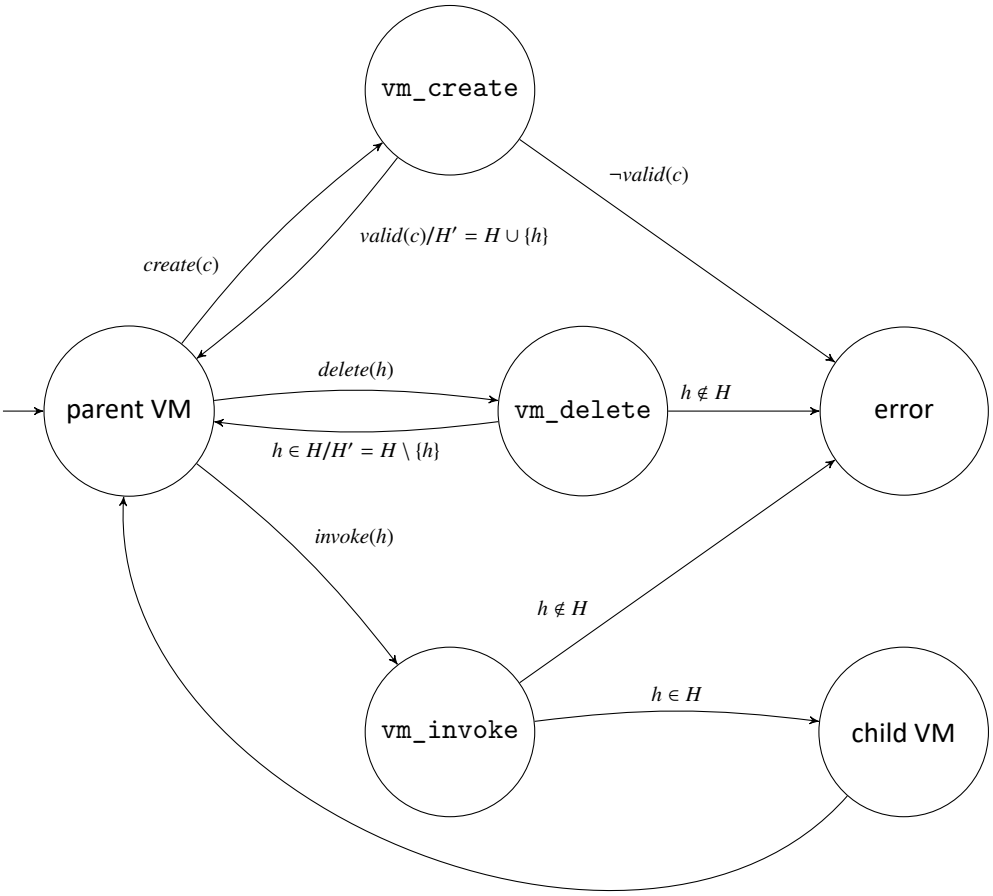

Figure 10: The state transition diagram for the dynamic VM instantiation logic.

The set of input actions for the automaton $\mathcal{A}$ is defined by the following grammar:

- for each possible configuration $c \in \mathcal{C}$, we have an action $create(c)$, corresponding to the creation of a VM with configuration c;
- for each possible handle $h \in \mathcal{H}$, we have actions $invoke(h)$ and $delete(h)$, which corresponds to the invocation and the destruction of the corresponding VM, respectively.

The correspondence between these input actions and the API functions illustrated in section 6.1 is clear.

6.3 Model checking

In order to formally prove some relevant properties for the automaton $\mathcal{A}$ it is necessary to encode it in a suitable specification language. As in chapter 4, we shall use the NUXMV model checker and its

associated language SMV, which supports (among others) specifications written in the Linear Temporal Logic formalism.

Due to some limitations in the SMV language, we had to modify slightly the automaton shown in Figure 10. The main changes are the following:

1. for the sake of simplicity, the configurations c with associated predicate $valid(c)$ have been replaced with a single input variable v, of boolean type, which directly encodes whether a supplied configuration is valid or not;
2. as the support for sets in SMV is quite restricted, we chose to model the set of handles H as an array a of boolean values of size N_{dyn}, such that for every $i \in \{1, \ldots, N_{dyn}\}$ the element $a[i]$ is true if and only if handle i is active;
3. finally, since the SMV language does not admit input variables with parameters, we had to decompose the input actions defined above into a triple of input variables (α, v, h) where α is an element of the enumerated type $\{create, delete, invoke\}$, v is the boolean variable mentioned in point 1 and h is a variable ranging over all possible handles. Clearly, the value of v is only taken into account when $\alpha = create$, and similarly for h with respect to the other two actions.

The resulting NUXMV code is shown in Listing 11.

Using this formalization we were able to prove some basic behavioral properties for the formal model of dynamic VM instantiation. In particular, we verified the validity of the following properties:

1. the error state is a sink;
2. a VM with a valid configuration is always instantiated;
3. an instantiated VM can always be executed;
4. a child VM can execute if and only if its handle is active.

```
MODULE main
  DEFINE N_dyn := 8;

  IVAR action : { create, delete, invoke };
  IVAR c_valid : boolean;
  IVAR h : 0..N_dyn;

  VAR state : {s_parent, s_child, s_create, s_delete, s_invoke,
        s_error};
  VAR handles : array 0..N_dyn of boolean;
  VAR saved_c_valid : boolean;
  VAR saved_h : 0..N_dyn;

  ASSIGN
    init(state) := s_parent;
    init(handles) := CONSTARRAY(typeof(handles), FALSE);
    init(saved_c_valid) := FALSE;
    init(saved_h) := 0;

    next(state) := case
      state = s_parent & action = create : s_create;
      state = s_parent & action = delete : s_delete;
      state = s_parent & action = invoke : s_invoke;

      -- Here we use nondeterminism of input variable h to
      -- choose a random handle for the new VM which is
      -- compatible with existing active handles
      state = s_create & saved_c_valid & READ(handles, h) =
            FALSE : s_parent;
      state = s_create & ! saved_c_valid : s_error;

      state = s_delete & READ(handles, saved_h) = TRUE :
            s_parent;
      state = s_delete & READ(handles, saved_h) = FALSE :
            s_error;

      state = s_invoke & READ(handles, saved_h) = TRUE :
            s_child;
      state = s_invoke & READ(handles, saved_h) = FALSE :
            s_error;

      state = s_child : s_child;
      state = s_child : s_parent;

      state = s_error : s_error;
      TRUE : state;
    esac;

    next(handles) := case
      -- vm_create, successful case
      state = s_create & saved_c_valid & READ(handles, h) =
            FALSE :
        WRITE(handles, h, TRUE);

      -- vm_delete, successful case
      state = s_delete & READ(handles, saved_h) = TRUE :
        WRITE(handles, saved_h, FALSE);

      -- every other transition leaves the set of active
      -- handles as it is
      TRUE : handles;
    esac;

    next(saved_c_valid) := case
      state = s_parent & action = create : c_valid;
      TRUE : saved_c_valid;
    esac;

    next(saved_h) := case
      state = s_parent & (action = delete | action = invoke) : h;
      TRUE : saved_h;
    esac;

-- Specification

-- 1. error state is a sink
LTLSPEC
  state = s_error -> G state = s_error;

-- 2. a dynamic VM with a valid configuration is instantiated
LTLSPEC
  action = create & c_valid -> X X (state = s_parent);

-- 3. a dynamic VM instantiated by a valid configuration can
      execute
LTLSPEC
  action = create & c_valid & X h = 1 & X X (action = invoke & h =
        1) -> F (state = s_child);

-- 4. a dynamic VM can execute if and only if its handle is active
LTLSPEC
  (action = invoke & h = 2 & X X state = s_child) -> (handles[2] =
        TRUE);
LTLSPEC
  (handles[2] = TRUE) -> (action = invoke & h = 2 & X X state =
        s_child);
```

Figure 11: The encoding to prove the correctness of the dynamic VM automaton.

7 Formal verification of digital hardware designs

Hardware primitives (e.g., MPU, Hypervisor extension, and Perimeter guard) are the fundamental building blocks on which the security of a computer system is based. Because of that, we want to ensure that they behave as expected; otherwise, the entire system might be compromised. We usually ensure that the digital hardware design works as expected with testing, which is easy to apply, but it is hard to cover the behaviour of the entire design. Formal methods promise to address this limitation by allowing better coverage of the design's behaviour. In this chapter, we look at the open source tools currently available to formally verify digital hardware designs, and we try to apply them to our designs, namely the AXI interconnect.

7.1 Motivation

We use tools every day. We drive a car to work, we use a stove to cook, and we use our phones to talk to our friends. Each time we use one of those tools, we expect the tool to behave in a certain way. When we drive a car, we expect the car to break when we hit the break; when we turn off a gas stove, we expect the stove to stop dispensing gas to the heating had; and when we send a message through our phone, we expect the program to deliver the message to the intended recipient. Any deviation of the tool from the expected behaviour can cause problems. Some problems can be efficiently mitigated, while others can cause harm, leading to injuries and financial damage.

In general, the question of how to prevent unexpected behaviour of a system is an open problem. We usually try to address this problem by testing; for example, we take a car, hit a break in different situations, and see what happens. We do the same in digital hardware design. To be sure that the design behaves as expected, we simulate it and see how it behaves when we provide specific input signals. Some behaviours are easy to test, but when the complexity of the design increases, it's harder and harder to test all possible behaviours. The problem we encounter here is that we only have a limited amount of engineering effort available to do the testing, which means that we are often not able to test the entire design. This means that we need to selectively choose which parts of the design we want to cover. Usually, we cover more crucial parts of the design. More effort that we use for testing, the more of the design's behaviour can be covered, and more confident we can be that the design behaves as expected. Because producing chips is expensive and we do not have a chance to fix the design after a chip is fabricated, we want to be confident that our designs work as expected. Therefore, we are prepared to spend a lot of effort on testing. In our experience, the amount of effort that is usually used on testing is 2/3 of the entire effort needed to produce a design, which is much more than is usually spent on testing software.

Spending that much effort just on testing is a lot. So, the question arises: Can we do better? Can we spend less effort on testing and cover more of the design's behaviour? Note that we are not interested just in theoretical results but in how we can apply them in practice.

Usually, when discussing how to ensure that hardware and software behave as expected, formal methods are mentioned as a promising toolset that can be used alongside the usual testing to achieve better coverage. This makes them an interesting starting point. In order to get a better idea if formal methods are useful in production, we decided to try out open source tools that are currently available and see how far we can get using them.

In the following chapter, we briefly go through the formal methods used by the industry and follow with a description of open source tools that we tested. Note that we have used SymbiYosys [20] more than any other tool as it is the most mature tool that we have encountered. Using it, we were able to formally

verify our AXI-lite interconnect, which is a production-ready digital design.

7.2 Formal methods in the industry

It seems that formal methods are widely adopted among hardware design companies. Some of the biggest companies that use them are Intel [21], AMD [22] and ARM [23]. Unfortunately, it turns out that most of these companies consider their testing practices to be a trade secret and do not often discuss them publicly in detail. Furthermore, the tools they use are partially or fully developed in-house and are not shared with the community. In our experience, this is a trend for most of the tools used in the digital hardware design industry, which is in stark contrast to the much more open-software community where most of the tools used to develop software are open source. The result of that is that there are not many open source tools out there that can be used to formally verify hardware designs in practice, with a few notable exceptions such as SymbiYosys [20]. Also, the available open source tools are often not production ready, which means that significant effort is needed to apply them to existing projects. The reason for this seems to be that the tools are mostly developed in an academic setting.

In general, the formal methods used in the industry fall into one of the following categories:

- fully-automated program analysis,
- model checking, and
- interactive theorem proving.

where the main difference between them is the amount of automation and what can be proven. The slides from Mike Dodds talk 'Proofs in the Wild: What's done today? What's close? What's far?' (December 2024) [24] give a great overview of the pros and cons for each of the categories.

Fully-automated program analysis are a set of fully automated methods that aim to check design's properties at scale. Such methods are, for example, used by advanced linters [25]. The main advantage of using them is that they can be used by non-specialists as they are easy to use, you just run them and get the results, where the downside is that they usually check only specific properties and they often provide false positive and false negative results.

Model checking is a set of methods that translate a design into a set of logical formulas that can be given to a solver to check if the design has a certain property. The translation and checking are fully automated, but the methods are known to work only for relatively small designs. For example, model checking can be used to verify processor arithmetic logic units and interconnects [26]. Model checking methods do not provide false positive or false negative results. Two tools that rely on model checking are SymbiYosys [20] and Siemen's Questa Verify Property [27].

Interactive theorem proving is a set of methods that allow us to describe our design in a formal language and reason about it using a proof assistant: a program that helps us construct proofs and checks that the proofs are valid. An example of such tools is Coq[1] [28], Isabelle [29], F* [30] and ACL2 [31], where three major projects that use interactive theorem proving are formally verified separation kernel seL4 [32], formally verified AAMP7G microprocessor [33], and Project Everest [34] where they are working to provide formally verified parts of a HTTPs stack, for example HACL* [35] - a formally verified library of modern cryptographic algorithms. A major advantage of using interactive theorem proving is that the language that we use to describe the design and its properties is more expressive, which allows us to verify more interesting properties, and that the interactive theorem proving can be applied to medium-sized systems. Furthermore, the proof assistant checks that the proofs are correct, avoiding false positive and false negative results. The downside of using interactive theorem proving is that verification and

[1] Also known as Rocq Prover.

proving require a lot of human effort and expertise; the effort used in such projects is often counted in used PhD years.

7.3 Open source tools

In this chapter, we discuss the open source tools we have tried. Because we design our hardware in Verilog, which is also true for most of the industry, we focused only on the tools that support formal verification of Verilog and SystemVerilog. We started our effort with Symbiyosys [20] as it is one of the most well-known open source tools for the formal verification of digital hardware designs. Also, it is the most mature tool that we found as we were able to use it to formally verify AXI-lite interconnect. For each tool, we will look at how the tool is used (workflow), available documentation, pros and cons, and the tool's maturity.

7.3.1 Symbiyosys

SymbiYosys [20], or *sby* for short, is an open source digital hardware verification tool maintained by YosysHQ [36]. sby is based on model checking and allows one to check if a program has a certain property by asserting the property in Verilog and passing the translated design with assertions to an SMT solver that then checks if the assertion holds. sby supports bounded and unbounded model checking.

Workflow: sby supports the following high-level workflow: (1) take a Verilog design that you want to verify and describe the properties that the design should have by using assert, assume and cover statements (or SystemVerilog assertions if using a paid version of sby), (2) run sby which returns a waveform for each assertion that fails, (3) check the waveform for why the assertion has failed, (4) fix the design and (5) repeat until there are no assertions that fail and you have covered all the properties you wanted to check.

In the background, each time you run sby (2), sby translates your design to a set of logical formulas in SMTv2 format and passes them to an SMT solver such as Z3 [37] and Boolector [38]. The SMT solvers tries to find cases when the formulas are satisfied, this is when one of the assertions in your design fails, and then translates the output of the SMT solver back to a waveform that you can view and figure out why an assertion has failed.

When describing the behaviour of your design, you use Verilog together with assert, assume and cover statements. The assume statement allows you to assume properties of a design and input signals, where the cover statement instructs sby to check if a certain behaviour can actually occur by finding a waveform that covers it. The two statements are usually used together, as we use the cover statement to check that the design can perform an expected behaviour under the specified assumptions; in other words, we use cover statements to make sure that we did not assume something that we did not intend to. For example, if we assume too much, some assertions will be trivially true as the design never comes to a situation where it can potentially violate them. This is necessary as there is no other way to check how the assumptions impact the behaviour of the design.

Pros/Cons: Specifying HW design using Verilog: Verilog is a hardware description language that is expressive enough to describe most of the hardware in a simple and efficient way [2]. However, the same cannot be said for using Verilog to test and formally verify hardware. In our experience, writing more complicated tests, for example, testing if transactions are correctly performed over an interconnect, comes with a significant overhead of used effort and lines of code, which seems to be due to Verilog not providing the needed language abstractions/features to easily implement tests. For example, Verilog lacks the idea of a structure (i.e., record) that can be used to store test-related data. Instead of defining

[2] With some exceptions, for example, connecting hardware modules needs to be done by hand, signal-by-signal, which is error-prone and time-consuming

a structure, you need to define an array where a specific set of bits holds a value. This, in general, is not a problem, as one could argue that the structures in other languages are usually implemented in a similar manner, but the problem arises when one wants to access the values stored in the array as you need to select each value by providing the starting and ending index of the value in the array. This requires that the developer tracks the indexes separately for each stored value, which is error-prone and time-consuming.

The lack of such abstractions (e.g. structures, dynamically sized associative arrays, and local variables) makes writing and maintaining test cases much more complicated than it needs to be. We encounter the same problems when trying to formally verify hardware designs with sby. In our experience, it is hard to describe a behaviour of a design by not having a sufficient level of abstraction, as you spent a lot of effort on trying to fit a relatively simple concepts (e.g. structures and dynamically sized associative arrays) into Verilog in such a way that it can also be efficiently used by the SMT solver. This makes specifying, proving, and maintaining proofs and specifications much more complicated. Ideally, we would use more expressive language, for example Coq [28] and F* [30], to describe and prove the properties of a design.

Some of the aforementioned problems are addressed in later versions of Verilog, called SystemVerilog, but sby does not support it. As far as we can see, the only improvement in expressiveness that sby provides is support for SystemVerilog assertions, which is only available in the paid version of sby.

Pros/Cons: How the assertions are checked: sby supports two ways of checking that assertions hold for a given design: bounded-model checking and proof-by-induction. Bounded-model checking can be used to check if an assertion holds for the first N cycles from the start of the simulation, where proof-by-induction can be used to check the assertion holds for all cycles by using induction. Both approaches have prose and cons.

Bounded-model checking is easy to use as it is to some extent similar to usual testing: the design is reset and then we check if an assertion holds for each clock cycle from the reset onward. In general, the problem with this approach is that the more cycles you check, the more time a SAT solver needs to check that assertions hold. After some ten cycles, the SAT solver might need 30 minutes or more to check that all the assertions hold for a single cycle. The amount of time that the solver need heavily depends on the number of free variables in the formula generated from your design. For each clock cycle, we extend the formula with additional free variables representing input signals of the design in that clock cycle, which means that the solver needs more time to check the formula after each clock cycle. There are ways to mitigate this, for example, by reducing the number of external inputs to the design, but with the larger designs and by wanting to check more cycles, this becomes the main problem you are trying to avoid. In contrast, using proof-by-induction does not have this issue as it only checks a small number of cycles (base step cycles and the induction step cycle).

The problem with the proof-by-induction approach is that we do not start checking if assertions hold from a reset state of the design. sby treats the internal state of the design as free variables and thus also performs induction on the states of the design that cannot be reached from the reset. This prevents us from treating the design as a black-box as we need to explicitly specify what is a valid internal state of the design. In our experience, this requires significant engineering effort. Also, each time that you change the internals of the design, you also need to change the description of what a valid internal state is. This is not the case for bounded-model checking. In our experience, both methods are useful but you need to know when to apply them to get the best result with the least amount of effort.

Documentation: The main documentation available for sby is a reference manual [39], which explains how to use sby and goes through a basic example of how to verify a design. Unfortunately, it does not provide any additional guidance on different approaches to verification; for example, it does not explain what is the difference between bounded and unbounded model checking, and what are the prose and cons of both approaches. It seems that the authors assume that the reader has some prior knowledge of

model checking and how it can be efficiently applied, which results in a high learning curve for a beginner who wants to be efficient with sby. This can be partially addressed by relying on other resources, for example, ZipCPU's blog [40] which has great posts on how to formally verify hardware in practice using sby while also explaining what kind of problems you usually encounter.

Maturity: sby is the most mature open source hardware formal verification tool that we found. It can be used out-of-the-box on small production-ready hardware designs in practice, which is great as it is an open source tool that is free to use. In our experience, it has an active community as YosysHQ regularly responds to issues posted on the project's GitHub page [20]. The workflow provided by sby can be used in production, but in order to scale it efficiently to bigger and more complicated designs, it would be great to refine the workflow and improve documentation. More specifically, we found two points that would make the tool easier to use and more useful in practice.

(1). We encountered two issues related to reporting which of the assertions has failed. The first issue [41] was that sby reported that a wrong assertion has failed when the same module was instantiated two times. This issue was quickly fixed by YosysHQ development team, so it is no longer a problem. The second issue [42] is that sby does not report exactly which assertion has failed when using for loop generate constructs. This means that one is not able to see for which iteration of the for loop the assertion has failed just from sby's log messages. This is not a problem for small designs as one can check the generated waveform, but when the designs get larger, figuring out what went wrong just from the waveform gets harder. This is especially problematic because larger Verilog designs rely on parameters and generators for configuration, so you will often encounter this problem. This causes unnecessary overhead for a routine workflow. Note that this problem is addressed with the Verific frontend available in the paid version of sby.

(2). When an assertion fails, the developer usually checks the generated waveform to figure out why it has failed. A failed assertion usually indicates that something is wrong with the design or with the specification. If something is wrong with the design, the design is fixed and sby is re-run. In our experience, it is hard to see if the problem was actually fixed just from re-running sby because the changes to the design often fail some other assertion. It would be much easier to confirm that a specific part of the design was fixed if a test case would be generated for each assertion that fails. sby currently does not support automatic test generation that work with the original design. We have confirmed that this is a problem with YosysHQ and posted a GitHub issue to note it as a feature request [43].

The automatic test generation might seem as a minor problem, but it is important if you want to avoid being locked to a specific tool. If you can get a test case for each failed assertion, the test case can be used without the tool, which is valuable even after you decide to stop using the tool. As part of the same consideration, it would be great that a specification that you write in Verilog with assume, assert and cover statements could also be used with other tools. As far as we can see, this is currently not possible with the open source tools.

Our contribution: During our investigation, we have reported issues 296 [41], 300 [43] and 306 [42] to SymbiYosys GitHub repository.

As far as we can see, SymbiYosys is currently the only open source tool that can be used out-of-the-box to formally verify small production-ready HW designs written in Verilog based on model checking. We are grateful to YosysHQ for making it available to the community.

7.3.2 EBMC

EBMC is an open source digital hardware verification tool based on model checking that supports Verilog and SystemVerilo where properties can be expressed with a fragment of SystemVerilog assertions. EBMC supports bounded and unbounded model checking.

Workflow: EBMC supports a similar workflow as SymbiYosys: (1) describe the designs behavior using

assertions, (2) run EBMC that returns a trace or waveform for each assertion that failed, (3) check the waveform for why the assertion has failed, (4) fix the design and (5) repeat until no assertion fails.

Pros/cons: Unfortunately, as SymbiYosys, EBMC does not support test generation for failed assertions.

Pros/cons: EBMC supports a fragment of SystemVerilog assertions which is useful as SystemVerilog assertions are more expressive as Verilog with assert statements. Using SystemVerilog assertions makes describing properties of your design less time-consuming and error-prone.

Documentation: EBMC's team provides short documentation on how EBMC can be used, but similar to SymbiYosys, they assume some prior knowledge about model checking and do not elaborate what are the best practices to use their tool, for example, when and how to use bounded and unbounded model checking.

Maturity: As far as we can see, EBMC has a small community of users and is mainly maintained by Daniel Kröning and his team. It aims to support Verilog and SystemVerilog. But after trying to use the tool, we encountered several issues, such as missing language features, that prevented us from using it on production designs; for example, vector part-select addressing was not fully supported [44]. These are minor issues that can be addressed by putting more effort into covering all Verilog and SystemVerlog features; but it is a problem, as it can prevent EBMC from being used in practice as it cannot be applied to all valid Verilog designs.

Our contribution: We have reported issues 751 [45], 747 [46] and 784 [44] to EBMC's GitHub repository [47]. All the issues were quickly addressed by the EBMC team.

7.3.3 Knox

Knox [48] is a formal verification framework that can be used to formally verify hardware and software. Knox allows us to show that a Verilog design implements a functional specification through information-preserving refinement (IPR). If we are able to show that IPR exists, then we know that the design meets the functional specification and no other information is leaked, through timing side-channels, that is not already leaked by the specification.

An interesting insight that we can take from Knox is that one can uses Yosys to translate a Verilog design into an SMTv2 representation, which can then be interpreted in Racket [49] as a domain-specific language. By using Rosette [50], we can then symbolically evaluate the design and verify that the design has the wanted properties.

Rosette [50] is a solver-aided programming language that extends Racket with language constructs for program synthesis and verification. Rosette verifies a design by translating the design into a set of logical constraints that can be solved by an SMT solver. To translate the design, Rosette symbolically evaluates the code to get a reduced set of logical constraints. As far as we can see, this makes verification with Rosette more efficient and thus allows verification of larger designs.

Workflow: Knox supports the following workflow: (1) Translate a Verilog design into SMTv2 form with Yosys. (2) Interpret the translated SMTv2 design in Racket using Knox. (3) Describe and (4) check the desired properties with Rosette. (5) If some of the checks failed, determine why the check failed from the counterexamples provided by Rosette. (6) Repeat until you have checked all the properties.

Using Rosette and Knox we can verify our Verilog designs in a similar fashion as with SymbiYosys: we can check that a property holds for the first N number of clock cycles after the design is reset.

Pros and cons: When using Rosette, we are able to specify the design properties in a subset of Racket, which, as far as we can see, should be easier than specifying the properties in Verilog as Racket provides a higher level of abstraction, for example, it supports structures and dynamically sized associative arrays.

In order to see if this is actually the case, we would need to evaluate this verification approach on larger designs.

Pros and cons: In order to efficiently verify hardware using Knox, one needs to be proficient in (1) Racket, (2) Racket's macro system (the mechanism that allows one to embedded domain-specific languages in Racket), (3) how Yosys translates Verilog designs into SMTv2, (4) how Knox interprets the SMTv2 translation, and (5) how Rosette works, alongside other more general formal verification knowledge that is usually needed to do model checking. In our experience, this is not trivial and requires a large time investment before a developer can efficiently verify a hardware design. Furthermore, as far as we know, Racket is not often used in the industry, which means that a developer will probably need to learn it before it can start with the verification.

Documentation: Unfortunately, Knox does not provide documentation explaining how to get started, but this can be determined through published articles and examples. Because of that, using Knox requires a bit of reverse engineering, especially when you try to determine how Knox interprets the SMTv2 representation of your design. This process is made easier by the great documentation provided by Rosette and Racket.

Maturity: Knox is a research project and, as far as we can see, does not have many users. The project's community is small, and the main contributor to the project is its author, Anish Athalye. In terms of features, the project is interesting, as it allows one to use Racket to write down the specification, and it leverages Rosette to allow fast verification. So far, we have not encountered any problem that would indicate that Knox could not be used on larger designs.

Our contribution: We have discovered one issue [51] and reported it on the Knox's GitHub page.

7.3.4 ACL2

ACL2 [31] is a programming language and an automated theorem prover that allows one to develop programs and prove their properties in a subset of Common Lisp. ACL2 is well known for its support for hardware verification and is also used in the industry, for example, by Intel, AMD and Centaur Technology [52]. ACL2 supports symbolic evaluation, and its proof system is based on term rewriting rules that are extended once the user proves a theorem.

Workflow: ACL2 with SV library [53] supports the following workflow: (1) Translate a Verilog design into an ACL2 representation. (2) Describe the design's properties in ACL2 and (3) prove that the properties hold.

ACL2 belongs to a family of interactive theorem provers that allow one to prove properties of a design by guiding the theorem prover. This means that proofs are more involved and require more effort as, to some extent, they need to be done by hand. This is in contrast to model checking where in order to prove the properties you just pass the property to a program that just confirms that the property holds or provides a counterexample. The advantage of using theorem provers is that we can prove more complicated properties and work with larger designs.

Unfortunately, we did not try out ACL2 and verify a design with it, but we mention it here because it seems as a promising option if one wants to verify HW using proof assistants.

Documentation: As far as we can see, ACL2 has great documentation [54] that covers how to use ACL2 and how to start with hardware verification [53].

Maturity: As far as we can see, ACL2 has a small but responsive community. The tool seems to be industrially tested, which makes it an interesting candidate for further verification attempts. Currently, the only downside that we can see for using ACL2 is that it has a steep initial learning curve, as the developer needs to become familiar with the ACL2 language and they also need to learn how to prove theorems with it.

8 The CROSSCON Certification Manifest

In this chapter, we present the *CROSSCON Certification Manifest*, a data structure which is designed to support formal reasoning about the safety and behavioral properties of the applications running on the CROSSCON stack. This data structure has been integrated in the CROSSCON Secure Update framework that is being developed as part of the WP3 activities, whose outcomes will be reported in the deliverable 3.3.

8.1 Structure of the Certification Manifest

The Certification Manifest is a structured list of formal properties satisfied by a binary application, together with a corresponding list of *proof certificates*. A proof certificate is a textual representation of a formal proof, written in some specified formal language [55]. Their use enables the agent in charge of the security of a device to directly verify the validity of such properties before executing (or installing) the application.

For the sake of clarity, we shall distinguish between a proof certificate, in the sense defined above, and a *proof descriptor*, which is a data structure whose purpose is to encapsulate a proof certificate together with all the information needed to verify it.

Field name	Field type
Property identifier	string
Component identifiers	list of strings
Language identifier	string
Proof certificate	string
Locality constraint	boolean
Verification servers	list of strings

Table 2: Fields in a proof descriptor.

The CROSSCON Certification Manifest consists of a list of proof descriptors. Each proof descriptor is a record consisting of the fields listed in Table 2. Let us explain in more detail the intended semantics of these fields.

- The **Property identifier** field contains a string that is used to uniquely identify the formal property to which the proof descriptor refers. Each property identifier must appear only once in each Certification Manifest.
- The list of **Component identifiers** is used to specify which components running on the device are targeted by the considered proof. Each element of this list must uniquely identify a corresponding software component in the CROSSCON stack.
- The **Language identifier** field contains a string that uniquely identifies the formal language in which the proof is expressed (including a version number if necessary). By requiring each proof descriptor to specify the language used in the corresponding proof, our proposal can remain fully agnostic about the particular tools that are used to generate proof certificates.
- The **Proof certificate** field contains the proof certificate itself, expressed in the formal language singled out by the Language identifier.
- The **Locality constraint** is a Boolean flag that, when set, requires the proof checking step to be per-

formed on the device, without involving any communication with the external world. This possibility can be useful for sufficiently powerful devices that, nevertheless, have no or very restricted access to the Internet, or when a verification server cannot be set up in a trusted way.

- Finally, the **Verification servers** field contains a (possibly empty) list of *verification servers* that can be used to verify the validity of the attached proof certificate. Clearly, this field only becomes relevant when the locality constraint flag is not set. Each string in the list is required to be a valid URI of a remote server that can be queried to perform the verification of the associated proof certificate. Again, specifying the servers on a proof-by-proof basis is useful because some languages and/or proof techniques may be supported only by some of the available backends. The presence of this list is optional and even when nonempty, its contents are meant to be advisory only; the owner of the device can always ignore the suggestions and require the use of a trusted server of his choice when it is available.

8.2 Purposes of the Certification Manifest

The Certification Manifest is a versatile structure that can be used for many purposes in the CROSSCON stack.

For instance, in the Secure Update process (whose final version will be described in more detail in deliverable D3.3), we propose to embed a Certification Manifest inside each update package. This Certification Manifest will contain a set of formal proofs concerning the update's contents. These proofs can be of two kinds:

- *standalone proofs*, aimed to certify that the code contained in the update satisfies some selected property;
- *differential proofs*, aimed to certify that the new component version replaced by the update satisfies the same properties as the old version that already runs on the device.

The Certification Manifest could also be integrated into other components of the stack.

9 Conclusions

In this document, we have presented the current version of the open formalization of the CROSSCON separation kernel and the security properties we have considered within CROSSCON. We started with the individual architectural components of CROSSCON and the possible implementation alternatives of the CROSSCON stack discussed in [2]. We proved that the CROSSCON separation kernel satisfies the properties of interest. For TEE and Hypervisor-less hardware, we performed a more detailed formalization, and we mechanically proved memory isolation of the design leveraging model checking techniques and satisfiability modulo theory. We included the formalization of the shared memory and the dynamic creation of sub-domains within a domain. Moreover, we also formalized the correctness of the CROSSCON hypervisor configuration file through SMT with the extraction of proofs to be added to the certification manifest. We also formalized the finite-state machine governing the creation of virtual machines (sub-domain) and formally verified some properties on this machine. We described the experience of using open-source verification tools to verify hardware components. Finally, we described the extension of an existing certification manifest to include CROSSCON-specific components (namely, proofs for security properties or bills of material).

As future work, our aim is to maintain this document to incorporate future extensions and to submit an excerpt of its content to scientific venues to valorize the research carried out.

References

[1] Marco Roveri, Jurij Mihelič, and Alberto Tacchella. *D2.2: CROSSCON Formal Framework - Draft*. 2023.

[2] Markus Miettinen, Shaza Zeitouni, Marco Roveri, Michele Grisafi, Žiga Putrle, João Sousa, David Cerdeira, Sandro Pinto, and Lukas Petzi. *D2.1: CROSSCON Open Specification - Draft*. 2023.

[3] Marco Roveri, Emanuele Beozzo, Michele Grisafi, Alberto Tacchella, Žiga Putrle, João Sousa, David Cerdeira, and Nikhilesh Singh. *D2.3: CROSSCON Open Specification - Final*. 2025.

[4] Ainara García, David Purón, Bruno Crispo, Hristo Koshutanski, Krystian Hebel, Maciej Pijanowski, Michał Żygowski, and Rafał Kochanowski. *D1.1: Use Cases Definition Initial Version*. 2023.

[5] David Purón, Ainara García, Rafał Kochanowski, Ziga Putrle, Yacine Felk, Emna Amri, Akos Milankovich, Gergely Eberhardt, Sandro Pinto, Bruno Crispo, Marco Roveri, Michele Grisafi, and Peter Ten. *D1.2: Requirements Elicitation Initial Technical Specification*. 2023.

[6] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive formal verification of an OS microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014), 2:1–2:70.

[7] *Sel4 - The Proof*. [Online]. Available: https://sel4.systems/Info/FAQ/proof.pml. Jan. 2024.

[8] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. *CHERIoT: Rethinking security for low-cost embedded systems*. Tech. rep. MSR-TR-2023-6. Microsoft, Feb. 2023. URL: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/.

[9] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture*. Tech. rep. UCAM-CL-TR-864. University of Cambridge, Computer Laboratory, Dec. 2014. DOI: 10.48456/tr-864. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-864.pdf.

[10] David Greve, Matthew Wilding, and W Mark Vanfleet. "A separation kernel formal security policy". In: *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*. 2003.

[11] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[12] John M. Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies 1*. Tech. rep. SRI International, 2005. URL: https://api.semanticscholar.org/CorpusID:8472202.

[13] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018.

[14] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1267–1329.

[15] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. "Automated assumption generation for compositional verification". In: *Formal Methods Syst. Des.* 32.3 (2008), pp. 285–301.

[16] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 3843–3860. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/grisafi.

[17] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 334–342.

[18] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - specification*. Springer, 1992. ISBN: 978-3-540-97664-6. DOI: 10.1007/978-1-4612-0931-7.
[19] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. "Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System". In: *CAV*. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 378–393.
[20] YosysHQ. *SimbiYosys GitHub repository*. 2025. URL: https://github.com/YosysHQ/sby (visited on 02/28/2025).
[21] *Micro-architectural modelling and verification of an x86 micro-processor*. 2024. URL: https://www.newton.ac.uk/seminar/44405/ (visited on 10/28/2024).
[22] David M Russinoff. "A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the amd athlon tm processor". In: *International Conference on Formal Methods in Computer-Aided Design*. Springer. 2000, pp. 22–55.
[23] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. "End-to-end verification of processors with ISA-Formal". In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer. 2016, pp. 42–58.
[24] *Proofs in the Wild: What's done today? What's close? What's far?* 2025. URL: https://mikedodds.github.io/files/talks/2024-12-5-proofs-in-the-wild.pdf.
[25] *Questa Inspect*. 2025. URL: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/autocheck/.
[26] *2019: AXI Meets Formal Verification*. 2019. URL: https://zipcpu.com/blog/2020/01/01/2019-in-review.htm.
[27] *Questa Verify Property*. 2025. URL: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/property-checking/.
[28] *The Rocq Prover*. 2025. URL: https://rocq-prover.org/ (visited on 02/28/2025).
[29] *Isabelle*. 2025. URL: https://isabelle.in.tum.de/ (visited on 03/08/2025).
[30] *Fstar*. 2025. URL: https://fstar-lang.org/ (visited on 03/08/2025).
[31] *ACL2*. 2025. URL: https://www.cs.utexas.edu/~moore/acl2/.
[32] Gernot Heiser. "The sel4 microkernel–an introduction". In: *The seL4 Foundation* 1 (2020).
[33] Matthew M Wilding, David A Greve, Raymond J Richards, and David S Hardin. "Formal verification of partition management for the AAMP7G microprocessor". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 175–191.
[34] *Project Everest*. 2025. URL: https://project-everest.github.io/ (visited on 03/08/2025).
[35] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A verified modern cryptographic library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1789–1806.
[36] YosysHQ. *Home page of YosysHQ team*. 2025. URL: https://www.yosyshq.com/ (visited on 02/28/2025).
[37] Microsoft Research. *Z3 theorem prover GitHub repository*. 2025. URL: https://github.com/Z3Prover/z3 (visited on 02/28/2025).
[38] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: 10.3233/sat190101. URL: https://doi.org/10.3233/sat190101.
[39] YosysHQ. *SymbiYosys (sby) Documentation*. 2025. URL: https://symbiyosys.readthedocs.io/en/latest/ (visited on 02/27/2025).
[40] Gisselquist Technology. *ZipCPU's blog*. 2025. URL: https://zipcpu.com/ (visited on 02/27/2025).
[41] Žiga Putrle. *Sby Github issue 296: Wrong assumption reported in summary*. 2025. URL: https://github.com/YosysHQ/sby/issues/296 (visited on 02/28/2025).
[42] Žiga Putrle. *Sby Github issue 306: Not able to identify which assertion failed when using generators*. 2025. URL: https://github.com/YosysHQ/sby/issues/306 (visited on 02/28/2025).

[43] Žiga Putrle. *Sby Github issue 300: Generated test banches cannot be used with Xcelium or Iverilog*. 2025. URL: https://github.com/YosysHQ/sby/issues/300 (visited on 02/27/2025).
[44] Žiga Putrle. *EBMC Github issue 784: Expression not recognized as constant when using indexed part-sele*. 2025. URL: https://github.com/diffblue/hw-cbmc/issues/784 (visited on 02/28/2025).
[45] Žiga Putrle. *EBMC Github issue 751: Expression not recognized as constant generators*. 2025. URL: https://github.com/diffblue/hw-cbmc/issues/751 (visited on 02/28/2025).
[46] Žiga Putrle. *EBMC Github issue 747: loop_generate_construct not supported*. 2025. URL: https://github.com/diffblue/hw-cbmc/issues/757 (visited on 02/28/2025).
[47] *EBMC Github repository*. 2025. URL: https://github.com/diffblue/hw-cbmc (visited on 02/28/2025).
[48] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Hardware Security Modules with Information-Preserving Refinement". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. Carlsbad, CA, July 2022.
[49] *Racket language homepage*. 2025. URL: https://racket-lang.org/.
[50] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 135–152.
[51] Žiga Putrle. *Knox Github issue 1: input* assigns Bool but the rest of the SMT2 model expects (BitVec 1)*. 2025. URL: https://github.com/anishathalye/knox/issues/1.
[52] Warren A Hunt Jr, Matt Kaufmann, J Strother Moore, and Anna Slobodova. "Industrial hardware and software verification with ACL2". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017), p. 20150399.
[53] *ACL2 SV tutorial*. 2025. URL: https://www.cs.utexas.edu/~moore/acl2/v8-6/combined-manual/index.html?topic=SV____SV-TUTORIAL.
[54] *ACL2 User manual*. 2025. URL: https://www.cs.utexas.edu/~moore/acl2/v8-6/combined-manual/index.html?topic=ACL2____TOP.
[55] Haniel Barbosa, Clark W. Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. "Generating and Exploiting Automated Reasoning Proof Certificates". In: *Commun. ACM* 66.10 (2023), pp. 86–95. DOI: 10.1145/3587692.