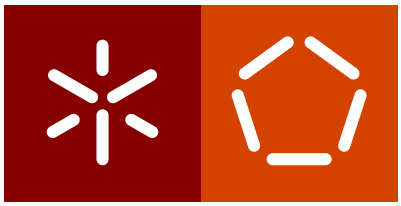
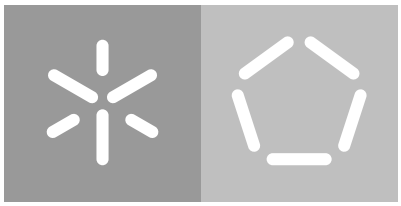


Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Ricardo João Rei Roriz

Enabling System Survival Across Hypervisor Failures

October 2018



Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Ricardo João Rei Roriz

Enabling System Survival Across Hypervisor Failures

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Sandro Pinto

October 2018

Declaração do Autor

Nome: Ricardo João Rei Roriz
Correio Eletrónico: a68536@alunos.uminho.pt
Cartão de Cidadão: 14655239

Titulo da dissertação: Enabling System Survival Across Hypervisor Failures
Ano de conclusão: 2018
Orientador: Professor Doutor Sandro Pinto

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

Universidade do Minho, 26/10/2018

Assinatura: Ricardo João Rei Roriz

Acknowledgements

This dissertation consecrates the culmination of my academic journey carried out over these six years. Thus, there are many people whom I want to thank that without their time, expertise, patience and support, I would not have completed this journey.

Firstly, I would like to thank my thesis advisor Dr. Sandro Pinto for the advices and independence, though always steering me in the right direction.

I wish to express my sincere thanks to Dr. Adriano Tavares for transforming this "pedreiro" into an embedded system engineer by sharing his knowledge. Thank you for showing me the beauty of this area.

I would also like to thank my "ESRGianos" colleagues: Ailton Lopes, Ângelo Ribeiro, Franciso Petrucci, Hugo Araújo, José Martins, José Ribeiro, José Silva, Miguel Silva, Nuno Silva, Pedro Machado and Sérgio Pereira for their feedback, cooperation and of course friendship. I also take this opportunity to express gratitude to all my friends, for their help and support.

Finally, I must express my very profound thankfulness to my mom, dad, sister, and my beloved Ariana Bezerra for providing me with unfailing support and continuous encouragement throughout my years of study, researching and writing this thesis. This accomplishment would not have been possible without your love. Thank you.

Abstract

Embedded system's evolution is notorious and due to the complexity growth, these systems possess more general purpose behaviour instead of its original single purpose features. Naturally, virtualization started to impact this matter. This technology decreases the hardware costs since it allows to run several software components on the same hardware. Although virtualization begun as a pure software layer, many companies started to provide hardware solutions to assist it.

Despite ARM TrustZone technology being a security extension, many developers realized that it was possible to use this extension to support development of hypervisors. With TrustZone, hypervisors can ensure one of the most important features in virtualization: isolation between guests. However, this hardware technology revealed some vulnerabilities and since the whole system is TrustZone dependent, the virtualization can be compromised.

To address this problem, this thesis proposes an hybrid software/hardware mechanism to handle failures of TrustZone-based hypervisors. By using the processor's abort exceptions and hash keys, this project detects system malfunctions caused by imperfect designs or even deliberate attacks. Additionally, it provides a restoration model by checkpoints which allows a system recovery without major throwbacks. The implemented solution was deployed on TrustZone-based LTZVisor, an open-source and in-house hypervisor, and the revealed results are appealing. With a 6.5% memory footprint increase and in the worst case scenario, an increment of 23% in context switching time, it is possible to detect secure memory invasions and recover the system. Despite of the hypervisor memory footprint increment and latency addition, the reliability and availability that the system bring to the LTZVisor are unquestionable.

Resumo

A evolução dos sistemas embebidos é notória e, devido ao aumento da sua complexidade, estes sistemas cada vez mais possuem um comportamento de propósito geral, em vez das suas características originais de propósito único. Naturalmente, a virtualização começou a ter impacto sobre este meio, uma vez que permite executar vários componentes de software no mesmo hardware, diminuindo os custos de hardware. Embora a virtualização tenha começado como uma camada de software pura, muitas empresas começaram a fornecer soluções de hardware para auxiliá-lo.

Apesar da TrustZone ter sido projetada pela ARM para ser uma extensão de segurança, muitos desenvolvedores perceberam que era possível usá-la para suporte ao desenvolvimento de hipervisores. Com a TrustZone, os hipervisores podem garantir uma das premissas mais importantes da virtualização: isolamento entre hóspedes. No entanto, esta tecnologia de hardware revelou algumas vulnerabilidades e, sendo todo o sistema dependente da TrustZone, a virtualização pode ficar comprometida.

Para solucionar o problema, esta tese propõe um mecanismo híbrido de software/hardware para lidar com as falhas em hipervisores baseados em TrustZone. Usando as excepções do processador e chaves de hash, este projecto detecta defeitos no sistema causados por imperfeições no design e também ataques intencionais. Além disso, este fornece um modelo de restauração por pontos de verificação, permitindo uma recuperação do sistema sem grandes retrocessos. A solução foi implementada no LTZVisor, um hipervisor em código aberto e desenvolvido no ESRG, sendo que os resultados revelados são satisfatórios. Com um aumento de 6,5% da memória usada e um incremento, no pior caso, de 23% no tempo de troca de contexto, é possível detectar invasões de memória segura e recuperar o sistema. Apesar do incremento de memória do hypervisor e da adição de latência, a confiabilidade e a disponibilidade que o sistema oferece ao LTZVisor são inquestionáveis.

Contents

List of Figures

List of Tables

List of Listings

Glossary

ACTLR	Auxiliary Control Register
AMP	Asymmetric Multiprocessing
APB	Advanced Peripheral Bus
API	Application Programming Interface
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CFI	Control Flow Integrity
CPSR	Current Program Status Register
CRC	Cyclic Redundancy Checks
DFAR	Data Fault Address Register
DFI	Data Flow Integrity
DFSR	Data Fault Status Register
DJB2	DJB2 algorithm
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
ESRG	Embedded Sistem R
ExT	External bit
FF	Flip-Flop
FIQ	Fast Interrupt Request
FNV	Fowler–Noll–Vo
FPGA	Field-Programmable Gate Array
FS	Fault status bits
GIC	Generic Interrupt Controller
GP	General Purpose
GPOS	General Purpose Operating System
IDE	Integrated Development Environment
IFAR	Instruction Fault Address Register
IFSR	Instruction Fault Status Register
IoT	Internet of Things

IP	Intelectual Proprety
IRQ	Interrupt Request
ISR	The Interrupt Status Register
LCG	Linear Congruential Generator
LPAE	Large Physical Address Extension
LR	Link Register
LTZVisor	Lightweight TrustZone-assisted hypervisor
LUT	Look Up Table
MMU	Memory Management Unit
MVBAR	Monitor Vector Base Address Register
NS	Non Secure
NSACR	Non-secure Access Control Register
OS	Operative System
OSs	Operative Systems
PC	Program Counter
PL	Programmable Logic
PMU	Performance Monitoring Unit
PS	Processing System
QSPI	Quad Serial Peripheral Interface
RAM	Random Access memory
ROM	Read Only Memory
RTOS	Real Time Operating System
SCR	Secure Configuration Register
SCTLR	System Control Register
SDBM	SDBM database library
SDER	Secure Debug Enable Register
SDIO	Secure Digital Input/Output interface
SDK	Software Development Kit
SE	Security Extensions
SMC	Secure Monitor Call
SoC	System-on-Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
SRAM	Static Random Access Memory
TTBR	Translation Table Base Register
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adopter

TZPC	TrustZone Protection Controller
VBAR	Vector Base Address Register
VE	Virtualization Extensions
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMs	Virtual Machines
VT	Virtualization Technology
XSDK	Xilinx Software Design Kit

1. Introduction

The accentuated growth of the industry has been forcing embedded systems towards more complex solutions. Nowadays, there is a wide range of embedded systems applications and domains, from basic consumer electronics [ABK09] and IoT solutions [PGP+17, OGP18], to aerospace control systems [HHY+12, PPG+17a]. Although the traditional definition is characterized by limited hardware with timing constraints, the functionality is increasingly making these systems tacking characteristics towards general-purpose. Also, with the processing power provided by the modern processors, the old idea of single-purpose is extinguished. By implementing multiple subsystems in the same platform, the resources sharing is possible, reducing significantly the cost. However, in addition to the increasing complexity is the security problem which became an important issue to solve in this area.

Due to these characteristics and the need for support heterogeneous operating systems (Real Time Operating Systems (RTOS) and General Purpose Operating System (GPOS), the virtualization technology arrived naturally as the best solution [Hei08, Kai09, AH10]. This technology consists of the encapsulation of each embedded subsystem in its own virtual machine, allowing isolation and fault-containment. Although the hardware is shared among various subsystems, each virtualized partition has its own virtualized resources. By creating this layer it is possible to define a hierarchy hardware resource accesses as well as pure virtualized resources. To monitor the VMs (Virtual Machines) actions, a hypervisor is required. It not only controls the layer between hardware and virtualized worlds, but also the information shared among them [SML10]. With Inter Partition Communication (IPC) mechanisms, virtualization offers cooperation tools between environments but without changing the demanded isolation [OMC+18] .

In the beginning of embedded system's virtualization, software-based solutions were exclusive, but due to strict timing requirements and constraints imposed by the real-time nature of such applications [ZMH15], companies like *ARM* and Intel have begun to provide hardware to support virtualization. Intel introduced

Intel Virtualization Technology (VT) [SK10], ARM presented ARM Virtualization Extensions (VE) and recently Imagination/MIPS released MIPS Virtualization and OmniShield technology [ZMH15]. However, due to the ubiquitous adoption of ARM-based processors in the embedded market, ARM solutions are more popular among the remaining ones. Despite not being design to assist virtualization, ARM also provides a security extension (ARM TrustZone [Lim09]) to its processors. It allows to run different security environments in the same processor at lower cost comparing with VE-enabled processors [PS18]. Interpreting the growth of TrustZone technology, several virtualization developers and newcomers provided monitor hardware-based solutions: low print hypervisors assisted by TrustZone security features.

The TrustZone-based hypervisors rely on extra security bits to virtualize worlds (33rd ARM processor bit) [PPG+17b]. Since they are processor bits, these hypervisors frequently create the illusion of a perfect virtualization solution. However, projects like CLKSCREW [TSS17] demonstrate that hardware-based hypervisors can also be vulnerable. By exploiting the processor's frequency, this project proves that it is possible to change the secure/non-secure bit of ARM-TrustZone registers, obliterating the isolation between worlds and eliminating the secure world control over non-secure world.

In software applications, fault tolerance mechanisms to detect, prevent and solve the problems are discussed from the very beginning [Ran75]. Various software mechanisms are developed to hypervisors like Xen [BDF+03], oriented to general purpose computing, and despite incrementing overhead they provide survivability to the system. The key difference between this type of hypervisor and embedded hypervisors is the abundance of hardware resources. Although general purpose hypervisors are designed with performance and footprint taken into consideration, the latter is not the main metric. On the other hand, embedded hypervisors are designed to provide virtualization with the lowest foot print possible. In this scenario, the most common mechanisms are limited to physical fault tolerance mechanisms (electrical faults e.g). To increase the mechanism complexity without increasing system overhead, dedicated hardware with health-monitoring purposes is the most desired approach.

1.1 Motivation

As Figure 1.1 exposes, there are three different hypervisor execution flows. The first one is the most common in TrustZone-based hypervisors. Due to its minimal

approach, the security is only based on error prevention.

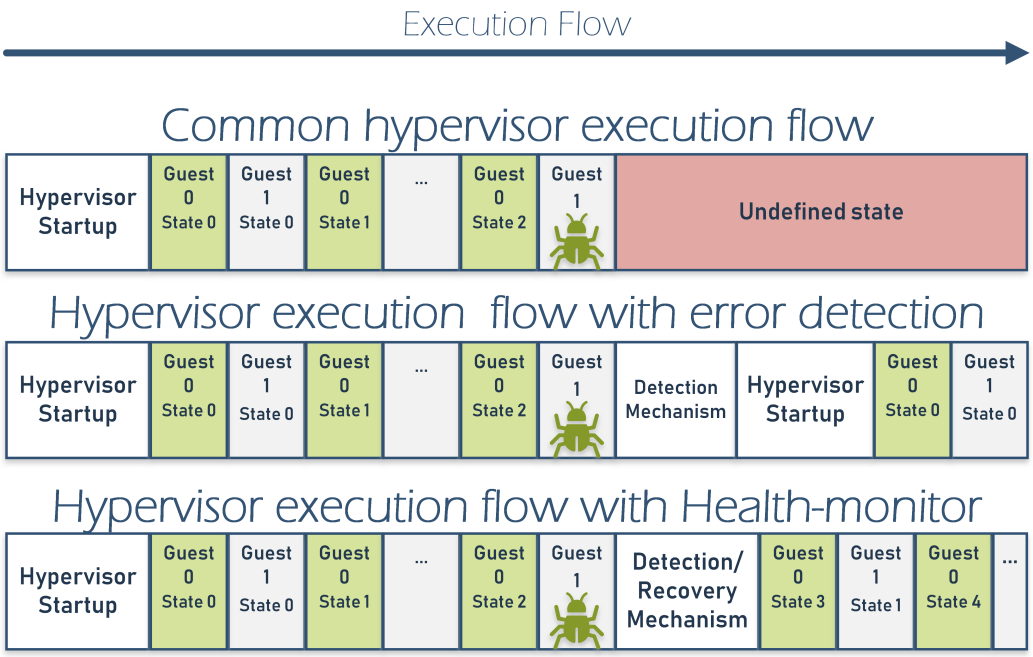


Figure 1.1: Motivational execution flow.

The other two types imply extra mechanisms: error detection and recovery. In an implementation with error detection only, the hypervisor detects the error but the only feasible action is the full restart of the system. Although not ideal, this implementation already prevents the malfunction of the system for indeterminable time. With the addiction of the recovery mechanism, the system keeps the same but the throwback produced is reduced since it allows to recover from an healthy state closer to the current state of the system.

This thesis intent to provide a hybrid hardware/software fault tolerance mechanism with an Health-monitor, on an open source and an in-house TrustZone-based hypervisor, the Light-weight TrustZone-assisted hypervisor (LTZVisor).

1.2 Goals

The project reported on this thesis aims to achieve three primary goals decomposable in more detailed sub goals:

1. Provide an exception handling to the LTZVisor. With this tool, the hypervisor user will know which part of the software is causing problems making the development more productive. The design of this feature needs to provide the following information:

   (a) Faulted World - If a fault occurs on the secure world or thenon-secure World;

   (b) Faulted processor mode - If the fault occur in Monitor mode or Supervisor.

   (c) Fault type - If it is Data Abort or Prefetch Abort;

   (d) Faulted subtype - If it is an alignment fault, cache maintenance, access fault, permission fault, and so on;

   (e) Faulted address - The address that causes the fault.

2. Create a mechanism to detect secure memory evasive attacks. As the CLKSCREW's project [TSS17] shows, it is possible to change secure memory from non-secure side masking the ARM's secure/non-secure bit, so the detection mechanism needs to:

   (a) Distinguish a legitimate secure memory access from non legitimate secure access;

   (b) Introduce the minimum of deterioration relative to the LTZVisor native performance.

3. Create a recovery mechanism responsible for recovering the hypervisor from memory failures with a minimum system throwback and completely LTZVisor independence, since the secure world is compromised. Although memory errors occur less often, they can be more disastrous and without an independent mechanism it cannot be handled without a full system reset.

In order to be LTZVisor independent and not introduce any overhead, these mechanisms should be hardware-based components making possible to run parallel mechanisms to detect non-legitimate memory accesses and full recover in case of failure.

1.3 Document Structure

This document structure serves the following order: Chapter 2 begins with an overview of the basic virtualization concepts, ARM Architecture and Trustzone technology, exception handling and Health-Monitors. It then proceeds by surveying existing hypervisors as well as their exception handling and health-monitor implementations. This chapter ends with an introduction to hashes and checksum

algorithms, being these possible Health-Monitor mechanisms to detect memory faults.

Chapter 3 provides a more detailed description about the platforms: the board ZYBO Zynq-7000, with more indepth information about the DMA and AXI features, and the target hypervisor, the LTZVisor. Chapter 4 addresses the design and implementation of this project. Chapter 5 exposes the system evaluation and the discussion of the results. Lastly, Chapter 6 provides a summary of this thesis, revealing the system limitations and from them outline solutions to future improvements.

2. Background, Context and State of the Art

As the main theme of this thesis is error recovery on embedded systems' hypervisors, this chapter will first expose a general definition of virtualization and hypervisors, ARM Architecture and TrustZone, exception handling and health-monitor, followed by some work done in this area. Even though there are many hypervisors, the focus will be on hypervisors that use hardware to assist virtualization, i.e. LTZVisor, SafeG [saf], Jailhouse [Tec13] and VOSYSmonitor [LCP+17]. In fault tolerance and health-monitor section, the basic terminology and concepts are explained to better understand the project's Health-Monitor implementation. Since the detection mechanism uses key comparison as sanity checker, the final part of this chapter will describe hash functions and checksums algorithms with the potential to be used.

2.1 Virtualization

The digital evolution is so accentuated that it is impossible to implement a modern system based on a simple Flip-Flop mindset due to the complexity required. So to convert an high complex design into a compound of electrical signals, the system needs to be analyzed as a hierarchy arrangement of abstraction layers and a well-defined interface between them [SN05]. At lower level, there is almost no abstraction since it is an electronic level. As we go up in the layers spectrum, the abstraction rise to the point that the layers become software based implementations such as code to run in the microcontroller, drivers to control the hardware or even Operating Systems (OS). They use the hardware infrastructure (communicating with the layers below) but in a such encapsulated way that they are unaware about how the hardware works.

The virtualization layer is responsible not only for hardware abstraction, but also enhances its features by creating unique virtual replicas of the hardware.

Thus, each software component has its own virtual replica. This makes possible to share the same hardware with different applications, run the same application on a different platform without large engineer efforts, or even distribute the same application for different platforms [POP+17]. In some cases, the virtualization layer also creates pure virtual components to meet the hardware requirements of the system. These replicas, also known as Virtual Machines (VMs), are described by Popek and Goldberg as an efficient and isolated replica of the real machine [PG74]. In the virtualization environment, two different terms, "host" and "guest" are used to distinguish where the software runs. The software that runs on the physical machine is called host software, and guest symbolises software that runs in a virtual world. The software responsible to create and monitor virtual machines on the host hardware is called a hypervisor or Virtual Machine Monitor. Popek and Goldberg [PG74] also describe this VMM as "a piece of software" with "three essential characteristics":

1. Equivalence: the provided virtualized environment should be essentially identical to the original machine. With this characteristic, guests OSs are directly used in VMs without any modification, minimizing costs of porting guest software to the VM.

2. Efficiency: It is a requirement that the virtual processor's main instructions be executed directly by the real processor without intervention of the virtualization software in order to not substantially reduce the guest' performance in relation to its native performance.

3. Resource Control: the VMM must have the full control of all system resources being impossible for a guest to monopolize or manipulate them. Only with a full resource control is possible to have the temporal and logical isolation between the machines.

Although the first hypervisor's designs were bare metal hypervisors, there are currently two types of hypervisors described in figure 2.1: bare-metal hypervisors that run directly on the native hardware and hosted hypervisors that run on top of an OS. An example of hosted hypervisors are software as Oracle Virtual Box [ora] or Java Virtual Machine (JVM) [JVM] that enables different systems to run Java programs. Nonetheless, due to rigorous timing and low footprint systems, embedded systems demand bare-metal hypervisors.

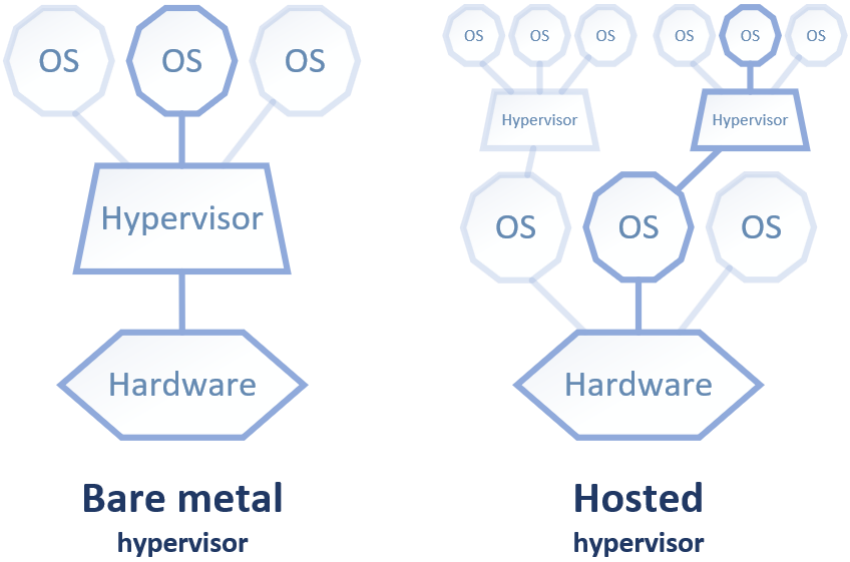


Figure 2.1: Hypervisor types. The hypervisors that run directly on hardware are defined as Bare metal hypervisors (left) and the ones that run on top of OSs are Hosted hypervisors (right).

2.2 ARM Architecture

The ARM architecture is well suited for embedded systems due to simplicity that it present comparing other architectures. Consequently, simple implementations lead to small implementations, thus providing devices with low power consumption. The ARM architecture is a Reduced Instruction Set Computer (RISC) architecture, as it incorporates the following features [Lim12]:

- A large uniform register file;
- Load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents;
- Simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only.

Also, this architecture provides enrichments that supply a good balance between high performance and low resources demand like: instructions that combine a shift with an arithmetic or logical operation, auto-increment and auto-decrement addressing modes to optimize program loops, load and store multiple instructions to maximize data throughput and conditional execution of many instructions to maximize execution throughput.

ARMv7 is the seventh version of the ARM architecture and it provides three different profiles: ARMv7-A (Application), ARMv7-R (Real-time) and ARMv7-M (Microcontroller). They present different features in terms of memory. The

ARMv7-A supports a Virtual Memory System Architecture (VMSA) based on a Memory Management Unit (MMU). Contrary, ARMv7-R supports a Protected Memory System Architecture (PMSA) based on a Memory Protection Unit (MPU). Althought both architectures afford mechanisms to split memory into different regions, specifying memory types and attributes, the MMU provides a virtual memory system. This feature abstracts memory for different processes, allowing dynamic memory allocation by operating systems.

Depending on the desired application, the ARMv7 instruction set can be expanded with extensions in order to provide extra features:

- Security Extensions (SE) - Also known as Trustzone. This extension provides a set of security features that facilitate the development of secure applications.
- Multiprocessing Extensions - This extension provides a set of features that enhance multiprocessing functionality. However, this is restrict to profiles ARMv7-A and ARMv7-R.
- Large Physical Address Extension (LPAE) - This extension provides an address translation system supporting physical addresses up to 40 bits. The LPAE is restrict to ARMv7-A profiles with Multiprocessing Extensions enable.
- Virtualization Extensions (VE) - This extension provides hardware support for virtualization with a virtual machine monitor, also called a hypervisor, to switch Guest operating systems. The VE requires the Secure Extensions extension.

2.2.1 ARM TrustZone

TrustZone is an hardware technology that allows the execution of guests with different levels of security in the same platform. On processors where VE is not available, TrustZone is considered the only hardware-based deployable approach in terms of virtualization. The TrustZone expansion to the new generation of Cortex-M processors [WFM+07] proves that the technology is even spreading to processors with limited resources.

With ARM TrustZone it is possible to separate the execution environment into two isolated worlds [Tru], allowing to run secure and non-secure applications in the same hardware. The 33rd processor bit (Non-Secure NS-bit) indicates the currently executing world. Also, some registers are baked for each specific world,

which expedites the transitions between two worlds without compromise their isolation.

In terms of architecture, the main features of the Security extensions are:

1. Monitor mode - This extra processor mode is implemented with the purpose of context switching between the Secure and Non-secure security states. Regardless the The Secure Configuration Register (SCR) secure bit, the software running in Monitor mode has access to both the Secure and Non-secure resources, even to system registers. Due to the high privileges, this mode only can be triggered by exceptions and by a special instruction named Secure Monitor Call (SMC).

2. Security Registers:

   - Secure Configuration Register (SCR)- The SCR is the register responsible to specify the security related parts of the system: the security state of the processor (Secure or Non-secure), for which mode the processor branches to if an IRQ, FIQ or external abort occurs (Abort mode or Monitor), and whether the Current Program Status Register (CPSR) F and A bits can be modified by the non-secure world.
   - Secure Debug Enable Register (SDER) - The SDER enables secure invasive and non-invasive debug.
   - Non-secure Access Control Register NSACR - The NSACR defines the Non-secure access permission to coprocessors.
   - Vector Base Address Register (VBAR) - The VBAR holds the exception base address for exceptions that are not taken to Monitor mode.
   - Monitor Vector Base Address Register (MVBAR) - The MVBAR holds the exception base address for all exceptions that are taken to Monitor.
   - Interrupt Status Register(ISR) - The ISR shows whether an IRQ, FIQ, or external abort is pending.

3. Secure Monitor Call (SMC) - The only way that the non-secure world accesses the secure world, which can be an access to data or routines, is through a special instruction called SMC. This instruction triggers an exception with predefined routines that run in monitor mode.

4. Exception model - With the addition of the monitor mode, two slightly different exceptions models are available that will be described on the next section.

2.2.2 Exceptions in ARMv7 with TrustZone

There are eleven exceptions in ARMv7 with Trustzone: the common six of the ARMv7 architecture 1) Reset, 2) Data Abort, 3) Prefetch Abort, 4) Fast Interrupt (FIQ), 5) Interrupt Results (IRQ), 6) Undefined instructions; plus the 7) Secure Monitor Call (SMC), 8)/9) banked data aborts and 10)/11) banked prefetch aborts, for monitor and for non secure-side.

A data abort exception occurs when a data transfer instruction attempts to load or store data at an illegal address [Lin12] which can be unmapped memory, unaligned memory or inaccessible memory. Additionally, this exception takes place for errors related with paging and translation on virtual memory environments.

Although the prefetch exception also occurs on memory accessing errors, it arises when the processor fetches an instruction from an illegal address. If a pipeline architecture is present, the instructions already in the pipeline continue to execute until the invalid instruction is reached and then a prefetch abort is generated [Lin12].

Figure 2.2 describes the two possible models that differ from each other in how external aborts are handled. In the first model (SCR.EA bit set to 0), external aborts are handled by an abort mode and each world is responsible to handle its own aborts. In the second model (SCR.EA bit set to 1), the Monitor is responsible to handle all external aborts regardless of the origin.

<table>
<tr><td rowspan="8">ARM-v7 Exception Model with Security Extensions</td><td colspan="4">External Aborts bit = 0 (SCR.EA)</td></tr>
<tr><td colspan="2">Secure Side</td><td colspan="2">Non Secure Side</td></tr>
<tr><td>Secure Abort Mode</td><td>Secure Guest</td><td>Non Secure Abort Mode</td><td>Non Secure Guest</td></tr>
<tr><td>- External Aborts (S)</td><td>- Alignment faults (S)
- MMU faults (S)
- Debug faults (S)</td><td>- External Aborts (NS)</td><td>- Alignment faults (NS)
- MMU faults (NS)
- Debug faults (NS)</td></tr>
<tr><td colspan="4">External Aborts bit = 1 (SCR.EA)</td></tr>
<tr><td colspan="2">Secure Side</td><td colspan="2">Non Secure Side</td></tr>
<tr><td>Monitor Mode</td><td>Secure Guest</td><td colspan="2">Non Secure Guest</td></tr>
<tr><td>- All External Aborts (S & NS)</td><td>- Alignment faults (S)
- MMU faults (S)
- Debug faults (S)</td><td colspan="2">- Alignment faults (NS)
- MMU faults (NS)
- Debug faults (NS)</td></tr>
</table>

Figure 2.2: ARM-v7 with Security Extensions Exception Model. Two models are described on the figure. At the model on the top, each world has an abort mode for itself. On the bottom model, the these exceptions are forced into the monitor despite the origin. Both modules share the same implementation on normal exceptions.

As defined in ARMv7 Arquitecture Reference manual [Lim12], external aborts are "errors that occur in the memory system, other than those detected by the MMU or Debug hardware. External aborts include parity errors detected by the caches or other parts of the memory system." On other hand, the other aborts are more usual and by ARM design, they are handled by the respective supervisor (Guest OS) of each world in both models.

Despite some of the register are not dedicated only to interpret exceptions, the architecture provides five registers in order to understand the aborts' origin:

- Saved Program Status Register (SPSR) has the purpose of saving the pre-exception value of the Current Program Status Register (CPSR), saving the processor status and control information that causes the exception. Since there is an SPSR for each processor mode, the CPSR is copied to the SPSR of the mode to which the exception is taken.

- Instruction Fault Status Register (IFSR) has the purpose of holding the status information about the last instruction fault. This register has two important parts: External bit (ExT), a bit that describes if it was an external abort or not; and Fault status bits (FS), five bits that describe the abort source. The valid encodings of these bits are present in the table 2.1.

- The Data Fault Status Register (DFSR) is similar to IFSR but its purpose is holding status information about the last data fault. Due to the similarity, the DFSR and IFSR share the same register structure and FS enconding values.

- The Instruction Fault Address Register (IFAR) has the purpose of holding the address of the access that caused a prefetch abort exception. Depending on the prefect abort source, this register can be invalid.

- The Data Fault Address Register (DFAR) is the data fault version of the IFAR.

Table 2.1: Fault Status encoding table. It is exposing the FS encoding and corresponding exception source. On the column *Abort type* is expressed which aborts types are enable for each source. The DF means Data fault and IF means Instruction fault.

Fault Status	**Source**	**Abort type**	**Note**
00001	Alignment fault	DF	-

00100	Fault on instruction cache maintenance	DF	-
01100	Synchronous external abort on translation table walk - First level	DF & IF	-
01110	Synchronous external abort on translation table walk - Second level	DF & IF	
11100	Synchronous parity error on translation table walk - First level	DF & IF	-
11110	Synchronous parity error on translation table walk - Second level	DF & IF	-
00101	Translation fault - First level	DF & IF	MMU Fault
00111	Translation fault - Second level	DF & IF	MMU Fault
00011	Access flag fault - First level	DF & IF	MMU Fault
00110	Access flag fault - Second level	DF & IF	MMU Fault
01001	Domain fault - First level	DF & IF	MMU Fault
01011	Domain fault - Second level	DF & IF	MMU Fault
01101	Permission fault - First level	DF & IF	MMU Fault
01111	Permission fault - Second level	DF & IF	MMU Fault
00010	Debug event	DF & IF	-
01000	Synchronous external abort	DF & IF	-
10000	TLB conflict abort	DF & IF	-
10100	Implementation defined	DF & IF	Lockdown
11010	Implementation defined	DF & IF	Coprocessor abort
11001	Synchronous parity error	DF & IF	-
10110	Asynchronous external abort	DF	-
11000	Asynchronous parity error on memory access	DF	-

2.3 Hypervisors Implementations

This section describes some hardware-assisted hypervisors implementations, exposing its characteristics. After analysing the hypervisors, the section ends with a discussion of its similarities and differences.

2.3.1 LTZVisor

The Lightweight TrustZone assisted Hypervisor (LTZVisor) is an open-source hypervisor developed to seek the benefits and limitations of using TrustZone hardware to assist virtualization [LTZ]. The LTZVisor provides a virtualization solution based on the two virtual execution environments, Secure VM and Non-Secure VM. In the Monitor mode, this hypervisor provides software tools for virtualization like scheduler and an Inter-VM Communication. Figure 2.3 describes these three main software components of LTZVisor's architecture (the hypervisor, the secure VM and the non-secure VM).

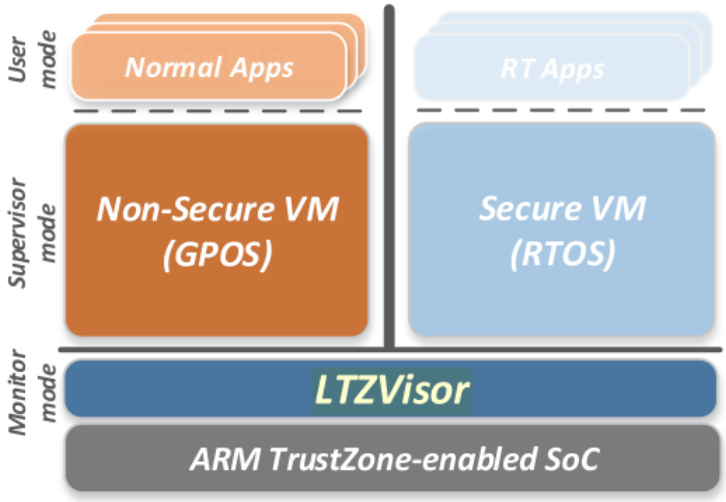


Figure 2.3: LTZVisor General Architecture [PPG+17b]. The LTZVisor hypervisor is a bare metal hypervisor and through TrusZone technology, it allows the concurrent execution of a GPOS and RTOS without violate the isolation between both.

The design of this hypervisor is based on three principles [PPG+17b]:

- Minimal implementation - LTZVisor assures it by thoroughly relying on the hardware support of TrustZone technology since it reduces the software components needed to create a fully functional hypervisor without loosing the features.

- Least privilege - Access to the resources (e.g., I/O devices, system services, etc) is only allowed if absolutely necessary. This principle is guaranteed by ARM TrustZone design itself. As described in the Trustzone chapter, two different worlds are implemented in terms of privileges, allowing resources to be confined at one or both worlds at the same time.

- Asymmetric scheduling - The adoption of an asymmetric scheduling policy, where the secure environment has a higher privilege of execution than the non-secure one, will ensure that timing requirements are met, even while executing real-time tasks.

This hypervisor does not only guarantee processing and memory isolation, but also pledge the devices through Device Partition. The technology allows for the devices to be configured as secure or non-secure, as well as ensures isolation when the device is shared between the two worlds. Being the receptacle of this thesis, this hypervisor will be described more in depth in the next chapter.

2.3.2 Jailhouse

Jailhouse is an open-source, real-time, non-scheduling, fully functioning and Linux-based hypervisor. It combines the operating system Linux with isolated purpose components, minimising the hypervisor's activity. Jailhouse was first developed by Jan Kiszka [Sin15] and later released to the public as open-source software. Despite this hypervisor is not TrustZone-based, it uses hardware to support virtualization.

The Jailhouse hypervisor's core acts as a Virtual Machine Monitor (VMM), but due to its design, it implements resource access control rather than resource virtualization. Instead of virtually isolated worlds, the system reserves real cores. As Figure 2.4 shows, the resources are split in root cells and non-root cells. The difference between them are the privileges of cell management, offering cell creation/destruction and hypervisor disabling tools to the root cells.

Although the GPOS in most cases are considerer non-privileged by other hypervisors, in this case the Linux is the root cell guest since it cooperates closely with Jailhouse. The other cells can be baremetal applications or RTOSs. Instead of sharing symmetrically multi-core processor resources between guests, this hypervisor drives each guest with their own set of resources [Sin15]. Thus, it implements Asymmetric Multiprocessing (AMP) without losing isolation.

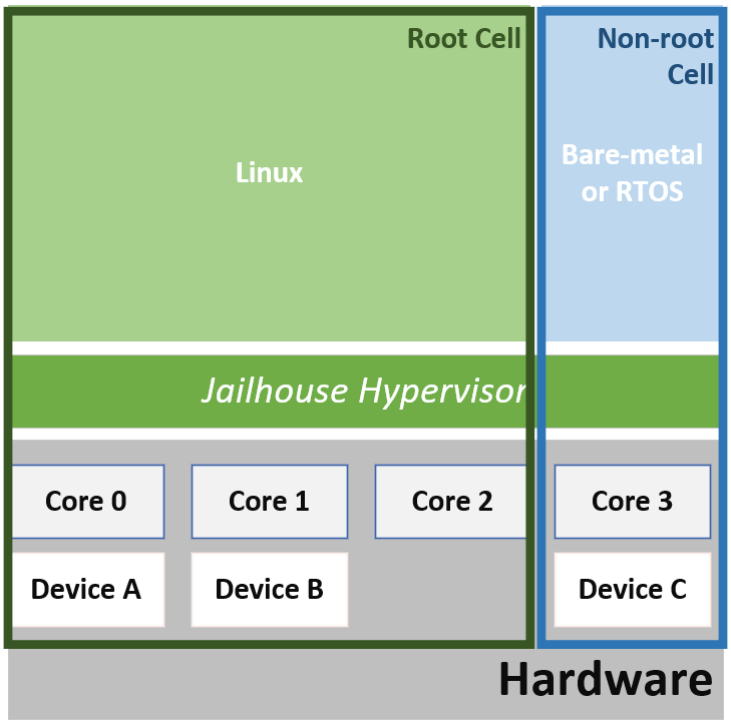


Figure 2.4: Jailhouse hypervisor Overview, based on [Kis14]. The Jailhouse hypervisor divide its guests in root (Linux) and non-root cells (bare-metals and RTOSs). The root cell, defined as green on figure, is privileged in terms of controlling the resources. Since this hypervisor do not provide resource sharing, the non-root cells (represented in light blue) are limited to used the resources provided by the root cell.

Despite of being a bare metal hypervisor, this Jailhouse is Linux dependent since it is the system that provide boot and hardware initialization. After the inicialization, the hypervisor acquire all the hardware resources (e.g., CPU(s), memory, PCI or MMIO devices), removes them from Linux and reassigns them to the new domain (other cell). Since Jailhouse only remaps and reassigns resources, once everything is set up, its ideal execution would to be only intervene if there was a case of access violation [RKLM17].

2.3.3 SafeG

SafeG was designed by TOPPERS[saf] as a dual-OS TrusZone-based monitor with the purpose of executing a RTOS and a GPOS concurrently. Like other Trustzone-based hypervisors, it relies on Trustzone for isolation and on the new processor mode (Monitor) to perform switches between the Trusted and Non-Trusted worlds. In this mode, the interrupts are disabled making it deterministic [SHT13].

One of the key features of this hypervisor is the full control of RTOS over the GPOS actions and scheduling time, making possible to implement a hypervisor independent scheduler. As shown in Figure 2.5, the scheduler slices the time into two different parts: RTOS running time and Cyclic Sched. The first one is reserved only for RTOS tasks while in the Cyclic Sched slice the processor can switch between guests. Like other VMMs oriented to real-time systems [YLH+08], the GPOS is only executed when the RTOS becomes idle, and in that time, the RTOS schedules the GPOS as a normal RTOS task [SHT13] since it is responsible for scheduling.

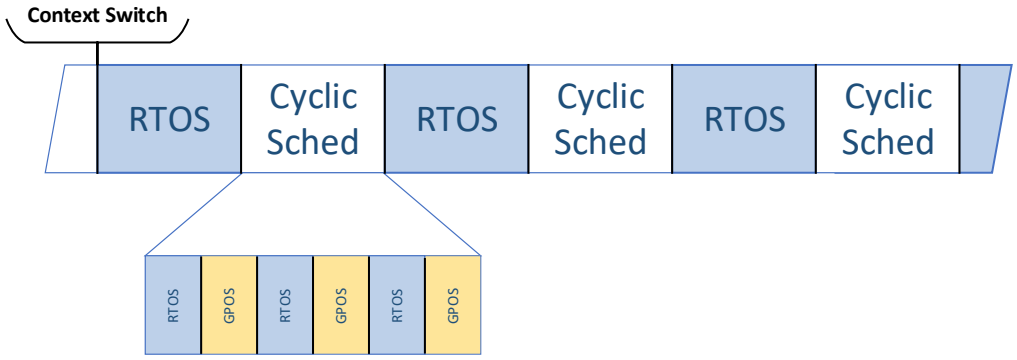


Figure 2.5: SafeG scheduler, based on [SHT13]. The SafeG scheduling policy is divided in two periodic slices: The RTOS scheduling time (represented in blue) and a Cyclic Sched (represented in white). The first portion is fully reserved to RTOS as long as in the Cyclic Sched, the RTOS can schedule the GPOS (represented in yellow) as well as its own tasks.

2.3.4 VOSYSmonitor

VOSYSmonitor [VOS], developed by Virtual Open Systems [LCP+17], is a software monitor which enables, just as the others aforementioned, the concurrent execution of a safety critical RTOS along with a GPOS. The VOSYSmonitor, similarly LTZVisor, was designed using the TrustZone architecture, insuring by design peripherals and memory isolation between both OSs, but allowing dynamical cores sharing.

Since the main goals of the VOSYSmonitor are performance related, the boot time is taken into consideration. By design, the VOSYSmonitor setup must be achieved in less than 1% of the full RTOS boot time. For instance, a RTOS boot time of 60 ms implies a setup performed in less than 600us regardless of the platform [LCP+17]. Additionally, the context switch is simplified. This hypervisor periodically transfers the execution from one world to the other, and as a result,

minimises context switches. Even so, most of the code executed is written in ARM assembly and only vital registers are saved in these switches.

2.3.5 Discussion

Being hardware-assisted hypervisors, it is possible to identify common design features:

- All support concurrent execution of a GPOS and an RTOS.
- They take advantage of hardware extensions in order to achieve isolation between guests, a low footprint hypervisor and very low execution overhead.
- Due to TrutZone design, they separate IRQs for the non-secure world and FIQs for the safe world.
- They provide time isolation of the RTOS creating a deterministic behavior for him.

Regarding interrupts, the analysed hypervisors separate IRQs for the non-secure world and FIQs for the secure world because through TrustZone design is possible to prevent the non-secure world side from disabling FIQ interrupts (IRQs can be disabled) and IRQs can be treated without the intervention of the secure world.

As for hypervisors like VOSYSmonitor and LTZVisor, when the processor is running in the secure world, the IRQs are turned off as it is considered that no interruption from the non-secure world should stop the normal working of the secure world. Nevertheless, when the processor is running in the non-secure world both IRQs and FIQs are enabled and FIQs have higher priority than IRQs because of their origin.

The Jailhouse is the hypervisor that provides most differences in terms of design due to the fact that the most privileged guest is the Linux. In summary, it is possible to conclude that the Jailhouse is a tool to extend hardware virtualization to Linux providing full guest control without loosing system performance.

2.4 Exception Handling

Anomalies may occur during program execution. Exception handling is the selection of handlers with predetermined actions to respond to those anomalies. Due to the fact that the exceptions can differ depending on the processor architecture, this work will focus on exceptions from ARMV7 without VE.

2.4.1 Exception Handling Implementations

Even though this section is about exception handling implementation in hypervisors, it will only bring to light SafeG handlers due to the architecture incompatibility of the others. Both Jailhouse and VOSYSmonitor are designed to run in ARMv7 with Virtualization Extensions or ARMv8 altering completely the exception handling design. These technologies provide levels of exceptions that separate monitor, guest and application exceptions, contrary to the ARMv7 with SE that provides only one for each world.

2.4.1.1 SafeG

Figure 2.6 illustrates the exception handling overview of SafeG hypervisor. It follows the normal design of the ARM-v7 architecture exceptions. The Undefined handler is a simple endless loop due to the difficult of understanding the origin and the exception consequences. Like the LTZvisor, this hypervisor is designed to define FIQs as secure interrupts and IRQ as a non-secure interrupts. To accomplish hypervisor atomicity, the FIQs are disabled when the processor is on monitor mode (when the tasks of the hypervisor are running) so they stay pendent until one of the guest is active. Like FIQs, the IRQs are disabled on monitor mode but also when the secure guest is running. This guarantees that the secure world is not interrupted by the non-secure guest. Only when the non-secure guest is functioning, in this case the GPOS, the IRQs are enabled and attended.

This hypervisor has a list of APIs that both secure and non-secure guest can call through Secure Monitor Call (an exeption to call the monitor) in order to configure some hypervisor features. They are monitored by the Notifier module, which is designed to avoid race conditions and security leaks. This module allows the creations of dynamic system calls by the secure guest but by default, the Notifier defines ten system calls:

- **setperm** - set permissions for a certain system call.
- **switch** - initiates a switch to the opposite world.
- **restarnt** - restart NT OS
- **getid** - return the ID of a system call.
- **signal** - signals an interrupt to the opposite world.
- **writel** - write specific address
- **regdyn** - register a dynamic system call.
- **setntpc** - set NS OS Program Counter
- **regnot** - register a notifier call.

- **readl** - read specific address

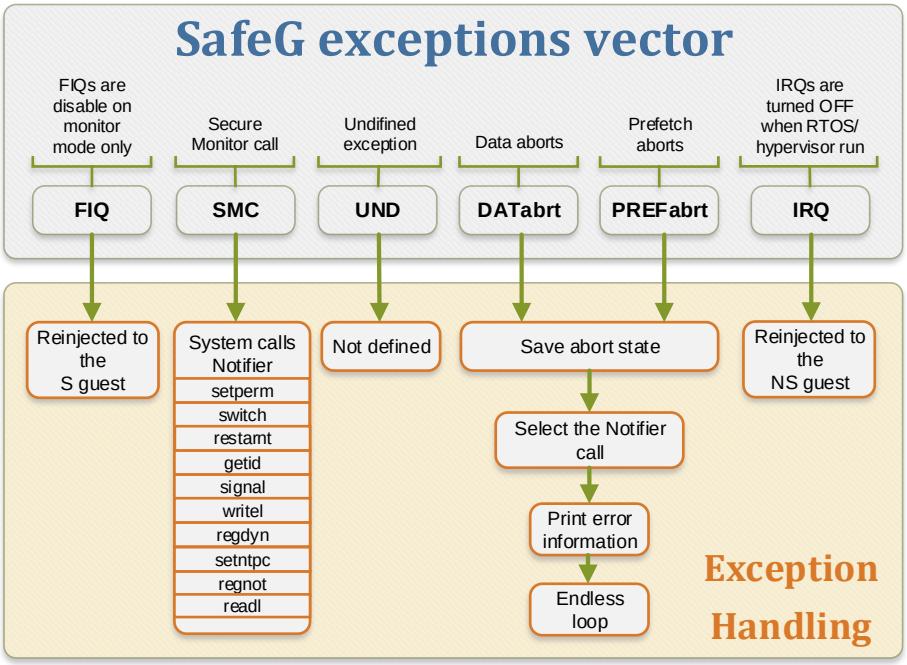


Figure 2.6: SafeG hypervisor Exception Handling Overview. This hypervisor has five defined exceptions (represented in grey): FIQ, IRQ, SMC, Data aborts and Prefetch aborts. The handlers are defined in orange.

To handle data aborts and prefetch aborts, this hypervisor provides the possibility of using guest handlers previously registered in the Notifier. If it is empty, the SafeG prints the error information and jumps to an endless loop.

2.5 Fault tolerance concepts and Health-monitor

The fault tolerance can be simply defined as the prevention of faults from becoming failures [ASTM08]. Since these terms are wrongly often used as synonyms, as they describe different concepts, a detailed description of them is needed [ALRL04]. Also the terms Reliability and Availability are important to define due to fact that they characterize the system performance in terms of fault tolerance [ASTM08].

- **Faults** - A fault is the defect that causes the error.
- **Errors** - An error is a corrupted system state that may cause a subsequent failure.

- **Failures** - A failure is an event that occurs when a system deviates from the correct work flow.
- **Reliability** - probability that a system will perform its intended function satisfactorily, for a specified period of time.
- **Availability** - probability that a system is performing its required function at a given point in time.

Implementation of fault tolerance involves in two different subsystems: error detection and system recovery. Errors detection can be a watchdog, to detect abused deadlines, or a more complexing mechanism like DFI (Data Flow Integrity)[LMTP18] and CFI (Control Flow Integrity)[WJ10] that detects not-expected data and instructions flows. System recovery aims to eliminate the error from the system state and may diagnose the fault, preventing it from being reactivated. The complexity of the mechanism is increased with the fault tolerance classification of the system. Critical systems in which a failure can cause life losses, the system's availability and reliability needs to be as high as possible [Sie91]. Improving the fault tolerance usually depends on implementing extra error detection mechanisms and even redundancy, features that add more variables to the system cost equation: cost/reliability and cost/availability.

2.5.1 Basic techniques in error handling

After the detection of an error state, the recovery system is triggered to solve the error. There are three general techniques for error handling: backward recovery, forward recovery and compensation.

- The backward recovery, also known as rollback technique, is a technique in which the system is restored to a previous assumed error-free state. In this technique, the system state is stored periodically in predetermined checkpoints, to be able to recover from them [Cri82][XRR+95][RP12].
- In the forward recovery, also called rollforward technique [RLT78], the system is taken from an error state to a healthy state but by rolling forward to a future checkpoint. Since the roll is not a rollback, the deadlines and real-time constrains are not corrupted. This technique is more system demanding due to the predictable behaviour needed on checkpoints creation [XR96].

- In the compensation technique, the system contains enough redundant information so that an error do not compromise it's normal flow. Normaly, this technique do not depend on error detection but due to the multiples implementations of the critical parts.

2.5.2 Hypervisor's Health-monitors and Recovery mechanisms

The hypervisor's health-monitor is responsible to monitor the hypervisor's work flow or, in other words, it is the hypervisor's fault tolerance mechanism. During hypervisor's execution, the Health-monitor will compare the hypervisor's actions with it own patterns or redundant parts of itself in hardware. Not only the actions but the hypervisor's time spent on them can be monitored to control his sanity. If the Health-monitor detects issues with the hypervisor, the Recovery System activates the mechanism to restore the hypervisor to a healthy state.

Using the above definitions, the health-monitor main job is to detect errors as soon as possible to prevent failures and restore the system to a healthy state, eliminating faults and improving the system's reliability and availability.

Concerning health-monitor implementations, hypervisors' health-monitors are limited since the main goal is to achieve a system with very low execution overhead. The sections below describe health-monitors implementation in hypervisors.

2.5.2.1 SafeG

The SafeG hypervisor supports a GPOS health monitoring with the ability to monitor, suspend, resume and restart the GPOS from the secure world side (RTOS). Monitoring the GPOS status from the RTOS is possible because the GPOS resides in Non-Trust space memory, which is accessible from Trust state. To support GPOS interrupt monitoring (when appear, the frequency and inter-arrival time), IRQs are first processed by SafeG, which implements a Secure Monitor mode vector table, before being forwarded to the GPOS.

2.5.2.2 VOSYSMonitor

The VOSYSMonitor developed a secure world monitoring mechanism, despite not being present in the latest versions of this hypervisor. The developed mechanism was based on a watchdog turned on whenever the processor entered the safe world. With each context switch, this counter is reset, measuring only the safe

world processor's time. This watchdog would activate a reset in the safe world if the time was greater than a certain threshold.

2.5.2.3 Discussion

Despite of the presented health monitors being taken into consideration, only one of them shares this thesis goal. SafeG provides a mechanism to control the GPOS from the secure side. This feature is considered a Health Monitor since it controls and monitor the non-secure guest health. However, it does not improve the secure world security.

The watchdog introduced by VOSYSMonitor's developers aims to control the time spent in the secure world execution, which completely removes the possibility of running non-secure code in the secure world schedule window. But as described in the hypervisor's presentation document, the mechanism failed for two reasons [LCP+17]: 1) Turning watchdog on and off adds a significant overhead to the context switch; 2) The ARM Physical Secure Timer was chosen for the watchdog, but due to the interrupts being treated as IRQs they can no longer be turned off once the processor is in secure mode, thus breaking the design of the hypervisor itself.

2.6 Non-Encrypted Hash functions and Checksums

In order to detect error states, this thesis sanity check is based on secure memory patterns. To detect secure memory faults it compares the current memory state with a sane state. In order to reduce time and resources of comparing all the of bytes of memory, it compresses the memory states into keys. Although this task will be implemented in hardware, the function to convert states to keys have to follow two main metrics: fast conversion and easy implementation. The two metrics are interconnected with each other in the following way of if the conversion is a complex algorithm, it can compromise the fast conversion metric. The conversion occurs twice per context switch, one for the healthy state and one for the unknown state, which it is mandatory to be as fast as possible. The second metric removes encryption algorithms from the list since they turn to be more complex. Also, considering that the generated keys are not accessible from outside of Health-monitor, the encryption is useless.

Due to the simplicity of the non-encrypted hash functions and the checksums, the sections below describes some of them.

2.6.1 FNV-1 and FNV-1a

The Fowler–Noll–Vo or FNV is a hash function created by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo designed to be fast while maintaining a low collision rate [Lan]. They took the main idea from a comment reviewing the IEEE POSIX P1003.2 in 1991 and due to the speed of the algorithm, it is used to hash large data. The flowcharts below describe the basic algorithm composed by a multiplication and an multiplexer operation, and the constants FNV_prime and FNV_offset_basis. The FNV-1a is a FNV-1 alternative hash function that differs only in the order of the operations. The FNV_prime constant consists on a prime number and it is dependent of the size of the key. The FNV_offset_basis is an offset to achieve better dispersion [FNV+11].

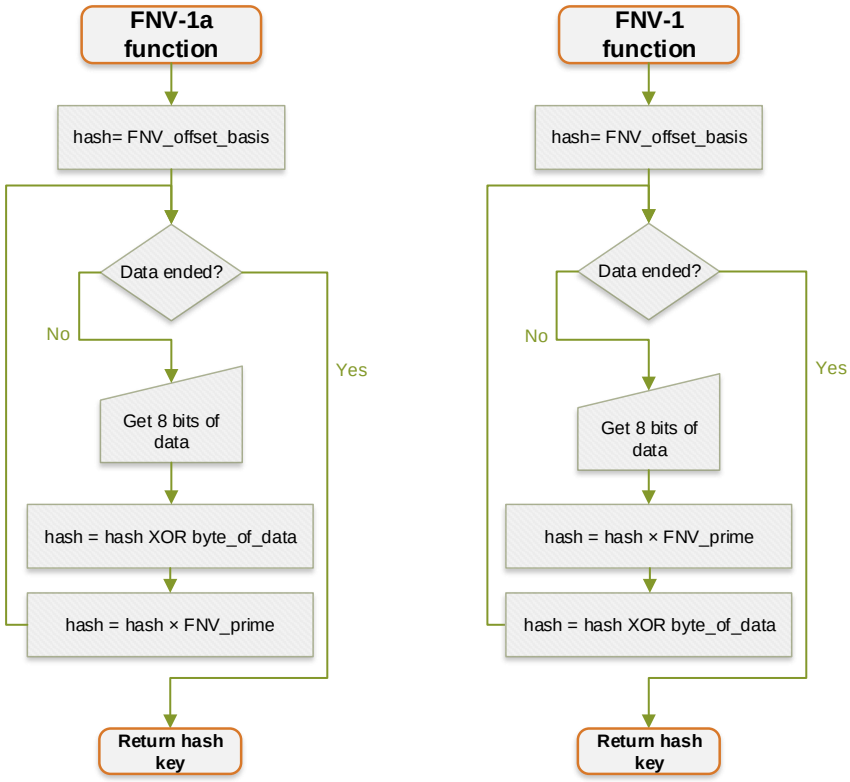


Figure 2.7: FNV-1a and FNV-1 algorithms flowchart. On the left, the FNV-1a hashes eight bits of data per cycle by executing: 1) a XOR operation between the byte and the previously produced hash key, 2) a multiplication between the resulting number and a predefined prime number. The function ends when all the data is fully hashed. On the right algorithm, the basics are the same but the operations are swapped.

To simplify the selection of the right constants for the hash function, the creators afford a table with the best pair (FNV_prime and FNV_offset_basis) for each desired size of the key.

Table 2.2: Suggested values for FNV constants

Size in bits	FNV prime	FNV offset basis
32	16777619	0x811c9dc5
64	1099511628211	0xcbf29ce484222325

2.6.2 SDBM

The SDBM hash algorithm was created for SDBM [SDB] database library that is a public-domain reimplementation of NDBM. NDBM is an Application Programming Interface (API) made to maintain key/content pairs in a database and it uses hash functions to allow a programmer store keys and data in the database tables. As it is possible to see in the flowchart of Figure 2.9, the function is a compound of simple mathematical operations and binary shifts.

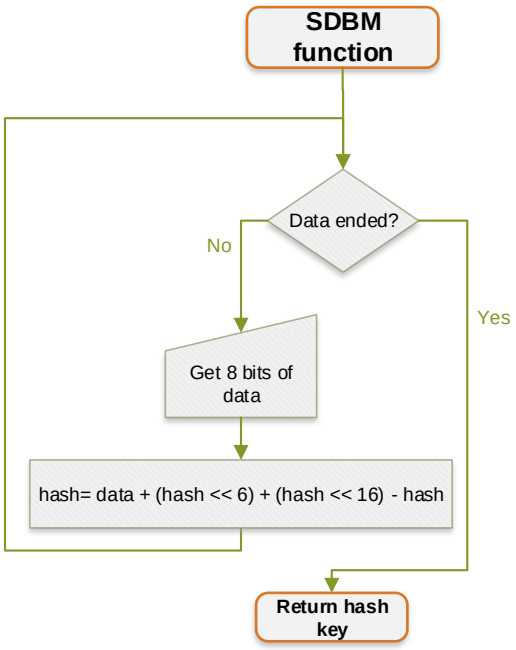


Figure 2.8: SDBM algorithm flowchart. The SDBM hash function is a manipulation of the previously produced key plus the input data untouched. It hashes eight bits of data per cycle and ends when all the data is hashed.

2.6.3 DJB2

The DJB2 hash function was reported by the mathematician and computer scientist Dan Bernstein in an open group with the form of $X = ((a \times X) + c) \bmod m$, where the X is the value produced, c the "increment", a a multiplier constant and m the "modulus" that truncates the output value.

This function is similar to LCG (Linear Congruential Generator) functions which are a class of functions that generate pseudo-random numbers proving the good dispersion of values produced [Knu97]. With an $\boldsymbol{m}$ of 2^{32} and the constant $\boldsymbol{a}$ having the value of 33 the function can be converted into a less operations demanding form: $X = (X << 5 + X) + c$.

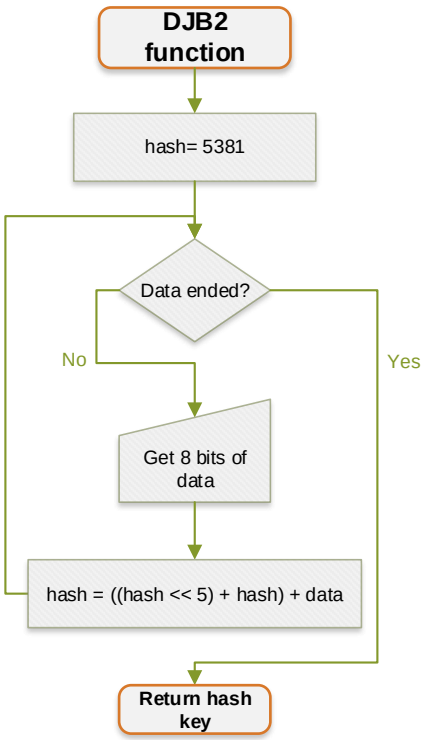


Figure 2.9: DJB2 algorithm flowchart. This algorithm uses the 5381 number as the start-up key value in order to achieve a better distribution. It hashes eight bits of data per cycle and ends when all the data is hashed.

2.6.4 Murmur

Austin Appleby created the Murmur hash in 2008 and since then several variants and versions of this hash were made and submitted online [App08]. The main difference from the others hashes to this one is the number of input bytes (8 bits for the others and 32 bits for this one) for each generated key. This characteristic

reduces the number of cycles needed to hash the same amount of data to a quarter but increase the hardware needed due to the redundancy. Figure 2.10 exposes the algorithm which despite the number of operations, it can be fast since some of them can be parallel operations due to their independence.

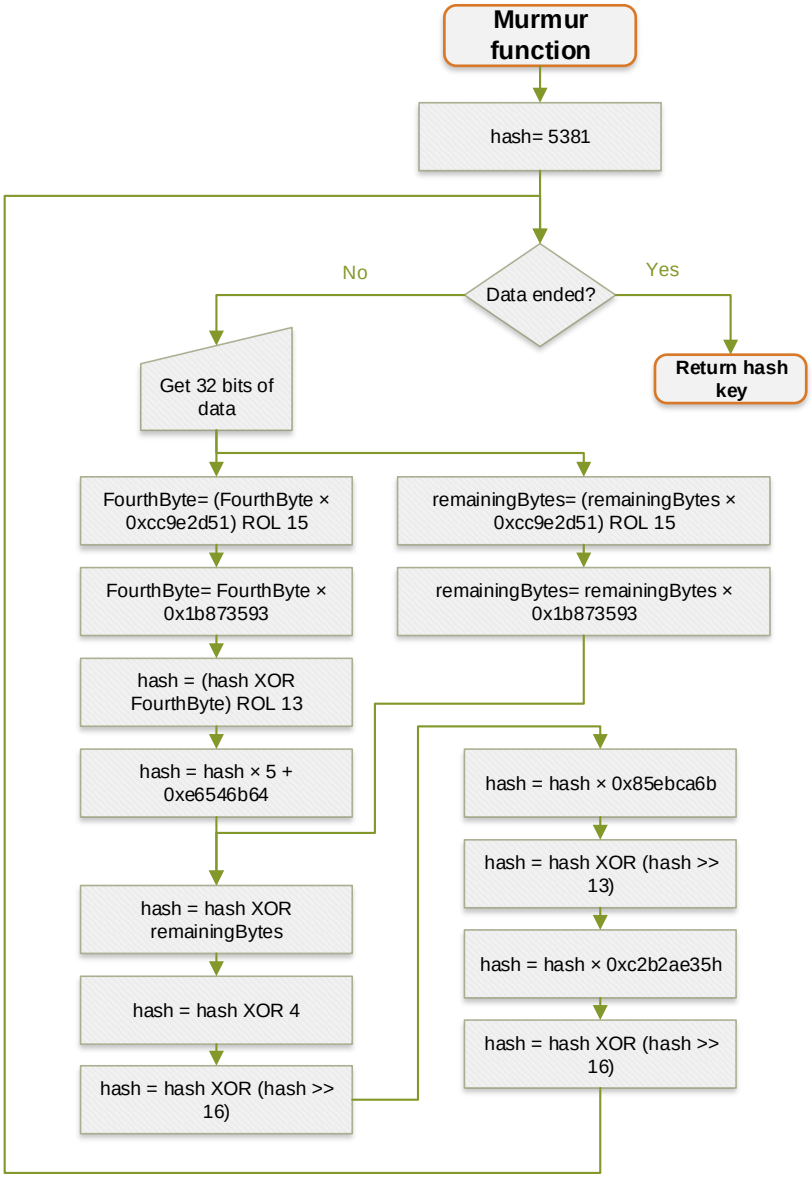


Figure 2.10: Murmur algorithm flowchart. This algorithm hashes 32 bits of data by modifying its fourth byte and the remaining ones in parallel operations. The key produced by Murmur algorithm is a 32 bit key.

2.6.5 CRC32 Checksum

CRC checksum was first proposed by W. Wesley Peterson in 1961 with the purpose of use redundancy for error detection in communication networks [PB61]. In this algorithm, the generated key results from a polynomial long division, where the message is the dividend, the polynomial is the divisor and the quotient is discarded. To reduce the time of the calculation, the implementation of this checksum uses a lookup table with the constants of the generator polynomial. Figure 2.11 describes the CRC32 algorithm that is a variant of this checksum for a 32 bits output value and also the lookup table algorithm. The variable *pos* is the table index used to get the right value from the lookup table.

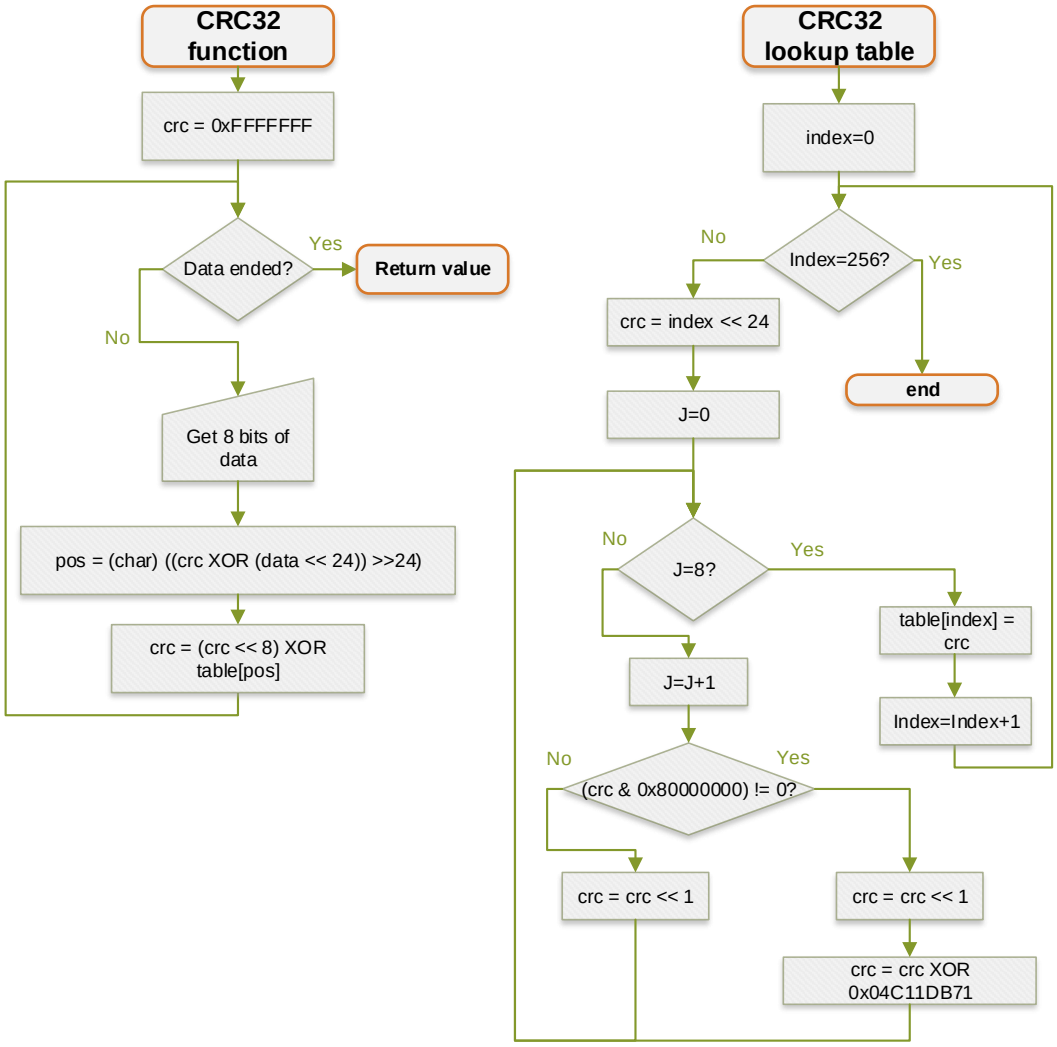


Figure 2.11: CRC32 and respective lookup table flowcharts algorithms. On the left is represented the CRC32 algorithm simplified by its lookup table. On the right, the algorithm to produce the lookup table.

2.6.6 Discussion

Although all algorithms share the same property of conversion of an input value to a shorter output value, CRCs and hash functions have different design purpose. The CRCs are designed to detect not forced errors in data [PB61] while general purpose hashes, like the above algorithms, are optimized to reduce bias in the output value even when the input is biased. Even if the CRCs algorithms are designed for error detection, the origin of hypervisor errors is unknown (they can be forced to errors on the NS side), which makes CRC a not so ideal option.

3. Platforms and Tools

This chapter exhibits the groundwork platforms that support this thesis. First will be described the Zybo board, the Zynq device where the entire system, LTZVisor plus Health-monitor is deployed. Lastly, will be exposed the LTZVisor, the hypervisor on which the exception handler is implemented.

3.1 ZYBO Zynq-7000 SoC

The Zybo (Zynq Board) is an entry-level embedded software and digital circuit development platform developed by DIGILENT [DIL] . This board is built around the Xilinx All Programmable System-on-Chip (SoC) Z-7010, which integrates a dual-core ARM Cortex-A9 processor with Xilinx [XIL] 7-series field programmable gate array (FPGA) logic. Attached to the processor and the FPGA, this board provides a rich set of multimedia and connectivity peripherals. To expedite the system design and deployment on the board, the Xilinx's Vivado [Viv] Design Suite as well as the ISE/EDK toolset provide full compatibility with this board. These features make the Zybo a complete development kit to handle diverse projects with no additional hardware needed.

3.1.1 Zynq-7000 family

As it is possible to see in Figure 3.1, Zynq-7000 family integrates two distinct systems: The Processing System (PS) and the Programmable Logic (PL). The first one is composed by resources to handle software: Dual or single-core ARM® Cortex™-A9 MPCore; On-chip memory; External memory interfaces; I/O peripherals and Programmable Logic interconnects.

The PL provides configurable logic in order to create dedicated hardware: configurable logic blocks (CLBs); configurable ports to the block RAM (BRAM); DSP slices with a 25 x 18 multiplier and 48-bit accumulator; an user configurable analog to digital convertor (XADC); Clock management tiles (CMT); a configuration

block with 256b AES for decryption and SHA for authentication; a configurable SelectIO™ technology and optionally GTP or GTX multi-gigabit transceivers and an integrated PCI Express® (PCIe) block [XI15].

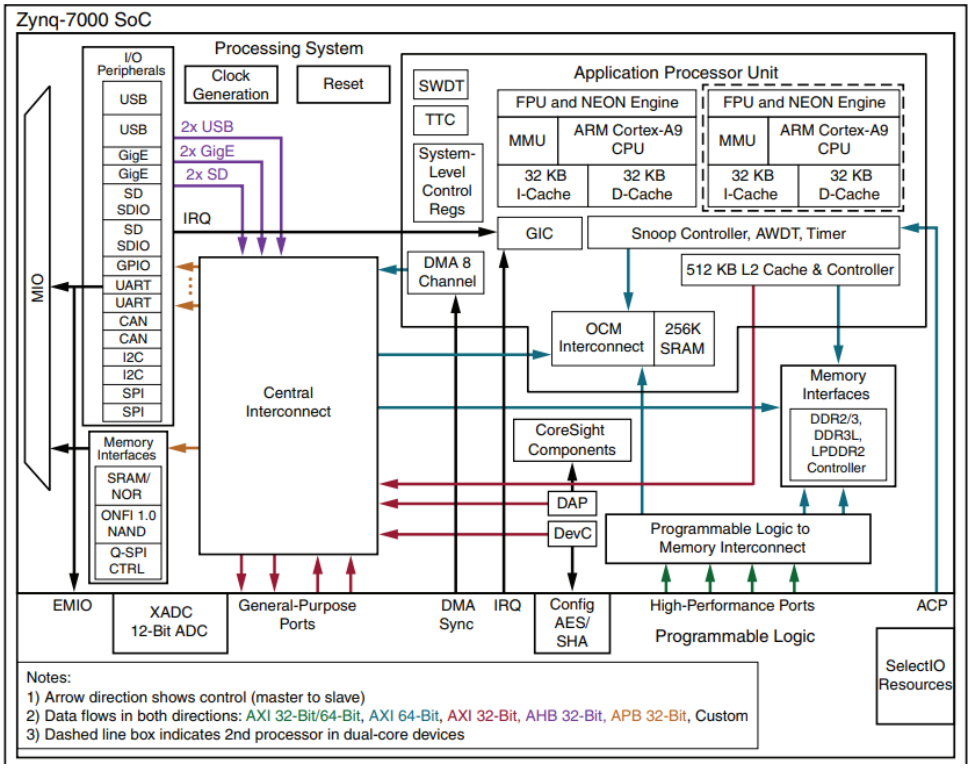


Figure 3.1: Zynq-7000 SoC overview [XI18a].

This system compound extends the functionality of simple processor with FPGA features, providing flexibility to the FPGA with the dynamic reconfiguration. This allows PL reconfigurations at run-time reducing hardware costs. To enhance both sides, this SoC provides three types of PL-PS communication based on AXI protocol [XI18b]:

- General Purpose (AXI_GP) - a communication with 32-bit data bus aims to genereal purpose without hight performance needs;
- High Performance (AXI_HP) - a comunnication with 32-bit or 64-bit data bus but with high bandwidth datapaths to the memories and FIFO buffers (allow burst transactions);
- Accelerator Coherency Port (AXI_ACP)- a low latency communication that allows memory access to from the PL with cache coherency.

3.1.2 AMBA Advanced eXtensible Interface

The AMBA Advanced eXtensible Interface (AXI) protocol is part of ARM AMBA, a microcontroller-oriented buses family. Being one of the standard protocols for ARM SoC, this protocol provides features like address/control and data separation, unaligned data transfers and burst-based transactions. They are achieved by implementing five independent transaction channels: two for addressing, two for data and one for control/confirmation. Figure 3.2 describes the basic write and read transactions behind AXI protocol [Lim03].

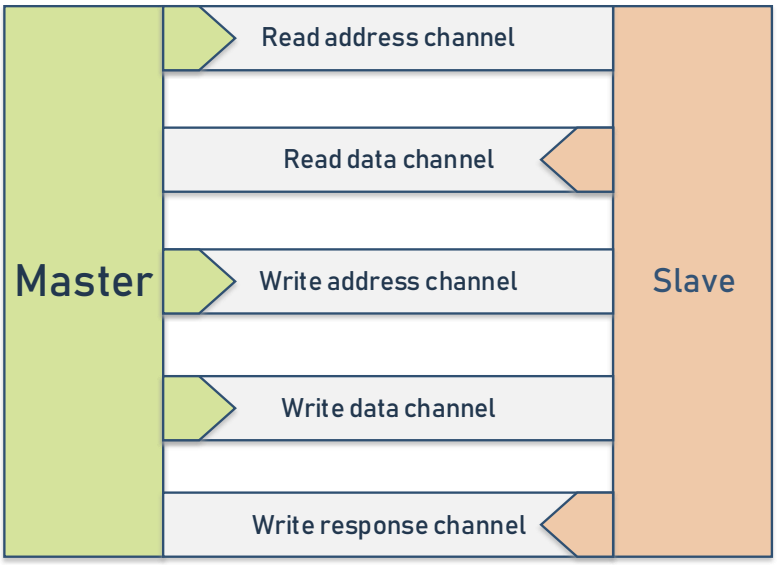


Figure 3.2: AXI communication overview.

In order to both master and slave be able to control the transmission rate, the handshake process is made by VALID/READY for each individual channel. As Figure 3.3 demonstrate, the information is set when the VALID (controlled by the source) and READY (set by the destination) are asserted simultaneously.

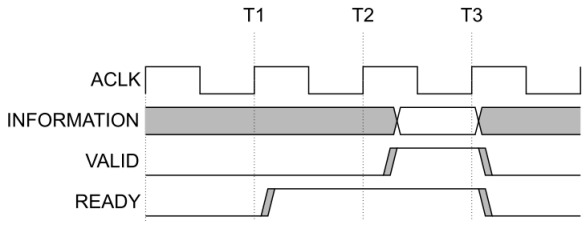


Figure 3.3: VALID/READY handshake.

Although the most recent version of AXI is AXI5, the analyzed AXI version is the second (AXI4) since it is the version provided by the Zybo board. The AXI4

can be characterised in three interface types according to the addressing type: AXI-Full and AXI-Lite as Memory-mapped interfaces, and AXI-Stream as Point-to-point interface. The AXI-Lite is the simplest AXI protocol since implements only the basic features: five channels and their independent signals [Lim03, XI11]. Also, this reduced protocol does not support burst transactions of more than one data access and its accesses use the full width of the data bus (32bits or 64bits). Among the signals below exposed, the protocol has two global signals: *ACLK* as the Clock signal that serves as tick and the *ARESETn* that resets the channels.

Table 3.1: AXI-Lite Address Write Channel Signals.

Signal	Descripton
AWVALID	Signal controlled by the master when it writes a valid write address.
AWREADY	Signal controlled by the slave when it is ready to accept a write address.
AWADDR [31:0]	Bus that holds the initial write address.
AWPROT [2:0]	Bus that defines the privileged state of the write access (00b means normal, 01b means privileged and 10b means secure).

Table 3.2: AXI-Lite Address Read Channel Signals.

Signal	Descripton
ARVALID	Signal controlled by the master when it writes a valid Read address.
ARREADY	Signal controlled by the slave when it is ready to accept a Read address.
ARADDR [31:0]	Bus that holds the initial Read address.
ARPROT [2:0]	Bus that defines the privileged state of the Read access (00b means normal, 01b means privileged and 10b means secure).

Table 3.3: AXI-Lite Data Write Channel Signals.

Signal	Descripton
WVALID	Signal controlled by the master when it writes a valid data on the Data Channel.
WREADY	Signal controlled by the slave when it is ready to accept a new data.
WDATA [31:0]	Bus that holds the data write.
WSTRB [3:0]	Bus that indicates which bytes of the write data bus are valid for each transfer of data.

Table 3.4: AXI-Lite Data Read Channel Signals.

Signal	Descripton
RVALID	Signal controlled by the slave when the data required is available on the Read Data Channel.
RREADY	Signal controlled by the master when it is ready to accept the data.
RDATA [31:0]	Bus that holds the data required.
RRESP [1:0]	Bus that indicates the transfer status.

Table 3.5: AXI-Lite Write Response Channel Signals.

Signal	Descripton
BVALID	Signal controlled by the slave when the response is available.
BREADY	Signal controlled by the master when it is ready to accept the response information.
RRESP [1:0]	Bus that indicates the write transfer status.

In comparison, the AXI-Stream protocol is slightly different. The address related signals are removed due to the Point-to-point characteristic. Also, only one data channel is present since the data always flows from the master to the slave. Attached to this channel to inform the slave that the packet is over, the new *TLAST* signal controls the data flux. The table 3.6 describes the each signal using the following terms:

- $\boldsymbol{n}$ - Data bus width in bytes
- $\boldsymbol{i}$ - Configurable TID width (8bits max)

- ***d*** - Configurable TDEST width (4bits max)
- ***u*** - Configurable TUSER width

Table 3.6: AXI-Stream Signals.

Signal	**Source**	**Descripton**
ACLK	Clock source	Clock signal.
ARESETn	Reset source	Reset signal (active LOW).
TVALID	Master	HIGH when the master is driving a valid transfer.
TREADY	Slave	HIGH when the slave is ready to accept.
TDATA[(8n-1):0]	Master	Data payload.
TSTRB[(n-1):0]	Master	Indicates if TDATA is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	Indicates if content of the associated byte of TDATA is processed as part of the data stream or ignored.
TLAST	Master	HIGH when the last packet is send.
TID[(i-1):0]	Master	Stream identifier.
TDEST[(d-1):0]	Master	Routing information.
TUSER[(u-1):0]	Master	User defined information that can be transmitted alongside the data stream.

3.1.3 AXI Direct Memory Access

The AXI Direct Memory Access (DMA) is an IP core developed by Xilinx in order to provide high-bandwidth direct memory accesses, taking advantage of the AXI-Stream data protocol. The AXI DMA provide two different utilisation modes: Register Mode and Scatter / Gather Mode that [XI18a]. The first is a simple interface that allows the control and configuration based on AXI-Lite registers although the Scatter/Gather Mode allows more complex configurations such as transfer noncontiguous blocks of data in one continuous transfer. The figure 3.4 describes a typical AXI DMA system configuration with the two modes associated channels and registers.

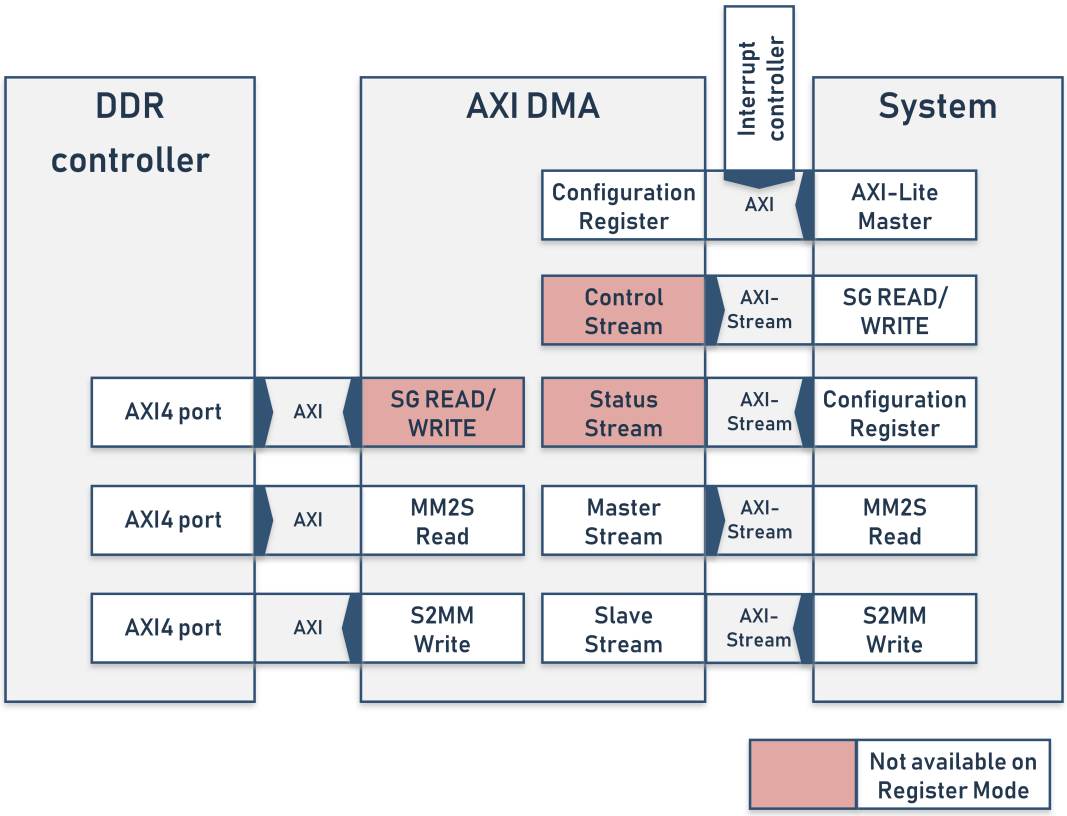


Figure 3.4: Typical AXI DMA system configuration.

Three different components are represented on the figure above: the DDR controller, the AXI DMA and the implemented device that will access the memory. The DDR controller provides two AXI ports that allows to access memory. Attached to these are the *MM2S Read* and *S2MM Write* channels from the AXI DMA module in order to read from memory and to write to memory correspondingly. Also, this module has a slave port (*Configuration Register*) to configure the communication streams from the device to the DMA and vice versa. These streams allows independent memory reads and writes from the device with the full support of the AXI Stream protocol.

3.1.4 TrustZone Architecture on the Xilinx Zynq-7000

In order to extend the ARM TrustZone feature to the board resources, the Zynq-7000 AP SoC includes a TrustZone module with 21 registers [XI14]. Each mapped register configures the security of a distinct board component, providing not only design flexibility (resource configuration as Secure or Non-Secure) but also resource sharing between different Trust levels.

3.1.4.1 SDIO, Ethernet, USB, QSPI, APB registers

The TrustZone module provides three different registers to configure the Secure Digital Input/Output interface: two registers (the *security2_sdio0* and *security3_sdio1*) to configure the security of SDIO0 and SDIO1 slaves, and one register (*TZ_SDIO*) to configure the SDIO controllers. In terms of Ethernet and USB, the module supplies two registers that configures their security (*TZ_GEM* and *TZ_USB* respectively). Concerning the Quad Serial Peripheral Interface, the module offers a register (*security4_qspi*) to configure the QSPI slave. Also, this module has two different registers to configure the Advanced Peripheral Bus (APB): one for configure the APB slaves security (security6_apb_slaves) and one for resources access control (security_apb).

3.1.4.2 SMC register

In order to control the Secure Monitor Call, the Trustzone module provides a register (*security7_smc*) to define the SMC as secure or non-secure.

3.1.4.3 AXI registers

To configure PS and PL masters for PS-PL communications, this module provides four registers: two to select the security of the PS General Purpose (GP) AXI masters (*security_fssw_s0* configures the master number 0 (*M_AXI_GP0*) security and *security_fssw_s1* configures the master number 1 (*M_AXI_GP1*) and two for PL masters (*TZ_FPGA_M* for general purpose and *TZ_FPGA_AFI* for high performance masters).

3.1.4.4 DMA registers

Since the TrutZone features are extended to the DMA, there are three registers to configure the DMA Controller security: one to define the DMA Controller operation state (*TZ_DMA_NS*); one to select the security state of the external interrupt generated by DMA Controller (*TZ_DMA_IRQ_NS*); and one for peripherals attached to the DMA Controller (*TZ_DMA_PERIPH_NS*).

3.1.4.5 Memory registers

To configure the memory security, four registers with 32 bit length are available: three to configure the On Chip Memory (OCM) and one for DDR. The *TZ_OCM_RAM0* register configures the first 128KB of OCM. Each bit represents the security status for a 4 KB page; the *TZ_OCM_RAM1* register is similar

to *TZ_OCM_RAM0*, but configures the second 128KB of OCM. Also, each bit represents the security status for a 4 KB page starting at 128 KB; the *TZ_OCM* register configures the third 128KB of OCM. Finally the *TZ_DDR_RAM* register that configures the DDR security. Each bit represents the security status for a 64 MB section.

3.2 LTZVisor

As previously introduced, the LTZVisor main goal is to afford virtualized environments providing coexistence between RTOS and GPOS [PTM16].

This section thoroughly describes the LTZVisor design, presenting how CPU virtualization and memory isolation are ensured, detailing how MMU and caches are configured, illustrating how device isolation is achieved and explaining how interrupts and time are managed in order to achieve real-time.

3.2.1 Virtual CPU

Since the LTZVisor is a Trustzone-based hypervisor, it relies on Trustzone for CPU virtualization. As mentioned before, the technology provides two virtualized CPUs of each hardware core: secure world and non-secure world. In the LTZVisor design, each guest OS runs in a different world to minimise the number of registers saved and restored in each partition-switching operation since each virtualized world contains an individual copy of banked registers. On the secure side, the Virtual Machine Control Block (VMCB) is composed by 16 registers:

- General purpose registers: R0-R12
- System mode registers: the Stack Pointer (SP), the Link Register (LR) and the Saved Program Status Register (SPSR)

Although the TrustZone provides a monitor mode to context-switching, it is not provided any additional registers for this mode in the secure world VMCB. On the non-secure side, the (VMCB) is composed by 25 registers:

- General purpose registers: R0-R12
- Supervisor mode registers: SP, LR and SPSR
- System mode registers: SP, LR and SPSR
- Abort mode registers: SP, LR and SPSR
- Undefined mode registers: SP, LR and SPSR

The reduced size of the secure VMCB endorse the secure side real-time features since it promotes faster partition switches from non-secure world to this world.

For the IRQ and FIQ modes, the General Registers (R8-R12), as well as the SP, LR and SPSR registers are not banked. So they are defined from the beginning in which world they belong.

Among the aforementioned registers, there are some registers shared by both worlds. Although they can be read from both worlds, they are only modifiable from the secure side. An example of these are: the System Control Register (SCTLR) and the Auxiliary Control Register (ACTLR) that provide control and configuration over memory, cache, MMU, AXI accesses, etc. Since they configure both worlds, the LTZVisor is responsible to manage these registers before the guests boot process as well as initialise both VMCBs.

3.2.2 Scheduler

Typically, the hypervisor scheduler and OS scheduler are detached from each other since they schedule different features: an hypervisor schedules guests while a guest schedules its own tasks. This model does not fulfil the real-time environment needs, so the hypervisor implements an asymmetric scheduler in order to only schedule the non-secure guest OS on the idle periods of the secure guest OS. This schedule process is carried by the secure side, with higher scheduling priority than the non-secure which provides a real-time environment to the secure partition.

3.2.3 Memory Partition

On TrustZone SoCs without VE, the MMU provides only a single-level of address translation instead of the traditional two-level that grants the execution of unmodified guest and ensures spacial isolation. To overcome the problem, the Trustzone allows for a memory configuration that separates the memory into different security segments. These memory regions can be defined with a specific granularity which in Zybo Zynq-7000 platform is 64MB. As Figure 3.5 illustrate, each bit of TZ_DDR_RAM selects the secure state of the corresponding 64MB section. Since the deployable platfrom has only 512MB of DDR RAM and, both hypervisor and the secure VM have low memory footprint, the first seven sections are configured as non-secure and the last section as secure.

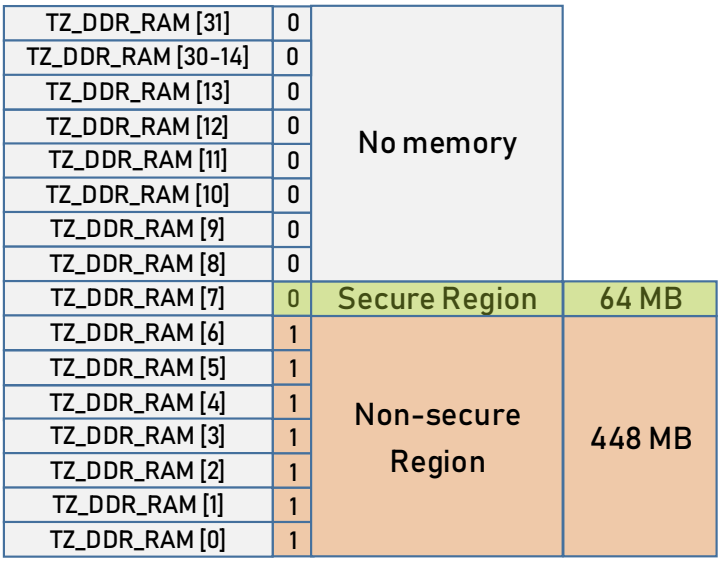


Figure 3.5: Memory partition in Zybo platform.

3.2.4 MMU and Cache Management

With the TrustZone extension, the processor provides two distinct interfaces to MMU, delivering separate virtual-to-physical memory tables for each world. The virtualized MMU has an individual copy of the Translation Table Base Register (TTBR) and an independent configuration. This virtualization feature accelerates the world switching since it removes extra software to validate translation lookaside buffer (TLB) entries.

At cache-level, the same TrustZone isolation is available. The processor caches also have the NS bit, not directly accessible by system software since it is set by hardware. This bit stores the processor security state that tries to access the memory. LTZVisor's performance is improved since there is no need to cache management mechanisms to eliminate memory leaks whenever guest-switches happen.

3.2.5 Device Partition

The devices partition is handled without the LTZVisor interference. The TrustZone allows devices to be configured as Secure and Non-Secure and with LTZVisor pass-through policy, the devices are managed directly by guests. To guarantee isolation, devices are not shared between worlds. Devices assigned to the RTOS are configured as secure and devices assigned to the GPOS are configured as non-secure ensuring that the GPOS cannot ruin the RTOS expected execution.

3.2.6 Interrupt Management

The Trustzone GIC supports the configuration of interruption as secure or non-secure. This provides provides an additional hierarchy between secure interruptions and non-secure interruptions.

The LTZVisor claims the FIQ as secure interrupts and IRQ as non-secure interruption due to the GIC interrupts design. The GIC allows the possibility of masking the IRQs when the secure guest is running, removing the interference caused in the Secure World by non-secure interrupts and assuring real-time. However, the FIQ are configured to interrupt both worlds as it is possible to see in Figure 3.6.

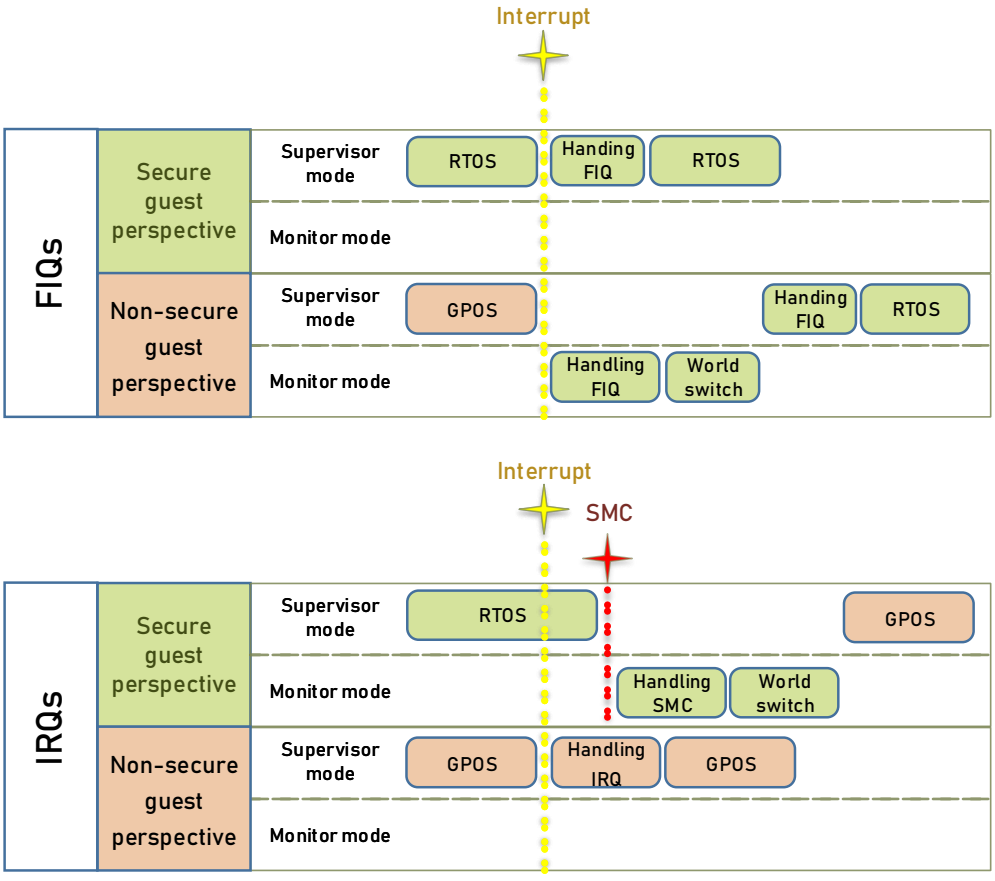


Figure 3.6: Interruption model in LTZVisor. On top, it is represented how FIQs are handled in LTZVisor. On the bottom part, the same is represented but for IRQs.

In order to improve the interrupt latency, the FIQ configuration changes depending on which world the processor is running:

- If the non-secure guest is running, the FIQ handler will run in monitor mode in order to provide the context-switch. The world switch is done and the secure guest handles the interruption.

- If the secure guest is running, the FIQ will be attended by the secure guest without hypervisor interference.

The figure above describes how FIQs and IRQs are handled depending on which guest is running. When the secure guest is running and an IRQ is triggered, it is ignored until the SMC interruption (interruption triggered by the scheduler) comes.

3.2.7 Time Management

In the hypervisor there are are two different levels of timing: the real level that provides the hypervisor tick and the virtual level normally provided by the hypervisor to the guests. The virtual level can be full virtualized or it can be an interface to the real level. It is considered virtualized when the guest becomes inactive, the timer stops and restart as soon as the guest becomes active again. On the other hand, as an interface, the timer runs independently from the guest but as soon as the guest becomes active the hypervisor needs to update the guest timing structure. The LTZVisor isolates completely the time of the worlds providing one hardware clock for each world. Since the TrustZone extends the security to the devices, the hypervisor configures the non-secure guest clock as a non-secure device. By providing two clocks LTZVisor guarantees that the timing structures of the two worlds are always updated and the RTOS miss none tick system.

3.2.8 Exception Model

In order to control both worlds exceptions, the LTZVisor exception model routes the external aborts (SMC exception included) to the monitor rather than to abort mode.

As described in the Figure 3.7, the monitor is responsible to handle all external aborts (defined as other than MMU and Debug faults). With this configuration the hypervisor can trace secure access attempts from the non-secure side, which is an non-secure OS malfunctioning indicator, and take action to prevent secure side failure.

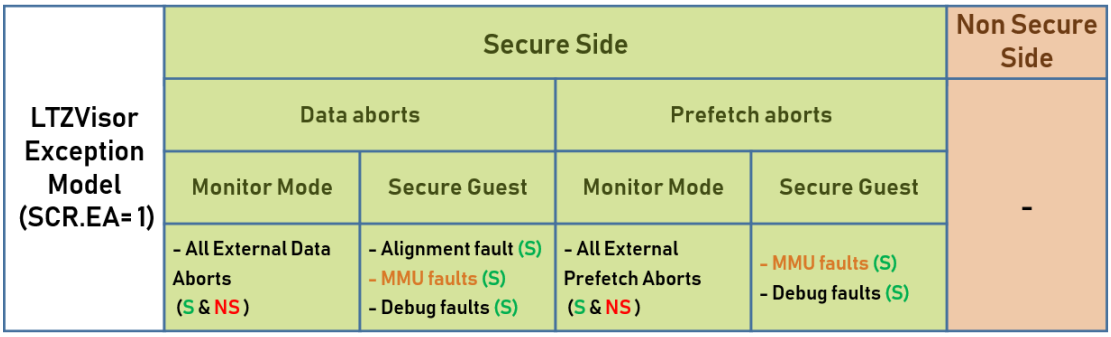


Figure 3.7: LTZVisor Exception Model. Secure exception are represented in green and non-secure exception in red.

4. Implementation

This chapter addresses the developed system implementation. Since this thesis has two different goals, one being the exception handling and the other the hardware mechanism, the implementation chapter is also divided accordingly. In the beginning of the exception handling section, it will be discussed the reason behind the implementation of the whole secure side exception handling instead of only monitor exceptions. After that, it is described the implementation of the exception handling and how the handler extracts information about the exception. The second section of the implementation chapter exposes the Health-monitor implementation divided in its modules. Afterwards, the integration with the LTZVisor is exposed as well as the *Health-monitor Interface*, an interface that provides Health-monitor access from the processing system. In the final part of this chapter, it is addressed the Intruder Module, a module that deliberately interferes with the secure memory.

4.1 Exception Handling

As described previously, the secure exceptions are divided in monitor exception and guest exceptions. Since the main goal is to provide an exception handling to the LTZVisor, it is mandatory that exceptions originated by the monitor are handled by hypervisor, not by secure guest. Considering that the secure exceptions contain hypervisor and secure guest exceptions, the LTZVisor is forced to have full control over the secure guest exception handler. Figure 4.1 illustrates the LTZVisor exceptions handler overview. It is possible to observe the four exceptions: two in the monitor exception vector and two in secure supervisor exception vector (Data and Prefetch aborts); and their structure.

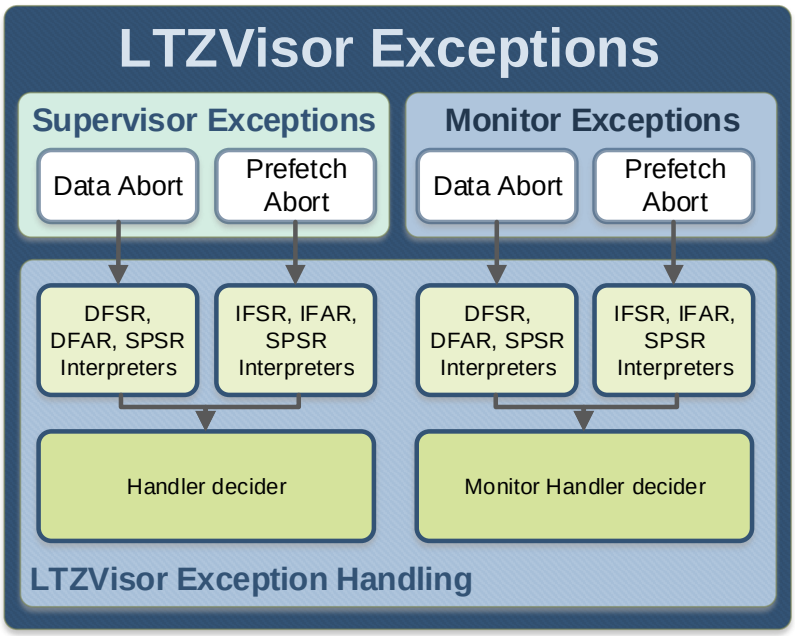


Figure 4.1: LTZVisor Exception Handling Overview.

The exception handling is divided in interpreters and handlers deciders. First each handler exposes the information about the exception by analysing the respective registers. Then, depending on the information collected, the handler decides the right action to handle the problem. Since there is a fault status register for each abort and not for each processor mode, both monitor and secure supervisor exceptions share the interpretation part of the handler. However, each processor mode has a different decider: the Supervisor exceptions have a Handler Decider and the Monitor exceptions have the Monitor Handler decider.

4.1.1 Secure Supervisor Data abort and Prefetch abort exceptions

The implementation of the secure supervised exceptions is delicate due to the fact that the secure guest and hypervisor exceptions are handled in the same exception vector. Although the Interpreters are shared, it is essential that the hypervisor attends its own exceptions. This way, interpreting which processor mode causes the exception is crucial to later take the right decision.

4.1.1.1 DFSR, DFAR, IFSR and IFAR interpreters

As mentioned before, the Fault Status Register stores important information about the exception like the DFSR and IFSR interpreters are responsible to extract

the information from the register and provide it to the hypervisor/user. Due to the registers uniformity, the DFSR and IFSR interpreters are similar, with the peculiarity of the DFSR providing an extra bit to differentiate if faulted instruction was a read or a write instruction. As depicted in Figure 4.2, there are five bits from the FSR that are take in consideration. The already explicated *WR* bit and the four Fault Status bits.

Figure 4.2: Fault Status register masked bits. The *FS* bits describes the fault status and the *WR* bit describes the type of the instruction (Write or Read).

The DFSR and IFSR are read into C variables as described in Listing 4.1. After masking the registers to get the *FS* bits, the value goes into a switch case with every fault status possible. Depending of the status value, the DFAR and IFAR can be valid or not.

Listing 4.1: Exception handle instructions to get the FSR and FAR registers. C code extract.

```
1 asm("MRC p15, 0, r3, c5, c0, 0" : "=r" (dfsr_value));
2 asm("MRC p15, 0, r3, c5, c0, 1" : "=r" (ifsr_value));
3 asm("MRC p15, 0, r3, c6, c0, 2" : "=r" (ifar_value) );
4 asm("MRC p15, 0, r3, c6, c0, 0" : "=r" (dfar_value) );
```

4.1.1.2 SPSR interpreter and NS bit

In order to extract information about the faulted processor state, the SPSR register is analysed. Although it is not an abort special register as the other mentioned before, the SPSR holds the information about the last execution processor mode. Since the last processor mode that executed was a faulted one, the information saved into the SPSR is important as it concerns the faulted state. The bits represented in Figure 4.3 are the ones to took into consideration from this register and the code to read its value is describe in Listing 4.2.

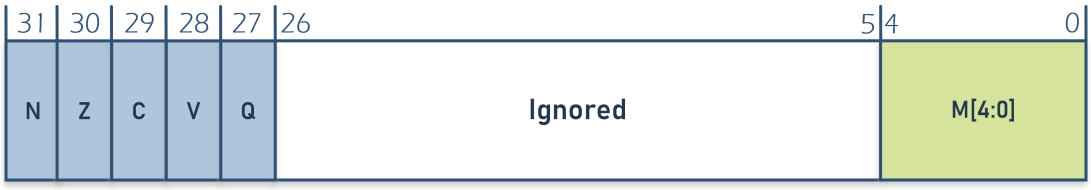


Figure 4.3: SPSR register masked bits. The **N**,**Z**,**C** and **V** represent the Negative, Zero, Carry and Overflow condition flag bits respectively. The Cumulative saturation bit (**Q**) and the Processor mode (**M**) are also extracted from this register.

Listing 4.2: Exception handle instructions to get the SPSR and NS bit. C code extract.

```
1 asm ("MRS r3, SPSR"  : "=r" (spsr_value) );
2 asm ("mrc p15, 0, r3, c1, c1, 0"  : "=r" (SCR_NS_BIT) );
```

After the SPSR register value being extracted, different masks are applied to get the bits values (**N**, **Z**, **C**, **V**, **Q**). To get the processor mode, the SPSR is masked with the 0x1f to obtain the last five bits. For each value a corresponding mode is attributed as Table 4.1 exposes.

Table 4.1: SPSR.M bit value interpretation.

Corresponding processor mode	Value
System	0x1f
Undefined	0x1b
Abort	0x17
Monitor	0x16
Guest (Supervisor)	0x13
System	0x12
IRQ	0x11
User	0x10

Although the secure state in Secure Supervisor exceptions is a redundant information (every exception that is handled here is necessarily from the secure side), the SPSR interpreter analyses the state anyway. Thus, both monitor and supervisor exceptions share the same interpreter. After getting the SCR, its last bit determines the security state of the processor.

4.1.1.3 Handler decider

After interpreting information, the handler determines the right action based on the faulting processor mode. As presented Figure 4.4, the *Handler decider* provides compatibility to both monitor and guest handlers.

Due to the disrupted nature of the hypervisor caused exceptions, its handlers are limited. By design, monitor exceptions are triggered when something not expected occurs to hypervisor. Consequently, the entire hypervisor is compromised and the only option is stop the execution. On the other hand, exceptions which were originated from the secure guest are not handled by the hypervisor as they are application-dependent. Instead of trying to recover from them, the hypervisor roots them to the secure guest. If nothing was defined by the secure guest, its execution is halted.

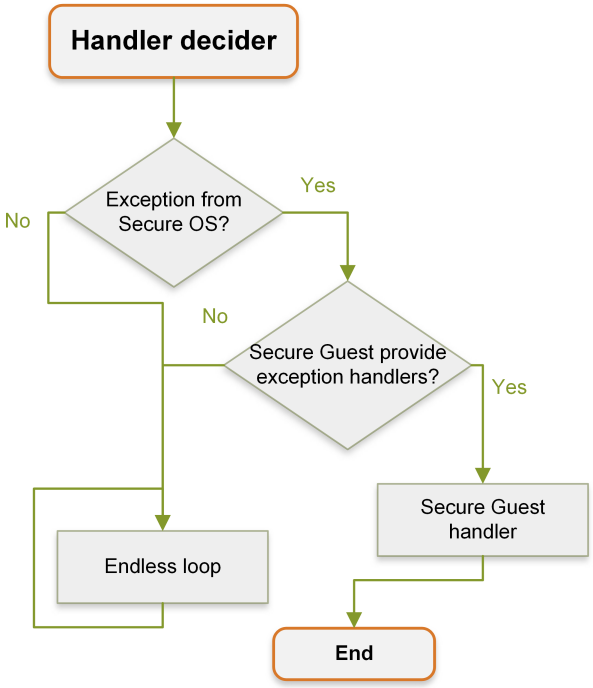


Figure 4.4: Handler decider flowchart.

4.1.2 Monitor Data abort and Prefetch abort exceptions

The external aborts' handlers share the same structure as the secure supervisor trapped exceptions. First, they provide information about the abort and then they are handle accordingly to their origin. In contrast to supervisor exceptions, the monitor exceptions come from different worlds, making the NS bit an important deciding factor. After parsing the information, the handler decides based on which world caused the abort:

- On non-secure external abort, it stops the non-secure guest execution;
- On secure external abort, the handler full stops the system execution.This is the only viable option because the secure external aborts are normally generated by bad secure side designs, making impossible to recover from it. Additionally, it is not recommended to run the non-secure application after the collapse of the secure world which involves hypervisor and secure guest.

4.1.2.1 Stopping the non-secure guest

In order to stop the non-secure guest from hampering the execution without modifying the expected hypervisor flow or increasing the time spent on context-switching, the non-secure guest execution is routed to a special LTZVisor section as demonstrate in Listing 4.3. This section is compiled alongside of the hypervisor but instead of being placed in secure defined memory it is attached to the non-secure guest code. After a non-secure external abort the non-secure PC register is loaded with the address of the code within the special section. Since the goal is to stop the execution, this code is a simple loop (Listing 4.4).

Listing 4.3: Hypervisor Linker script with the new non-secure monitor section. Linker code extract.

```
1 MEMORY
2 {
3 ...
4 DDR_MNS  (rwx)  : ORIGIN = 0x1BFFFF00, LENGTH = 0xEE
5 }
6 .monitornscode : {
7 _monitornscodestart = .;
8 *(.monitornscode)
9 } > DDR_MNS
```

Listing 4.4: Code to stop the non-secure guest execution. Assembly code extract.

```
1 .section .monitornscode, "awx"
2 LOOOP:
3 b .
```

Figure 4.5 shows the flow of the non-secure world execution. It is possible to observe that besides changing the non secure side execution, the flow outside of the non-secure guest is not really affected.

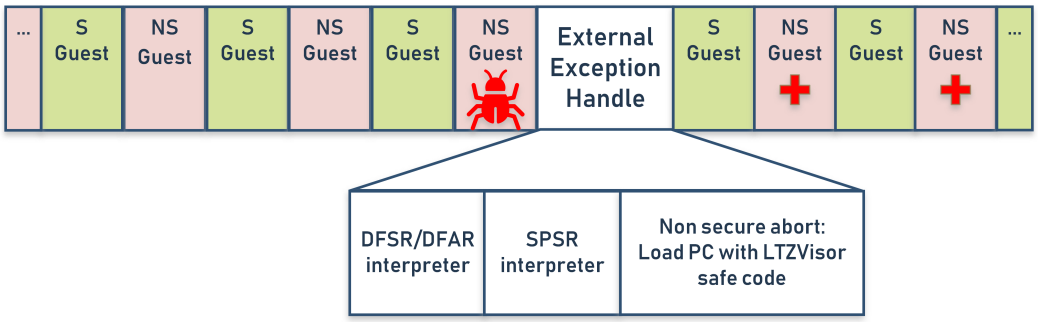


Figure 4.5: Non secure guest Execution flow. After the exception, the NS Guest runs the hypervisors's non-secure code (symbolised with a red cross).

4.2 Health-Monitor

The Health-monitor is the hardware package that includes the mechanisms to detect and recover from secure memory faults. Although they work in cooperation, the implementation of these mechanisms are completely independent. This allows for implementing different detection methods as substitutes or to work alongside this thesis mechanism, but maintaining the recovery system intact. As Figure 4.6 shows, the Health-monitor is composed by four hardware modules due to the four main tasks needed.

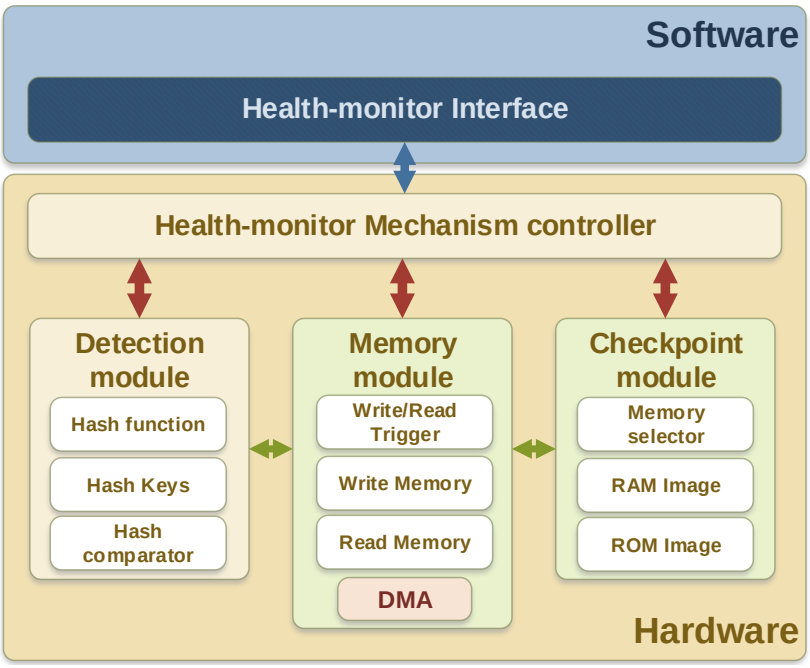


Figure 4.6: Health-Monitor Overview. The four main Health-monitor's components: 1) Detection Module, 2) The Recovery mechanism, composed by the Memory and Checkpoint modules (in green), 3) the Controller and 4) the Interface.

First, it is mandatory to have a detection module that flags anomalies which it is the Detection module purpose. In order to be able to read and write on the system memory, the Memory module was designed. The Checkpoint module is responsible for saving and controlling the memory check points. These checkpoints are healthy secure images gathered in run time. Since this module accommodates ROM and RAMs, it supplies the Memory module with a sane memory in case of failure. Together, they are the recovery mechanism. Lastly, in order to control all modules, a control unit is added to the Health-monitor.

4.2.1 Detection Module

The Detection module is responsible to detect memory anomalies based on data provided by the Memory module. The basic idea behind this module is to produce two different hash keys based on the secure world memory: one before and one during the non-secure world execution. By comparing the two keys, it is possible to detect secure memory irregularities caused by the non-secure guest. To not produce false positives when the secure guest change its memory, the keys are produced each time that the non-secure world is scheduled. Figure 4.7 depicts the Detection module overview.

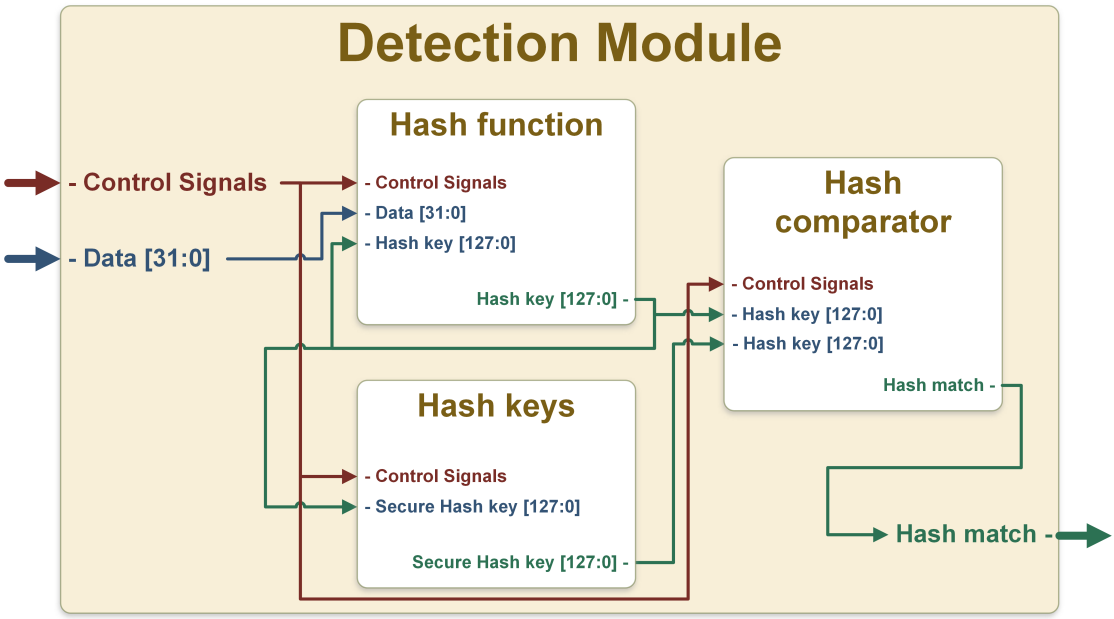


Figure 4.7: Detection module overview with the representation of the three sub-modules: Hash function, Hash Keys and Hash comparator. The red signals describe signals sent by the control unit; The blue signals are inputs; The green signals are outputs.

This module is divided in three subcomponents:

- Hash function - The implemented hash function converts the 32 bits of data received into a 128 bits hash key. Since all analysed functions produce 32 bits from 8 bits, this sub-module combine four hash functions in order to achieve the 128 bit output. To produce the full memory corresponding key, the secure memory is sliced into 32 bits. Although each slice produce a key, they are reintroduced in the hash function. Consequently, only the final one is take into consideration.

- Hash keys - After the hash function finalises the first key, it is stored to later be compared. The Hash keys sub-module is responsible to do exactly this, providing a 128 bit register that when signalled stores the input value.

- Hash comparator - After the Hash function produces the second key, the Hash comparator sub-module compares the outputs of the Hash funtion and the Hash keys. If the keys match, the signal *Hash match* is set to 1.

4.2.2 Memory module

With the DMA AXI technology, the Memory module creates an additional abstraction level for the remaining modules that require access to memory. Since the goal is to read and write the secure memory altogether, this module resumes the memory access in two possible operations:

- Read Mode - When this module is on Read mode, 32 bits of secure memory data is sequential outputted into the output data port. It begins on the first secure memory address and ends on the last used one.

- Write Mode- On Write mode, the module works the same way as in Read mode but instead of read, it sequentially writes the supplied data into the secure memory.

In order to achieve these modes, this module is fragmented in four sub-modules, as Figure 4.8 expose:

- AXI DMA - The IP provided by Xilinx to access the DMA technology. There are five AXI ports on this IP: one slave to configure the DMA, two masters to access the DDR memory (one to read and one to write) and two stream channels to get and set the data to the DMA.

- Write/Read Trigger - Based on the control signals sent by the Control Unit, this sub-module controls the DMA channels. Since the AXI DMA is configured by an AXI Lite communication, the sub-module implements an AXI Master to access the AXI DMA registers, configuring the channels attributes (e.g the bandwidth) and triggering the channels transferences.

- Read Memory and Write Memory - Both of these sub-modules are interfaces. The Read Memory sub-module is responsible to extract the data from the AXI read stream into the data output. On the other hand, the Write Memory sub-module is the opposite of the Read Memory. It prepares the AXI write stream with the data from the input.

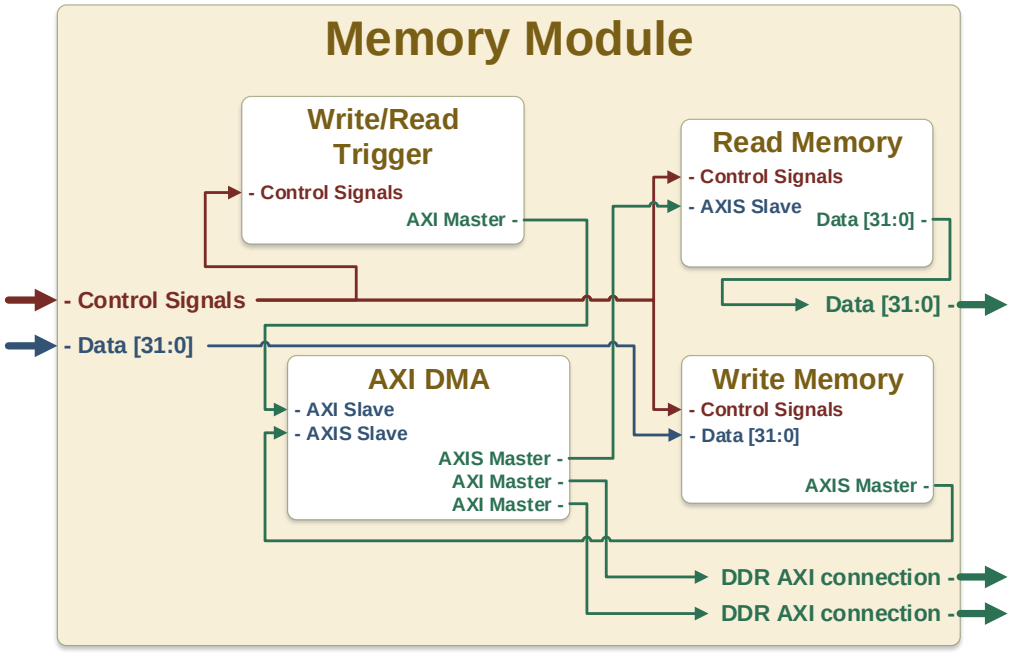


Figure 4.8: Memory module overview with the representation of its four sub-modules: Write/Read Trigger, AXI DMA, Write Memory and Read Memory. The red signals describe signals sent by the control unit, the blues are inputs and greens are outputs.

Figure 4.9 exposes the data flow between the modules and sub-modules. As the sequence diagram describes, the data flow created by these sub-modules allow *Detection* and *Checkpoint* modules to perform writes and reads on the secure memory without addressing related issues. When the *Control Unit* selects the Read mode, the *Write/Read Trigger* triggers the DMA channel to perform a read transfer. Then, the provided Xilinx's *AXI DMA* module signals the Read module

that there is new data to receive. It is through this last sub-module that the *Detection* module starts to receive the secure data.

On an Write operation, the Write/Read Trigger triggers the DMA channel to perform a write transfer with the data provided by the Checkpoint module.

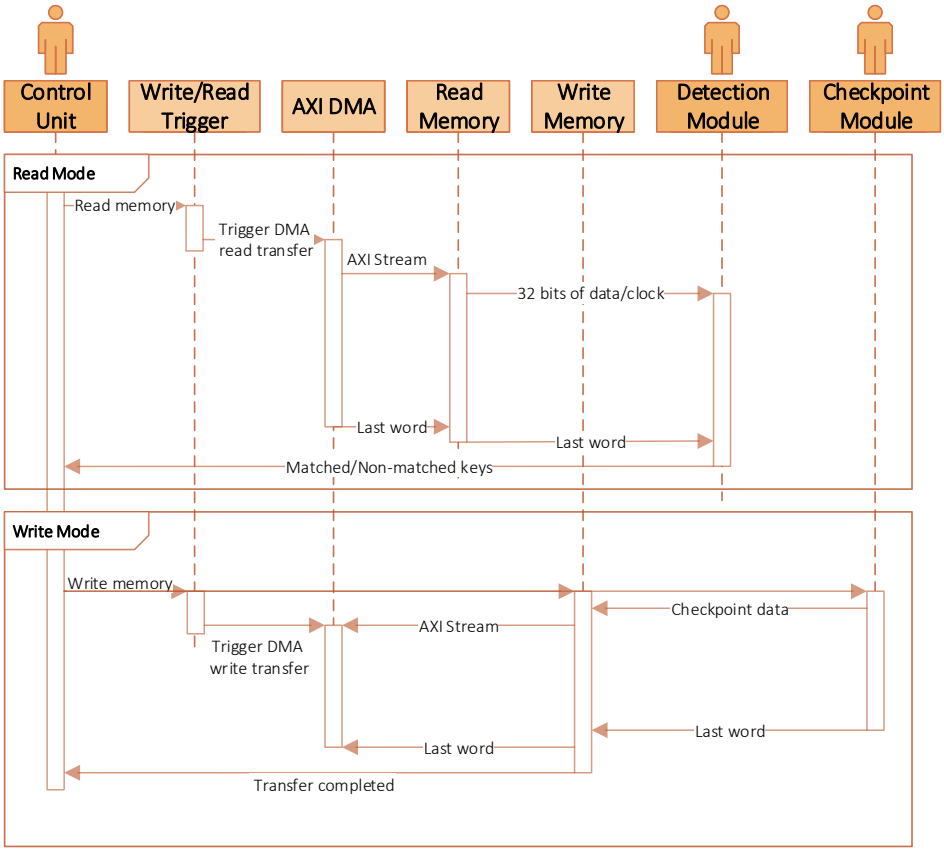


Figure 4.9: Memory module sequence diagram. There are three agents represented in this figure: the Control Unit, the Detection module and the Checkpoint module. Also, the sub-modules of the Memory module are represented: Write/Read Trigger, AXI DMA, Read Memory and Write Memory. On diagram top part, it is described the data flow when this module is on Read Mode. On the bottom, Write mode flow is exposed.

4.2.3 Checkpoint Module

The Checkpoint module is responsible to provide a sane memory image when a failure occurs. This healthy image is gathered in two different situations: before either world execution and when the detection mechanism confirms that the running state of the secure side was not changed outside its scheduling time. In order to keep both, each image is saved into a different dedicated memory. The first healthy image is pre recorded into a ROM making it not editable. Although

it is the purest state of the secure memory, recovery into this state creates an enormous throwback to the system which it is not ideal. On the other hand, the checkpoint image confirmed by the detection mechanism is stored in a RAM. The throwback on this image is almost none since it provides the last healthy state of the memory.

Figure 4.10 shows how both states savers are connected by the Memory Selector. This sub-module is responsible to select the right data to use on the recovery (if it is from ROM or RAM) and it controls the address inputted into the ROM and RAMs.

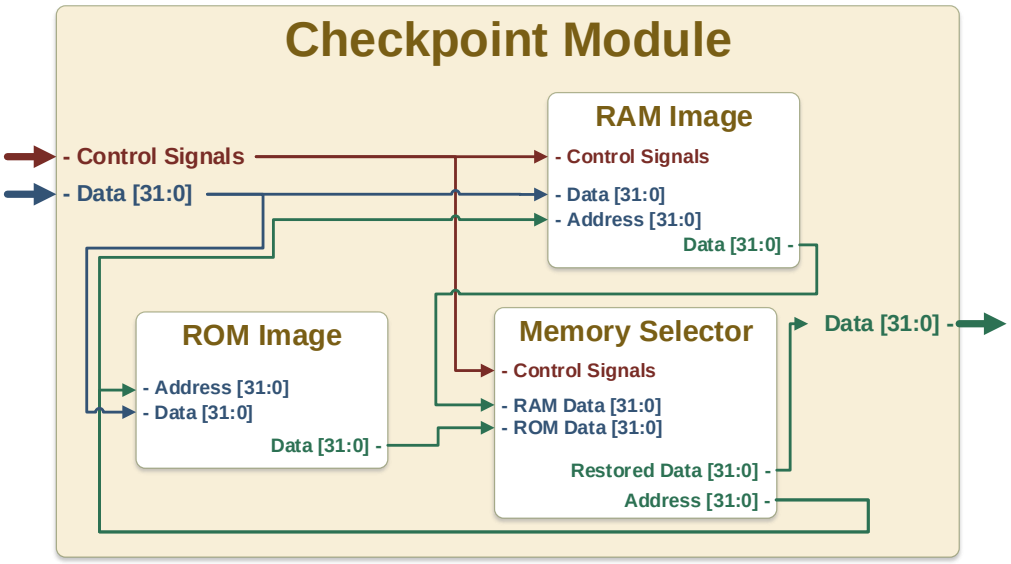


Figure 4.10: Checkpoint module overview with the representation of its three sub-modules: ROM Image, RAM Image and Memory Selector. The red signals describe signals sent by the control unit, the blues are inputs and greens are outputs.

Since the most demanding part of the system in terms of timing is getting the secure memory, this module can not add an extra repetition of this process (one to get the first key, one to get the second key and a hypothetical extra to store the checkpoint). Consequently, the secure memory is stored during the non-secure guest scheduling window which means that any image stored will be always undefined until the key comparison. To not store an unhealthy state on top of the last healthy state, this module affords two RAMs instead of one. Therefore, an healthy state is guaranteed in one RAM and the still undefined state is not discarded. To optimize the checkpoint saving process, not having to copy from one RAM to the other, a rotation method is used. As Figure 4.11 describes, neither RAM 0 or RAM 1 is set as a healthy state holder. The healthy state holder switch

between the two. After the detection module decrees the memory's veracity, the undefined data RAM becomes the next healthy state holder and the older holder receives the new undefined data in the next Health-monitor analysing cycle.

Due to board resource limitation, the implementation of the ROM and RAMs was not straitforward. The FPGA allows two types of memory that differ which board resource is implemented: BRAM and Distributed RAM. The Block of RAM or BRAM technology is a fast and small internal memory that can be used as RAM and/or ROM. Since the board only provides 240KB of BRAM and the hypervisor plus secure guest normal footprints only occupy less than 1% of the entire secure memory, saving all the 64MB is not reasonable. Instead, three 150 KB memories are implemented.

In fact, not all BRAM is available to implement the memories. After the deployment of the other modules, only 150 KB of BRAM was left. In order to implement them, some extra Logic Cells are converted to memory: the ROM is full implemented in LUTs (Logic Cells converted) since its usage is reduced; The RAMs implementations are sliced, half implemented in BRAM and half implemented in LUTs. Although this increases the read/write operations complexity due to address decoding, the half/half method has the benefit of maximize the BRAMs usage to implement memories, reducing the LUTs used as memory.

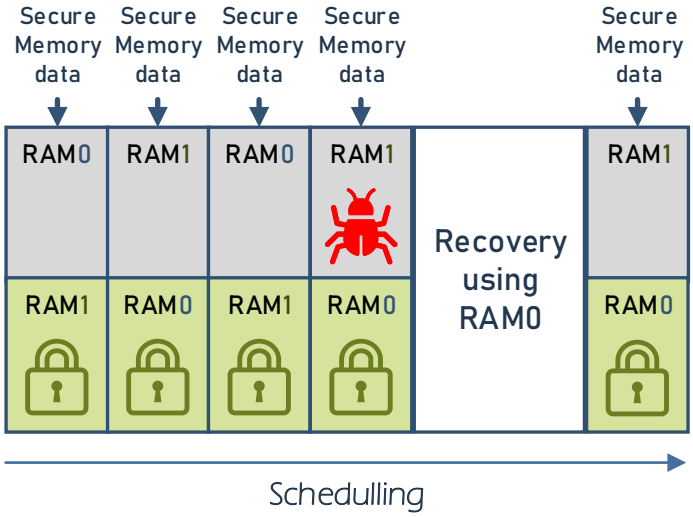


Figure 4.11: With the RAM swap method, the recovery mechanism guarantees that the actual state and the healthy state are saved without an extra memory transfer between RAMs.

4.2.4 Health-monitor Mechanism Controller

The Health-monitor Mechanism Controller is the control unit of the Health-monitor which is responsible to synchronise and control the entire hardware. This controller is implemented based on a finite state machine with the nine states:

- RESET - As the name describes, the *RESET* state resets the entire hardware: it clears the saved key, all RAM images and the hash match bit. It also stops the active AXI streams transfers (read from memory or write to memory).
- IDLE - In this state, the Health-monitor holds its functionality and waits for an action.
- FUNCTION - When the Memory module signals that there is new data to read, the *FUNCTION* state enables the Hash Function (Detection module) and it initiates the hash key production.
- SAVE HASH - In the *SAVE HASH* state, the first produced key is stored in the Detection Module to later be compared.
- COMPARE - The *COMPARE* state is responsible for enabling the key comparator, testing if the secure data is still untouched.
- ERROR - This state activates the error flag, notifying the hypervisor that the secure memory is corrupted.
- NEW READ - In the *NEW READ* state, the Memory module triggers a new DMA read transfer.
- RAM RECOVERY - The *RAM RECOVERY* state is responsible to trigger a new DMA write transfer, using the checkpoint image to restore the secure memory.
- ROM RECOVERY - The *ROM RECOVERY* state is similar to *RAM RECOVERY*, however it restores the secure memory by using the original image.

As Figure 4.12 shows, the control unit has three possible execution flows: Reset, Normal execution and Recovery execution. In the Reset execution flow, the control unit jumps between the IDLE and RESET states, resenting the entire system. After the reset, the control unit advances into the IDLE state, entering in Normal execution.

When the Memory module signals that there is new data to read, the *FUNCTION* state enables the Hash Function (Detection module) and it initiates the hash key production. While the memory is not completely read, the control unit stays on this state. After the key is finished, the following state can be the *COMPARE* or *SAVE HASH*, depending on which type of the key was produced. If

the Hash Funtion input data was the secure image right after the secure world execution, or in other words if it is the first key produced, the *SAVE HASH* is the following state; If not, the successive state is the *COMPARE*.

In the *SAVE HASH* state, the key is stored in the Detection Module to later be compared. Subsequently, the Controller progresses into *NEW READ* state. On the other hand, the *COMPARE* state is taken when two keys are formed and the comparator needs to test if the secure data is still untouched. From here, two outputs are possible: the keys match and the next state is the *NEW READ*; or the keys are not the same and the recovery mechanism needs to intervene. In order to initiate the key cycle again, the *NEW READ* state actives the Memory Module Read Trigger. Then Control Unit waits for new data in the *IDLE* state.

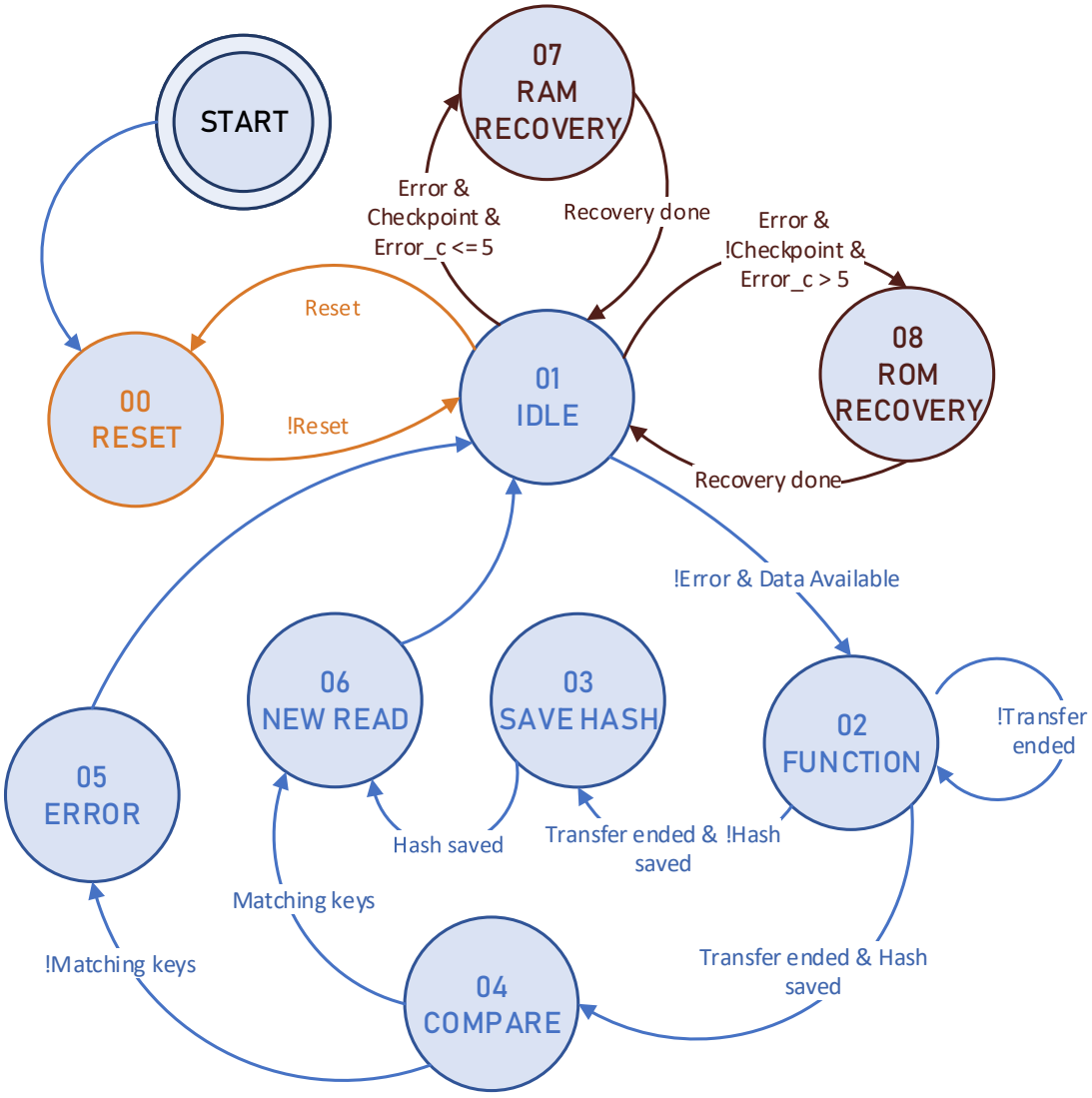


Figure 4.12: Control Unit State Machine. There are three execution flows: Reset represented in orange, Normal execution represented in blue and Recovery execution represented in brown.

Contrary to the *NEW READ*, the *ERROR* shifts the controller into the recovery execution by setting the error flag. After the hypervisor triggers the Recovery execution, two outputs are possible: *RAM RECOVERY* and *ROM RECOVERY*. The choice between them depends on whether there is a healthy image checkpoint and the *Error_c* counter. This variable counts the consecutive recovers from the same state. If the state leads into the failure more then certain number (by default five), this state is considered corrupted and the system will recover from the beginning (ROM).

Figure 4.13 overviews the control unit operation. As it is possible to observe, the Health-monitor mechanism is idle until the hypervisor schedules the non-secure side. The first key is produced right after the context-switch, and the others are produced and compared during the non-secure guest execution. If one of these comparisons flags an error, the mechanism notifies the hypervisor and enters in IDLE state until the LTZVisor's execution returns, signalling the mechanism to starts the recover. The normal exception is established after the mechanism unlock the hypervisor from the error state loop.

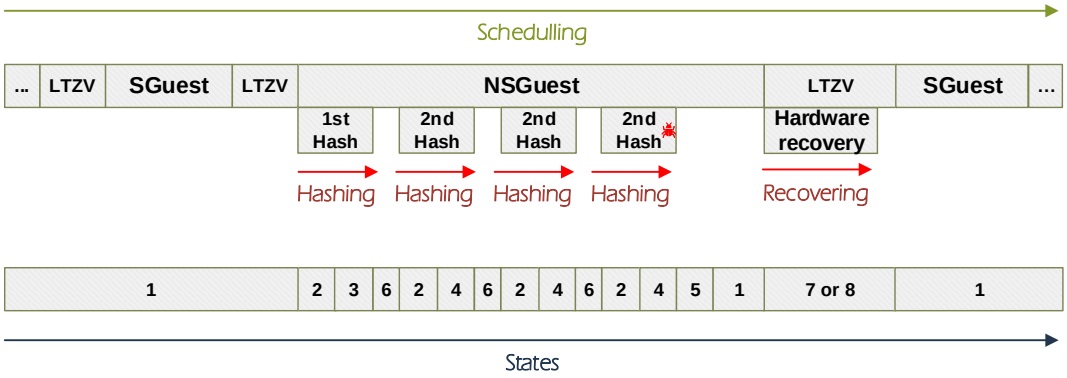


Figure 4.13: Control Unit actions and states overview.

4.3 LTZVisor Integration and Health-monitor interface

The Health-monitor's interference in LTZVisor is limited to the scheduler and configuration interface. Since LTZVisor's schedule is affected in every guest-switch, it is mandatory that the code addition is as small as possible. To achieve

this, the Health-monitor has a group of memory mapped registers and the communication LTZVisor/Health-monitor is based on reading and writing on these registers.

Listing 4.5 displays the code that is responsible to trigger the Health-monitor hash function before the non-secure guest execution. The *addr_DMA* is the Health-monitor register address that holds the number of bytes read in the AXI Read transfer which depends of the secure world compiled size. To trigger a new transfer, the register must be set to zero before the correct number of bytes. After this code, the processor mode is changed and the PC is loaded with non-secure guest instructions.

Listing 4.5: Health-monitor trigger code. Assembly code extract.

```
1 .macro HM_TRIGGER
2 push    {r0 , r1}
3 ldr r0, addr_DMA
4 ldr r1, reset_
5 str r1, [r0]    @ reset DMA tranfer
6 ldr r1, bytes_
7 str r1, [r0]    @ Start DMA tranfer
8 pop     {r0 , r1}
9 .endm
```

When the monitor is called to context-switch into the secure guest, it first runs the code presented in the listing 4.6. The hypervisor signals the Health-monitor that the context-switching is happening. Then, it reads the error register to confirm that the recover is fully completed, there are no errors or if a recovery is pendant. If the register is not cleared (set to zero when the recovery is done) the hypervisor execution is halted in the checking loop, waiting a secure restored image to continue the execution.

Listing 4.6: Health-monitor error checker. Assembly code extract.

```
1 .macro HM_SANITY_CHECKER
2 push    {r0 , r1}
3 ldr r0, addr_HMCS
4 ldr r1, NS_
5 str r1, [r0]    @ CS Signal
6 ldr r1, addr_HM
7 _checking_loop:
8 ldr r0, [r1]
9 cmp     r0, #1
```

```
10 bne      _checking_loop
11 pop      {r0 , r1}
12 .endm
```

The Health-monitor's interface is a group of APIs to configure and get hardware information. Since the registers are memory mapped, they use C pointers to read and write into them. They are divided in two categories: the configurations registers and the informative registers. Two of the first type were already mentioned in Listings 4.6 and 4.5. These are the only configuration registers used outside the proper API.

As described in the listing 4.7 the *ltzvisor_HM* function configures the AXI DMA streams and the Control Unit. Using the *reg* pointer to point to the registers address, it first configures the Read and Write channels (e.g. defining the length of the data bus and the burst size) and then the addresses. The address value is received from *___startup_start* which is a linker variable that defines the first address of the secure memory. To configure the Control Unit, the Health-monitor has a configuration register that defines the number of bytes to transfer. Like the *___startup_start*, the *__secure_size* is a linker variable that defines the size of the memory used by the secure world.

Listing 4.7: Healthmonitor Configuration APIs. C code extract.

```
1 void ltzvisor_HM (void){
2   uint32_t * reg;
3
4 /* Config DMA MM2S channel */
5   reg = (uint32_t *) (AXI_DMA_BASEADDR+c_MM2S);
6   *reg = 65539;
7   reg = (uint32_t *) (AXI_DMA_BASEADDR+sa_MM2S);
8   *reg = (uint32_t) &__startup_start;
9
10 /* Config DMA S2MM channel */
11   reg = (uint32_t *) (AXI_DMA_BASEADDR+c_S2MM);
12   *reg = 65539;
13   reg = (uint32_t *) (AXI_DMA_BASEADDR+da_S2MM);
14   *reg = (uint32_t) &__startup_start;
15
16 /* Set number of bytes to transfer */
17   reg = (uint32_t *) (CONTROLUNIT_BASEADDR+
18   NUMBER_OF_BYTES_REGISTER);
19   *reg = (uint32_t) &_secure_size;
20 }
```

In the informative part, The Health-monitor provide four registers: the Recovery Clock Cycles (RCC), the Hashing Clock Cycles (HCC), the Number of Checkpoints (NoC) and the Number of Restores (NoR). The first register stores the number of cycles spent in a recovery. This value is measured from the beginning of the recovery process until the recovery done flag is activated. Similarly to this register, the Hashing Clock Cycles holds the number of cycles spent in hashing the secure memory. Since the hash function was designed to provide a throughput of 32 bits of data per cycle, the register value only depends the size of the secure memory used. Althought the cycle unit is desired in the hardware module (hardware clock frequency may vary depending on the implementation), in software the value needs to be converted into seconds. To achieve this, the API divides both values by the hardware clock frequency.

Listing 4.8: Healthmonitor Informative APIs. C code extract.

```
1 uint32_t RCC_HM (void){
2   uint32_t * reg = (uint32_t *) 0x83C10000; //RCC addr
3   return ((*reg)/HFC); //microseconds
4 }
5
6 uint32_t HCC_HM (void){
7   uint32_t * reg = (uint32_t *) 0x83C10004; //HCC addr
8   return ((*reg)/HFC); //microseconds
9 }
```

The Number of Checkpoints (NoC) and the Number of Restores (NoR) registers, as their name indicate, hold the number of times that the mechanism create checkpoints and restores the memory. The hypervisor uses this information to keep track of its state along the time.

4.4 Intruder Module

In order to test the the hardware mechanism, an extra module was developed. Althought it is possible to mess up the secure side from the non-secure, the task is not simple and requires especial system characteristics. To bypass these procedures, an hardware module with secure privileges is implemented, mimicking the illicit non-secure access to secure memory. Since it is defined as secure, the Trustzone does not signals the access due to the fact that is not illegal. This way, the Intruder Module acts exactly like a non-secure guest which somehow masks its execution as being secure.

As Figure 4.14 shows, this module is composed by two different inputs types: the control inputs that are controlled by hardware (Control Unit and User since the board provides a binary switch to trigger the event) and the configuration inputs. These configuration inputs are mapped registers that can be set by software or by hardware using the AXI protocol. They provide flexibility on Health-monitor tests since the registers configure the address that will be modified as well as the data to write on it. This module also has a AXI master port which connects the module to the DDR.

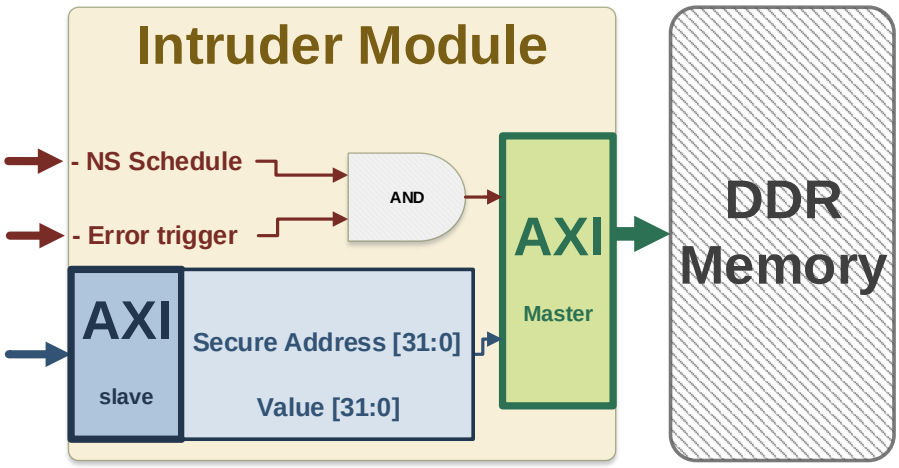


Figure 4.14: Intruder Module Overview. This module is responsible to execute a secure write operation when both control signals are actived (when the trigger is up and when the non-secure guest is running).

5. Evaluation and results

This chapter evaluates the Exception Handling and the Health-monitor implementations. Even thought the Exception Handling is a powerful tool, the hypervisor code size is affected. In the first section of this chapter, this problem is addressed by exposing how this thesis affects the LTZVisor Memory footprint and the context-switching performance.

On Chapter 2 are detailed hashing and CRCs algorithms for error detection. In the section *Hashes and CRCs Evaluation tests*, these algorithms are compared in terms of collisions and dispersion, concluding which is the best fitting method for the sanity checker. After that, the complete Health-monitor mechanism is analysed based on Hardware costs, providing information about the board resources spent to implement the mechanism. A case study finalizes this chapter. There, it is exhibited a side by side comparison between the raw LTZVisor and, the LTZVisor plus this thesis' implementation.

The results were taken on Zybo board with the standard frequency values: the processor ARM Cortex-A9 running at 600MHz and the programmable logic clock at 100MHz. The LTZVisor runs two bare-metal guests (a secure guest and a non-secure guest running a logging code) and the entire system is compiled using the ARM Xilinx toolchain.

5.1 Memory footprint

To extract information about memory footprint, the [size] tool of ARM Xilinx toolchain was used. As it can be seen in Table 5.1 , three different measures are take into consideration: i) the LTZVisor running one bare-metal guest in each security side printing a generic log message, ii) the same LTZVisor configuration but with the Health-monitor enabled and iii) the LTZVisor running the two guests with all the features enabled (Exception Handling and Health-monitor).

Table 5.1: Memory footprint (bytes).

Image	.text	.data	.bss	total
LTZVisor plus two bare-metal guests with logging application code	19812	452	66188	86452
LTZVisor plus two bare-metal guests with logging application code and Health-monitor enabled	20458	452	66192	87102
LTZVisor plus two bare-metal guests with logging application code, Health-monitor enabled and Exception Handling enabled	25466	452	66196	92114

Using the first measurement as a reference value, it is possible to observe a memory increase by adding the Health-monitor and the Exception Handling to the system, inducing an increment of nearly 6.5% on the memory footprint. As expected, the Health-monitor increment is lower than the Exception handling due to the nature of the implementation. Despite the obvious increase, these features enrich the functionality of the hypervisor creating a trade-off between memory/functionality for the LTZVisor user.

5.2 Context switching performance

The Performance Monitoring Unit (PMU) component was used in order to evaluate the guests context switch time. By putting a specific instruction at the beginning of the context switch and other at the end, this component measures the number of clocks spent during the context switch. Since the results were gathered in clock cycles, it was converted to microseconds using the processor's frequency (600MHz). Each value represents the average value of hundred collected samples.

As it is possible to observe on tables 5.2 and 5.3, the Health-monitor increases the time spent on switching guests. When the hypevisor switches from the SGuest to the NSGuest, the Health-monitor trigger code adds 0.74 µs to the overall time, representing an increment of 23%. On the other hand, the time addition of the Health-monitor restore is not so apprehensive. Since the overall time spent on switching from the NSGuest to the SGuest is higher, the increment percentage is only 7%.

Table 5.2: Performance values: Switching from SGuest to NSGuest.

Description	Number of clock cycles	Time (µs)
Without Health-monitor	1891	3.15 µs
With Health-monitor	2336	3.89 µs

Table 5.3: Performance values: Switching from NSGuest to SGuest.

Description	Number of clock cycles	Time (µs)
Without Health-monitor	4396	7.36 µs
With Health-monitor	4740	7.9 µs

5.3 Hashes and CRCs Evaluation tests

The analysed algorithms were already designed considering its dispersion and collisions. However, the results may alter depending on the data input. In order to determine the best algorithm for the data errors detection, it was designed a test with the following principals:

1. All the algorithms have an input of 200 kB of data which is 1.25% of the board secure memory by design. This value is twice the size of the LTZVisor's secure side.

2. For each algorithm iteration, the output is saved to check its collisions (algorithm output repetition).

3. The 200 kB of data is addressed as a big compacted data and not fragmented. From the algorithm point of view, it needs to iterate from the first byte until the end, using the result from the last iterations as a feedback for the new ones.

4. Since the input is real hypervisor's data, the algorithm with the less collisions and in case of close values, the best dispersion is considered the best suit.

As it is possible to observe in Table 5.4, the number of collisions are close between algorithms that share similar characteristics. Also, due to the fact of all algorithms are based on simple combinational functions, the number of cycles spent in each function iteration is the same.

Table 5.4: Algorithms evaluation result.

Algorithm	Number of collisions	Number of cycles	Keys produced
FNV-1	48	one setup + one per Byte	200 000
FNV-1A	61	one setup + one per Byte	200 000
SDBM	75	one setup + one per Byte	200 000
DJB2	78	one setup + one per Byte	200 000
CRC32	102	one setup + one per Byte	200 000
Murmur	6	one setup + one per four Byte	50 000

To understand the distribution, the key value space (all possible values that a key can have) is divided in 256 zones. It is considered that an algorithm has a good distribution if the 200 000 produced keys (50 000 to *Murmur* algorithm) are spread into the 256 zones equally. So, for each algorithm was counted the number of keys in each zone and this number was divided by the expected number of keys. Note that in this case, a perfect distribution algorithm exhibits the value 1 in all zones.

The 5.1 shows the results in a Box Plot graph. All the algorithms demonstrate values near 1 in the zones. The worst result are the Murmur's values. It presents a maximum value of 1,1878 which means that there is a zone in this algorithm where the number of keys are more 19% of the expected value. In contrast, it presents a minimum of 0.8089, representing 19% less keys of what it is supposed.

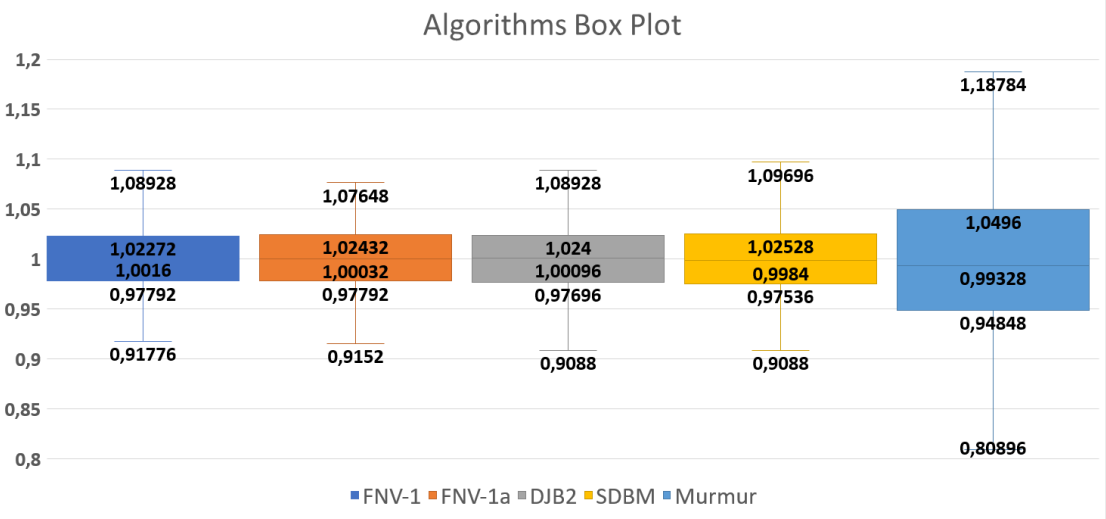


Figure 5.1: Algorithms Box-and-Whisker Plot.

The distributions for the other algorithms are similar. Despite the maximum and minimum values differ a little, the upper and lower quartiles are very much alike the same.

Since the number of collisions is more related to the number of keys produced than the number of input bytes, the collision test is not conclusive in terms of comparing the Murmur algorithm to the others. However, through the analysis of the results it is possible to conclude that *FNV-1* and *Murmur* are algorithms with good performances. The *CRC* performance is expected due to the input data characteristics and the design of the algorithm itself. As normal in memories, the input data includes large blocks of empty data and due to its design, the output key tend to be biassed when the input trend is the same. Regarding the remaining hashes they present worst results in collision number and similar values in dispersion. In fact, they are close to *FNV-1* which proves that they can be alternative solutions or in case of applying redundancy to the system, they serve as secondary algorithms.

5.4 Hardware Costs

To extract the hardware costs, the Vivado post-implementation utilisation report was analysed. The resources are divided in eight categories: Slice LUTs, Slice Registers, Slice, LUTs as Logic, LUTs as Memory, Block RAM Tile and DSPs. The Slice LUTs value describes the number of LUTs required by the implementation. To better understand the LUTs usage, the utilisation report divides LUTs used as Distributed Memory and LUT used as Logic. Like the Slice LUTs, the Slice Registers represent the same thing but at Flip-Flops level. Since each slice has four LUTs and eight FFs (Flip-Flops), the Slice value represents the number of slices used in the implementation overall. Depending of the mechanism timing needs, the number of slices can vary since it is not required the completion of each slice used. Also, the BRAM and DSP resources of the FPGA are shown.

As it is possible to identify in table 5.5, 91% of the board slices are used to deploy the hardware mechanism. There are two main reasons behind this high value: the combinational nature of the hash function and the memories implementation. The first case can be bypassed with pipeline registers. Although they create an initial delay and increase the Slice registers part, these registers allow bigger signal travelling, providing a better slice reallocation. On the other hand, the memories usage is inevitable. Despite the Block RAM being almost fully used,

the resultant memory size does not fulfil the implementation requirements. As a result, the remaining memory was implemented as LUTs, increasing this value.

Table 5.5: Resource utilization.

Resource	Available	Used	Used (%)
Slice LUTs	17600	13377	76%
Slice Registers	35200	9367	27%
Slice	4400	4008	91%
LUT as Logic	17600	9249	53%
LUT as Memory	6000	4128	69%
Block RAM Tile	60	54.5	91%
DSPs	80	12	15%

5.5 Case study

In this section a case study was made in order to understand the improvement that this project provides to the LTZVisor. First, it is analysed the exception handling improvement, allowing a side by side comparison between this project's implementation and the raw LTZVisor version. Then, in order to evaluate the Health-monitor implementation, it is simulated a secure side intrusion on the native LTZVisor and on the monitored LTZVisor.

5.5.1 Exception Handling

The following subsections describe how differently the enhanced LTZVisor reacts to exceptions. Since the data and prefetch aborts handlers are similar and processed the same way after parsing the information, only one of them is present in each processor mode, minimising the number of tests performed. It was introduced faulty code on the three agents (monitor and both guests) in order to produce every exception. On the secure guest and monitor aborts, the code was changed so that a value was written to/read from a non-aligned address. The same implementation was used for secure external aborts, but instead of using non-aligned addresses, the code writes the value into an unmapped region. In contrast, the code to trigger the non-secure external abort had a different design. Rather than writing into unmapped memory, the non-secure guest tries to read a memory value that was defined as secure. As result, the TrustZone triggers an external data abort.

5.5.1.1 Raw LTZVisor exceptions

The native LTZVisor already provides the minimum support for exceptions in every processors mode. However, the exception handlers are generic infinite loops with the purpose of stopping the system execution. For the user point of view, the guests are stopped when an exception occurs due to the absence of their log messages. Nonetheless, without any information, understanding the fault origin is a difficult task.

5.5.1.2 Enhanced Hypervisor Data Abort

With the exception handling enhancement, the information interpreted by the handlers is sufficient to understand why the system is failing.

As describe in Figure 5.2, the exception was triggered due to a deficient read access by the hypervisor (the address read is unaligned), resulting in a secure data abort. After printing the information about the type and the processor mode responsible for the abort, the system execution is stopped.

```
Exception
-> LTZVisor: MMU, Debug or Alignment Data Abort Exception
  * DFSR (Data Fault Status Register): 0x1
 * Type:
          * Alignment fault
          * Fault occurred on a READ access
          * Faulting address 0x83c00009

 * SPSR (Saved Program Status Registers): 0x1d6
         * Negative condition code flag: 0
         * Zero condition code flag: 0
         * Carry condition code flag: 0
         * Overflow condition code flag: 0
         * Cumulative saturation flag: 0
         * Endianness execution state bit (LE:0/BE:1): 0
         * Flawed processor mode: Monitor
         * Trustzone state (S->0/NS->1): 0
```

Figure 5.2: LTZVisor Hypervisor Data Abort output.

5.5.1.3 Enhanced Secure Guest Data Abort

Figure 5.3 shows the LTZVisor output when a Secure Guest Data abort occurs. The normal execution is present until the iteration number 10 of the secure guest. Instead of printing the "Hello" message, the secure guest tries to write into the

address 0x83c10001. Being an unaligned address, the Secure supervisor Data abort is triggered, exposing information about the type, the address and the faulting processor. Since the secure guest exceptions have not been defined, the hypervisor stops the execution of both worlds.

```
Secure World Hello :8
Non-Secure World Hello :6

Secure World Hello :9
Non-Secure World Hello :7


------------------------ Exception ------------------------
-> LTZVisor: MMU, Debug or Alignment Data Abort Exception
  * DFSR (Data Fault Status Register): 0x801
 * Type:
          * Alignment fault
          * Fault occurred on a WRITE access
          * Faulting address 0x83c10001

 * SPSR (Saved Program Status Registers): 0x600001d3
          * Negative condition code flag: 0
          * Zero condition code flag: 1
          * Carry condition code flag: 1
          * Overflow condition code flag: 0
          * Cumulative saturation flag: 0
          * Endianness execution state bit (LE:0/BE:1): 0
          * Flawed processor mode: Guest (Supervisor)
          * Trustzone state (S->0/NS->1): 0
-------------------------------------------------------------
```

Figure 5.3: LTZVisor Secure Guest Data Abort output.

5.5.1.4 Enhanced Secure Guest and Hypervisor External Abort

The secure guest and the hypervisor share the same external abort handler. The reason behind it is the dependency that the hypervisor and secure guest had. An external abort represents a serious malfunction of the entity behind the abort which means that independently of each processor mode the entire security side is compromised. Nonetheless, knowing about the abort is important. As Figure 5.4 describes, both guests are executing normally until the secure guest tries to write in the unmapped memory address 0x93c10000. The external abort is triggered, the information exposed and the entire system is suspended.

```
Secure World Hello :8
Non-Secure World Hello :6

Secure World Hello :9

------------------------ Exception ------------------------
-> LTZVisor: External Data Abort Exception
  * DFSR (Data Fault Status Register): 0x808
 * Type:
          * Synchronous External Abort
          * Fault occurred on a WRITE access
          * Faulting address 0x93c10000

 * SPSR (Saved Program Status Registers): 0x600001d3
          * Negative condition code flag: 0
          * Zero condition code flag: 1
          * Carry condition code flag: 1
          * Overflow condition code flag: 0
          * Cumulative saturation flag: 0
          * Endianness execution state bit (LE:0/BE:1): 0
          * Flawed processor mode: Guest (Supervisor)
          * Trustzone state (S->0/NS->1): 0
-------------------------------------------------------------
```

Figure 5.4: LTZVisor Secure Guest External Data Abort output.

5.5.1.5 Enhanced Non-Secure External Data Abort

As it can be seen in Figure 5.5, the non-secure guest runs normally until its 10th iteration. As soon as it tries to read the secure memory, the non-secure external Data abort is triggered. Similar to the other exceptions, the information is exposed. However, the type of the abort is Unknown, making impossible to understand which address the non-secure guest was trying to read. Besides the lack of the type information, the faulted processor mode is fully exposed. Since the faulting world was the non-secure, the hypervisor stops its execution without compromising the secure guest (the secure guest continues to function after the exception occurs).

```
Secure World Hello :12
Non-Secure World Hello :9

Secure World Hello :13

------------------------ Exception ------------------------
-> LTZVisor: External Data Abort Exception
  * DFSR (Data Fault Status Register): 0x0
 * Type:
          * Unknown
          * Fault occurred on a READ access
 * SPSR (Saved Program Status Registers): 0x60000193
          * Negative condition code flag: 0
          * Zero condition code flag: 1
          * Carry condition code flag: 1
          * Overflow condition code flag: 0
          * Cumulative saturation flag: 0
          * Endianness execution state bit (LE:0/BE:1): 0
          * Flawed processor mode: Guest (Supervisor)
          * Trustzone state (S->0/NS->1): 1
-----------------------------------------------------------
Secure World Hello :14
Secure World Hello :15
```

Figure 5.5: LTZVisor Non-Secure Guest External Data Abort output. The Fault Status value of TrustZone aborts is zero, the same as Unknown aborts.

5.5.1.6 Conclusion

Although the exception handling increases the size of the LTZVisor significantly, the improvement is obvious compared to the raw LTZVisor version. The information provided about the exception and the faulting state is much higher which creates a better foundation for the developer that will use the LTZVisor. Additionally, the separation between hypervisor and secure guest faults in the secure supervisor exception not only grants isolation between secure exceptions but also it provides the handlers coexistence. To finalize, the "Stop the non-secure guest execution" feature allows secure side survivability since it removes the full execution stop after a non-secure side error.

5.5.2 Health-monitor

Like in the exceptions section, a comparison between LTZVisor with and without Health-monitor is analysed in the following subsections. To perform tests,

the guests' code was simple printing "Hello" plus the respective iteration number. This last value allows two features: enables a visible secure value as the target of the Intruder Module and visible recovery indicator.

5.5.2.1 LTZVisor without Health-monitor

Without the Health-monitor, the LTZVisor is completely unaware of the secure side break. As Figure 5.6 exposes, the two guests work normally until the secure guest iteration number 34. This iteration was the last sane state of the secure side since its value was changed by the Introduder Module, simulating a non-secure side attack. For the hypervisor point of view, nothing unusual happened and both guests continue their execution normally. The shown test was controlled in order to alter the secure code without creating a exception fault.

```
Secure World Hello :33
Non-Secure World Hello :30

Secure World Hello :34
Non-Secure World Hello :31

Secure World Hello :450
Non-Secure World Hello :32

Secure World Hello :451
Non-Secure World Hello :33
```

Figure 5.6: LTZVisor without Health-monitor output.

5.5.2.2 LTZVisor with Health-monitor

The same procedure was made on the LTZVisor with the Health-monitor but the Introduder Module changes the secure guest value in the 137th iteration. Instead of continuing the system execution, the Health-monitor detects that the secure image was corrupted (by comparing the hash keys) and actives the restore mechanism. Since the last healthy state was the iteration number 137, the secure guest re-does the same print and continues its normal execution. The red rectangle in Figure 5.7 highlights the throwback produced by the Health-monitor.

Figure 5.7: LTZVisor with Health-monitor output.

In this test, the secure guest code was slightly modified, as it is possible to observe in Figure 5.8. Attached to the normal "Hello" function, a subroutine reads the Health-monitor recovery values using the APIs and prints its values. In order to not spam the output with large data, the subroutine only runs every 25 secure iterations.

Figure 5.8: Health-monitor information output.

5.5.2.3 Conclusion

As it is possible to verify in the figures above, the improvement is notorious. The Health-monitor does not only detects the memory intrusion but also restores the secure image to a healthy state. Without this tool, the secure guest is fully exposed after the TrustZone security fails, resulting two possible scenarios: A successful secure intrusion and a failed intrusion. The first scenario is described in the test above where the secure guest was unaware of the intrusion and the execution runs normally. This is the worst scenario since the "secure" guest is running but controlled by the non-secure guest. If the intrusion alters the code in such a way that triggers an abort, the error is detected and the execution is stopped. It is considered a better scenario due to the fact that the secure guest is not running a faulty code.

Figure 5.8 expresses the Health-monitor values. As it is possible to see, the system takes 21384 PL clocks (213µs) to hash the secure side (86KB). Since the minimum number of keys for the correct functioning of the Health-monitor are two, the non-secure execution time has to be higher than 426µs (2 x 213µs). In this case, the scheduling time is not a problem due to the fact that the non-secure guest runs almost 5 milliseconds each time. However for bigger secure images and/or shorter scheduling times, this value can be unworkable. The minimum non-secure execution time that this Health-monitor supports is calculated by the following equation:

$$MinNSexec(\mu s) = \frac{SecureImage(Bytes) + 2}{2 \times PLfrequency(MHz)}$$

6. Conclusion

The presented thesis sought to deepen the problem of TrustZone hypervisors failures in embedded systems. In this study, the failures were divided in two main groups: the processor's detected faults and undetected faults. The former group are within the processor exceptions, which are triggered whenever a system malfunction is detected. The two exceptions contemplated by this project are Data aborts and Prefetch aborts. The former exceptions are data-related flaws such as read/write operations in unmapped or unaligned memory. On the other hand, the prefetch aborts handle the instruction flow failures, for instance a processor jump into an unaligned code segment. Since this thesis uses the ARMv7 architecture with security extensions, these exceptions are present on each world (secure world and non-secure world), as well as on monitor mode.

In this thesis, informative handlers were implemented for the secure side, to exposing the faulted world, the faulted processor mode, the type of the fault and even the address that causes the fault. In addition, the non-secure world is controlled by the handler, stopping its execution whenever the non-secure world is itself the source of the problem.

For processors' undetected faults, this thesis implements an Health-monitor that controls the sanity state of the secure side. This hardware mechanism is divided in two parts: the detection and the recovery mechanisms. In order to detect faults, the detection module generates hash keys based on the secure memory values. By producing and comparing the keys during the non-secure guest execution, this module guarantees that the secure memory cannot be altered by others rather than itself. Depending on the detection module verdict, the recovery module acts differently. After a secure memory validation, this module stores the secure memory into a dedicated memory, providing a future restoring point to the system (checkpoint). In contrast, if an hash key disagreement is detected, the module triggers the recovery using the last valid checkpoint saved. To prevent faulting loops, this module also provides other dedicated memory (ROM) with the unmodified secure image. In last case scenario, the secure memory is restored

with this image.

Although the main goal of this thesis is to protect the secure memory as a whole, it is possible to achieve more granularity with small system modifications. By running the detection mechanism during every guests execution (Secure guest and non-secure guest) and by having different keys for each section of the memory, the mechanism can detect not only non-secure intrusion into the secure memory, but also secure guest intrusions into the hypervisor memory. In other words, if the non-secure world is running, the keys generated by the hardware mechanism are based on the whole secure world, since the goal is to detect possible secure memory changes. However, if the secure guest is the one that is running, the detection mechanism generates keys exclusively from the hypervisor memory, preventing hypervisor manipulation from the secure guest or non-secure guest.

In conclusion, this thesis reinforces the TrustZone hardware virtualization with an extra harware mechanism that detects and recovers from failures, creating an extra security dimension: manipulation window. This concept allows defining time windows so the memory can be manipulated. In this case, the windows are the software component scheduling time and by defining them, the system creates the demanded isolation between hypervisor and secure guest that the TrustZone cannot provide.

6.1 Future work

This thesis stands as an open project where work can be developed in the future, providing a groundwork for future research on health-monitoring in TrustZone-assisted hypervisors. The following points are possible system improvements in terms of power consumption, algorithm efficiency and extra functionalities:

- **Sleep the processor** - Rather than stopping the non-secure guest execution by forcing it to run an endless loop, the handlers can use the processors registers to make the processor sleep during the supposed guest execution time. This provides a better solution in terms of power saving without altering the LTZVisor structure. The processor recovers from its sleeping mode when the FIQ that triggers the secure execution time is taken, allowing the normal execution of the secure guest and LTZVisor.

- **Split the secure memory into hypervisor and secure guest sections** - As already mentioned, splitting the secure memory section in two (hypervisor and secure guest) allows a distinct approach in terms secure world security.

The implemented Health-monitor protects the secure side memory as a block from non-secure side masked attacks, but with this improvement each secure constituent owns its own section and consequently its own key, allowing separated detection windows. This results in an extra isolation layer between hypervisor and segure guest codes.

- **Increasing the throughput** - Although the detection mechanism was designed considering the input data size of the hash functions, limiting the throughput to 32 bits per clock is low especially when the DMA technology allows to transfer of 1024 bits per clock cycle. In order to increase and guarantee the 1024 bits of throughput, more parallel hashing is needed. Clearly, this will increase considerable the resources used.

- **Hardware timeout** - Since the Health-monitor's control unit actions are heavily dependent of the LTZVisor scheduler, creating a mechanism that detects the scheduling defects is a must. There are two phases that are critical to the Health-monitor: the ending and restarting of the secure guest execution. The first one triggers the key generation but due to the fact that this is happening during a secure to non-secure world switching, this critical point is considered safe. On the other hand, the non-secure to secure world switch can be problematic. If the non-secure world alters the secure world code in such a way that it prevents the secure guest execution, the hypervisor's trigger to recover/save the checkpoint may never occur. The proposed solution for this problem consists in implementing an hardware watchdog to control the non-secure guest execution time, forcing the recovery mechanism when it detects malfunction.

- **Include the processor state in the checkpoints** - By adding the processor's registers value in the Health-monitor secure checkpoints, the mechanism becomes more secure and flexible. More secure because it expands the sanity checker to processor registers. It allows a more flexible recovery since the recover procedure is not limited to errors detected in the Health-monitor. Through the Health-monitor APIs, processor aborts would also have access to secure world restoration.

References

[ABK09] A. Acharya, J. Buford, and V. Krishnaswamy. Phone virtualization using a microkernel hypervisor. In *2009 IEEE International Conference on Internet Multimedia Services Architecture and Applications (IMSAA)*, pages 1–6. IEEE, December 2009. doi:10.1109/IMSAA.2009.5439460.

[AH10] A. Aguiar and F. Hessel. Embedded systems' virtualization: The next challenge? In *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, pages 1–7, June 2010. doi:10.1109/RSP.2010.5656430.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, volume 1, pages 11–33, Los Alamitos, CA, USA, January 2004. IEEE Computer Society Press. doi:10.1109/TDSC.2004.2.

[App08] A. Appleby. MurmurHash, final version, 2008. URL: https://tanjent.livejournal.com/756623.html.

[ASTM08] F. Afonso, C. Silva, A. Tavares, and S. Montenegro. Application-level fault tolerance in real-time embedded systems. In *2008 International Symposium on Industrial Embedded Systems*, pages 126–133, June 2008. doi:10.1109/SIES.2008.4577690.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, Harris T, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SIGOPS Oper. Syst. Rev.*, volume 37, pages 164–177, New York, NY, USA, October 2003. ACM. doi:10.1145/1165389.945462.

[Cri82] F. Cristian. Exception Handling and Software Fault Tolerance. In *IEEE Transactions on Computers*, volume C-31, pages 531–540, June 1982. doi:10.1109/TC.1982.1676035.

[DIL] Digilent - Electrical Engineering Store, FPGA, Microcontrollers and Instrumentation. URL: https://store.digilentinc.com/.

[FNV+11] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen. The FNV non-cryptographic hash algorithm. In *Ietf-draft*, 2011.

[Hei08] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM. doi:10.1145/1435458.1435461.

[HHY+12] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H.-W. Jin. Full virtualizing micro hypervisor for spacecraft flight computer. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6C5–1–6C5–9, October 2012. doi:10.1109/DASC.2012.6382393.

[JVM] Java Virtual Machine. URL: https://www.java.com/en/.

[Kai09] R. Kaiser. Complex embedded systems - A case for virtualization. In *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 135–140, June 2009.

[Kis14] Jan Kiszka. Real Safe Times in the Jailhouse Hypervisor, 2014. URL: http://events.linuxfoundation.org/sites/events/files/slides/ELCE-2014-Jailhouse.pdf.

[Knu97] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Lan] Landon Curt Noll. FNV Hash. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[LCP+17] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In *ECRTS*, 2017. doi:10.4230/LIPIcs.ECRTS.2017.6.

[Lim03] ARM Limited. AMBA AXI ™ and ACE ™ Protocol Specification AXI3 ™ , AXI4 ™ , and AXI4-Lite ™ ACE and ACE-Lite ™. Technical report, 2003. URL: https://silver.arm.com/download/download.tm?pv=1198016.

[Lim09] ARM Limited. ARM Security Technology. Building a Secure System using TrustZone Technology ARM. *ARM white paper*, page

108, 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[Lim12] ARM Limited. ARMv7-A and ARMv7-R manual. Technical report, 2012.

[Lin12] Proceedings of the Linux Symposium, 2012. URL: http://landley.net/kdocs/mirror/ols2012.pdf#page=93.

[LMTP18] J. Lopes, J. Martins, A. Tavares, and S. Pinto. DIHyper: Providing Lifetime Hypervisor Data Integrity. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 645–650, June 2018. doi:10.1109/ISIE.2018.8433832.

[LTZ] LTZVisor. URL: https://github.com/tzvisor.

[OGP18] D. Oliveira, T. Gomes, and S. Pinto. Towards a Green and Secure Architecture for Reconfigurable IoT End-Devices. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 335–336, April 2018. doi:10.1109/ICCPS.2018.00041.

[OMC+18] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto. TZ-VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 708–713. IEEE, 2018. doi:10.1109/ISIE.2018.8433781.

[ora] Oracle VM VirtualBox. URL: https://www.virtualbox.org/.

[PB61] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. In *Proceedings of the IRE*, volume 49, pages 228–235, January 1961. doi:10.1109/JRPROC.1961.287814.

[PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Communications of the ACM*, volume 17, pages 412–421, 1974. doi:10.1145/361011.361073.

[PGP+17] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares. IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. In *IEEE Internet Computing*, volume 21, pages 40–47, January 2017. doi:10.1109/MIC.2017.17.

[POP+17] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares. Lightweight multicore virtualization architecture exploiting ARM TrustZone. In *IECON 2017 - 43rd Annual Conference of the*

IEEE Industrial Electronics Society, pages 3562–3567, October 2017. doi:10.1109/IECON.2017.8216603.

[PPG+17a] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. In *IEEE Computer Architecture Letters*, volume 16, pages 158–161, July 2017. doi:10.1109/LCA.2016.2617308.

[PPG+17b] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.4.

[PS18] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. In *ACM Computing Surveys*, volume preprint, 2018.

[PTM16] S. Pinto, A. Tavares, and S. Montenegro. Space and Time Partitioning with Hardware Support for Space Applications. In *DASIA 2016 - Data Systems In Aerospace*, volume 736 of *ESA Special Publication*, page 19, August 2016.

[Ran75] B. Randell. System Structure for Software Fault Tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM. doi:10.1145/800027.808467.

[RKLM17] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). In *CoRR*, volume abs/1705.06932, 2017. arXiv:1705.06932.

[RLT78] B. Randell, P.A. Lee, and P. Treleaven. Reliability issues in computing system design. In *ACM Computing Surveys (CSUR)*, volume 10, pages 123–165, June 1978.

[RP12] R. Ragel and S. Parameswaran. Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors. *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 875–880, 2012. doi:10.1109/DATE.2012.6176621.

[saf] SafeG Hypervisor. URL: https://www.toppers.jp/en/safeg.html.

[SDB] Apache Portable Runtime Utility Library: SDBM library. URL: https://apr.apache.org/docs/apr-util/0.9/group__APR_

_Util__DBM__SDBM.html.

[SHT13] D. Sangorrín, S. Honda, and H. Takada. Dual Operating System Architecture for Real-Time Embedded Systems. In *Journal of Chemical Information and Modeling*, volume 53, pages 6–15, 2013. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004.

[Sie91] D. P. Siewiorek. Architecture of fault-tolerant computers: an historical perspective. In *Proceedings of the IEEE*, volume 79, pages 1710–1734, December 1991. doi:10.1109/5.119549.

[Sin15] V. Sinitsyn. Jailhouse. In *Linux J.*, volume 2015, Houston, TX, April 2015. Belltown Media.

[SK10] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *EuroSys'10*, page 209, 2010. doi:10.1145/1755913.1755935.

[SML10] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In *2010 Second International Conference on Computer and Network Technology*, pages 222–226, April 2010. doi:10.1109/ICCNT.2010.49.

[SN05] James E. Smith and Ravi Nair. Chapter one - introduction to virtual machines. In James E. Smith and Ravi Nair, editors, *Virtual Machines*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 1 – 26. Morgan Kaufmann, Burlington, 2005. doi:https://doi.org/10.1016/B978-155860910-5/50002-1.

[Tec13] Siemens Corporate Technology. Jailhouse: Static System Partitioning and KVM, 2013. URL: https://www.linux-kvm.org/images/b/b1/Kvm-forum-2013-Static-Partitioning.pdf.

[Tru] ARM TrustZone. URL: https://www.arm.com/products/security-on-arm/trustzone.

[TSS17] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium*, pages 1057–1074, 2017.

[Viv] Vivado Design Suite. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[VOS] VOSYSMonitor Hypervisor. URL: http://www.virtualopensystems.com/en/products/vosysmonitor/.

[WFM+07] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing

Embedded Security on Dual-Virtual-CPU Systems. *IEEE Design & Test of Computers*, 24(6):582–591, November 2007. doi:10.1109/MDT.2007.196.

[WJ10] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010. doi:10.1109/SP.2010.30.

[XI11] Xilinx and Inc. AXI Reference Guide UG761 (v13.1). Technical report, 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[XI14] Xilinx and Inc. TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC. Technical report, 2014.

[XI15] Xilinx and Inc. Zynq-7000 All Programmable SoC Software Developers Guide. Technical report, 2015. URL: https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf.

[XI18a] Xilinx and Inc. AXI DMA v7.1 LogiCORE IP Product Guide (PG021). Technical report, 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[XI18b] Xilinx and Inc. Zynq-7000 SoC Technical Reference Manual (UG585). Technical report, 2018. URL: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[XIL] Xilinx. URL: https://www.xilinx.com/.

[XR96] J. Xu and B. Randell. Roll-forward error recovery in embedded real-time systems. In *Proceedings of 1996 International Conference on Parallel and Distributed Systems*, pages 414–421, June 1996. doi:10.1109/ICPADS.1996.517589.

[XRR+95] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 499–508, June 1995. doi:10.1109/FTCS.1995.466948.

[YLH+08] S. Yoo, Y. Liu, C.-H. Hong, C. Yoo, and Y. Zhang. MobiVMM:

a virtual machine monitor for mobile phones. In *Proceedings of the 1st Workshop on Virtualization in Mobile Computing, MobiVirt '08*, pages 1–5, December 2008. doi:10.1145/1622103.1622109.

[ZMH15] S. Zampiva, C. Moratelli, and F. Hessel. A hypervisor approach with real-time support to the MIPS M5150 processor. In *Sixteenth International Symposium on Quality Electronic Design*, pages 495–501. IEEE, Mars 2015. doi:10.1109/ISQED.2015.7085475.