Cross-platform Open Security Stack for Connected Device

D2.2 CROSSCON Formal Framework - Draft

Document Identification			
Status	Final	**Due Date**	29/02/2024
Version	1.0	**Submission Date**	20/02/2024

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

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

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	28/11/2023	M. Roveri (UNITN)	Initial draft structure.
0.2	09/01/2024	M. Roveri (UNITN)	Initial draft of chapters 1 and 2.
0.3	10/01/2024	M. Roveri (UNITN), J. Mihelič (BEYOND)	Initial draft of CROSSCON separation kernel formalization.
0.4	16/01/2024	J. Mihelič (BEYOND)	Revised chapter 3.
0.5	16/01/2024	M. Roveri (UNITN)	First version chapter 1.
0.6	21/01/2024	J. Mihelič (BEYOND)	Further revision of chapter 3.
0.7	22/01/2024	M. Roveri (UNITN)	Finalized first draft of chapter 4.
0.8	22/01/2024	M. Roveri (UNITN)	Finalized first draft of chapter 5.
0.9	23/01/2024	M. Roveri (UNITN)	Overall pass to fix many typos.
0.10	24/01/2024	M. Roveri (UNITN)	Wrote the executive summary, document structure, introduction to section 3.4.
0.11	29/01/2024	J. Mihelič (BEYOND)	Revised chapter 2 and 3.
0.11	29/01/2024	M. Roveri (UNITN)	Revised chapters 2, 3, and 5.
0.12	29/01/2024	A. Tacchella (UNITN)	Revised chapter 3.
0.13	30/01/2024	M. Roveri (UNITN)	Fixed some descriptions in chapter 3.
0.14	31/01/2024	J. Mihelič (BEYOND)	Revised chapter 3.
0.15	05/02/2024	M. Roveri (UNITN)	Revised chapters 1, 2, executive summary to address comments from the internal review.
0.16	06/02/2024	J. Mihelič (BEYOND), M. Roveri (UNITN)	Revised chapters 3 and 4 to address comments from the internal review.
0.17	15/02/2024	M. Roveri (UNITN)	Final version and approved, ready for quality control.
0.18	20/02/2024	Juan Andrés Alonso (ATOS)	Final version based on quality control.
1.0	20/02/2024	H. Koshutanski (ATOS)	Final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Marco Roveri	15/02/2024
Quality Manager	Juan Andrés Alonso (ATOS)	20/02/2024
Project Coordinator	H. Koshutanski (ATOS)	20/02/2024

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
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
MCU	Micro Controller Units
REE	Rich Execution Environment
RTOS	Real-Time Operating System
SMT	Satisfiability Modulo Theory
TCM	Trusted Computing Module
TEE	Trusted Execution Environment

Executive Summary

The purpose of this document is to provide the basis for the definition of a formal specification framework for the design and deployment of trusted Internet of Things (IoT) applications considering all different aspects. We will also investigate (as a future work for the rest of the project activities) the structure of a ”certification manifest” to accompany any application. In particular, here, we will perform a preliminary definition of the concepts of safety and assurance for IoT applications. We will take advantage of temporal logic specifications complemented with security concepts to prove security properties. We will rely on existing frameworks and verification tools (possibly adapted) to address the identified verification problems. In this document, we present the initial draft of the CROSSCON security properties and the initial formalization of the CROSSCON separation kernel together with manual proofs of some of the properties. Furthermore, we provide formalization and automatic proofs for the case where we use TEE and virtualization-less architecture together.

1 Introduction

The purpose of this document is to provide preliminary results related to the formalization of the CROSSCON stack considering the results of task 2.1 which aims to provide the initial picture of the CROSSCON architecture. The role of this document is to provide an initial draft towards the formalization of the CROSSCON stack, discussing in detail the security properties we aim for, and their initial formalization in the scope of the CROSSCON separation kernel.

This document will be used as a basis for the formal technical work of the project and will be subsequently refined to a final formal specification later in 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 [1], the result of the first working period of Task 2.1. In turn, it is related to the definitions of use cases in the deliverable D1.1 [2] and the technical specification of the corresponding general requirements as documented in the deliverable D1.2 [3].

The work packages on the CROSSCON security stack (WP3) and extensions for accommodating domain-specific hardware in CROSSCON (WP4) will benefit from the formal specification laid out in this document.

The draft of the formal specification will be updated during the project to accommodate possible changes and insights gained during the project work and documented as a final version of the CROSSCON formal specification in the deliverable D2.4.

1.2 The CROSSCON high-level architecture

As thoroughly discussed in Deliverable D2.1 [1], the current CROSSCON high-level abstract architecture can be graphically represented as shown in Figure 1.

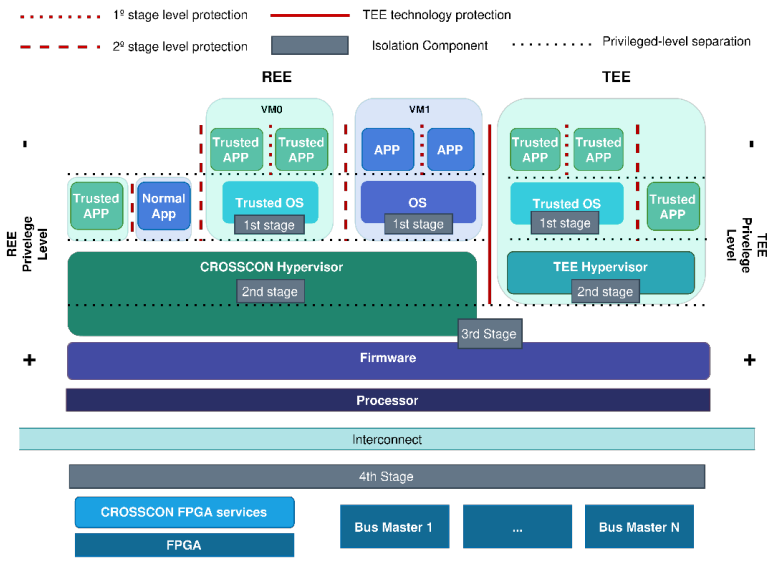

Figure 1: The CROSSCON high-level architecture.

As discussed in D2.1 [1], we distinguish two high-level cases: i) deployment in the case of a Rich Execution Environment (REE) with different REE privilege levels or ii) deployment in the case of the presence of a Trusted Execution Environment (TEE) also with different TEE privilege levels. In both cases, at the lower levels of the CROSSCON stack, we have the ISA that varies depending on the CPU architecture families (E.g., RISC-V, Arm with all their variants) considered, each equipped with possibly different capabilities (E.g., the presence of a TEE, privilege levels). On top of the Instruction Set Architecture (ISA), we consider the possible presence of firmware (which may exist or not, depending on the CPU architecture and the needs of the above 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 initial draft of the security properties. In Chapter 3 we discuss the first draft 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. Finally, in Chapter 5 we draw conclusions for this period and outline future work.

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 [4], 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.

We notice that from a thorough reading of the documentation, 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 [5]:

- The hardware operates as intended, meaning we assume 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 [5] for additional details.

2.1.2 CHERIoT security properties

Capability Hardware Extension to RISC-V for Internet of Things (CHERIoT) [6] is an Instruction Set Architecture and software model built on top of CHERI [7][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, for example, MIPS and Arm, CHERIoT is tight to RISC-V architectures only, and extension to other families is not trivial. We remark that being CHERIoT based on RISC-V, the proposed approach is limited to RISC-V processors only, and extension to other families is not trivial. We remark that 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) [8] 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 [9]. 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 integrity of Y': Y cannot be corrupted by X through M.

It seems natural 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 prohibits the expression of special high-integrity users who are trusted to downgrade the information, for example, by declassifying it from the high to low mark. Therefore, the so-called *intransitive noninterference* (non-interference under an intransitive security policy has been studied in the literature [10].

2.2 The CROSSCON proposal

For the CROSSCON approach, we will adopt a slight variant of the seL4 classical security properties: confidentiality, integrity, and availability. This choice is motivated by the fact that these properties are not as generic as those of the GWV model and are not as specific to RISC-V as in the CHERIoT case. In addition, confidentiality and integrity are both captured in the CIA triad, which is an established model designed to guide policies for information security. Thus, this allows us to properly capture the spirit of the CROSSCON stack to be independent of the low-level hardware architecture. 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 having been explicitly given the entity read access to the data.

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 the authorized use of resources by another entity. This is similar to the case of seL4, meaning that the different CROSSCON elements will eventually get the resources needed if they are 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.

Noninterference: 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 sections), we will provide formal counterparts of the above natural language security properties. We remark that this is an initial draft and that we may revise this initial proposal throughout the remaining time of the project.

In the formalization, similarly to the other considered approaches, we will assume the following:

- Unless strictly necessary we keep temporal aspects out of the formal specification, thus considering static properties.
- We assume that Hardware (HW) behaves correctly.
- To start with, 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 some of the security properties, we envisage leveraging on adaptation of existing verification techniques, for example, model checking [11], theorem proving, for example, Satisfiability Modulo Theory (SMT) [12] and compositional reasoning [13], along the lines followed in the preliminary work on limited HW architectures [14].

3 The CROSSCON separation kernel formalization

In this chapter, we present our approach towards a 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 mechanisms, such as support for at least two processor privilege levels and a memory protection unit or memory management unit, the implementation of the separation kernel is usually based on the hardware and software co-designed approach. However, 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.

The environment provided by the separation kernels consists of one or more *domains* isolated via *protection mechanisms*. We offer (and later in this document 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. Afterwards, 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

$$\sigma = (M, R, PC, PL, \mathcal{D})$$

where the components of the state σ are as follows:

- A mapping $M : A_M \to V_M$ 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 $R : A_R \to V_R$ 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 $PL : \{\mathsf{S}, \mathsf{U}\}$ that identifies the current privilege level where S indicates the privileged level (i.e., separation kernel) and U indicates the unprivileged level (i.e., user).
- A variable $\mathcal{D} : \{1, \ldots, N\}$ 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. For simplicity, 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.,*

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

The idea of the following assumption is to simplify the semantics of machine instructions; e.g., incrementing the program counter register changes it to the next instruction.

Assumption 3. *Uniform encoding and representations: machine instructions and operands, etc., use W bits to represent.*

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 state.

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

$$\sigma = (M, \sigma'_1, \ldots, \sigma'_P) \quad \text{where} \quad \sigma'_i = (R_i, PC_i, PL_i, \mathcal{D}_i).$$

Here, σ_i is a sub-state of the i-th core.

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 of a multi-core state into a single-core state, i.e.,

$$\sigma_i \equiv \sigma(M, \sigma'_i).$$

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

3.2.3 Core assignment

The component $\mathcal{D}$ of the single-core state σ specifies the active domain. Depending on this setting, the separation kernel configures the corresponding protection mechanisms. If, for a particular application, multiple domains are required, then domain switching must also be implemented. To put it differently, for a single-core machine running an application that requires multiple domains, only dynamic core assignment makes sense.

Now, consider a multi-core machine where the component $\mathcal{D}$ of the i-th sub-state σ_i specifies the active domain for the i-th core. The inverse of getting the domain for the i-th core is to obtain all the cores assigned to a given domain $d \in \{1, \ldots, N\}$. For this purpose, we define a function

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

In our model, we do not allow sharing of the cores between different domains. We state this in the following assumption.

Assumption 4. *No two domains share a core, that is,*

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

Core assignment can be *static*, that is, fixed throughout execution, as well as *dynamic*, that is, variable during execution. In the latter case, every change of $\mathcal{D}$ must still obey Assumption 4.

Assumption 5. *Static core assignment disallows changing the active domain, that is, the component $\mathcal{D}$ of a state (or a sub-state). The dynamic core assignment allows one to change the active domain if the Assumption 4 is satisfied.*

The consequence of static core assignment is that DOMSWITCH machine instruction cannot be used (see later in Section 3.6.5). We leave as future work the extension to support dynamic core assignment.

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 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:

- Code sub-region represented by a set $C_d \subseteq A_d$ of addresses and containing the instructions that can be executed.
- Data sub-region represented by a set $D_d \subseteq A_d$ of addresses and containing data that can be read or written to.

Assumption 6. *Separation of memory regions:*

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

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

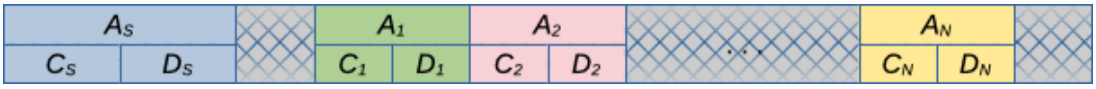

Figure 2: An example of a memory map.

From these assumptions follow some observations regarding the disjointness of the specific sub-regions.

Lemma 1. *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 7. *Statically assigned memory-mapped I/O, that is,*

$$E_d \subseteq D_d \quad \text{for all domain} \quad 1 \leq d \leq N.$$

3.3.3 Effective domain

The separation kernel provides domain separation by isolating memory regions. In particular, exactly one domain is active at a time, which means that in addition to the memory region A_0 of the separation kernel, the memory region $A_{\mathcal{D}}$ of the domain identified by the register $\mathcal{D}$ is also accessible. Of course, the access rules for these two regions, i.e., A_0 and $A_{\mathcal{D}}$, depend on the current privilege level.

When there is 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, other conditions may also cause the computation to halt.

To model instructions that our model is able to execute we assume 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: an instruction itself is read from the memory.

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

Unprivileged level: $PL = \mathsf{U}$

- R – read: Data may be loaded from regions $C_{\mathcal{D}}$ or $D_{\mathcal{D}}$.
- W – write: Data may be stored in the region $D_{\mathcal{D}}$.
- X – execute: Instructions may be fetched from the region $C_{\mathcal{D}}$.

Privileged level: $PL = \mathsf{S}$

- 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.

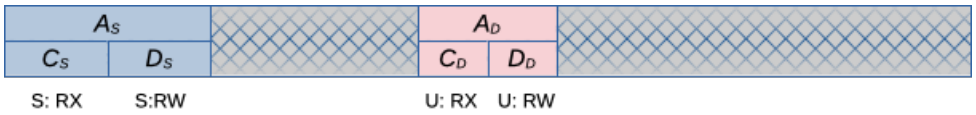

Figure 3: The protection provided by separation kernel.

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 CROSSCON Hypervisor never allows or performs such access.

Assumption 8. *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.,

- $\mathrm{canr}(a) \equiv PL = \mathsf{U} \wedge a \in A_{\mathcal{D}} \vee PL = \mathsf{S} \wedge a \in A_S$,
- $\mathrm{canw}(a) \equiv PL = \mathsf{U} \wedge a \in D_{\mathcal{D}} \vee PL = \mathsf{S} \wedge a \in D_S$,
- $\mathrm{canx}(a) \equiv PL = \mathsf{U} \wedge a \in C_{\mathcal{D}} \vee PL = \mathsf{S} \wedge a \in C_S$.

Assumption 9. *Protection mechanism guarantees. An instruction*

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

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

3.4 Instruction semantics

In the following, without loss of generality, we will consider three main instructions, namely: LOAD r, a (that load on the register r the content of the memory at address a), STORE r, a (that writes in the memory at address a the content of the register r), and JUMP a (that modifies the program counter

to continue the execution at the value specified at address a[1]). These instructions, together with instructions to perform arithmetic operations (e.g., addition, subtraction, shift) on registers, are sufficient to model any possible complex instruction. Thus, in the following, we will consider only these three types of instruction and will use $\mathsf{I}(a)$ to indicate an instruction that requires access to the address a (for reading, writing, or executing).

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 $[\![\cdot]\!] : \sigma \to \sigma$, which maps a state to a state (that is, it specifies how the instruction modifies the state to which it is applied). In the definitions, we usually omit σ and 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 9.

Assumption 10. *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* $M[a]$*, is bound to the* $\mathsf{canr}(a)$ *predicate, i.e.,*
$$\neg\mathsf{canr}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$
- *Write obligation: Any write to the address* a*, i.e., writing to the value* $M[a]$*, is bound to the* $\mathsf{canw}(a)$ *predicate, i.e.,*
$$\neg\mathsf{canw}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$
- *Fetch obligation: A fetch of the instruction* $\mathsf{I}(a)$ *from the address* a*, i.e., fetching from the* $M[a]$*, is bound to the* $\mathsf{canx}(a)$ *predicate, i.e.,*
$$\neg\mathsf{canx}(a) \Rightarrow [\![\mathsf{I}(a)]\!] = \bot.$$

We remark that this assumption is a direct trivial refinement of Assumption 9, and this statement is a trivial consequence of the two assumption's formulations.

3.4.2 Ordinary (unprivileged) instructions

The definitions of the selected instructions are as follows.

$$[\![\mathsf{LOAD}\ r, a]\!] \equiv \begin{cases} R[r] \leftarrow M[a] & \mathsf{canx}(PC) \wedge \mathsf{canr}(a) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{STORE}\ r, a]\!] \equiv \begin{cases} M[a] \leftarrow R[r] & \mathsf{canx}(PC) \wedge \mathsf{canw}(a) \\ \bot & \text{otherwise} \end{cases}$$

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

Consider instruction LOAD. It is easy to check that it satisfies all three obligations:

[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.

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

Other more complex instructions can be modeled in terms of the three simple instructions considered, or their semantics can be defined to satisfy the Assumption 10. For example, consider an instruction LOAD.REL r_d, r_s, a that loads in the register r_d the content of the memory at the location specified by $R[r_s] + a$, where r_s is a register and a is a memory address.

$$[\![\text{LOAD.REL } r_d, r_s, a]\!] \equiv \begin{cases} R[r_d] \leftarrow M[a + R[r_s]] & \text{canx}(PC) \wedge \text{canr}(a + R[r_s]) \\ \perp & \text{otherwise} \end{cases}$$

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 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

Observe that 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 2. *Consider an instruction* I(a) *that accesses (read/write/fetch) the memory at the location* $a \in A_M$. *If* $a \notin A_S \cup A_{\mathcal{D}}$, *then* $[\![\text{I}(a)]\!] = \perp$.

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

3.5.1.2 Valid access

Therefore, since the addresses $a \notin A_S \cup A_{\mathcal{D}}$ are invalid, let us focus on the addresses $a \in A_S \cup A_{\mathcal{D}}$ and see if they are all valid. We consider $a \in C_S \cup D_S \cup C_{\mathcal{D}} \cup D_{\mathcal{D}}$, 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 $\perp$, then the access is invalid, and $[\![\text{I}(a)]\!] = \perp$. The main two rows, denoted by SK and DOM, contain the rules for the regions of the separation kernel and the effective domain, respectively.

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 $PL = \text{S}$ must hold for

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	$A_\mathcal{D}$	$C_\mathcal{D}$	$PL = \mathsf{U}$	$\bot$	$PL = \mathsf{U}$	$C_\mathcal{D}$
		$D_\mathcal{D}$	$PL = \mathsf{U}$	$PL = \mathsf{U}$	$\bot$	

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, $a \in A_S \cup A_\mathcal{D}$.

3.5.2.1 Read access

Lemma 3. *When an instruction* $\mathsf{I}(a)$ *successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \bot$*, reads from the memory at address* $a \in A_S \cup A_\mathcal{D}$*, we have*

$$a \in A_S \iff PL = \mathsf{S} \quad \text{and} \quad a \in A_\mathcal{D} \iff PL = \mathsf{U}.$$

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

$$PL = \mathsf{U} \wedge a \in A_\mathcal{D} \quad \vee \quad PL = \mathsf{S} \wedge a \in A_S$$

holds.

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

3.5.2.2 Write access

Lemma 4. *When an instruction* $\mathsf{I}(a)$ *successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \bot$*, writes to the memory at address* $a \in A_S \cup A_\mathcal{D}$*, we have*

$$a \in D_S \iff PL = \mathsf{S} \quad \text{and} \quad a \in D_\mathcal{D} \iff PL = \mathsf{U}.$$

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

$$PL = \mathsf{U} \wedge a \in D_\mathcal{D} \vee PL = \mathsf{S} \wedge a \in D_S$$

holds.

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

3.5.2.3 Fetch access

Lemma 5. *When an instruction* $\mathsf{I}(a)$ *is successfully, i.e.,* $[\![\mathsf{I}(a)]\!] \neq \perp$*, fetched from the memory at address* $a \in A_S \cup A_{\mathcal{D}}$*, we have*

$$a \in C_S \iff PL = \mathsf{S} \quad \text{and} \quad a \in C_{\mathcal{D}} \iff PL = \mathsf{U}.$$

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

$$PL = \mathsf{U} \wedge a \in C_{\mathcal{D}} \vee PL = \mathsf{S} \wedge a \in C_S$$

holds.

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

Corollary 6. *If* $a = PC$ *then we have*

$$PC \in C_S \iff PL = \mathsf{S} \quad \text{and} \quad PC \in C_{\mathcal{D}} \iff PL = \mathsf{U}.$$

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 $\mathsf{canw}(a)$ 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 $PL = \text{S}$ 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 $PL = \text{U}$ for $a \in D_{\mathcal{D}}$.

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 $a \in C_{\mathcal{D}}$, 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 2, any instruction semantics equals $\perp$.

3.5.3.2 Confidentiality

Confidentiality often refers to a security guarantee of any kind of access to objects, i.e., *an object cannot be accessed by non-authorized subjects*, or, in other words, *a subject must be authorized to access an object*.

Taking into account our model, the access to the memory is monitored through read, write, and fetch operations, respectively, the functions canr(a), canw(a), and canx(a). Since write is just another type of access, the integrity is subsumed by confidentiality by including the integrity constraints. Additional 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 in 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 (and written to) at the privileged level. The row $a \in D_S$ contains $PL = \text{S}$ in the read (and write) column.

Domain data confidentiality: Domain data memory can be read from (and written to) at the privileged level. The row for $a \in D_S$ contains $PL = \text{U}$ in the read (and write) column.

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

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

We can define a *stricter form of confidentiality* by observing that, generally, a confidentiality guarantee considers only accesses that reveal the data. However, in our case, we can easily extend this definition to also consider data mutation by observing the write column of Table 1. Here, we observe that canw() and also canx() represent a stricter form of access rights than canr(). In other words, if canr() can provide confidentiality, then canw() and canx() can also. We state this more formally with the following lemma.

Lemma 7. *For all $a \in A_M$ we have*

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

Hence, one could argue for confidentiality (but with less granularity, so only for memory regions A_S and $A_{\mathcal{D}}$) only by observing the read column of Table 1.

3.5.3.3 Isolation

First, we consider the separation kernel and show that its memory region is always accessed by a code running at the privileged level.

Corollary 8. *Consider an instruction* $\mathsf{I}(a)$ *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 3 and 4 together with Corollary 6. □

Next, we consider the active domain and show the same property in a similar way.

Corollary 9. *Consider an instruction* $\mathsf{I}(a)$ *that is fetched from the* PC *address and that performs read or write access to the address* $a \in A_{\mathcal{D}}$*. We have*

$$a \in A_{\mathcal{D}} \iff PL = \mathsf{U} \iff PC \in C_{\mathcal{D}}.$$

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

Corollary 10. *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* $i \neq \mathcal{D}$*, then* $\mathsf{I}(a) = \bot$*.*

Proof. Consider a proof for read access. Since $i \neq 0$ and $i \neq \mathcal{D}$, $\text{canr}(a)$ 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 a concept of safe storage that is used to store and retrieve the execution context of domains. The storage is safe in the sense that ordinary instruction 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:

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

Here, the former stores the execution context of the active domain in its respective safe storage, while the latter retrieves it.

Assumption 11. *Invariance of the push/pop functions. Let X be a property of state σ, e.g., a component PC, PL, or D. We assume*

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

3.6.2 Interrupt handling

In the following, we analyze the assumptions and the semantics of interrupt handling in the defined CROSSCON separation kernel. To formalize interrupt handling, we consider the following assumptions.

Assumption 12.

- *We assume non-reentrant interrupts. When the interrupt handler is triggered, the interrupts are disabled until the handler finishes.*
- *The call to the interrupt handler is performed with a virtual instruction* IRQ_i*, where i is the interrupt number.*
- *The interrupt handler returns via the* IRET instructions.
- Interrupts are disabled while the handler is running, that is, IRQ disables interrupts and IRET enables them.
- We implicitly model interrupt enable/disable by ensuring that the handler code does not include IRQ instructions.
- The IRQ instruction can be arbitrarily inserted after every instruction if interrupts are enabled, that is, after instructions that are not part of the interrupt handler.
- We assume that the interrupt handler code is correct: its final instruction is IRET, and no virtual IRQs are inserted.
- Assume that $intentry_i \in C_S$ is the interrupt entry point, that is, the address of the handler for the interrupt number i.

In real systems, an interrupt may arrive anytime, and it is handled in between two instructions. We model this by a virtual instruction that can be inserted anywhere in the program trace. However, since during the execution of the interrupt handler code the interrupts are usually disabled, we disallow the insertion of this virtual instruction into its code.

We define the virtual instruction representing the call of the interrupt handler as

$$[\![\text{IRQ}_i]\!] \equiv \begin{cases} \text{pushContext}(\sigma) \\ PL \leftarrow \text{S} \\ PC \leftarrow intentry_i \end{cases}$$

N.B. We assume that IRQ is not part of the interrupt handler code, so no check is necessary to disable the interrupts.

Interrupt handler ends (must end) with

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

3.6.3 User-mode software interrupts

In some systems, interrupt handlers may also be triggered by user programs. We model this by the following instruction

$$[\![\text{INT}_i]\!] \equiv \begin{cases} [\![\text{IRQ}_i]\!] & PL = \text{U} \wedge \text{canx}(PC) \\ \bot & \text{otherwise} \end{cases}$$

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 $sentry \in C_S$ 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. Thus, such a handler is part of the trusted code base.

Additionally, the handler is non-reentrant. 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.

$$[\![\text{SCALL}]\!] \equiv \begin{cases} SL \leftarrow PC, PL \leftarrow S, PC \leftarrow sentry & PL = U \wedge \text{canx}(PC) \\ \bot & \text{otherwise} \end{cases}$$

The SRET instruction returns to the caller: it changes to an unprivileged level and restores the original PC from the SL.

$$[\![\text{SRET}]\!] \equiv \begin{cases} PL \leftarrow U, PC \leftarrow SL & PL = S \wedge \text{canx}(PC) \\ \bot & \text{otherwise} \end{cases}$$

3.6.5 Domain switch

This is the instruction that provides a safe domain switch.

$$[\![\text{DOMSWITCH } d]\!] \equiv \begin{cases} D \leftarrow d & PL = \text{S} \wedge \text{canx}(PC) \wedge d \in \{1, \ldots, N\} \\ \bot & \text{otherwise} \end{cases}$$

Observe that this instruction may only be executed at the $PS = \text{S}$ privilege level. Therefore, the domain may be switched only during the hypervisor call or during interrupt handling.

4 Formalization for a TEE and Hypervisor less hardware

As highlighted in [1], numerous low-cost, small-size, and low-power consumption hardware devices emerged as possible deployment 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 [1], within CROSSCON we will adopt a software-based methodology to ensure the isolation between normal applications and trusted applications. This approach entails the implementation of PISTIS [14] and allows for memory isolation, remote attestation, and secure code update services, all fortified by robust security assurances. The architecture in this case will then be the one shown in Figure 4.

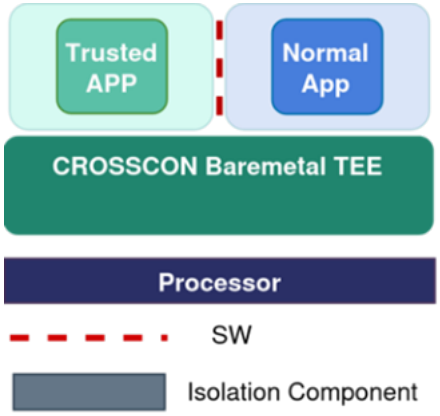

Figure 4: TEE and virtualization less architecture.

The memory map of Figure 2 lifted in the case of TEE and virtualization less architecture can be represented in a pictorially way as in Figure 5.

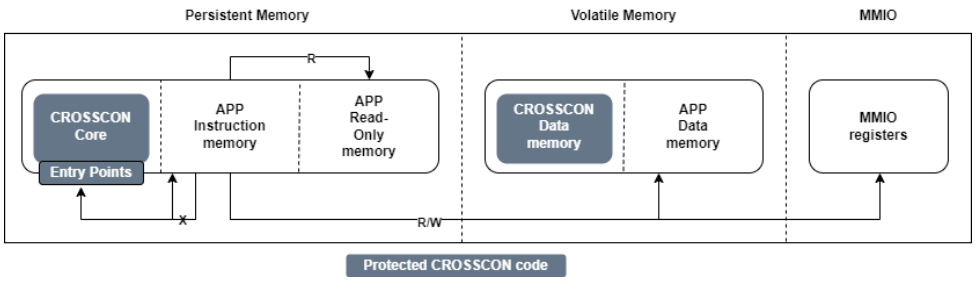

Figure 5: An example of a refined memory map for the TEE and virtualization less case.

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 for 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 of as composed of sequences and combinations 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.*

We can safely 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 introduce 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; iii) and finally introduce the 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 need also 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, and 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:*

- $\texttt{Read}_{sf}(M, i)$ *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;*
- $\texttt{Write}_{sf}(M, i, V)$ *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;*
- $\texttt{Goto}_{sf}(M, i)$ *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 11. *Given an access policy AP, the* safe primitive instructions $\texttt{Read}_{sf}$, $\texttt{Write}_{sf}$, *and* $\texttt{Goto}_{sf}$ *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 12. *Let AP be an access policy, 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} utilizing Theorem 11 (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 a safe one, and if so, it does the jump 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 11 and 12

Proof of Theorem 11. In order to prove Theorem 11, we formalized the three operations in NuXmv [15] (a state-of-the-art symbolic model checker), and for each of the three formalizations we considered some Linear Temporal Logic (LTL) [16] 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 [17] (as is typically done in compilers and in 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 listing 4.2 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 ($\forall$)).

Listing 4.1 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
```

Listing 4.1: 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. Listing 4.3 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 Listing 4.4.

Proof of Theorem 12. The proof of Theorem 12, proceeds by induction on the structure of the application P_{sf} leveraging on Theorem 11.

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 11. 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 11), 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)
```

Listing 4.2: 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)
```

Listing 4.3: 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)
```

Listing 4.4: The encoding to prove the correctness of the $\texttt{Write}_{sf}$

5 Conclusions

In this document, we have presented the first draft of the open formalization of the CROSSCON separation kernel and of the security properties that we will consider within CROSSCON. We started from the individual architectural components of CROSSCON and the possible implementation alternatives of the CROSSCON stack discussed in [1]. We proved for the CROSSCON separation kernel the properties of interest. For the 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.

As future work, among the ones envisaged in the tasks, we will consider the extension of the separation kernel formalization to consider dynamic reconfigurations of the memory layout (E.g., in response to an update or because of specific requests). Furthermore, we will automate the verification of the CROSSCON configuration file (see Deliverable [1]) using SMT techniques, and define a proper certification manifest to accompany CROSSCON applications.

The resulting finalized version of this document will then be reported in the final version of the CROSSCON Formal Specification deliverable [18]. We are also planning to submit an excerpt of the content of this document to scientific venues to valorize the carried out research.

References

[1] 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.

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

[3] 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. *Deliverable D1.2: Deliverable D1.2: Requirements Elicitation Initial Technical Specification*. 2023.

[4] 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.

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

[6] 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/.

[7] 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.

[8] 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.

[9] 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.

[10] John M. Rushby. "Noninterference, Transitivity, and Channel-Control Security Policies 1". In: 2005. URL: https://api.semanticscholar.org/CorpusID:8472202.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] CROSSCON Team. *Deliverable D2.4: CROSSCON formal specification - Final Version*. 2025.