PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO

LUCIANO LORES CAIMI

SECURE ADMISSION AND EXECUTION OF APPLICATIONS IN NOC-BASED MANY-CORES SYSTEMS

Porto Alegre
2019

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SECURE ADMISSION AND EXECUTION OF APPLICATIONS IN NOC-BASED MANY-CORES SYSTEMS

LUCIANO LORES CAIMI

Thesis presented as partial requirement for obtaining the degree of PhD in Computer Science at Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2019

Ficha Catalográfica

C134s Caimi, Luciano Lores

Secure Admission and Execution of Applications in NoC-based Many-cores Systems / Luciano Lores Caimi . – 2019.
121.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. NoC-based Many-Core System. 2. Security. 3. Mutual Authentication. 4. Application Admission. 5. Secure Zones. I. Moraes, Fernando Gehm. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Luciano Lores Caimi

Secure Admission and Execution of Applications in Noc-Based Many-Cores Systems

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on September 13th, 2019.

COMMITTEE MEMBERS:

Prof. Dr. Altamiro Amadeu Susin (UFRGS)

Prof. Dr. Cesar Albenes Zeferino (UNIVALI)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

Prof. Dr. César Augusto Missio Marcon (PPGCC/ PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/ PUCRS - Advisor)

ACKNOWLEDGMENTS

Dedico este trabalho a duas mulheres especiais, minha mãe, Maria de Lourdes Caimi e minha filha, Amanda Liczbinski Caimi. A elas uma palavra... Amor.

Uma jornada como esta não se percorre sozinho. Foram muitas pessoas que caminharam comigo e que tornaram mais fácil chegar ao destino. Quero deixar aqui registrado minha gratidão e meu obrigado.

Agradeço ao meu orientador, professor Fernando Ghem Moraes, pela confiança que depositou em mim, por todo o processo de orientação, pelo tempo e paciência dedicados.

Agradeço a minha família, especialmente minhas irmãs (Claudia, Flávia e Ana), sobrinhas e sobrinhos pelo carinho, suporte, incentivo e convívio, essenciais para manter a saúde e o equilíbrio emocional ♡.

Agradeço a Carin pelo incentivo no início da jornada. Meu muito obrigado e gratidão a Reni pelo carinho e apoio. Agradeço ao meu amigo Vinícius M. Fochi, parceiro de jornada e de trabalho. Agradeço aos colegas do GAPH que contribuíram diretamente com meu trabalho: Marcelo Ruaro, Eduardo Wachter e André Del Mestre. Agradeço aos antigos colegas e hoje amigos que fiz durante o doutorado, obrigado Jean, Felipe, Jurinha, Korol, Tanauan, Gustavo, Ramon, Cataldo, Anderson e também os amigos do futebol semanal.

Agradeço aos professores do PPGCC pela dedicação e competência: Marcon, Amory, Ney, Hessel, Avelino, De Rose, Gustavo. Obrigado ao Régis e ao Diego da secretaria do PPGCC pelo pronto auxílio que sempre recebi.

Agradeço a Universidade Federal da Fronteira Sul e aos programas de fomento à qualificação docente pelo suporte e auxílio recebidos.

ADMISSÃO E EXECUÇÃO SEGURA DE APLICAÇÕES EM SISTEMAS DE MULTIPLOS NÚCLEOS BASEADOS EM REDES INTRA-CHIP

RESUMO

A adoção de sistemas *múltiplos núcleos* torna a preocupação com a proteção de dados um requisito crítico de projeto de taos sistemas devido ao compartilhamento de recursos e a execução simultânea de várias aplicações na plataforma. Uma aplicação que processa dados confidenciais pode ter sua segurança prejudicada por um processo malicioso. A literatura contém várias propostas para proteção de sistemas *many-core*, concentrando-se principalmente na proteção da execução da aplicação ou no acesso a memórias compartilhadas. No entanto, uma solução que englobe todo ciclo da aplicação, incluindo a admissão da aplicação, sua execução e o acesso a periféricos, é uma lacuna a ser preenchida. Esta Tese discute três questões relacionadas à segurança em sistemas *many-core* basados em redes intra-chip: a admissão segura de aplicações, a prevenção do compartilhamento de recursos durante sua execução, e o acesso seguro a dispositivos externos. Esta Tese propõe um conjunto de protocolos e mecanismos, aplicados em tempo de execução, para abordar estas questões. Antes da requisição de admissão de aplicações as entidades responsáveis pela admissão são autenticadas. Uma entidade autenticada pode implantar aplicações, necessitando apenas da verificação de um Código de Autenticação de Mensagem (MAC - *Message Authentication Code*) para garantir a integridade da aplicação. As aplicações são mapeadas em zonas seguras contínuas opacas (*OSZ* - *Opaque Secure Zones*), com a reserva de todos os elementos de processamento e recursos de comunicação. Todos os fluxos de tráfego que devem atravessar as *OSZ* são redirecionados para o exterior das *OSZ*. Essa abordagem de isolamento evita ataques de negação de serviço (DoS - *Denial-of-Service*), ataques de temporização e falsificação, e garante a confidencialidade e integridade dos dados. Os dispositivos externos também são autenticados, permitindo o uso de uma chave compartilhada dedicada para criptografar as trocas de mensagens com periféricos. No que diz respeito à admissão de aplicações, o *overhead* dominante da proposta corresponde às etapas de computação e verificação do MAC, o que resulta na latência para iniciar uma aplicação segura por alguns milissegundos. Relativamente à execução da aplicação, a avaliação mostra um impacto insignificante no tempo de execução de aplicações seguras e não seguras, mesmo na presença de vários caminhos reroteados e respectivo reencaminhamento e retransmissão de mensagens. A proteção do cabeçalho da mensagem e da carga útil durante o acesso a periféricos corresponde ao

custo principal na latência de comunicação. O que se apresenta é um conflito entre o custo da criptografia (hardware ou software) versus a latência adicional na comunicação com o periférico. Esta Tese avança o estado da arte na área de pesquisa de sistemas *many-core* baseados em redes intra-chip, uma vez que engloba mecanismos de segurança para todo ciclo da aplicação. O mecanismo leve de autenticação mútua entre entidades externas e o *many-core* e a utilização de um MAC para proteger o código-fonte da aplicação são inovações propostas na Tese. A proteção da execução das aplicações sem mecanismos criptográficos através de *OSZ*, evitando o compartilhamento de recursos computacionais e de comunicação, representa outra contribuição desta Tese.

Palavras-Chave: Sistema many-core baseado em rede intra-chip, Segurança, Autenticação Mútua, Admissão de Aplicações, Zonas Seguras.

SECURE ADMISSION AND EXECUTION OF APPLICATIONS IN NOC-BASED MANY-CORES SYSTEMS

ABSTRACT

The adoption of many-cores systems introduces the concern for data protection as a critical design requirement due to the resource sharing and the simultaneous executions of several applications on the platform. A secure application that processes sensitive data may have its security harmed by a malicious process. The literature contains several proposals to protect many-cores against attacks, focusing for example in the protection of the application execution or the access to shared memories. However, a solution covering the application lifetime, including its admission, execution and peripheral access, is a gap to be fulfilled. This Thesis discusses three security-related issues: the secure admission of applications, the prevention of resource sharing during their execution, and the safe access to external devices. This Thesis proposes a set of protocols and mechanisms, executed at runtime, to tackle these issues. The application admission authenticates trusty entities. An entity authenticated might deploy applications, requiring only a Message Authentication Code (MAC) verification to guarantee the application integrity. Secure applications are mapped into Opaque Secure Zones (*OSZ*), with the reservation of *all* Processing Elements (PEs) and communication resources. All traffic flows that should cross the *OSZ* are rerouted to the outside of the *OSZ*. Such isolation approach avoids Denial-of-Service (DoS), timing, and spoofing attacks and guarantees data confidentiality and integrity. External devices are also authenticated, enabling the use of a dedicated shared key to encrypt the peripheral exchange messages. Concerning the application admission, the dominant overhead corresponds to the MAC computation and verification steps, that results in the latency to start a secure application by a few milliseconds. Concerning the application execution, the evaluation shows a insignificant impact on the execution time of secure and non-secure applications, even in the presence of several broken paths and the respective rerouting and retransmission of messages. Protection of message header and payload during the peripheral access correspond to the main overhead in the communication latency. The concern is the trade-off between the cryptography cost (hardware or software) versus the additional latency in the communication. This Thesis advances the state-of-the-art on the NoC-based many-core systems research area since that encompasses security mechanism to entire application lifetime. The lightweight mechanism to mutual authentication between external entities and the many-core, and a Message Authentication Code to protect the application'

source code are innovations proposed in the Thesis. The protection of application execution without cryptographic mechanisms through the *OSZ*, avoiding both communication and computational resources sharing represent another contribution of this Thesis.

Keywords: NoC-based Many-core System, Security, Mutual Authentication, Application Admission, Secure Zones.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

3PIP – Third Part Intellectual Property

AES – Advanded Encryption Standart

AET – Application Execution Time

API – Application Program Interface

ATZ – ARM TrustZone

CAM – Content Addressable Memory

CPU – Central Processor Unit

DH – Diffie-Hellmann

DMA – Direct Memory Access

DMNI – Direct Memory Network Interface

DOS – Denial-of-Service

DDOS – Distributed Denial of Service

ECC – Elliptic Curve Cryptography

ECDH – Elliptic Curve Diffie-Hellmann

EOP – End-of-Packet

FPGA – Field Programmable Gate Array

GMP – Global Manager Processor

GPPC – General Purpose Processing Cores

HT – Hardware Trojan

HTTPS – Hyper Text Transfer Protocol Secure

I/O – Input/Output

ILP – Integer Linear Programming

IOT – Internet of Things

IP – Intellectual Property

ISA – Instruction-Set Architecture

KDC – Key Distribution Center

LFSR – Linear-feedback Shift Register

LMP – Local Manager Processor

MAC – Message Authentication Code

MITM – Man-in-the-Middle

MMU – Memory Management Unit

MPSOC – Multiprocessor System on Chip

MCSOC – NoC-based many-core SoC

NAS – Numerical Aerodynamic Simulation

NI – Network Interface

NOC – Network on Chip

OSZ – Opaque Secure Zones

OS – Operating System

P + P – Prime + Probe

PE – Processor Element

PRN – Pseudo-Random Number

PRNG – Pseudo-Random Number Generator

PS – Packet Switch

PK – Private Key

PUK – Public Key

PUF – Physical Unclonable Function

RISC – Reduced Instruction Set Computing

RTL – Register Transfer Level

SBR – Segment-based Routing

SBR-SZA – SBR Security Zone Awareness

SCA – Side Channel Attack

SDN – Software Defined Network

SLR – Systematic Literature Revision

SSL – Secure Sockets Layer

SR – Source Routing

SP – Slave Processor

SWS – Sliding Window Search

SZ – Secure Zone

TEE – Trusted Execution Environment

TLS – Transport Layer Security

TNP – Temporal Network Partitioning

VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

WBAN – Wireless Body Area Network

WML – Wrapper Mask Logic

ZKP – Zero Knowledge Proof

CONTENTS

1. INTRODUCTION

Many-core SoCs (System on Chip) are platforms providing high connectivity and massive parallelism for running a wide variety of applications. The use of many-cores systems is continuously increasing, due to the growing number of integrated processing cores into a single chip and the number of fields where they are applied to. A many-core may be classified as heterogeneous or homogeneous (symmetric and asymmetric). Heterogeneous many-cores employ PEs (Processor Elements) with distinct architectures and organizations (like general-purpose processors, graphics processing unit, dedicated hardware accelerators, memories, among others) [Esmaeilzadeh et al., 2012]. Symmetric homogeneous many-core correspond to systems with all PEs having the same architecture and organization. Asymmetric many-core are a particular case of homogeneous systems, where the PEs share the same ISA (Instruction-Set Architecture), but not the organization [ARM, 2013].

A many-core SoC contains PEs or IPs (Intellectual Property) modules interconnected by complex communication infrastructures, such as hierarchical buses or NoC (Network on Chip) [Popovici et al., 2010].

An NoC contains routers and links and is responsible for exchange data and control messages between PEs and peripherals. The routers receives packets from the processors and links assist to reach the packets to other routers on the routing path, which is decided by the routing algorithm. Network Interfaces (NI) connect IPs to the NoC routers. The routing units constitute the underlying communication infrastructure of the system, where multiple interconnected routers define network topologies [Hemani et al., 2000][Benini and Micheli, 2002].

In the context of this work we adopt the term *MCSoC* to describe the NoC-based many-core SoC.

As the adoption and complexity of *MCSoCs* increases, the concern for data protection appears as a new design requirement [Baron et al., 2013]. A *MCSoC* may be used in scenarios where availability is a critical factor and downtimes must be minimized. These systems may also handle sensitive information; thus it is necessary to protect this data from unauthorized access.

According to [Ramachandran, 2002], not only data protection, unauthorized access and availability are concerns on *MCSoC* design. The following seven security principles are generally accepted as the foundation of a good security solution being the three first principles mandatory features:

- Confidentiality: the property of non-disclosure of information to unauthorized processes, entities or users;
- Availability: the protection of assets from DoS (Denial-of-Service) threats that might impact the system availability;

- Integrity: the prevention of modification or destruction of an asset by an unauthorized entity or user;
- Authentication: the process of establishing and validity a claimed identity;
- Authorization: the process of determining whether a validated entity is allowed access to a secured resource based on attributes, predicates or context;
- Auditing: the property of logging the system activities at levels sufficient for reconstruction of events;
- Nonrepudiation: the prevention of any participant denying his role in the interaction once it is completed.

MCSoCs, beyond their inherent scalability, provide massive parallelism and high performance to the users. In such systems, several applications execute simultaneously, sharing computation (processors) and communication (routers and links) resources. This resource sharing lead to security and trust problems, requiring the design of solutions to avoid malicious entities exploring vulnerabilities and breaking some security principle.

The system (hardware and Operating System) and applications are exposed to malicious attackers at different moments of the applications lifetime which can compromise their admission and execution due to the resource sharing feature of *MCSoC* [Real et al., 2016a] [Kinsy et al., 2017] [Reinbrecht et al., 2018].

The application's lifetime encompasses three main phases: (a) before the execution we need to admit the application to the *MCSoC*, mapping the tasks and reserving resources for execution; (b) during the application execution the computation resources run the algorithms and use communication resources to inter-task cooperation and peripherals access; (c) finally, at the end of the execution, the resources release must be provided to enable its use by another application.

The execution of an application with security constraints comprises at least three assumptions. The first one is the secure admission of the application, to guarantee the object code integrity. The second assumption regards the application execution in an runtime environment protected from attacks. The third assumption is related to the protection of the communication with peripherals and shared memories (Figure 1.1).

The *application admission* corresponds to the object code transfer from an off-chip entity to the *MCSoC*. With respect to security in this process, the many-core must trust on the entity transmitting the application and the integrity of the application must be verified to avoid the insertion of malicious code. Examples of attacks include Trojan Horses, man-in-the-middle and spoofing whilst solutions to these issues exist for the Internet, computer networks, and software in general [Hanka and Wippel, 2011][Kuntze, 2013][Khernane et al., 2016].

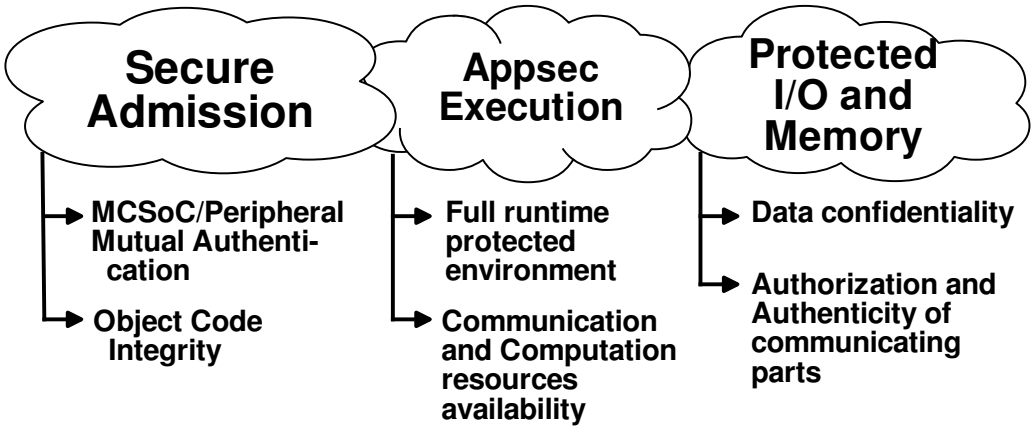


Figure 1.1: Security constrains related to the secure application execution.

At *execution time*, a malicious attacker may have access to sensitive computation or communication data and, therefore, a secure application (App_{sec}) that processes sensitive data will have its security harmed by a malicious process. Examples of attack on such systems include DoS, timing attack, side-channel attack and information leakage [Hu et al., 2015] [Sepúlveda et al., 2015c][Real et al., 2016b].

Adoption of firewalls, encryption mechanisms and resource isolation through secure zones are common strategies to deal with the security threats. The mechanisms used to create such secure zones include encryption, routing algorithm, logical and temporal isolation. Particularly, the scope of this Thesis uses Opaque Secure Zones[1], defined as:

Definition 1. Opaque Secure Zone (*OSZ*) is a continuous isolated area in the system, with a rectilinear shape, with the PEs and routers reserved to execute a single application, without computation and communication resource sharing.

Relative to application *communication with external devices*, unauthorized access to instructions and data in shared memory and peripherals can compromise the applications' execution, due to attacks aiming the information tampering or the information leakage. Other examples of attack include DoS and timing attack [Grammatikakis et al., 2015b][Reinbrecht et al., 2017].

Most proposed security solutions applied to *MCSoC* consider only one of three mentioned assumptions, limited to the application execution (computation or communication protection) and eventually the shared memory access. The concern about memory access is considered from the communication point of view. Proposals regarding the application admission and the access to peripherals are scarce in the *MCSoC* research field. A low-cost protocol for secure application admission and communication with external devices targeting *MCSoCs* is still an open research problem. The state-of-the-art chapter details this statement.

[1] The Opaque Secure Zone definition is introduced in this Thesis. Details are presented in Section 4.3.

1.1 Threat Model

The resource sharing of *MCSoCs* components introduces vulnerabilities to the applications running on it. These vulnerabilities compromise the security principles throughout the applications lifetime, by example:

- *Integrity*: a malicious entity can change the source code of the tasks, at the application admission, inserting Trojan Horses or backdoors aiming a shady behavior of the application during its execution.
- *Confidentiality*: unauthorized access to the data by writing or reading. With different applications sharing the *MCSoCs*, a malicious application can be loaded and executed by a given processor, accessing the memory to retrieve or leak critical data through malicious PEs or peripherals.
- *Availability*: disruption of the system by overloading resources. A malicious application generating packets with a high injection rate can produce this attack, overloading the communication infrastructure.
- *Authentication*: before the application admission, the *MCSoC* must have confidence that the entity that want to run an application in the many-core system proofs its identity.
- *Authorization*: the deployment of an malicious application without authorization enables innumerable threats in the system compromising its integrity, availability and integrity. At execution and communication phases, unauthorized access to peripherals or resources inside the *OSZ* also compromise these features.

Table 1.1 presents the security principles addressed by the Thesis to each phase of the application lifetime. This work does not cover the nonrepudiation and audition principles ('-' signal). The principles addressed mitigating attacks are showed using '◊' signal. The signal '⧫' inform that the security principle is fully address, avoiding the attack.

At the *MCSoCs* context, it is possible to explore such vulnerabilities with attacks that compromise the system by using:

- *Denial-of-Service - DoS*: disruption of the system by overloading resources, compromising its availability. A malicious application task generating packets with a high injection rate can produce this attack, overloading the communication infrastructure.
- *Distributed Denial-of-Service - DDoS*: similar to DoS, uses multiples tasks to attack and disrupt the system by overloading resources, compromising its availability. A malicious application running in distinct PEs can coordinate an attack to a specific router overloading its communication capacity.

Table 1.1: Security principles addressed during the application phases in this Thesis. Legend: -: not addressed, ♦: attack prevented; ◊: attack mitigated

Principles	Application phases		
	Admission	Execution	I/O Access
Availability	◊	♦	◊
Confidentiality	-	♦	♦
Integrity	♦	♦	♦
Authentication	♦	-	♦
Authorization	♦	♦	-
Nonrepudiation	-	-	-
Audition	-	-	-

- *Timing attack*: explores the communication collision between the sensitive traffic and the attacker traffic. The latency interference induced by malicious traffic can provide to the attacker some information about the timing, frequency, and volume of the secure communication.
- Hardware Trojans: a malicious modification of the system's hardware (e.g., inserted into the NoC) aiming to sniff and leakage sensitive data.
- *Spoofing*: a malicious application successfully falsifies its identity to obtain unauthorized privileges.
- *Hijacking*: an attempt to alter the system configuration to execute a set of abnormal tasks along with normal system operation (e.g. during the load of the operating system or an application).
- *Man-in-the-Middle - MitM*: an attack where the attacker secretly relays and alters the communication between the external entity and the *MCSoC*, in such way that each one believes they are directly communicating with each other. This enable the attacker send malicious data or obtain secret information of the *MCSoC*.
- *Trojan Horse and backdoor*: the tampering of the task's source code during the admission of the application can insert malicious code that, during the application execution, can compromise the availability of the *MCSoC* and the confidentiality of the data.

Side channel attacks, as power-monitoring or electromagnetic attacks, are not considered in this proposal. However, the feasibility of such attacks in a system with dozens of processors is unlikely to occur.

Table 1.2 details the attacks prevented or mitigated by our Thesis regarding the secure principles and the application phase.

Table 1.2: Security principles with respective attacks addressed during the application phases in this Thesis.

Principles	Application phases		
	Admission	Execution	I/O Access
Availability	DoS, DDoS	DoS, DDoS, Trojan Horse	DoS, DDoS
Confidentiality	-	Timing attacks, HT, Trojan horse, Backdoor	Data leakage MitM
Integrity	Trojan horse, Backdoor Hijacking, Spoofing	Trojan horse, Backdoor Hijacking, Spoofing	Spoofing
Authentication	MitM	-	Spoofing
Authorization	MitM	Trojan horse, Spoofing	
Nonrepudiation	-	-	-
Audition	-	-	-

1.2 Thesis Statement

It is feasible to develop a protocol to secure applications, including the application admission, runtime isolation of computation and communication resources, and access to external devices (memory and peripherals). Compared to other works available in the literature, the proposed protocol presents a smaller area, smaller communication latency, and execution time overhead.

1.3 Motivation

MCSoCs are becoming the solution to meet the high-performance demand of embedded systems while maintaining the power consumption restrictions during its execution. Examples of modern architectures with a large number of processors interconnected by NoC includes the Mellanox family TILE-Gx72 (72 cores) [Mellanox Tecnhlogies, 2018], Intel Knights Landing [Sodani et al., 2016] and Oracle M8 (32 cores) [Oracle, 2017], Kalray array (256 cores) [Dinechin et al., 2014], and KiloCore chip (1,000 cores) [Bohnenstiehl et al., 2016]. Recently, even complex architectures follow the *MCSoC* trend, such as Intel Xeon i9 (18 x86-processors) [Intel, 2018].

The protection of sensitive data is an ever-increasing concern today with the increased exposure of *MCSoCs*. While running sensitive applications, the splitting and sharing strategy performed in *MCSoC* enable a malicious application to explore the vulnerabilities due to computational and communication resources sharing. Attacks at *MCSoCs* enable to modify the system behavior, extract sensitive information, or disable its operation. In addition, the use of *MCSoCs* in areas where a failure could be catastrophic (e.g., medical devices, automotive and aerospace industry) require the adoption of security mechanisms.

1.4 Objectives

The main objective of this Thesis is to define a set of protocols, executed at runtime, to enable applications with security concerns to run in *MCSoCs* tackling the following issues: application admission, secure execution, and access to peripheral devices.

The following specific objectives must be fulfilled to help achieving the main objective:

I **External entities authentication**
Definition of a secure method to establish the communication of the *MCSoC* with external hardware components. The goal of this step is to authenticate these components and define a secret symmetric key that enables external hardware to send sensitive application and data with integrity guarantees during the application deploy and confidentiality guarantees during the communication with peripherals.

II **Application admission**
Definition of the location and the shape of the *OSZ* to make the application task mapping inside the region. This step requires new heuristics to determine the best location for the *OSZ* according to the application's task number, the *MCSoC* free resources, and the mapping heuristics into *OSZ*.

III **Internal transfer of the secret key and object code verification**
Definition of a secure mechanism to send the secret symmetric key to all processors belonging to the *OSZ*. This mechanism must avoid plaintext key exchange, making hard a possible attack to the task's object code. The external entity requesting the secure application execution sends the tasks' object code to the mapped processor appending a Message Authentication Code (MAC). At the reception, the processor confirms the object code integrity, and then the application execution can proceed.

IV **Close the *OSZ* and secure application execution**
The isolation of the *OSZ* occurs after loading the tasks on the processors. This work uses spatial and temporal isolation, i.e. the processors of the *OSZ* are exclusively dedicated to run the secure application. After isolation, the application can start without encrypting its messages since external traffic that should cross the *OSZ* is blocked. In the same way, any traffic belonging to the sensitive application does not leave the *OSZ*. To meet this requirement a rerouting algorithm is required to circumvent the messages that should cross the *OSZ* region.

V **Communication with Peripherals**
The communication with external devices is part of most applications. To enable data communication with peripherals we propose a mechanism that enable distinguish I/O packets from data packets allowing communication with peripheral without expose the

resources inside the *OSZ*. A secure application can expose sensitive application's data with the communication to external devices. To meet this specific objective, it is necessary to use encryption and the private key previously created. The challenge is the trade-off between the cryptography cost (hardware or software) versus the additional latency in the communication due to this procedure.

VI **Application release**
At the end of the secure application execution, the operating system of each processor in the *OSZ* must clear the task's memory area to avoid the leakage of sensitive data. After the memory cleaning, the processors in the boundary of *OSZ* can accept the message flow crossing again. When the system manager receives the notifications about the release of resources of all processors of *OSZ* these processor become available to other applications.

1.5 Original Contributions

The original contribution of this Thesis is the complete isolation at runtime of the application using secure regions and dynamic rerouting, protecting computation and communication resources from attacks. The proposal isolates a given system region using wrappers, instead of firewalls, reducing, comparatively, the hardware cost. In addition, given the fact that the application is isolated, there is no need to encrypt packets inside the *OSZ*, reducing the communication latency between application's tasks.

Other original contributions are the lightweight methods adopted. The authentication of trusty entities avoids the cost of key exchange protocols for each application admitted in the system since the key exchange occurs once for each entity. For each new application admitted in the system, only a MAC verification is required. The execution of the application inside a reserved region eliminates the need of firewalls and data encryption (except when communicating with peripheral devices).

1.6 Document Organization

The remaining of this document is organized as follows. Chapter 2 presents the state-of-the-art regarding *MCSoC* security issues and a discussion of the main related works. Chapter 3 describes the system architecture presenting the main hardware and software modules that support the used security mechanisms. Chapter 4 presents a general view of the mechanisms adopted in this Thesis to support a secure application lifetime. Chapter 5 details the proposed mechanism to ensure a secure admission of the application into the *MCSoC*, including peripherals authentication. Chapter 6 details the *OSZ* creation, the application execution, the reroute mechanism used to circumvent *OSZ* avoiding the

data traffic of another applications and the *OSZ* release. Chapter 7 details the access to peripherals from the *OSZ* and the mechanism used to protect the data exchange. Chapter 8 presents the general conclusion of the Thesis, directions for future work and the Author's publications during the Ph.D. period.

2. STATE-OF-THE-ART

The *MCSoC* research field is broad, with several research groups working in this area. Concerns related to NoC security is not a recent thread, with publications from the beginning of 2000's, such as [Gebotys and Gebotys, 2003], [Avizienis et al., 2004], [Coburn et al., 2005] and [Evain and Diguet, 2005].

This section presents the literature review on *MCSoC* security. Emphasis is given to the mechanisms used to protect the system on the different lifetime phases, considering the positioning of the mechanisms: at the NoC level; at the many-core level; in other systems or research area. Table 2.1 summarizes this approach, presenting some prevention mechanism found in the literature review to each execution lifetime phase and the respective positioning.

Table 2.1: Security mechanisms proposals to distinct application lifetime phase with respective positioning level

App. lifetime phase ↓	NoC	*MCSoC*	other area
Admission	–	–	[Khernane et al., 2016] Zero Knowledge Proof
	[Sepúlveda et al., 2018] MAC	–	[He et al., 2012] ECDH protocol
Execution	[Ancajas et al., 2014] Packet Certification	–	–
	[Rajesh et al., 2015] Auditing and Firewall	[Isakovic and Wasicek, 2013] Secure Zones and Sym. encryption	–
	[Fernandes et al., 2016] Routing Scheme	[Sepúlveda et al., 2017] Sym. and Asym. Encryption	–
	[Oliveira et al., 2018] Symmetric Encryption	[Sharma et al., 2018] Symmetric Encryption	–
I/O Access	[Grammatikakis et al., 2015b] Firewall	–	–
	[Reinbrecht et al., 2017] Routing Scheme	–	–

During the application admission phase, authentication mechanisms are applied to the entities in order to ensure the integrity of the application's object code (Section 2.1). With respect to the application's execution, the review focus on the protection of the communication resources, the computation resources, and works with both protection mechanisms (Section 2.2). The review of proposed mechanisms to protect the access to peripherals (mainly shared memory) is presented in Section 2.3. Each related work is summarized according to the following structure: (*i*) the attacks employed to obtain some advantage; (*ii*) the threats mitigation methods; (*iii*) the operation of the method; (*iv*) the cost regarding the area, power or latency of the proposed mechanisms.

Section 2.4, present a discussion, positioning this Thesis w.r.t. the state-of-the-art, showing some issues and gaps found in the reviewed works.

2.1 Protecting the application admission

The application admission corresponds to the object code transfer from an off-chip entity to the *MCSoC*. With respect to security in this process, each actor (external entity and *MCSoC*) must confirm the other part's identity, and the integrity of the application must be verified to avoid the tampering of the application's object code. Solutions to these issues exist for the Internet, computer networks, and software using techniques such as ZKP (Zero Knowledge Proof) [Khernane et al., 2016], DH (Diffie-Hellman) protocol based on ECC (Elliptic Curve Cryptography) [He et al., 2012] and MAC [Sepúlveda et al., 2018].

(i) Zero Knowledge Proof protocol

In [Khernane et al., 2016], the Authors present a secure lightweight and energy efficient authentication scheme for WBAN (Wireless Body Area Network) called BANZKP. The scheme is based on a ZKP protocol and a commitment scheme to authenticate the sensor nodes in the WBAN network. They used ZKP to confirm the identity of the sensor nodes while the commitment scheme deals with replay attacks. According to the authors, after the authentication success, an encryption mechanism (not presented in the paper) provides the message privacy protection. Evaluation uses the Omnet++ simulator. Authors claim that the proposed method reduces the memory consumption by 56.13% and energy consumption by 94.11% when compared with the alternative protocol TiniZKP.

(ii) ECDH protocol

He et al. [He et al., 2012] present an authentication scheme for session initiation protocol based on ECDH applied to multimedia services. The proposed scheme consists of three phases: system setup; registration; and authentication. At the setup phase, the system generates a point (P) over an elliptic curve using a large prime number, an integer (K_s) as the secret key, then calculate the P_{pub} = K_s x P as a public key (over the finite field F_P and modular arithmetic over P), and publish these parameters, except K_s. At registration phase the user became a new legal user sending the username and password, and the server computes two secret values, the first based on the user password and another based on K_s value. Then the server calculates and stores a password verifier based on these two values. At the authentication phase, the user sends a request using the username and a value (R_1), calculated with P_{pub}; the server responds with a challenge using (R_1); the user validate the server's challenge and, if it holds, compute a response that involves the user password, the R_1, the server's challenge, and the username. Finally, the server validates the answer to accept the user's request. After the mutual authentication, a shared session key is calculated in both sides using common values. The paper presents a security analysis and a performance analysis based on the number of elliptic curve scalar multiplications,

modular multiplications, modular inversions, and one-way hash functions operations used on the scheme.

(iii) Message Authentication Code

Sepúlveda et al. [Sepúlveda et al., 2018] propose a runtime mechanism based on MAC and PUF (Physical Unclonable Function) to provide memory integrity and authentication. The MAC uses the SipHash algorithm. The proposed mechanism prevents code injection and memory modification attacks, including spoofing, reallocation and replay attacks to the off-chip components. The mechanism is divided into three stages: key generation; MAC initialization and application installation; and operation. At the key generation stage a key for each application is derived using a challenge PUF and a random number. These are used to compute the helper data, used for generation and reconstruction of the key. Application identifier, PUF challenge, and helper data are used by the Authentication Controller module (Figure 2.1) to trigger the regeneration of the application-specific key which is required for MAC initialization during application installation as well as during the SoC operation.

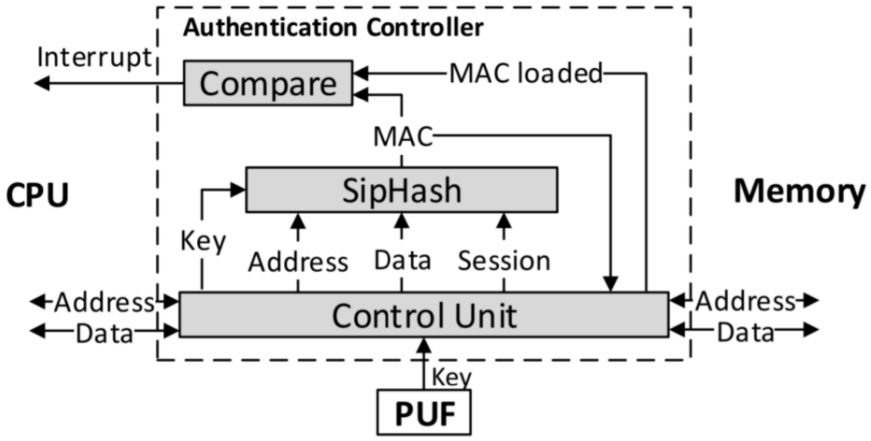


Figure 2.1: The Authentication Controller module. [Sepúlveda et al., 2018].

At the MAC Initialization and Application Installation stage, before the normal system operation, the MAC for all applications that are going to be executed are computed and stored in the off-chip memory. During SoC operation stage, code/data may be migrated to another IP core or write back to the main memory. In such situations, MACs are computed in the Authentication Controller and stored in the off-chip memory. During a read access, the loaded data is used to recompute the MAC, which is then compared with the MAC previously stored in memory. For an authentic memory line the computed MAC matches to the MAC stored in memory. The Authors implement the mechanism using Xilinx Nexys4 FPGA (Field Programmable Gate Array) and the evaluation FPGA resource utilization shows the following overhead of the modules when compared with the baseline system: +132% of FFs (Flip-Flops) and +135% of LUTs to the entyre secure PUF module; +19% of FFs (Flip-Flops) and +17% of LUTs to the authentication Controller module. Performance evaluation shows a

low impact on the application degradation due to the mechanism: up to 25% (write intensive Qsort application).

2.2 Protecting the application execution

The literature presents a diversity of mechanisms used to protect communication, computation and memory accesses in *MCSoCs*. Mechanisms protecting the communication include: (*i*) firewalls; (*ii*) secure zones; (*iii*) routing schemes; (*iv*) temporal network partitioning; (*v*) cryptography; (*vi*) packet certification. To protect computation the main mechanism used is spatial and/or logical isolation. The mechanism protecting memory accesses include routing schemes and firewalls.

2.2.1 Protecting communication

The communication subsystem are attacked by Hardware Trojans (HT) or malicious processes are employed to attack the communication subsystem to obtain information leakage or break the system security. The main reported attacks are DoS [Hu et al., 2015][Sepúlveda et al., 2015a] [Real et al., 2016b], HT [Rajesh et al., 2015][Ancajas et al., 2014], timing side-channel attack [Fernandes et al., 2016][Wassel et al., 2014], and attacks to confidentiality and integrity [Rajesh et al., 2015][Wang and Suh, 2012].

We describe bellow works that use different mechanisms to protect the communication subsystem, although the computation is still exposed due to resource sharing that is not directly addressed in the proposed mechanisms.

(i) Firewalls

Rajesh et al. [Rajesh et al., 2015] propose a runtime latency auditor for NoCs, called RLAN, to dynamically monitor the on-chip resources availability and properly filter the malicious traffic. The goal of the proposed method is to prevent HT and mitigate attacks to the availability of the NoC. According to the Authors, RLAN is a non-invasive technique that can work without any modification with third part IP NoCs. The method is implemented at the NI (Network Interface), as shown in Figure 2.2.

The principle adopted by the RLAN design is that packets traversing routes with significant overlap (spatial similarity) around the same time (temporal similarity) have comparable latencies. The method includes two major steps: (a) all packets are tagged with a timestamp to enable latency computation; (b) creation of Source/Destination traffic in RLAN. The second step creates a control traffic to compute the reference latency, using it to detect

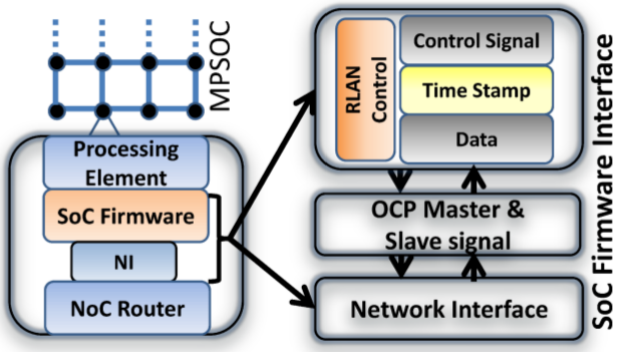


Figure 2.2: Block diagram of RLAN's network interface [Rajesh et al., 2015].

attacks by HTs or malicious processes. The authors use BooKSim 2.0 Simulator to evaluate the performance of the method, and the 45nm TSMC standard cell library to evaluate power and area. Results show that RLAN incurs an overhead of 12.73% in area, 9.84% in power and 5.4% in terms of network latency when compared to the baseline NI.

Hu et al. [Hu et al., 2015] propose a three-level firewall to provide access control, authentication, and availability of the communication system, preventing information leakage and DoS attack. The proposed method makes a design time analysis of the traffic and the NoC architecture [NaNoC, 2015] applying Integer Linear Programming (ILP) to select the levels and position of the firewalls: (a) between a PE and a router or; (b) between routers. Figure 2.3 illustrate the method in a smartphone application.

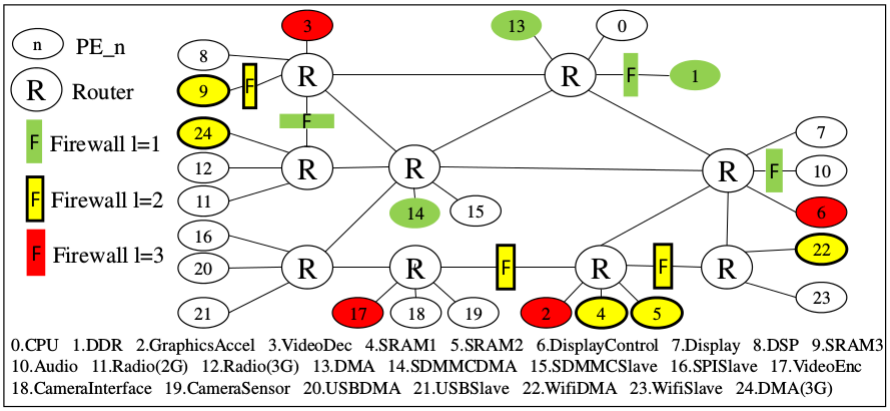


Figure 2.3: Example of optimized NoC topology for a smartphone application [Hu et al., 2015].

The goal of the Authors is to reduce the communication overhead required for security information in packets' headers. The authors do not present the area, power or performance costs. Results show a 30% to 63% overhead reduction in header size of packets when the firewalls are positioned between routers over a standard solution, i.e., firewalls connected to the NIs.

(ii) Routing Scheme

Sepúlveda et al. [Sepúlveda et al., 2015a] present a runtime method to prevent timing side-channel attacks and information leakage. The work proposes two mechanisms: adaptive routing and random arbitration. The proposed method assumes that a malicious task in the path of a memory access may extract sensitive data from the temporal behavior of the communication flow. To prevent the temporal behavior extraction, the first mechanism implements a random arbitration in the routers to remove the temporal correlation of malicious injected traffic and the memory access. The second mechanism is the adaptive West-First routing method, which allows, in some situations, to make turns to escape from blocking conditions. Thus, the secure traffic deviates from the malicious traffic in the path automatically to get a free one. Results show that random arbitration and adaptive routing maintains an average throughput of 0.41 and 0.6 flits/cycle, respectively, to one secure traffic flow with several malicious injection rates. The area overhead due to the random arbitration and adaptive routing is 11% and 5%, respectively. The power overhead due to the random arbitration and adaptive routing is 9% and 8%, respectively.

(iii) Secure Zones - Routing Scheme

Fernandes et al. [Fernandes et al., 2016] propose a design time method that enables the creation of Secure Zones (*SZ*) based on the routing algorithm to mitigate DoS and timing side channel attacks. The authors extend the Segment-based Routing (SBR) to security purposes creating the SBR Security Zone Awareness (SBR-SZA) that enables the creation of *SZs*. After running the SBR-SZA, the Region-based Routing Algorithm (RBR) creates routing restrictions avoiding shared paths between different applications and deadlock free paths.

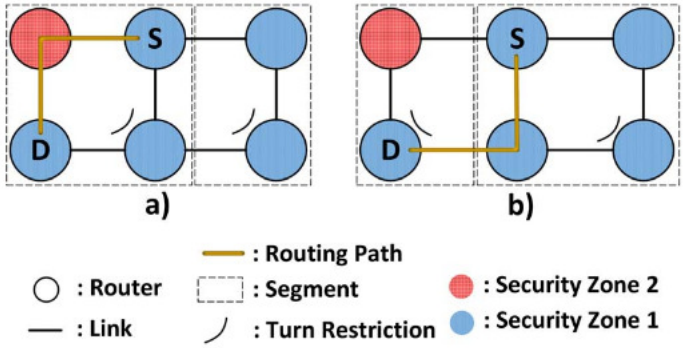


Figure 2.4: Example of two SBR segment computations: a) the path among S and D goes through an insecure element due to a routing restriction; b) a set of restrictions in the segments enables a secure path between S and D. [Fernandes et al., 2016]

The evaluation method uses two synthetic scenarios and the Numerical Aerodynamic Simulation (NAS) benchmark [NASA, 2015] to measure the overhead in the size of

the routing tables and the communication latency penalty. Results show up to 16.56% overhead in the size of the routing tables.

(iv) Secure Zones - Encryption

Sharma et al. [Sharma et al., 2018] proposes an encryption mechanism for zone-to-zone secure communication. The authors present a runtime protocol (PF-ID-2PAKA) that generates private/public par keys for each IP core of the SoC. The secure zones are created dynamically having an anchor node responsible by the secure communication with other zones. The protocol enables dynamically creation of session keys used to encrypt the message flow between the anchor nodes, protecting the communication.

(v) Temporal Network Partitioning

Temporal network partitioning (TNP) employs explicit flow separation to avoid interference of low-priority flows in high priority flows. The goal of the approach is to mitigate DoS, timing side-channel attacks and information leakage [Wassel et al., 2014] and [Wang and Suh, 2012].

In [Wassel et al., 2014] and [Wassel et al., 2013], the authors propose a design time method to create domains of noninterference to prevent DoS and timing attacks. Noninterference means that packet injection from one domain can never have any effect on the packets delivered from other domains. The domains are implemented using virtual channels, and the noninterference is obtained through bounded priority arbitration, called surf scheduling at each router port. In this schedule, a packet waits until it can be forwarded in one dimension (e.g. X-direction) and then does not experience any wait at any downstream router in this dimension. After finishing the first dimension, the packet might experience another wait until it can be forwarded to the next dimension. The authors call this *schedule surf scheduling* because a packet is like a surfer who waits to ride a wave to some location and then waits to ride another wave.

Figure 2.5 shows an example where the router forwards the flits in a single clock cycle. The arbiter schedules the flows according to the sequence: two white; one gray; one black (the white flow receives 50% of the bandwidth). A packet (the white box under the node S) belongs to the white domain and is sent from the node S to the node R. The Figure 2.5 contains six consecutive cycles (a to f). At $T = 1$, the packet is forwarded to the S port in the Y-dimension (which is scheduled to forward white packets). It keeps moving in the y-dimension until $T = 3$, when it needs to move in the X-dimension on the W port. The packet waits two cycles ($T = 4$ and $T = 5$) until it is the white domain's turn on the W port, and finally it is forwarded to its destination on $T = 6$.

The proposed SurfNoC was implemented in BookSim 2.0, a cycle accurate simulator. The latency evaluation shows that the overhead of surf scheduling is almost independent of network size (average number of hops), leading to a constant overhead of 19 cycles (ex-

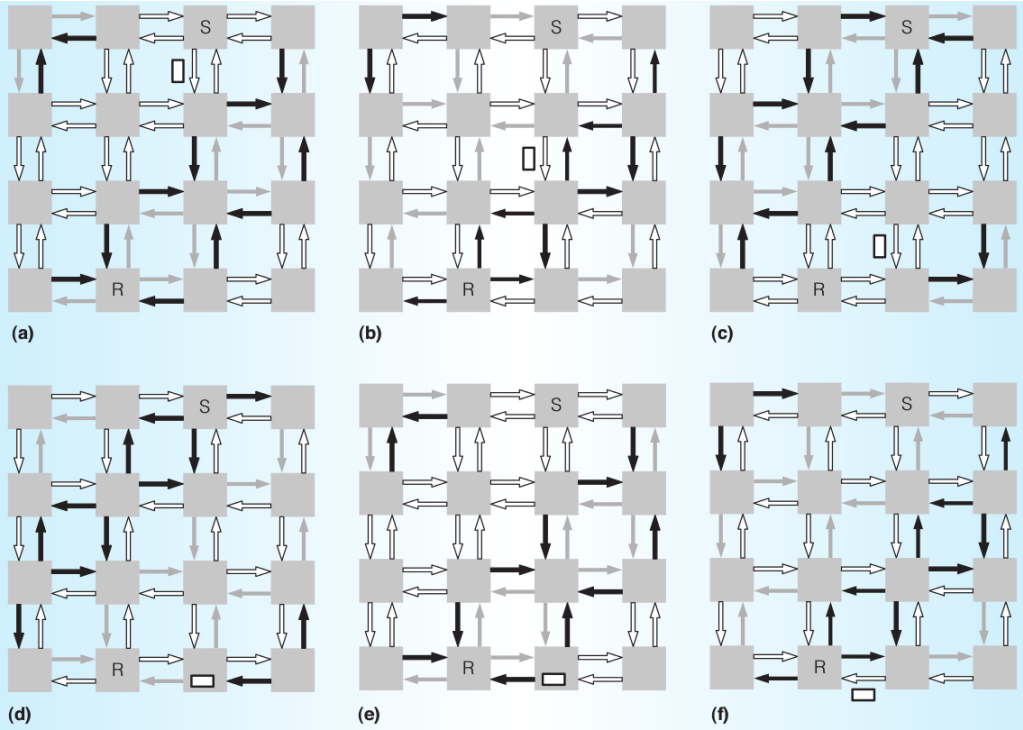


Figure 2.5: Example of Surf scheduling in a 16-node 2D mesh with three application domains (denoted by white, gray, and black). [Wassel et al., 2014]

cept for 16 nodes) because the packet wait time depends only on the number of dimensions and domains. The drawback of the proposed method is that increasing the number of domains also increases the number of virtual channels, increasing the router area an power consumption.

Wang et al. [Wang and Suh, 2012] propose a design time priority-based arbitration and a static limit mechanism to provide protection against information leakage and DoS. The idea is to assign high-priority to low-security traffic, in such way that its behavior is not affected by high-security traffic. With this scheme, when flows from two different security levels compete for the router traversal, the low-security flow always wins due to the arbitration police, avoiding a malicious task to infer timing information over the high-security flow. Virtual channels are statically allocated to each security domain to remove interference in buffers. To prevent a possible DoS attack due to this unfair arbitration scheme, the authors include an additional mechanism that monitors and limits the amount of the low-security traffic regardless the amount of high-security flows. The proposed method evaluation uses the Darsim Simulator. As expected, results show that the method increases the performance in low-security flow and decreases the high-security flow.

(vi) Cryptography

Ancajas et al. [Ancajas et al., 2014] present a runtime method to protect information leakage from HTs. The authors propose the Fort-NoC, a three-layer security mechanism. These security measures are introduced in the NI of the NoC. The first layer is the Data

Scrambling (DS), which makes the HT activation harder. This is done with XOR cipher encryption [Churchhouse, 2002] to encrypt the data before sending it to the NoC. The second layer is the Packet Certification (PC) that attach an encrypted tag at the end of the packet before injecting it into the NoC. This is obtained creating a random lookup table dynamically at the boot-time that creates a 16-bit unique identifier for each node in the system. Based on the destination node of a packet, each data packet embeds a tag containing the translated identifier of the destination node from the lookup table. The third layer is the Node Obfuscation (NObf) that decouples the source and destination nodes of a communication to increase side-channel resilience. The NObf is obtained with task migration. The evaluation of the area shows an overhead of 0.34% and 9.57% to PC and DS respectively. The power consumption evaluation shows no overhead to PC technique and overhead of 5.8% to DS technique. According to the authors, the performance evaluation shows an overhead of 5.9%.

Silva and Zeferino [Silva and Zeferino, 2017] propose the use of an AES block and a KDC (Key Distribution Center), adding authenticity and confidentiality in the message flow of the SoCIN NoC. The KDC share different master keys with each node (the authors do not explain how this is executed). During the operation, when a node A wants to send a sensitive message to node B, the node A send a session key request to the KDC. This request inform the communicating nodes (A and B) an is encrypted using his master key of node A. The KDC decrypt the request and generate (using a LFSR) a session key. The KDC sends the session key generated to node A encrypted with its master key and another message with the session key to node B encrypted with the note B's master key. At this point the two communicating nodes (A and B) have a session key to encrypt the message at source node A and decrypt message at target node B. Figure 2.6 illustrate the process.

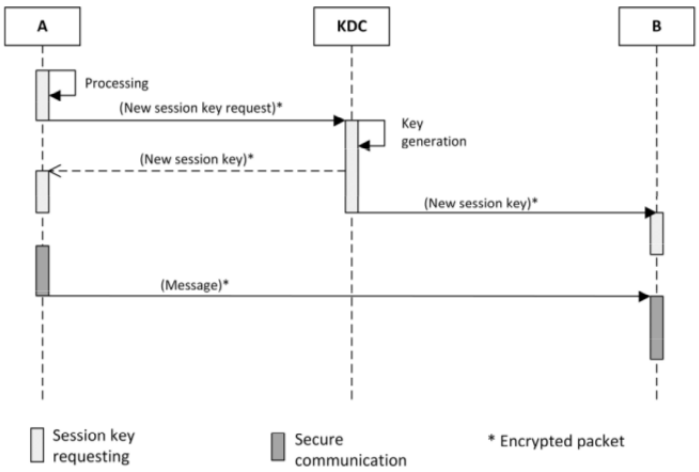


Figure 2.6: Establishment of a secure session in [Silva and Zeferino, 2017].

At the paper, it is not clear at the work if each message exchange needs a new key session or the same key session is used by all messages from node A to node B. The evaluation shows that the communication using the security mechanism is from 7 to

17.6 times slower if the secure session was already established or not, respectively. The hardware cost evaluation using Xilinx ISE sinthesis toll shows the additional costs in LUTs, FFs and BRAMs equal 233.5%, 188.6%, and 687.5%, respectively.

Oliveira et al. [Oliveira et al., 2018] propose an architecture that includes a firewall capable of filtering incoming and outgoing NoC traffic, an AES cipher block to encrypt the NoC flow and, an auxiliary NoC that use a Hamiltonian path to configure the firewall rules and distribute the keys. Figure 2.7 presents the interfaces connected to the firewall. The router and the NI signals are the same. Instead of changing the interfaces, state machines in the firewall manage the flow control signals. The firewall may encrypt or not the packets according to an identifier in the packet.

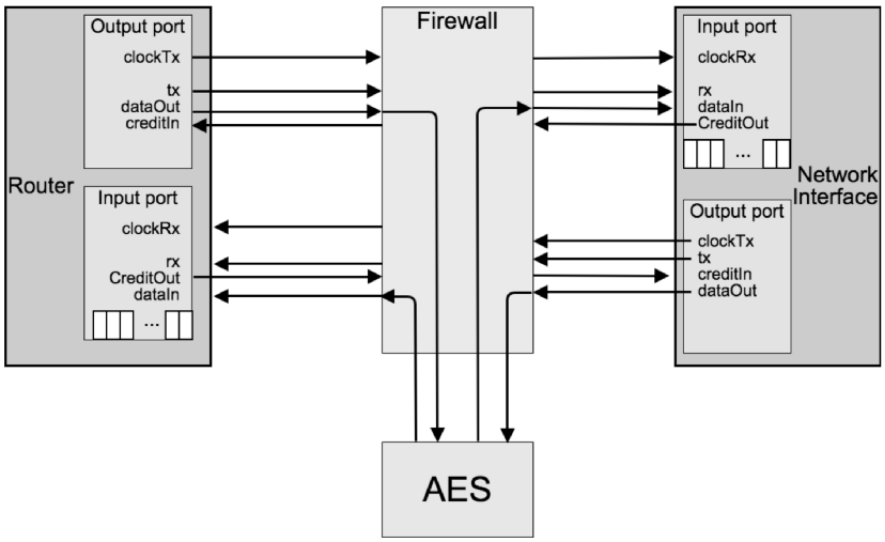


Figure 2.7: Representation of the Firewall connections. [Oliveira et al., 2018]

The firewall plus the AES module increases the router area by 193.7% and latency increases (*i*) in the best-case scenario (disturbing traffic in the path of the ciphered traffic) by 126.3%; (*ii*) in the worst-case scenario (contention due to the simultaneous need of encrypt and decrypt packets by the same AES block) by 395.92%.

In Kinsy et al. [Kinsy et al., 2017], the authors introduced Hermes, a secure multicore computing architecture framework. The proposed scheme claims to prevent DoS, virtual channel and physical memory attacks by creating a virtualization layer that isolates computing threads based on system and user-defined trust levels and security policies.

Hermes achieves both hardware and software views of secure processing by grouping processors into physical zones called wards and virtual logical zones called islands. The wards have different secure level defined at design time and each one have a anchor node responsible by its key management.

The environment protects the physical memory using an MMU (Memory Management Unit) with access restrictions based on two tables (Access Code Table and Base Table Entries) that provide a firewall behavior to the MMU (Figure 2.8.a). Hermes achieve com-

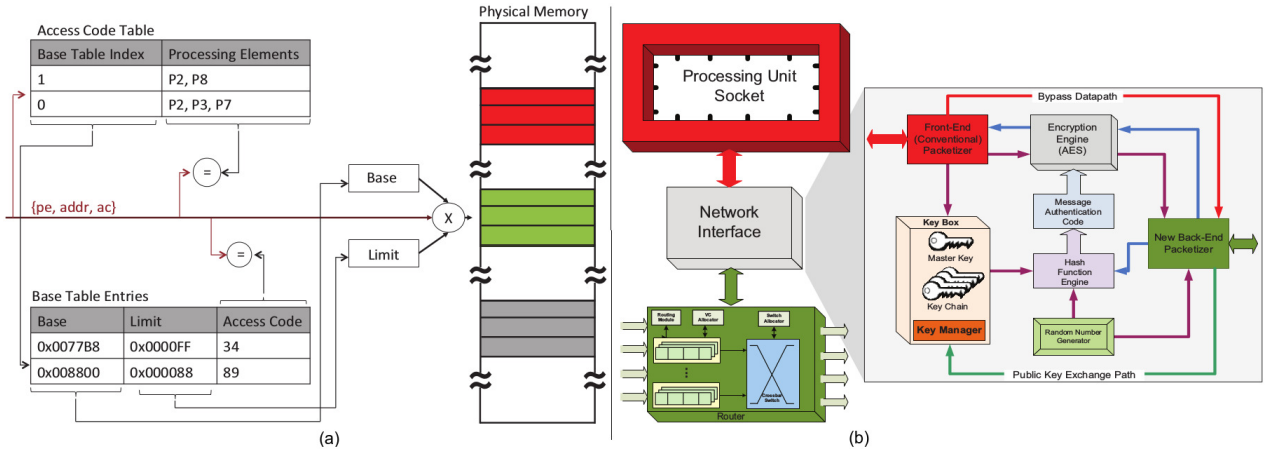


Figure 2.8: (a) Memory access mechanism. (b) Hermes PE architecture with detailed NI. [Kinsy et al., 2017]

munication protection using several mechanisms: (*i*) DH protocol distributes public keys to anchor nodes of the physical zones, with join and leave operations to distribute the keys to the logical zones; (*ii*) a AES module in the NI to encrypt the traffic flow when required; (*iii*) a key manager in the NI to select an appropriate key during the operation; (*iv*) MAC and hash modules to generate session keys (Figure 2.8.b). Beyond those mechanisms, the environment provides a routing algorithm to prohibit or limit the traversal of zones by non-member generated traffic.

According to the Authors, the hardware overhead to fully implement the security features of Hermes architecture is 17%. The performance results on the SPLASH-2 benchmarks presents an overhead of 1% to 9% across all the benchmarks when compared to the non-secure baseline architecture.

(vii) Packet Validation

Boraten et al. [Boraten and Kodi, 2016] propose a runtime packet-security (P-Sec) method, including a packet validation technique to protect compromised NoC architectures from fault injection side channel attacks (potentially DoS) and HTs by merging two error detection schemes, namely algebraic manipulation detection (AMD) and cyclic redundancy check (CRC) codes. According to the authors, in normal operating environments (not under attack) CRC is capable of detecting faults in packets since the fault rates are low. For cores sending sensitive data over the network, P-Sec is turned ON and they switch from CRC to AMD mode to protect sensitive packets from fault injection attacks.

Figure 2.9 shows, at the top, the changing between normal operation (CRC) and secure operation (AMD). In the same Figure, at the middle, the logical arrangement of two modules and, at the bottom, the packet structures in the distinct operation modes. In the AMD technique the number of redundant bits is a function of the message length and the size of a random number internally used.

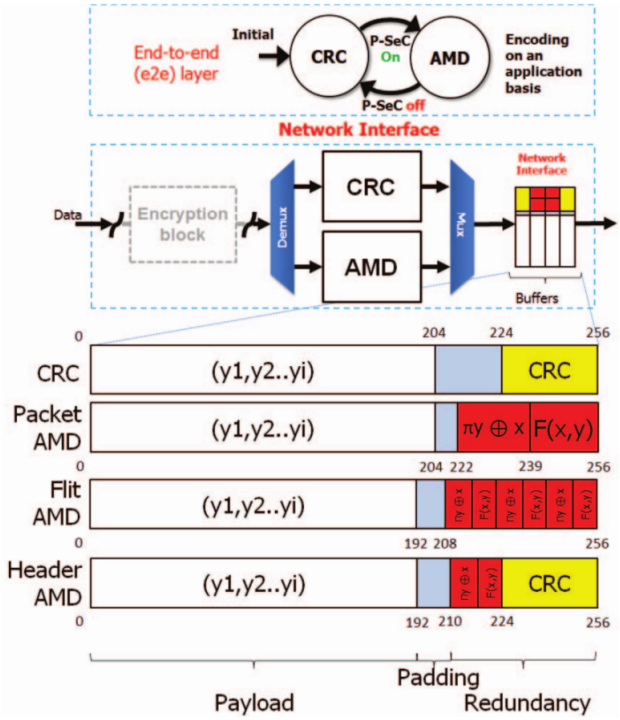


Figure 2.9: Overview of P-sec architectural blocks and structure of packets [Boraten and Kodi, 2016].

The work uses CRC messages with 204 bits length and CRC of 32 bits length; AMD messages with 204 bits length, random number of 17 bits and redundancy of 17 bits (204,17,17). The evaluation implements four methods (AMD, CRC-32, JTEC-QED and SECDED). Compared to SECDED, the smallest method, the power consumption is 2.68x and 8.36x greater to CRC-32 and AMD, respectively. The evaluated area, in comparison with SECDEC is 3.5x and 16.1x greater to CRC-32 and AMD, respectively. Performance results indicate P-Sec reduces overhead compared to Fort-NoCs.

2.2.2 Protecting Computation

The computation protection encompass mechanisms to avoid the processors sharing between distinct applications or deeply embedded into the processor micro-architecture mechanisms. The review shows that logical and spatial isolation are adopted [Real et al., 2016b][ARM, 2018]. When just processors or clusters isolation is used, the communication system remains exposed to attacks. Solutions based on processor micro-architecture mechanisms such as obfuscated instruction execution [Fletcher et al., 2012] or PUF-based authentication architecture [Hoffman et al., 2015] are not addressed in this work.

Real et al. [Real et al., 2018] and their previous works [Real et al., 2016a][Real et al., 2016b] propose a logical and spatial isolation of sensitive applications through the dynamic creation of *SZs* to mitigate DoS and cache SCA attacks at runtime. The architecture uses the MPSoCSim [Wehner et al., 2015], a Mesh NoC where each router is connected to

a cluster with 4 processors (with local memory), 1 shared memory and 1 shared bus (see Figure 2.10).

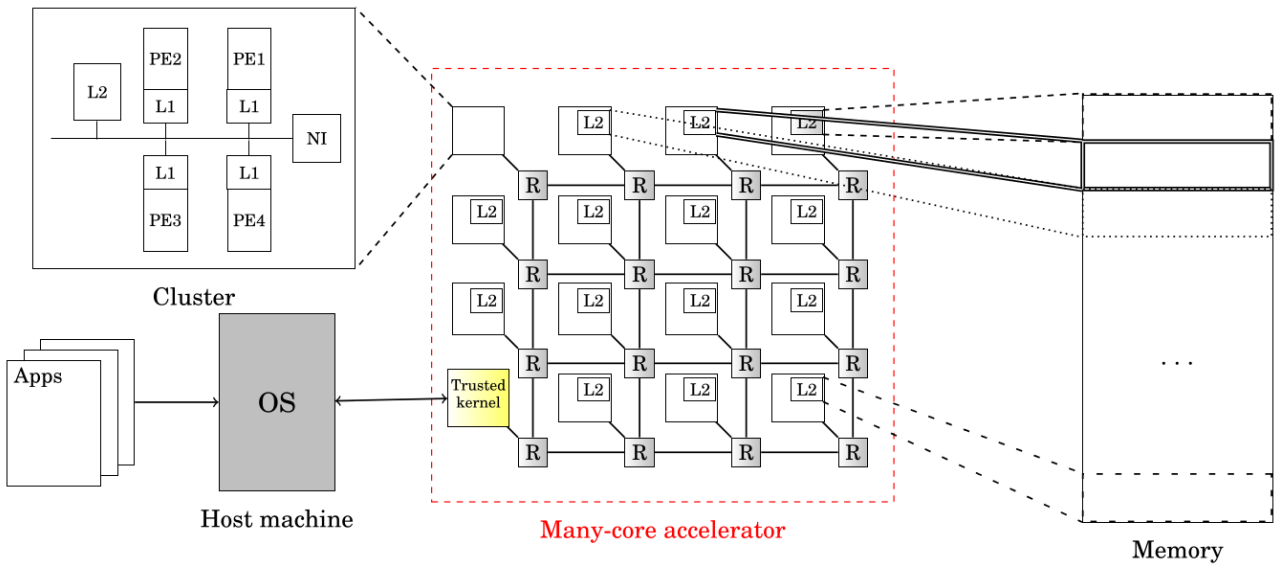


Figure 2.10: Overview of architecture [Real et al., 2018].

Only cluster resources are isolated by the *SZ*. If a task needs to communicate with a task in another cluster the message is sent through an insecure channel.

The evaluation focus is on the creation of secure zones with different deployment strategies (number of clusters in the *SZ*) and execution scenarios (number of isolated application and their priorities). According to the chosen deployment strategy and execution strategy, either the isolated applications performance or the non-isolated applications performance can be penalized. The performance overhead of the proposed mechanisms increases with the number of required secure zones. The worst-case shows an execution-time overhead up to 35.86% over the baseline.

ARM processors provide the ARM TrustZone (ATZ) [ARM, 2018], a hardware support for the creation at runtime of Trusted Execution Environments (TEEs) and therefore the isolation of applications in the same processor. This feature creates two virtual processors and two Memory Management Units (MMU), allowing to execute a secure and a non-secure application simultaneously. However, at any instant, only a single domain in the system is secured. TTE allows the secure partition of shared memory controlling memory accesses to avoid data extraction and change (mitigating confidentiality and integrity attacks). Nevertheless, in multicore and many-core systems architectures, applications running on different processors share resources such as the communication infrastructure (NoC, buses) and memory. Thereby, with TTE, applications running on different processors are not protected from each other since sharing the communication infrastructure leads to possible leakage of information.

2.2.3 Protecting Computation and Communication

This section includes works that protect both computing and communication simultaneously [Sepúlveda et al., 2017][Isakovic and Wasicek, 2013]. These works use firewalls or encryption mechanisms along with isolation.

(i) Secure Zones - Partition and encryption

Isakovic et al. [Isakovic and Wasicek, 2013] obtain computation and communication protection using spatial isolation with encryption mechanisms. The authors propose an architectural partitioning of the MPSoC resources at design time to provide availability, confidentiality and integrity. The Authors adopt security components like a *secure microkernel* and a *secure channel* infrastructure that includes cryptography and firewalls. The Authors propose to migrate the security functions from application components to the security components. To obtain a secure environment for applications, the Authors use Spatial Isolation of applications and secure channels (encryption) to data exchange. The proposed method uses the ACROSS MPSoC architecture [Salloum et al., 2013], but do not detail the implementation methods and how the protocols work. Figure 2.11 shows the block diagram of architecture (TISS means Trusted Interface Subsystem). The authors shows a use-case of the proposal on a Engine Control Unit (ECU) but do not present results of area, power consumption or latency.

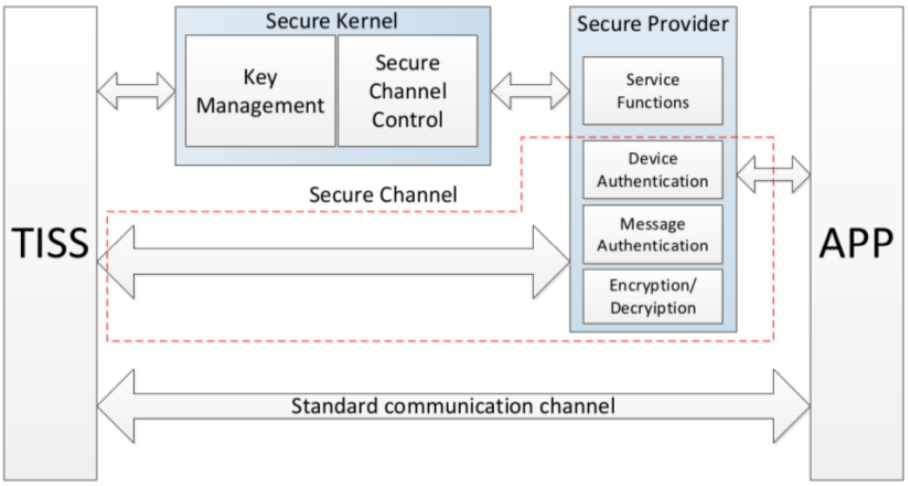


Figure 2.11: Block diagram of Secure Communication Architecture [Isakovic and Wasicek, 2013].

(ii) Secure Zones - Spatial isolation and encryption

Sepúlveda et al. [Sepúlveda et al., 2017] also protect computation and communication resources using spatial isolation with encryption mechanisms. In the incremental work [Sepúlveda et al., 2017] [Sepúlveda et al., 2016] [Sepúlveda et al., 2015b] and [Sepúlveda et al., 2015c] the authors propose an NoC-based architecture that implements runtime disrupted *SZs* using three cryptographic techniques: Hierarchical Diffie-Hellman, Hierarchical

Tree-based Diffie-Hellman and mapping key predistribution scheme. The method prevents attacks to availability, confidentiality, and integrity of the system. The architecture adopts two NoCs: (a) data NoC, used by the application data; (b) service NoC used to exchange the security control packets (key exchange, firewall rules, etc.). After mapping the application, one of the key agreement protocol is executed between the mapped PEs using the service NoC. The encryption/decryption is obtained XORing the message with the shared key. Figure 2.12 presents an example with one disrupt *SZ*, two malicious routers and one infected IP.

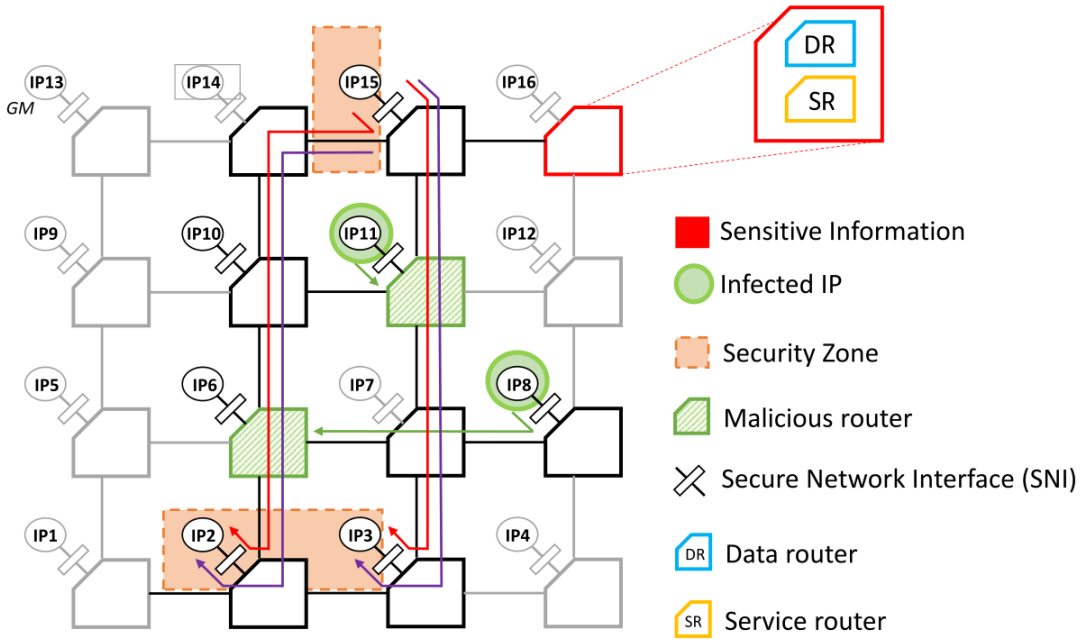


Figure 2.12: Security zones at MPSoCs interconnected through a two-level NoC (Service and data NoC). [Sepúlveda et al., 2017]

The architecture was modeled in SystemC-TLM and RTL-VHDL, and the evaluation uses the SHOC simulation environment. Despite the use of cryptographic primitives like Diffie-Hellmann, the authors show an area overhead of 20% and a power consumption overhead of 12.7% over a baseline implementation. The latency evaluation overhead shows a dependency with the number PEs in the SZ, between 30% and 55% over the baseline implementation.

2.3 Protecting the Memory Access

When the MPSoCs architecture has shared memories, it is necessary to protect the memory accesses. The protection must prevent unauthorized tasks to read or write some memory blocks and avoid information leakage in the communication subsystem.

(i) Routing Scheme

Reinbrecht et al. [Reinbrecht et al., 2017] and their previous works [Reinbrecht et al., 2016a][Reinbrecht et al., 2016b] propose the improvement of the Prime + Probe (P + P) attack introduced by Osvik [Osvik et al., 2006], expanding it to the communication structure of the MPSoC (i.e. the NoC). The (P + P) attack is a timing side-channel attack that exploits the information leakage through communication timing behavior. The attack is executed over the interaction between the shared cache and the victim IP that is reading the AES secret key.

The proposed attack has two variations: (i) (P + P) firecracker; (ii) (P + P) arrow. The authors shows results to both attacks. In the (P + P) firecracker technique, 14 cryptographic tasks were needed to complete the attack, properly recovering 12 of 16 bytes from AES key. The (P + P) arrow technique performed 256 encryptions in the attack and recovered 9 of 16 bytes from AES key.

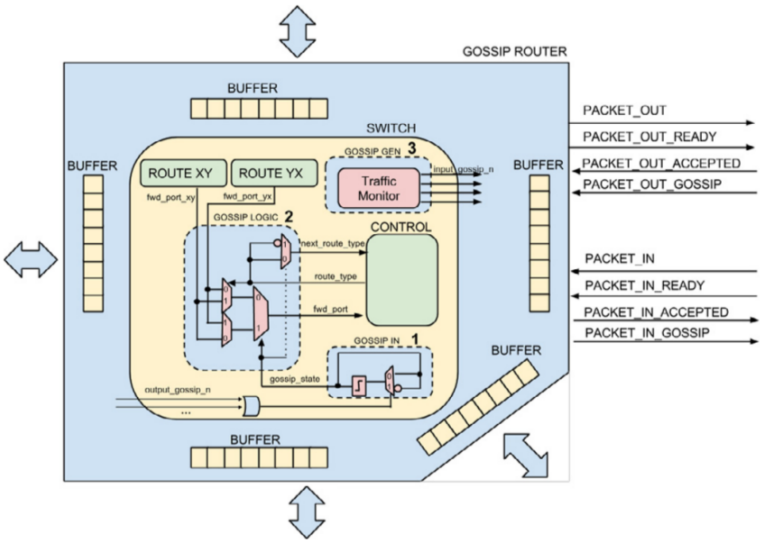


Figure 2.13: Gossip router microarchitecture. (1) Gossip In block; (2) Gossip logic; (3) Gossip generator [Reinbrecht et al., 2017].

To mitigate the (P + P) attacks, the authors propose the Gossip NoC. The Gossip NoC combines two strategies to protect the MPSoC against timing side-channel attacks: (*i*) detection, which includes a bandwidth monitoring and a gossip message generation in the presence of an abnormal behavior that enables the second strategy; (*ii*) protection, triggered when any gossip message is received and which is able to modify the route of the packet (XY routing algorithm to the YX). Figure 2.13 shows the Gossip router microarchitecture.

The work Reinbrecht et al. [Reinbrecht et al., 2017] shows that the Gossip router increases the logic area of the NoC about 21%. The unprotected NoC represent 35% of the MPSoC, became 42.5%, meaning that the effective logic overhead in the system was 7.5%. When calculating the same impact for power, the authors obtain 16.2% of power overhead over the baseline router and 1.18% of power overhead in the entire system.

Another cache collision attack in NoC-based SoC is presented in [Reinbrecht et al., 2018]. The authors show the attack efficacy to fully recover an AES key. The paper do not bring up a mechanism to mitigate the attack in the work.

(ii) Firewall

Grammatikakis et al. [Grammatikakis et al., 2015a] and their correlated works [Grammatikakis et al., 2015b][Papadimitriou et al., 2015] [Grammatikakis et al., 2014] propose a source-side firewall at the NI which, by checking the physical address against a set of rules, rejects untrusted CPU requests to the on-chip memory. The Authors claim that the proposed method protect to DDoS and threats related to data leakage, confidentiality, integrity, and availability. The firewall architecture has three modules: (a) the operating mode controller (OMC), that accepts, decodes and dispatches NoC firewall commands; (b) the segment-level rule-checking (SLRC), that module processes incoming memory accesses and configuration commands; (c) the interrupt unit (INTU) that accepts interrupt requests from the OMC and SLRC modules and reports interrupt contexts to the CPU. Figure 2.14 presents a top-level block diagram of proposed firewall.

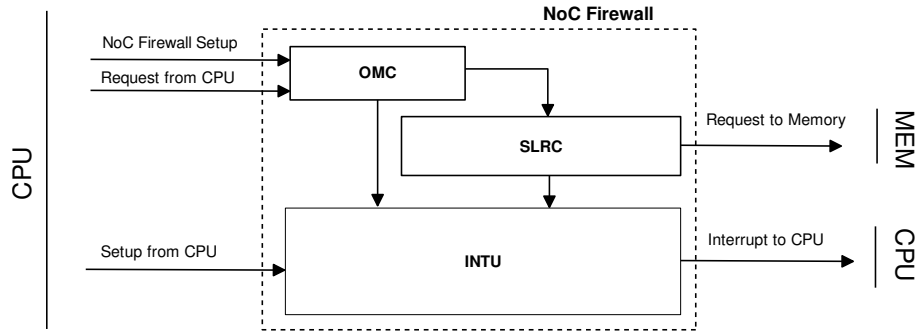


Figure 2.14: Top-level NoC firewall [Grammatikakis et al., 2015a].

The evaluation environment uses the STNoC, a ring-based NoC topology. First a time-annotated RTL description was implemented, then a set of GEM5 simulations are executed. Results shows a reduction on power consumption by 30% due to the fact that malicious requests are prevented from entering the network, thus fewer packets are released, resulting in lower network traffic and smaller queues and activities. Due to the same reason the delay to the packets transverse the STNoC decrease 20.47%.

2.4 Discussion

Proposals to provide mutual authentication of entities (users, sensors, peripherals) and application deploy exists in other computing areas (wireless sensor networks, WBAN, multimedia services, cloud computing, etc.), however, according to the literature review these issues have scarce attention in *MCSoC* research area.

The recent work Sepúlveda et al. [Sepúlveda et al., 2018] deals partially this issue presenting a MAC mechanism based on PUF to protect the memory from data forgery. The application admission protection is not directly addressed since that the paper doesn't discuss the authentication of the entities and the secure transfer of the object code to the memory, focusing on the protection during the memory access provided by the MAC scheme, that includes the tasks object code.

The SLR shows that during the application execution phase, most works related to the security of *MCSoCs* protect just the communication subsystem. The main mechanisms used are firewalls, temporal network partition, routing schemes and secure zones. Table 2.2 summarizes the characteristics of state-of-the-art proposals. The first column presents the primary author, year and reference paper of proposed mechanism. The second and third columns indicate if the proposed mechanism protects the communication and/or computation. The fourth column presents the protection mechanism. The fifth column shows which attack or malicious behavior is mitigated by the proposed mechanism. The sixth and seventh columns summarize what happens at design-time and runtime for each proposed mechanism. Table 2.2 also has three blocks, each one presenting works related to the protection of different application lifetime phase (admission, execution, peripheral access).

Several works ([Hu et al., 2015], [Fernandes et al., 2016], [Wassel et al., 2014]) adopt design time methods. Methods deployed at design time enable the adoption of sophisticated and robust algorithms to provide solutions to the security problem since they do not have limitations related to the computation time of the heuristics. However, design time methods are not applicable in dynamic workload scenarios. Thus, these methods are limited to scenarios where the workload is known beforehand, without any change during the life cycle of the system. In the review, only [Wassel et al., 2014] addresses this issue suggesting that their proposed method can be used in aerospace or medical devices. According to the authors, these fields require high performance, security and have static workloads.

The most common and intuitive approach to protect communication refers to encryption mechanisms. The review shows the use the AES modules incorporated to the NI, ciphering and deciphering the message flow in works such as [Silva and Zeferino, 2017], [Oliveira et al., 2018], [Kinsy et al., 2017] or, less robust encryption modules using XOR logic gates ([Ancajas et al., 2014], [Sepúlveda et al., 2017]). This approach provides data confidentiality but still expose the traffic to DoS and timing SCA attacks. Firewall and TNP (Temporal Network Partition) try mitigate this issues.

The use of firewalls ([He et al., 2012], [Rajesh et al., 2015], [Grammatikakis et al., 2015b]) ensure access control to the communication system, avoiding DoS attacks and minimizing the possibility of data extraction by a malicious process. Wassel et al. [Wassel et al., 2014] and Wang et al. [Wang and Suh, 2012] use TNP to provide temporal and logical traffic isolation avoiding the interference on secure flows, enabling communication availability and timing SCA attacks protection.

Table 2.2: State-of-the-art summary.

Proposal	Protection Comput.	Protection Comm.	Method	Prevent - Provide	Design Time	Runtime
Sepúlveda (2018) [Sepúlveda et al., 2018]	No	Yes	PUF and MAC	Access control; Data integrity and authenticity	–	Key regeneration MAC verification
Rajesh (2015) [Rajesh et al., 2015]	No	Yes	Firewall	Access control; Confidentiality	–	Traffic monitoring and rules activation
Hu (2015) [Hu et al., 2015]	No	Yes	Firewall	Access control; Confidentiality	Application mapping; Firewall positioning and rules configuration	–
Sepúlveda (2015) [Sepúlveda et al., 2015d]	No	Yes	Routing Scheme Random arbitration	Timming SCA ; and DoS prevention	–	arbitration obfuscation traffic monitoring and routing change
Fernandes (2016) [Fernandes et al., 2016]	No	Yes	Secure Zone: table-routing algorithm (RBR/SBR)	Timing SCA and DoS attacks mitigation;	Runs RBR and SBR algorithms to mapping tasks, calculate paths and router tables	–
Wassel (2014) [Wassel et al., 2014]	No	Yes	Temporal Network Partition	Timing SCA and DoS attacks mitigation;	Task mapping and configuration of static schedule arbitration	–
Ancajas (2014) [Ancajas et al., 2014]	No	Yes	Encryption, Authentication Obsfucation	Hardware trojan attack; Confidentiality and authentication	–	Encryption, table-based authentication, taks migration
Boraten (2016) [Boraten and Kodi, 2016]	No	Yes	Packet validation	Hardware trojan attack; Data integrity	–	Change operation mode; generate and verify coded packets
Silva (2017) [Silva and Zeferino, 2017]	No	Yes	Encryption	Data Confidentiality; Authentication	–	Key distribution; Cipher and decipher messages
Kinsy (2017) [Kinsy et al., 2017]	No	Yes	Firewall; Encryption	Memory access control; Data leakage and integrity; DoS	Secure wards arrangement	Key distribution; Cipher and decipher messages
Oliveira (2018) [Oliveira et al., 2018]	No	Yes	Firewall; Encryption	Data Confidentiality	–	Key distribution; Cipher and decipher messages
Real (2018) [Real et al., 2018]	Yes	No	Secure Zone: spatial and temporal isolation	Data integrity; Confidentiality inside the Cluster	Application Priority levels	Application mapping; *SZ* creation;
ARM (2008) [ARM, 2018]	Yes	No	Secure Zone: spatial and logical isolation	Access control; Data integrity;	Application development using ATZ API	Switch to TTE mode and police execution
Isakovic (2013) [Isakovic and Wasicek, 2013]	Yes	Yes	Secure Zone: Spatial isolation and encryption	Access control; Authentication; Data integrity	Secure Kernel (keys and secure channel management) and secure mechanisms (protocols, algorithms) provided	Secure channels creation; key exchange; Firewall rules execution
Sepúlveda (2017) [Sepúlveda et al., 2017]	Yes	Yes	Discontinuous Secure Zone: cryptography and firewalls	Access control; Authentication; Confidentiality	–	Key exchange; cipher and decipher messages
Reinbrecht (2017) [Reinbrecht et al., 2017]	No	Yes	Routing Scheme	Timing SCA;	–	Find baseline communication level; Communication monitoring; Change routing police
Grammatikakis (2015) [Grammatikakis et al., 2015a]	No	Yes	Firewall	DDoS; Data leakage;	–	Firewall rules configuration and execution
Our Proposal	**Yes**	**Yes**	**ECDH; Opaque Secure Zones; Rerouting; Symmetric Encryption**	**DoS; Spoofing; Data integrity; Authentication; Timming attacks; MitM**	**Cluster size**	**Entities authentication; Application admission; Opaque Secure Zones; Rerouting; Peripheral data encryption**

The works offering protection to computation ([Real et al., 2016b], [ARM, 2018]) or protecting computation and communication simultaneously ([Isakovic and Wasicek, 2013], [Sepúlveda et al., 2017]) adopts temporal, logical or spacial isolation as main mechanism.

The isolation enables the creation of secure zones. According to review, secure zones are defined at design time ([Fernandes et al., 2016]) or at runtime ([Isakovic and Wasicek, 2013], [Sepúlveda et al., 2017]). The techniques to create secure zones include encryption ([Sepúlveda et al., 2017]), the routing algorithm ([Fernandes et al., 2016]), firewalls ([Kinsy et al., 2017]) or, spatial isolation ([Real et al., 2018]). The proposed mechanisms found in the literature implement secure zones providing logical ([Sepúlveda et al., 2017], [Sharma et al., 2018]) or spatial isolation ([Fernandes et al., 2016], [Real et al., 2018]), al-

though remain sharing the communication resources, remaining exposes to attacks such as, DoS, timing SCA, HT or data eavesdropping, depending on the mechanism used.

In the reviewed works, just Isakovic et al. [Isakovic and Wasicek, 2013] discuss the needs of an explicit partitioning in the prevent resources. According to the Authors the application level don't need to implement mitigating resources directly. These mechanisms must be implemented at hardware and microkernel level to be used by the application level.

The protection of the application communication with peripherals and shared memories avoids unauthorized access to instructions and data, which may also compromise the resources availability, due to DoS attacks, cache-based SCA attacks or tampering. Mechanisms to mitigate these threats employ techniques such as firewalls [Grammatikakis et al., 2015b], routing scheme [Reinbrecht et al., 2017] and MAC [Sepúlveda et al., 2018].

Our proposal applies different defense mechanisms to protect both computation and communication encompassing all phases of application lifetime. At the application admission, a mutual authentication mechanism between the *MCSoC* and a MAC mechanism guarantee peripherals authenticity and application integrity. During the application execution, the defense mechanism is the spatial reservation of computation and communication resources, resulting in an Opaque Secure Zone (OSZ). The key feature enabling the runtime creation of OSZs is a rerouting mechanism responsible for deviating any traffic traversing an OSZ. The last mechanism is the access to peripherals using a secure protocol to open access points in the OSZ border, and lightweight encryption mechanisms.

Table 2.3 presents the overhead of distinct proposals (first column) related to the area (second column), power consumption (third column) and latency (fourth column). The fifth column shows another overhead parameter eventually presented by the schemes.

Even though all proposed mechanisms use *MCSoC*, according to [Bjerregaard and Mahadevan, 2006] a comparative cost analysis is hard to make. This is the case for the performance evaluation, area and power consumption overhead analysis due to the distinct platforms and the comparison happening against a baseline architecture of one's own work. Also, the baseline system adopted for each work and the workload varies. As an example, [Sepúlveda et al., 2017] uses encryption mechanisms to protect the communication and a dedicated service NoC to key exchange and firewall rules configuration, presenting area overhead of 20%, i.e. less than [Reinbrecht et al., 2017], which use a router logic to generate and detect gossip messages, with 21% of area overhead. These works do not detail the baseline system, making the comparison difficult and imprecise.

The area overhead is between 9.9% and 233% of the correspondent baseline implementation. In general, methods that protect communication using AES modules present higher overhead, as Silva et al. [Silva and Zeferino, 2017] (233%) and Oliveira et al. (193%). An initial expectation is that adopting encryption methods would imply in highest area and power costs. However, the reviewed proposed mechanisms ([Ancajas et al., 2014], [Sepúlveda et al., 2017]), use only simple XOR-based methods to encrypt the communication. While

Table 2.3: Cost overhead of the proposals.

Proposal	Overhead			
	area	power	latency	other
Sepúlveda (2018) [Sepúlveda et al., 2018]	152%	–	–	performance: 1% to 9%
Ancajas (2014) [Ancajas et al., 2014]	9.9%	5.8%	5.8%	–
Rajesh (2015) [Rajesh et al., 2015]	12.7%	9.8%	5.4%	–
Sepúlveda (2015) [Sepúlveda et al., 2015a]	11.0%	9.0%	–	–
Fernandes (2016) [Fernandes et al., 2016]	–	–	–	routing tables: 15.56%
Silva (2017) [Silva and Zeferino, 2017]	233.0%	–	700% to 1760%	–
Kinsky (2017) [Kinsy et al., 2017]	17.0%	–	9.0%	–
Real (2018) [Real et al., 2018]	–	–	–	execution-time: 35.8%
Sepúlveda (2017) [Sepúlveda et al., 2017]	20.0%	12.7%	30% to 55%	–
Oliveira (2018) [Oliveira et al., 2018]	193%	–	126.3% to 395.9%	–
Grammatikakis (2015) [Grammatikakis et al., 2015b]	–	-30%	-20.5%	–
Reinbrecht (2017) [Reinbrecht et al., 2017]	21.0%	16.2%	–	–

such strategies imply good results regarding area and power consumption, the papers do not discuss how secure or how strong is this encryption technique.

The power consumption is strongly related to the workload and the method used to mitigate the threats. The evaluation in Grammatikakis et al. [Grammatikakis et al., 2015a] show a power consumption reduction of 30% when compared to the system under attack. With the proposed method, several packets are dropped by the firewall in the NI, saving power resources. However, the impact of the firewall with normal operation is not evaluated. Other works that evaluate the impact of the proposed methods in power consumption shows an increase between 5.8% with the packet certification presented in [Ancajas et al., 2014] and 16.2% with the gossip method proposed in [Reinbrecht et al., 2017].

As presented in Chapter 1, we argue that the security concerns to run a sensitive application must deal with the application admission, the computational and communications aspects of their execution and the memory and I/O access. The state-of-the-art review shows that the proposed works do not present a systemic solution that includes all these aspects.

Most works consider only one of these aspects, limited to the application execution (computation or communication protection) and the memory access. The concern about memory access is considered from the communication point of view. The application admission is neglected presenting just one work protecting the memory integrity.

This Thesis stands-out from related works because it covers all phases required to execute an application with security constraints using runtime mechanisms to tackle these issues: application admission, execution and peripheral access. The protocol authenticates trusty entities and creates an entity key. The authenticated entities may deploy applications on the *MCSoC*, with an attached MAC (Message Authentication Code), which ensures at the same time integrity and authentication. Reserving and isolating computation and communication resources inside a region of the *MCSoC* guarantees the secure execution. During the application execution the protection to the peripherals access is obtained with lightweight encryption mechanism and packet integrity checking. After the application execution, the resources are released and the memory is erased to avoid information leakage.

3. BASELINE PLATFORM

This chapter introduces the *MCSoC* baseline platform used in this work, based on the Hermes MultiProcessor System (HeMPS) [Carara et al., 2009][Woszezenki, 2007]. The baseline platform and this work are both developed at the *Hardware Design Support Group* (GAPH) research group [GAPH, 2018].

The presentation of the hardware and software models is required in order for the reader to understand the assumptions made in the next Chapters related to the methods to add security in *MCSoCs*. The first part of chapter presents the hardware model, with emphasis on the Control NoC. This Control NoC, developed in cooperation with members of the research group ([Wachter et al., 2017]), corresponds to the first Thesis contribution, and it is a key module to make feasible the methods presented later. This Control NoC enables the implementation of the *OSZ*s because it is the hardware allowing to find the paths circumventing them. The second part of this chapter details the software model, with emphasis on the operating systems and how applications are described.

3.1 Hardware Model

Figure 3.1.a overviews the hardware components of the *MCSoC* used in this work. The system contains two regions: the GPPC (General Purpose Processing Cores), and peripherals. The GPPC contains a set of identical PEs that execute general purpose applications. Peripheral are specialized cores, which provide I/O interface and hardware acceleration for tasks running on GPPC. Examples of peripherals include accelerators for image processing, communication protocols (e.g., Ethernet, USB), and Application Injectors (*AppInj*).

The baseline architecture uses local scratchpad memory, without caches neither shared memories. The goal of using this simple memory model is twofold: (a) to reduce the power consumption related to cache controllers and NoC traffic (transfer of cache lines); (b) to reduce the complexity of the system model, enabling to simulate systems with tens of PEs. The architecture may support shared memories connected as peripherals, enabling access a largest data memory space.

Peripherals are connected to the boundaries of the GPPC [Ruaro et al., 2018] and, in an *MCSoCs*, it may occur at any location of the NoC, at external routers, or at unused ports of the mesh NoC (e.g., South ports of bottom routers). We adopted the last option, resulting in a regular floorplan for PEs, with peripherals distributed along the GPPC boundary.

The main hardware platform features are:

- NoC-based system: the platform contains two NoCs. A PS (Packet Switch) data NoC and a control NoC. Both adopts 2D-mesh topology. The data NoC uses duplicated

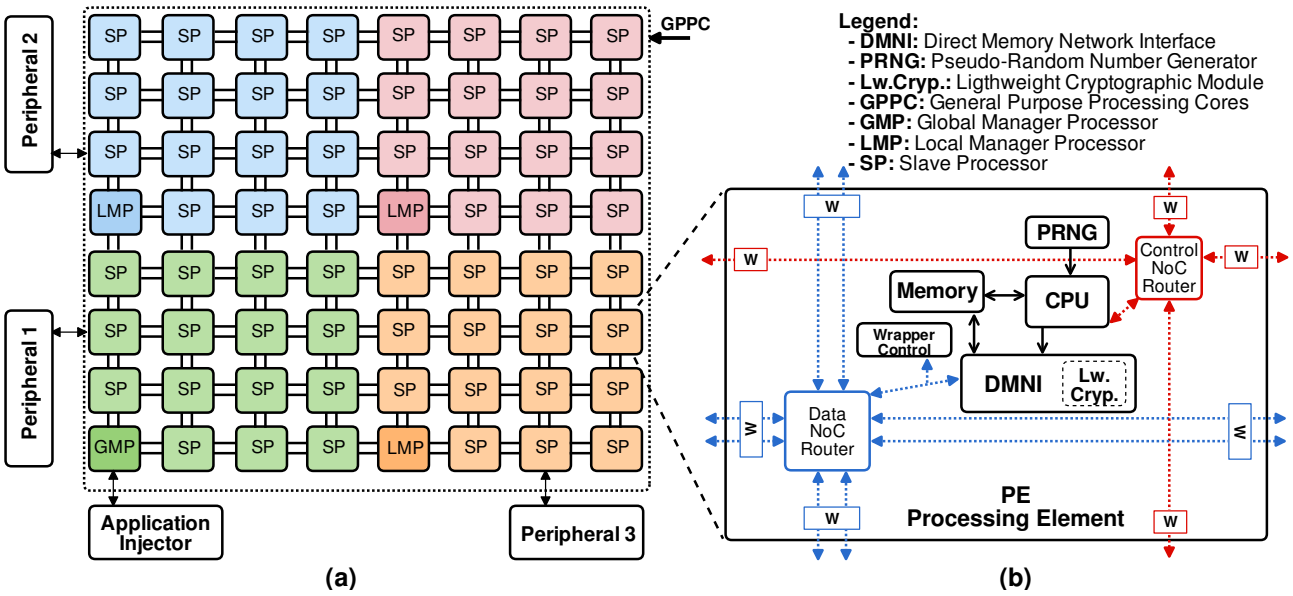


Figure 3.1: Overview of *MCSoC* hardware model.

physical channels, enabling the adoption of distinct routing algorithms per physical channel (detailed in Section 3.1.1). The control NoC uses broadcast as the default transmission mode (detailed in Section 3.1.2). The two NoCs are completely disjointing without any hardware or software dependence in their accesses.

- Homogeneous system: all PEs have the same hardware architecture (Figure 3.1.b) with a PS (Packet Switch) NoC router, a broadcast NoC router, a private memory, a MIPS-like processor, a DMNI (Direct Memory Network Interface) module, a PRNG (Pseudo-random Number Generator) module, a set of wrapper cells connected to each router link and a wrapper control module.

- Distributed memory: each PE has a true dual-port scratchpad memory for instructions and data, while message-passing performs the communication between PEs.

The CPU component adopted is the Plasma, which implements the MIPS ISA. The DMNI module is a network interface with DMA (Direct Memory Access) capabilities [Ruaro et al., 2016]. The PRNG module uses the Trivium [Canniere and Preneel, 2006], a stream cipher based PRNG to generate a 64-bit pseudo-random number (PRN). The local memory is a true dual-port scratchpad, storing code and instructions. If some application requires a larger memory space than the one available in the local memory, it is possible to connect shared memories to the system as a peripheral.

The system requires at least one peripheral, the *AppInj*. This peripheral is responsible for transmitting applications to be executed in the GPPC using the data NoC.

The control flow signals of all links contain wrapper cells or simply *wrappers*. The function of the wrapper cells is to isolate a given link. The granularity of the isolation is at the link level. For example, it is possible to block only the west link and continue to

transmit through the other links. The wrapper closes the control NoC and the data NoC links simultaneously, ensuring that a given message/packet will be discarded in both NoCs.

The activation of the wrapper cells occurs by the OS using a memory mapped register at each PE, enabling to isolate a given link. Each bit of the wrapper register enables/disables a given port wrapper of the PE. The wrapper acts over the control flow signals of each NoC port (8 ports for the data NoC and 4 ports for the control NoC). Thus, the wrappers' area overhead is negligible since its implementation requires a small number of gates, a register, and an FSM.

Sections 3.1.2 and 7 presents the behavior of the wrappers in the control NoC and data NoC respectively. The wrapper module is also described in Section 7.2.

3.1.1 Data NoC

The data NoC transfers *data messages*, exchanged by applications. The data NoC extends the NoC Hermes [Moraes et al., 2004] adopting duplicated physical channels, flit width equal to 16 bits, input buffering, round-robin arbitration, credit-based flow control, wormhole packet switching, simultaneous support for distributed XY routing and source routing (SR).

The use of duplicated physical channels ensures deadlock avoidance and full routing adaptivity. The number of virtual or replicated channels required to avoid deadlocks is a function of the network topology. For example, two virtual or replicated channels are sufficient to avoid deadlocks in a 2D-mesh topology [Linder and Harden, 1991]. The flit width is half of the original in the Hermes NoC to minimize the area overhead due to the duplicated physical channel adoption.

The standard routing mode between PEs is the distributed XY routing algorithm. The data NoC also supports SR such that it is possible to determine alternative paths to circumvent broken paths due to an *OSZ*. The mechanism to found an alternative path to use in the SR is presented in Section 3.1.2.

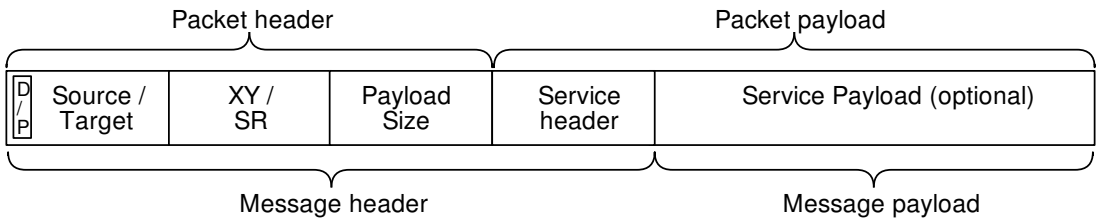


Figure 3.2: Packet and message structures - a flag (D/P) in the target address field differentiates *data* packets from *peripheral* packets.

The data NoC differentiates *data* packets from *peripheral* packets. Data packets are those exchanged by tasks running in PEs, and peripheral packets are those transferred

between a task and a peripheral. A *peripheral* packet arriving in a boundary PE goes to the peripheral, and not to the DMNI.

A *data* packet, from the NoC point of view, has a header and a payload (Figure 3.2). The packet header content controls the data NoC behavior, such as, routing, open and close internal switching, and arbitration. While in [Carara et al., 2009] the packet header have two fields (target and payload size), we adopt three fields to support the SR, the rerouting mechanism and the communication with peripherals: (*i*) the source/target address with data (D) or peripheral (P) packet flag; (*ii*) the XY or SR field that indicate the turns on each router when use SR or the source/target address when use XY routing (repeat the first field) and; (*iii*) the payload size.

From the task point of view, a message is used by the kernel with two fields: (*i*) the message header to control the data exchange between tasks or with peripherals trough data such as, producer task ID, consumer task ID, service (e.g., message delivery, request for a message, task mapping, task allocation), message timestamp and, (*ii*) the payload, an optional field, with the task or peripheral data. It may contain, for example, user data or the object code of a task.

3.1.2 Control NoC - BrNoC

The *BrNoC* [Wachter et al., 2017] is a dedicated NoC, decoupled from the data NoC. The BrNoC has the same topology of the data NoC, enabling to control each port individually (e.g., the North port in the dedicated NoC has an equivalent North port in the data NoC). The default transmission mode is the broadcast because it enables to reach PEs in case of disabled links, to notify several PEs with one message, and to transmit with low latency control messages.

In a broadcast, when a given port receives a message, it is processed and broadcasted to the neighbor routers (ports N, S, E, W), except to the port it came from. According to the transmission mode, the message may be transmitted to the port connected to the NI (local port). The broadcast acts as a wave traveling through the NoC. The BrNoC supports four transmission modes:

- ***brTgt*** (broadcast with a target): a specific PE is the target of this message. The message is broadcasted to all routers, but only the PE with the target address consumes it. This mode may be used to find a new path after a message discard; notify a specific PE to execute some action. The broadcast ensures that the message will be delivered even if a link/router is faulty or disable.

- ***brAll*** (broadcast to all PEs): all PEs consume the message. Therefore, all PEs are interrupted, and the message type defines the action the PE should execute. This

mode may be used to freeze the tasks of a given application; send commands to PEs of same application ID; set wrappers to define a secure zone.

- ***brWt*** (broadcast without a target): all *BrNoC* routers consume the message, without notifying the NIs. This mode executes actions related to the *BrNoC* management, as clearing specific data structures.

- ***unicast***: this message is an answer to a ***brTgt*** message. The ***unicast*** message follows the path defined by the ***brTgt*** message, in the reverse order to reach the source PE (backtrack process). This mode may be used to return a new path. Due to the limited payload size, each *BrNoC* router in the path sends a unicast message to the source router so that the fault-free path can be completely received.

Figure 3.3 presents the internal architecture of a *BrNoC* router, for a 2D-mesh topology. The router contains two control FSMs (Finite State Machines), two round-robin arbiters and a centralized CAM (Content Addressable Memory) memory. In addition, routers have a small area footprint since they do not have input buffers (the CAM acts as a buffer shared by all input ports, storing all flits received for all ports), and each flit encapsulates a single message.

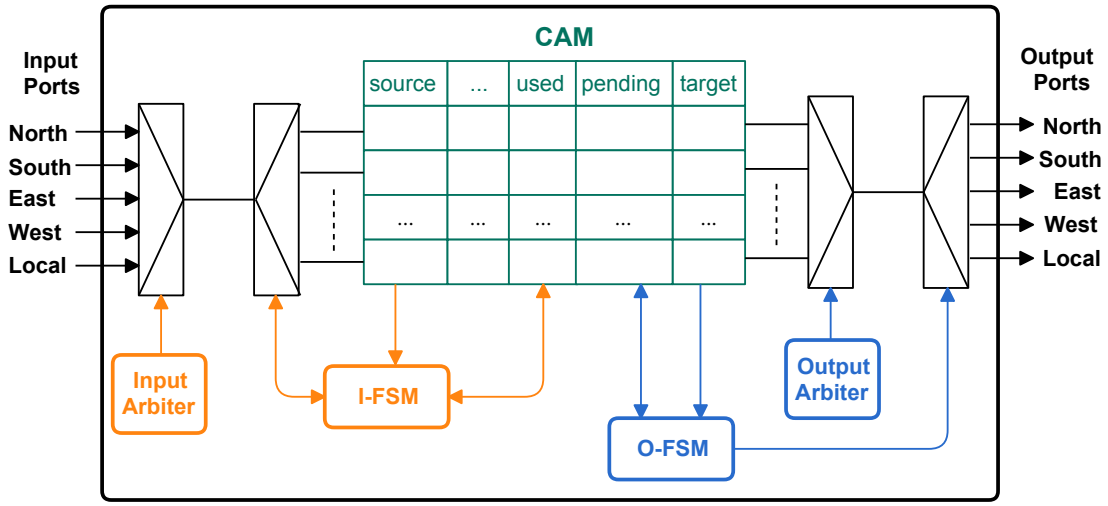


Figure 3.3: *BrNoC* architecture.

The wrappers are connected to the control flow signals (*req*, *ack* in the control NoC - Figure 3.4). The control flow signals traverses the wrapper if it is disabled. Considering the activation of the wrapper the *int_in_req* signal (internal value of the input request) is masked to 0 even if the *in_req* value is 1 in the external side of the PE. The value 1 in the wrapper value also set the *out_req* regardless of the *int_out_ack* value. This actions disable the message request from the neighbor PE and set the *out_ack* value to release it. From the neighbor PE the message was delivery. Equivalent behavior occurs when the request is generate internally (*int_out_req* signal).

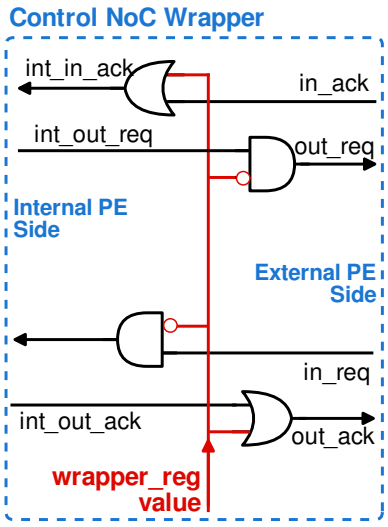


Figure 3.4: Control NoC Wrapper logic

Figure 3.5 details the flit structure (37 bits) and one CAM row (51 bits). Each CAM row stores the flit contents (to enable the broadcast) and control fields. The *flit* structure contains the fields: *message ID* (identification); *source address*; *target address*; message *type* (defines the action to execute and the transmission mode); message *payload*. The *tag* to search in the CAM is the tuple *{msg ID, source address}*. Each brNoC link contains the flit structure plus the *req*, *ack* and *nack* signals.

The CAM size definition (number of rows) occurs at design time, and it is not a function of the system size, ensuring scalability. Smaller CAMs can increases the delay in handling the messages, while larger CAMs reduces this delay at the cost of larger silicon area. The payload size may increase at design time to support services requiring larger data to transmit. The payload size is also a trade-off between the amount of data to transmit and the silicon area.

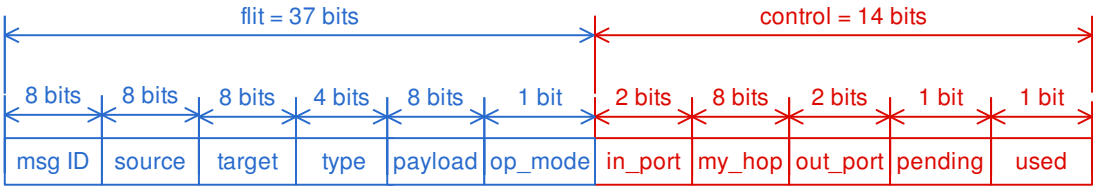


Figure 3.5: Message (*flit*) and one row of *BrNoC* CAM memory.

The control structure of one CAM row contains the fields: *op_mode*, *in_port*, *my_hop*, *out_port*, *pending* and *used*. The *pending* field signalizes the presence of a message to be handled. The *used* indicates that the row is in use. The *in_port* stores the port identification from where the message comes from. The ***unicast*** mode uses the fields *my_hop* and *out_port*.

The control NoC has two operation modes (*op_mode* field): *global* and *restrict*. The *global* operation mode enables the control messages to pass through the wrappers, even if they are enabled. This operation mode permits PEs inside a secure zone to exchange mes-

sages with manager PEs. The *restrict* operation mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded. This mode enables a path discovery mechanism by the control NoC.

The I-FSM receives incoming messages and if necessary stores the message in a CAM row. A handshake protocol (*req*, *ack*, *nack*) controls the I-FSM which is initially in an idle state, waiting for incoming messages (*req* asserted in a given port). The *input arbiter* chooses an input port to handle. Three conditions may assert the *ack* signal: (c_1) the *tag* is not in the CAM, and there is space in the CAM; (c_2) the *tag* is in the CAM; (c_3) failed or isolated port, where a wrapper force the *ack* signal. The assertion of the *nack* occurs when the *tag* is not in the CAM, and there is no space in the CAM. The router receiving the *nack* unsets the *req* and tries later (action discussed in the O-FSM). When condition (c_1) is satisfied the I-FSM execute the following actions:

- stores the message in a free position of the CAM;
- asserts the *pending* field to signalize that the message should be broadcasted;
- asserts the *used* field to signalize that the CAM row contains a valid message;
- stores the port identification selected by the arbiter in the *in_port* field (the size of *in_port* and *out_port* fields are a function of the number of router ports);
- in a search for a source-target path, the payload contains the distance from the current router address to the source address. This value is incremented and stored in the *my_hop* field.

Condition (c_2) ensures that requests to already visited routers are discarded, avoiding cyclic transmissions (i.e., deadlocks), and the end of the broadcast when all routers were visited.

The O-FSM handles the messages stored in the CAM, using the same handshake protocol. The *output arbiter* chooses a row to handle, according to the asserted pending fields. All broadcast modes propagate the message to the neighbor routers, except the *in_port*. According to the broadcast mode, the message also goes to all local ports (***brAll***), or to the local port that matches the router address with the target field (***brTgt***). The *pending* field is cleared when all broadcasted ports answer with an *ack*. If some broadcasted port answer with a *nack* the arbiter selects another CAM row, enabling the selection of the current row again. An example of message type using ***brWt*** propagation is the CLEAR, responsible for freeing a CAM row, by clearing the *used* field. The ***unicast*** message uses the *in_port*, *my_hop* and *out_port* fields to answer a ***brTgt*** message. The ***unicast*** message forwards the message to the port defined in the *in_port* field.

Figure 3.6 presents an example of the procedure to find a new path using the control NoC. In this scenario, Router 1 communicates with Router 15 (XY path), but an *OSZ*

interrupts the communication. Using the control NoC, Router 7 starts a BROKEN_PATH message to Router 1 (not showed in Figure 3.6).

Table 3.1: Examples of Messages Types and Purposes of the *brNoC*.

Purpose	Message Type	Transmission Mode	Operation Mode
Path Discovery	BROKEN_PATH	***brTgt***	Global
	SEARCH_PATH	***brTgt***	Restrict
	BACKTRACK	***unicast***	Restrict
Security	SET_SECURE_ZONE	***brAll***	Global
	END_SECURE_ZONE	***brAll***	Global
System Management	START_APP	***brAll***	Global
	END_TASK	***brTgt***	Global

The BROKEN_PATH is defined at the type field (Figure 3.5). Table 3.1 shows examples of messages types with the respective transmission mode, operation mode and purpose.

When the message reaches the Router 1, it starts a SEARCH_PATH message to find a new path to Router 15 (Figure 3.6.a, red arrows).

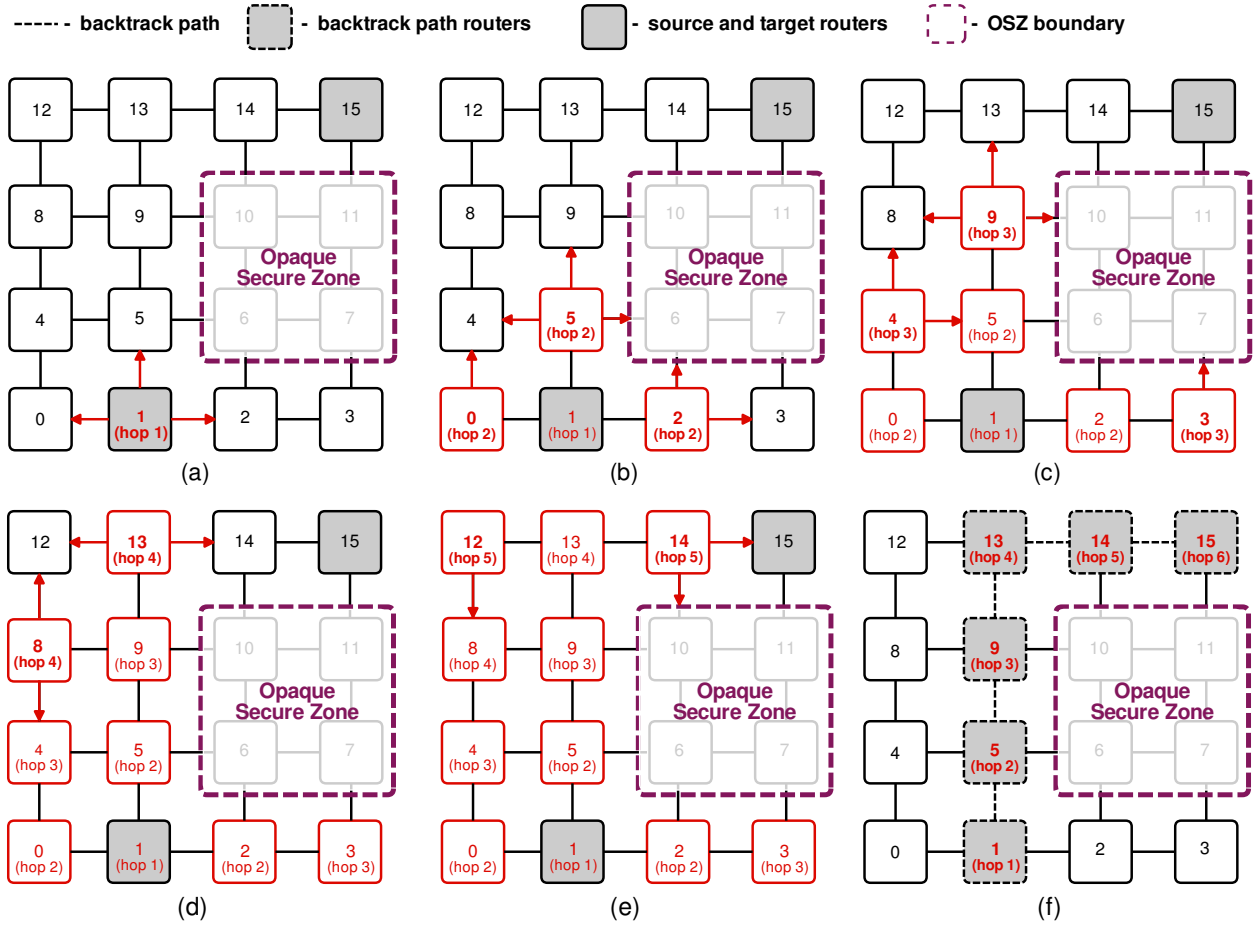


Figure 3.6: Example of path discovery using the *BrNoC*.

Next, Routers 0, 2 and 5 receive the message through ports East, West and South, respectively. Then, these routers broadcast the received message to their neighbors (Figure 3.6.b). As Router 6 has the wrapper activated in ports South and East and the SEARCH_PATH message uses the restrict mode the message is ignored in Router 6 (the input *req* signal is masked and the output *ack* signal is force to high - Figure 3.4).

In Figure 3.6.c, Routers 3, 4 and 9 receive the message from ports East, South, and South, respectively, and broadcast to their neighbors. Note that the message sent by the Router 4 is discarded in Router 5 because this router already have received the message from the same source (*msg ID*/*source address* stored in CAM). Router 7 ignore the message because the wrapper is active in the port South and East and Router 10 because the wrapper is active in the ports West and North.

Next, Figure 3.6.d, the message is received in ports East and South of Routers 8 and 13, respectively, and sent to their neighbors. The Router 4 discard the message because it has received the message previously. Router 10 ignores the message from Router 9 because the wrapper is active in ports West and North.

In the next hop (Figure 3.6.e) the message is received in Routers 12 and 14. Router 12 send to Router 8 that ignores the message because has received it previously. Router 10 ignores the message from Router 14 because the wrapper is active in ports West and North. Finally, in Figure 3.6(f) the SEARCH_PATH message reaches the target, starting the answer step, with BACKTRACK messages.

In the answer, each router in the path sends a BACKTRACK message to the source router. Initially, Router 15 sends a BACKTRACK message to Router 1 through the West port (information stored in the *in_port* field). Next, Router 14 propagates the first message, and then transmits a new BACKTRACK message to Router 1, with the payload having the contents of the *out_port* field. Each router in path repeats this process, propagating the previous BACKTRACK messages and sending a new one. The *my_hop* field controls the process, finishing when the source router receives all BACKTRACK messages (*my_hop*=1). Therefore the source router receives a number of BACKTRACK messages equal to the number of hops in the path to the target. Each one of these messages contains the port to reach the destination router in the payload. For the example the source PE (Router 1) receive the following *out_port* values form the BACKTRACK messages: [W E E N N N].

When Router 1 receives a BACKTRACK message, the PE is interrupted to compute a hop of the source routing path to Router 15. After receiving all BACKTRACK messages, the Router 1 compute the source routing path and resends the lost message. All subsequent packets to this destination use the source routing path, which is stored in an OS structure. The process to find a new path to a given target is executed once, only when the fault is detected.

3.2 Software Model

Scalability at the hardware level comes from PEs executing several tasks in parallel, using the NoC to transmit multiple flows concurrently. However, large systems require high-level management for controlling the deployment of new applications, monitoring resources usage, manage task mapping and migration, and can execute self-adaptive actions according to systems constraints. Thus, to achieve a scalable design, HeMPS adopts cluster-based decentralized management [Castilhos et al., 2013]. Clusters are virtual regions in the GPPC, with a set of slave processors (S_{PE}) and one manager PE (M_{PE}). S_{PE}s execute applications' tasks, while M_{PE}s manage the clusters.

The management occurs at the M_{PE} and S_{PE} levels, executed by the kernel running in those PEs, as depicted in Figure 3.7.

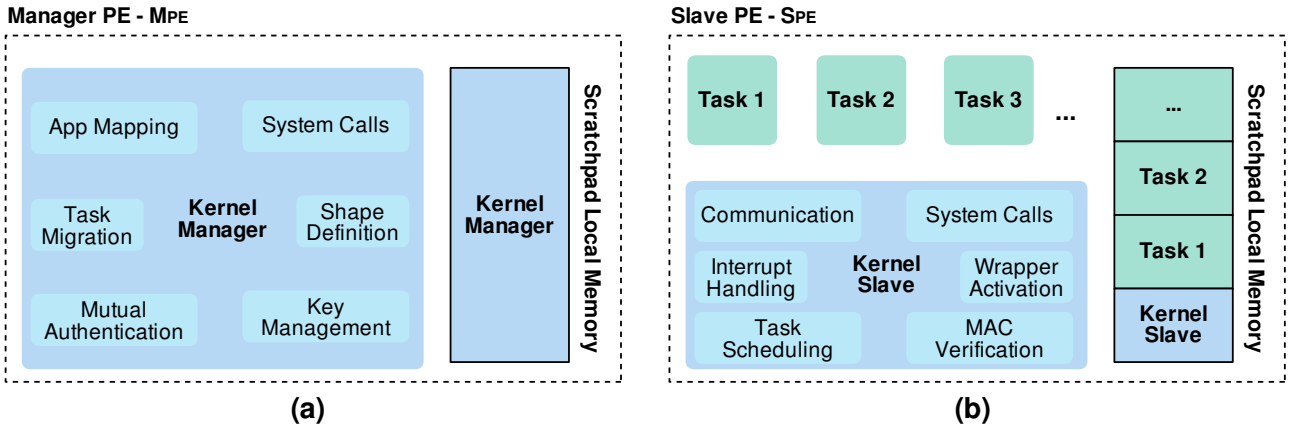


Figure 3.7: Overview of the kernels: (a) M_{PE} kernel manages the system and do not execute users' tasks; (b) S_{PE} kernel manage users' tasks.

At the M_{PE} level, as presented in Figure 3.7.a, the local memory is reserved to the kernel, without executing user's tasks. The M_{PE} executes heuristics as task mapping, task migration, monitoring, authentication and key management.

At the S_{PE} level, as displayed in Figure 3.7.b, a multi-task kernel acts as an Operating System. The platform adopts a paged memory scheme to simplify the kernel design. Examples of actions executed by the kernel include task scheduling, inter-task communication (message passing), interrupt handling.

Both manager kernels are written in C language. Only a small part of the code is written in assembly language, responsible for executing context saving and handling hardware and software interruptions.

Applications are also written in C language. They are modeled as task graphs $A = < T, P, D, S >$, where $T = \{t_1, t_2, ..., t_m\}$ is the set of application tasks corresponding to the graph vertices; $P = \{p_1, p_2, ..., p_n\}$ is the set of peripherals corresponding to the graph vertices. The D set represents the application descriptor which contains the communicating

pairs $\{(t_i, t_j), (t_i, p_r), (t_j, p_s), ..., (t_m, p_n)\}$ with $(t_i, t_j, ..., t_m) \in$ T, $(p_1, p_2, ..., p_n) \in$ P. A pair (t_i, t_j) denotes the communication from task t_i to task t_j ($t_i \rightarrow t_j$), and a pair (t_i, p_r) denotes the communication from task t_i to peripheral p_r ($t_i \rightarrow p_r$). The S value indicates if the applications execute in normal mode (value 0) or secure mode (value 1) and also is enhanced in the application descriptor. Figure 3.8 present an application m.

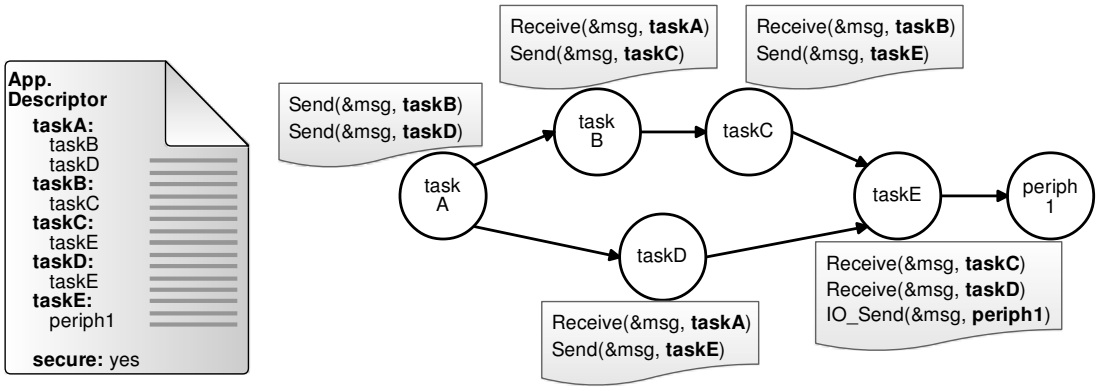


Figure 3.8: Application task graph example.

Tasks communicate using message passing (MPI-like) primitives. The API provides two primitives: a non-blocking *Send*() and blocking *Receive*(). The main advantage of this approach is that a message is only injected into the NoC if the receiver request data, reducing network congestion. To implement a non-blocking *Send*(), a dedicated memory space in the kernel, named *pipe* [Carara et al., 2009], stores each message written by tasks. The *pipe* is a communication channel where messages are stored and consumed in an ordered fashion. Within this work, the pipe is a memory area of the kernel reserved for message exchanging, where messages are stored in an ordered fashion and consumed according it. Each pipe slot contains information about the target/source processor, task identification and the order in which it is produced.

At the lower level, the kernel communicates with the data NoC with *data_request* and *data_delivery* packets. The *pipe* and a message buffer enables packet retransmission to inter-task communication and inter-manager communication respectively.

The support for I/O communication uses a second API, with *IO_Receive*() and *IO_Send*() primitives, using a master/slave communication model. The PE is the communication master and the peripherals the communication slaves. At the lower level, the kernel communicates with the data NoC with *IO_request*, *IO_delivery*, and *IO_ack* packets. The *IO_Receive*() primitive uses the *IO_request* at the PE side and the *IO_delivery* at the peripheral side. The *IO_Send*() primitive uses *IO_delivery* at the PE side and the *IO_ack* at the peripheral side.

4. GENERAL VIEW OF SECURITY MECHANISMS

This chapter presents in Section 4.1 a general view of the security mechanisms proposed to each phase of the application lifetime. This is required in order to provide a holistic perspective of this Thesis approach before detailing the methods on the next chapters. Section 4.2 presents a condensed view and an explanation of the protocol phases used to mitigate the security issues during the applications' lifetime. Section 4.3 presents a classification of the available techniques to create secure zones as found in the literature, using an orthogonal criteria. Section 4.4 presents the requirements of the *MCSoC* environment to implement the proposed methods in this Thesis, and the restrictions imposed by this environment.

4.1 Defense Mechanisms

As discussed in Chapter 1 the execution of an application with security constraints comprises at least three premises: (*i*) application admission; (*ii*) application execution; (*iii*) access to peripherals.

The application admission presents two security issues: (a) the authenticity of each actor (external entity and *MCSoC*), i.e., the guarantee that the other part is whom it says to be; and (b) the integrity of the application, to avoid the tampering of the object code with threats such as backdoors, trojan horses or spoof behavior.

In this work, the application admission uses a mutual authentication mechanism based on ECDH (Elliptic Curve Diffie-Hellman) to guarantee the authenticity of the *AppInj* that wants to run an application into the *MCSoC*, and a MAC (Message Authentication Code) to ensure the integrity of applications during the object code transfer.

Elliptic Curve Diffie-Hellman (ECDH) is an Elliptic Curve variant of the standard Diffie-Hellman protocol based on the algebraic structure of elliptic curves over finite fields. ECDH is a key agreement protocol that allows two actors, each having an elliptic curve public–private key pair, to establish a common secret key over an insecure channel [Hankerson et al., 2003].

The Elliptic Curve Cryptosystem (ECC) security is based on the discrete logarithm problem, i.e. the problem to solve the discrete logarithm in reasonable computational time. In a simplified view, let E be an elliptic curve and A, B be its points such that $B = n.A = \sum_{i=1}^{n} A$ – for some n. The computational effort to compute B from A when n and A are know is low. The task to find an n when A and B are know, is called the discrete logarithm problem for elliptic curves. No efficient algorithm to compute discrete logarithm problem for elliptic curves is known. Detailed discussion about ECC can be obtained in [Koblitz, 1987], [Schneier, 1996], [Hankerson et al., 2003], [Certicom Research, 2009].

Elliptic Curve Cryptosystems present the highest strength per bit of the public-key cryptosystems. In other words, the number of bits needed to obtain the same security is smaller in ECCs. ECC not only uses smaller keys for equivalent strength compared to traditional public-key cryptosystems like RSA but also the key size disparity grows as security needs increase. The advantages that can be obtained with smaller keys include speed (faster computations), processing power, storage, bandwidth, and circuit area, relevant parameters when resources are constrained [Gupta et al., 2002].

The aim of a MAC is to prevent an adversary from modifying a message sent by one actor to another, without the actors detecting that a modification has been made in the message [Katz and Lindell, 2014]. This task is only possible if the communicating actors have some common secret unknown by the adversary.

The difference between a hash function and a MAC is that the former provide just data integrity, while the latter guarantees data integrity and authentication. Briefly, a MAC code is a way of combining a common secret key with a message so that the receiver can authenticate that the sender of the message has the common secret key and no-one who does not know the secret key could have sent or changed the message.

According to [Mouha et al., 2014] some of the most commonly used algorithms are CMAC, HMAC, and UMAC. CMAC is based on a block cipher, usually AES or Triple-DES, whereas HMAC uses a hash function such as SHA-2, SHA-3, and UMAC is based on a universal hash function combined with a standard cryptographic primitive such as a block cipher or a hash function.

This Thesis uses the SipHash algorithm to generate a MAC of each application's task object code due to the good performance when compared to non-cryptografic hashes, according to the algorithm's authors ([Aumasson and Bernstein, 2012]) and the C reference implementation being available using MIT public licence. SipHash is a cryptographic MAC consisting of modular additions, rotations, and xors (ARX) and uses a 128-bit key, producing a 64-bit output and has an internal state of 256 bits size [Aumasson and Bernstein, 2012]. The process is split into three stages: initialization, compression, and finalization. SipHash is parameterizable using c SipRounds in the compression and d SipRounds in the finalization stages. A specific instantiation of SipHash is called SipHash-c-d. Dobraunig et al. [Dobraunig et al., 2018] presents a differential cryptoanalysis of SipHash-2-4 and conclude that it is indistinguishable from a pseudo-random function.

To support the use of PRN, a hardware module implements a PRNG based on Trivium. Trivium generates a 64 bit PRN from an 80-bit seed and an 80-bit Initialization Vector (IV), and has been selected as part of the portfolio for low area hardware ciphers (Profile 2) by the eSTREAM project [Canniere and Preneel, 2006]. It is not patented and has been specified as an International Standard under ISO/IEC 29192-3 [ISO/IEC , 2012]. The access to PRN uses a mapped register enabling the CPU to read its value directly.

The second premise regards the application execution in a runtime environment protected from attacks. This is obtained with a runtime mechanism that creates Opaque Secure Zones (*OSZ*) dynamically into the *MCSoC*, enabling temporal and spatial isolation of applications, preventing the communication and computation resource sharing.

The execution mechanism includes: *(i) OSZ* shape definition and positioning; *(ii)* wrapper activation; *(iii)* retransmission of lost packets in and out the *OSZ* boundaries; and *(iv)* launch application. Figure 4.1 illustrates a simplified view of the approach. In Figure 4.1(a), the *MCSoC* contains one application in execution, App_1. Next, the manager processor maps an App_2, activating the wrappers at the boundary of the *OSZ*. At this moment (Figure4.1(b)), the App_1 traffic is blocked by the *OSZ*. Figure 4.1(c) shows the App_2 executing in the *OSZ*, and the App_1 traffic circumventing the region.

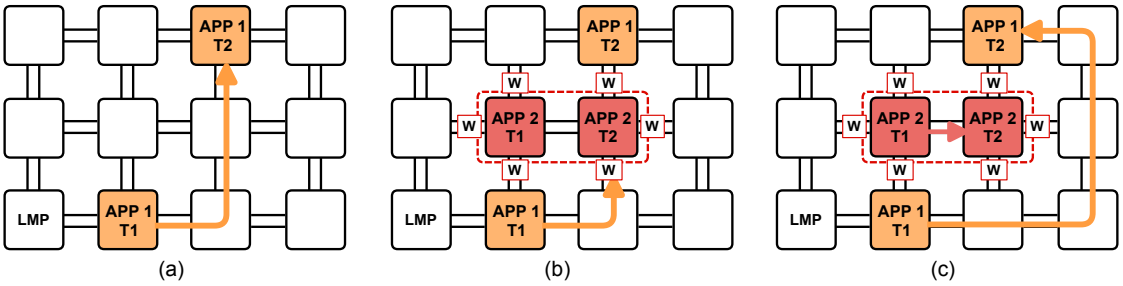


Figure 4.1: Secure zone and dynamic reconfiguration of routing paths.

The third premise is related to the protection of communication with peripherals and shared memories. As presented in Section 3.1.1 the data NoC distinguish the packets with a flag in the header field (first flit of the packet). This feature enables to block all data packets arriving at the boundary of the *OSZ* (in both directions) and to apply selective control of I/O packets. Besides the control of the I/O flow at the boundary of the *OSZ* the protection of data traffic is obtained with lightweight cryptography applied to the message payload and a MAC based on SipHash algorithm to protect the service header of the message.

The MAC on the service header inhibits an adversary to forge the message source or tampering the control information present in the service header such as task ID, timestamp, peripheral ID. The lightweight cryptography prevents data leakage during the communication with peripherals.

The protection of the payload has both software and hardware implementations. The software-based implementation uses PRESENT [Bogdanov et al., 2007] and AES [NIST, 2001] encryption mechanisms. The hardware-based implementation (the *lw_crypt* module in Figure 3.1) uses the SIMON [Beaulieu et al., 2015] and AES encryption modules.

None of the software or hardware cryptographic module is mandatory in the Thesis. The focus is to provide a secure environment to the application execution using a set of algorithms with strong security properties and low cost relative to area overhead and latency in the application execution. Thus, the modular implementation enables to replace the cryptographic modules according to distinct security requirements.

Table 4.1 summarizes the defense mechanisms, positioning them according to the Thesis objectives (Section 1.4), and where they are detailed in the subsequent chapters.

Table 4.1: Defense mechanisms positioning with objectives and Thesis chapters.

Defense Mechanism	ECDH	MAC on task's object code	OSZ	Lightweight Cryptography
Objectives (Section 1.4)	I - External entities authentication	II - Application admission and III - Key exchange	IV - Close the *OSZ*	V - Communication with Peripherals
Thesis chapter	Chapter 5 Section 5.1	Chapter 5 Section 5.2	Chapter 6	Chapter 7

4.2 General view of the protocol phases

As presented in Section 1.4 the objective of this Thesis is to propose a set of protocols, executed at runtime, to enable applications with security concerns to run in *MCSoCs* tackling the following issues: application admission, secure execution, and access to peripheral devices. Figure 4.2 summarizes the protocol phases. The names of the phases are presented on the left side of Figure 4.2, in blue color.

At the Setup phase each actor is responsible for generating a key pair {Pk, PuK} suitable for elliptic curve cryptography. Then, the public key and the ID pair {PuK, ID} of each external entity (*AppInj* and peripherals) is loaded at the *MCSoC* . Each external entity also load the {PuK, ID} pair of the *MCSoC* . The Authentication phase uses the {PuK, ID} pair during the mutual authentication phase.

The result of each of these phases, setup and authentication, are: (*i*) the external entities (peripherals and *AppInj*) authenticated in the *MCSoC* ; (*ii*) a common session key (K_e) shared between the peripherals and the *MCSoC*.

The Admission phase has three goals: (*i*) define the Opaque Secure Zone (*OSZ*) shape and location; (*ii*) ask to the *AppInj* the tasks' codes; and (*iii*) transmit to the SPs that will receive the task's codes the K_e of the *AppInj* and peripherals used by application to, respectively, verify the MAC and encrypt the messages (**1** to **5** in Figure 4.2).

The *AppInj* uses K_e to compute the MAC using the SipHash algorithm [Aumasson and Bernstein, 2012] during the allocation phase of the protocol. After receiving the destination of App_{sec}'s tasks, the *AppInj* uses K_e to generate a MAC to each task's object code. Thus, the allocation sends the task's object code with the respective MAC appended – **6**. The SP receives the task's object code and, using the same K_e, compute the MAC locally. If the computed MAC match with the received MAC the object code integrity is verified.

If the previous phase succeeded for all tasks, the application might execute – **7**. The Close *OSZ* phase proceeds by activating the wrappers surrounding the *OSZ* ("W" in Figure 3.1) and start the execution of the application – **8**. The wrappers discard all messages that should traverse the *OSZ*. The control NoC transmits to the source of the discarded

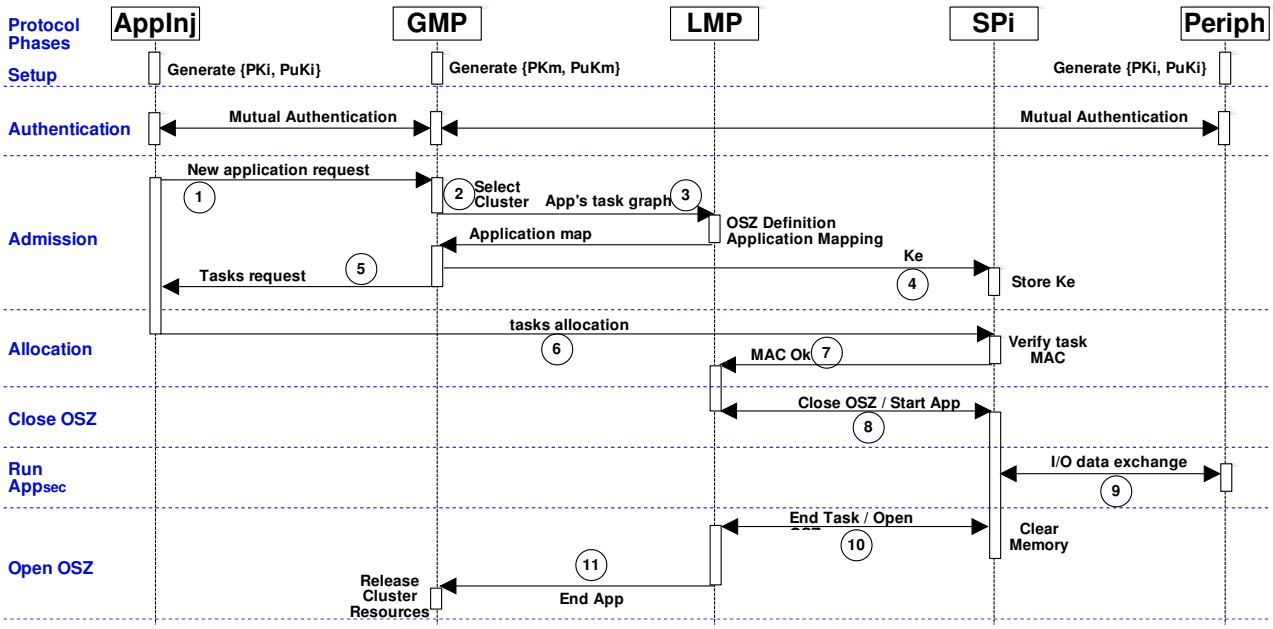


Figure 4.2: Protocol general view.

messages a retransmission request. The non-secure applications use the control NoC to find an alternative path to circumvent the *OSZ* and retransmit the non-delivered messages using a rerouting mechanism.

The I/O access occurs during the application execution, and the concerns are (*i*) to enable the communication without compromise the security, (*ii*) to protect the service header of message against tampering and, (*iii*) to protect the message payload against eavesdropping – **9**.

The Open *OSZ* phase cleans the memory contents of the SPs inside it to prevent any information leakage to be used by an attacker – **10**. Also, the SP's Operating System erase the value of K_e key and release the wrapper opening the *OSZ*. Finally, the LMP clears the internal structures to release the cluster resources previously allocated to the application and sends a message to the GMP that release its internal structures about the cluster resources – **11**.

4.3 Secure Zones

Resource sharing is a native feature of *MCSoCs*. Different applications may execute in the same processor, share the NoC links, as well as shared memories. This feature is the source of issues related to security. Secure Zones (*SZ*) is an approach adopted to reduce resource sharing.

Methods employed at design time enable the adoption of sophisticated and robust algorithms to provide solutions to the security problem since they do not have limitations related to the heuristics' computation time. However, design time methods are not applicable

in dynamic workload scenarios. Thus, these methods are limited to scenarios where the workload is known beforehand, without any change during the system lifetime.

The literature presents several methods to create *SZ*s. It is possible to classify such methods using a set of orthogonal criteria:

- **Creation time:** the definition of the *SZ* occurs at design time [Fernandes et al., 2016] or runtime [Real et al., 2018][Sepúlveda et al., 2017].
- **Shape:** the *SZ* may be discontinuous [Sepúlveda et al., 2017] [Fernandes et al., 2016][Sharma et al., 2018] or continuous, with a rectangular[Isakovic and Wasicek, 2013] or rectilinear shape [Caimi et al., 2018b].
- **Communication sharing:** the *SZ* may allow flows belonging to sensitive applications to share NoC links [Sepúlveda et al., 2017][Fernandes et al., 2016][Sharma et al., 2018] or the flow inside the *SZ* is forbidden to other applications.
- **Computation sharing:** the *SZ* may allow that tasks belonging to sensitive applications share the same processor [Sepúlveda et al., 2017] or applies resource reservation to sensitive application [Real et al., 2018][Fernandes et al., 2016][Sharma et al., 2018].
- **Methods:** the methods used by the *SZ*s include cryptography [Sepúlveda et al., 2017][Sharma et al., 2018], routing algorithms[Fernandes et al., 2016], spatial and temporal isolation[Real et al., 2018], rerouting[Caimi et al., 2017a].

Figure 4.3 presents examples of *SZ*s according to the previous classification. Discontinuous *SZ*s (*SZ*2) require more efforts to prevent attacks (encryption or routing schemes) due to the flows exposure, while continuous *SZ*s can imply internal fragmentation when using a rectangular shape, due to the reservation of resources without effective use (*SZ*1). A rectilinear shape (*SZ*4) prevents internal fragmentation but needs dedicated routing mechanisms to avoid flows crossing the boundary of the region.

The use of continuous *SZ* (*SZ*1 and *SZ*4) still exposes the communication to attackers because flows belonging to other applications can transverse the *SZ* allowing DoS, HT and timing attacks.

According to the previous classification, Opaque Secure Zones (*OSZs*) are created at runtime, have a rectilinear shape, without computation and communication resource sharing. The SPs of the *OSZ* are reserved to execute a single secure application (*SZ*3, in Figure 4.3). The only resource sharing exception is the communication with I/O devices.

The creation of an *OSZ* must avoid unreachable SPs. For example, in Figure 4.3 considering all *SZ* as opaques, the top right SPs are unable to communicate with manager PEs. Thus, processors' selection should consider not only the availability of resources but also prevent unreachable SPs. This is achieved by adding a padding around *OSZ*s to

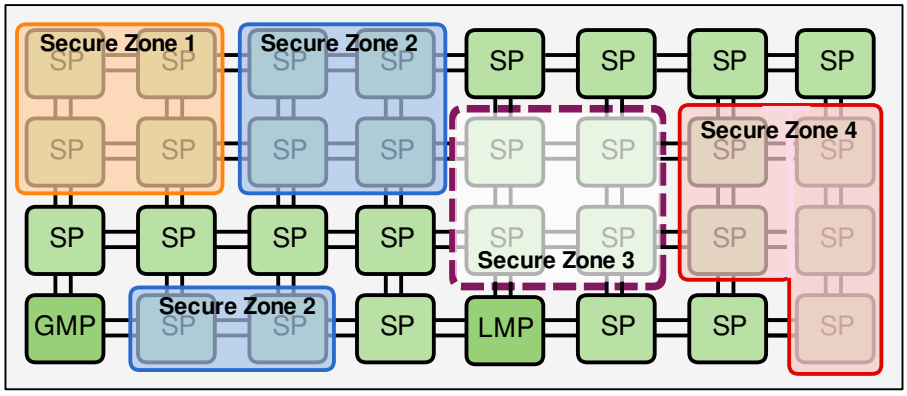


Figure 4.3: *SZ*1: continuous and rectangular, *SZ*2: discontinuous, *SZ*3: cont., rect., and opaque, *SZ*4: cont. and rectilinear.

prevent unreachable SPs. Also considering the access restriction, when a peripheral is connected to the *MCSoC* border, none *OSZ* must be placed using the same router were the peripheral is connected.

The method that enables *OSZ* is the dynamic rerouting mechanism. The rerouting mechanism ensures that the App_{sec}'s traffic stays inside the *OSZ*, and deviates all traffic that should cross the *OSZ*.

The *OSZ* mechanism corresponds to the second contribution of this Thesis since no other work adopts the communication resource reservation, enabling the use exclusive of the NoC links by the App_{sec}.

4.4 Requirements and restrictions

4.4.1 Requirements

This section presents the requirements to implement the security mechanisms proposed in this Thesis. They are mandatory resources that must be available in others *MCSoCs* that wants to implement the methods herein presented.

- **Hierarchical architecture**: as presented in Chapter 3 the system has a hierarchical architecture organized in clusters, with two types of PEs: (*i*) manager PEs, responsible by cluster control and management; (*ii*) PEs executing applications' tasks - slave processors (SP). There are two distinct manager PE, global manager (GMP) and local managers (LMP). The GMP and LMP assume management functions with respect the secure application execution, as the communication with external entities (*AppInj* and peripherals) to authenticate each other, key management and, compute, find and create the *OSZ*.

- **Task migration support**: The manager PE is in charge to find a continuous region with free SPs to execute the App_{sec}. In the absence of a continuous region, the system must support task migration to release SPs in such a way to create the region.
- **Wrappers surrounding PEs**: the wrapper are the key mechanism to the creation of *OSZ* and the isolation of the *MCSoC* communication resources. It's implementation use simple AND, NOT, and OR logic gates attached to the flow controls signals of the NoCs, in addiction to a set of mapped registers to activate the wrappers.
- **Secure mechanism to exchange control messages with *OSZ*s**: With the *OSZ* communication isolation, a secure mechanisms to control message exchange between LMP and SPs must be provided. The control NoC (*brNoC*) achieves this issue providing an independent and not shared mechanism to exchange control messages beetwen manager PEs and SPs. *BrNoC* is disjoint from the data NoC and is inaccessible at application level (only the OS has access to the *brNoC*).
- **Rerouting and retransmission support**: with the wrapper activation and the creation of the *OSZ* all traffic crossing the *OSZ* is discarded requiring the retransmission and the rerouting support. It requires a dedicated NoC (as *brNoC*) to search paths to circumvent the *OSZ*s, and a data NoC with support to source routing. At the software level, the OS of each SP should be able to resend packets that hit the border of an *OSZ* and were discarded.
- **PRNG support**: the use of PRNG is necessary in different moments of the proposed solutions, such as during the ECDH protocol and the key distribution to SPs inside the *OSZ*. This way, a software or hardware PRNG solution is required in the system to support the secure features operations. PRNG is not mandatory, alternative mechanism like PUF (Physical Unclonable Functions) are used to generate random numbers [O'donnell et al., 2004][Sepúlveda et al., 2018].
- **Peripherals support using Master/Slave communication model**: the model based on master/slave communication prevents unauthorized data traffic inside the *MCSoC*, since that the application always starts the communication.
- **Method to inject applications into the MCSoC**: the *MCSoC* must have at least one NoC port available to connect an application injector responsible by the applications' deploy.

4.4.2 Restrictions

The baseline architecture adopted imposes architectural restrictions:

- the communication model is message exchange, with applications modeled as application task graphs (Figure 3.8);
- the memory model is Distributed Private Memory (DPM), without local caches and shared memories;
- the OS is preloaded in the SP's private memories, i.e., there is no bootloader to read the OS from an external memory.

In this context, it is out of the scope of this Thesis Centralized Shared Memory (CSM) or Distributed Shared Memory (DSM) architectures [Girão et al., 2011], as well as the communication model with threads.

Since that the OS is preloaded in the private memories, boot attacks [Jacob et al., 2017][Rouget et al., 2017] are also out of the scope of this Thesis.

Work also assumes that hardware components of the platforms are secure. This leads to two restrictions. The first one, attacks originated from Hardware Trojans are out-of-the-scope of the Thesis. Second, attacks that disturb the hardware from side channels attacks (as forcing values in some links or wires) are also not considered in the Thesis.

5. THE MUTUAL AUTHENTICATION, ADMISSION AND ALLOCATION PHASES

As presented in the state-of-the-art, the *MCSoC* literature disregards the secure application admission. While robust solutions exist for distributed systems and the internet, the requirement of the *MCSoCs* such as memory limitation, and reduced latency to start the applications inhibits the adoption of these solutions on the *MCSoCs*.

This chapter details the protocol steps to guarantee the entire secure application deploy. Three main phases are detailed: (*i*) the mutual authentication; (*ii*) the application admission and; (*iii*) the application's tasks allocation. The mutual authentication and key agreement phases (Section 5.1) runs just once to each peripheral. At the end of this steps, a common secret key is shared between the GMP and the peripheral. Section 5.2 presents the secure admission, detailing the algorithm to define the *OSZ* shape and positioning, the common secret key distribution mechanism, and the application's task allocation protocol. Each new application execution requested by the *AppInj* runs the admission, allocation and key exchange phases of the protocol. Section 5.3 evaluates the mechanisms and protocols of the secure application admission. Finally, section 5.4 presents the final remarks of the chapter.

5.1 Mutual Authentication for MCSoCs

This section presents the authentication scheme for many-cores. The proposed scheme consists of two phases: System Setup phase and Mutual Authentication phase. The system setup runs offline and is responsible by communication parts register (peripherals and many-core). The mutual authentication runs once to each external entity and is responsible for guaranteeing that the other part is whom it says to be.

5.1.1 System Setup phase

At the Setup phase, both External Entity (E_i) and the *MCSoC* (M) initially compute a key pair suitable for elliptic curve cryptography, step **1** in Figure 5.1. The key pair consists {Pk, PuK} of a private key (Pk) and a public key (PuK), represented by a random multiplier number and a point over an Elliptic Curve, respectively. Next, each one publishes their respective ID and PuK (step **2**). This phase ends with each part loading the pair {PuK, ID} of each other (step **3**), using a secure channel.

In this context, the External Entity could be any peripheral connected to the boundaries of the GPPC (General Purpose Processing Cores) through the data NoC, such as the *AppInj*, accelerators and communication modules.

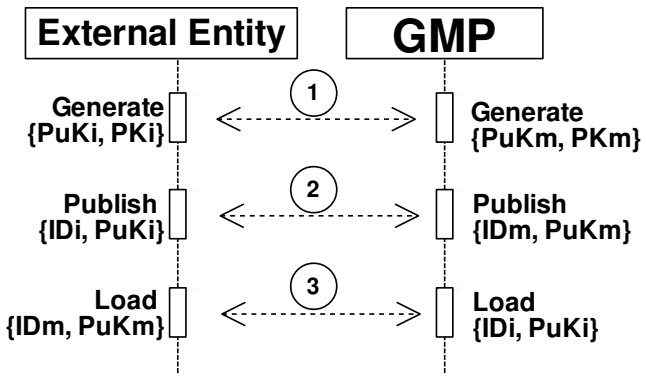


Figure 5.1: Sequence diagram of Setup phase of protocol.

A secure channel to publish and load the public keys may avoid man-in-the-middle and hijacking attacks. This process can be done using mechanisms like https, TLS/SSL or VPNs and, is out of the scope of the Thesis. However, the protocol deployed in the next section ensures that even if the public keys use unsecured channels, the Mutual Authentication phase guarantees that only the legitimate actors involved in the process will correctly exchange data.

5.1.2 Mutual Authentication phase

As shown in Figure 3.1, external entities may deploy secure applications to execute in the *MCSoC*. This phase provides mutual authentication between an external entity (the *AppInj*, or other peripheral), and the *MCSoC*.

If an entity wants to send an App_{sec} to run in the *MCSoC*, it must first execute the mutual authentication, before sending the application code.

Figure 5.2 shows the mutual authentication protocol. The arrows denote messages through the data NoC, the parenthesis content is the data message, and the external parenthesis index is the key used to encrypt the message.

Initially, the GMP sends a request authentication message to External Entity (step **1** in Figure 5.2). The External Entity returns a message encrypted by the *MCSoC* public key (PuK_m) loaded at the Setup phase (**2**). The request message contains the entity identifier and a random number, i.e., the pair $\{ID_i, nonce_i\}$. Note that any entity, even in an attempt to insert a malicious application, may send this message because the *MCSoC* has a published public key.

The GMP decrypts the request message using its private key (PK_m), verifying the received ID_i. If ID_i is valid, the *MCSoC* sends a reply message encrypted by the entity public key (PuK_i) – step **3**. This message contains the tuple $\{ID_m, nonce_i, nonce_m\}$, where ID_m is the *MCSoC* ID, and $nonce_m$ is a random number generated by the *MCSoC*. Otherwise, if the entity ID_i is not found, the reply message is encrypted using the *MCSoC* public key

(PuK_m) since there is no public key associated with the received ID_i. The reply message is always sent to avoid information leakage.

At the entity side, before decrypting the message, if the received $nonce_i$ does not match, this means that the connection with the *MCSoC* is compromised because other actor tried to forge the *MCSoC* ID. A correct received $nonce_i$ corresponds to a correct authentication of the *MCSoC*, once just the lawful pair {PuK_m, PK_m} can encrypt/decrypt the initial request message.

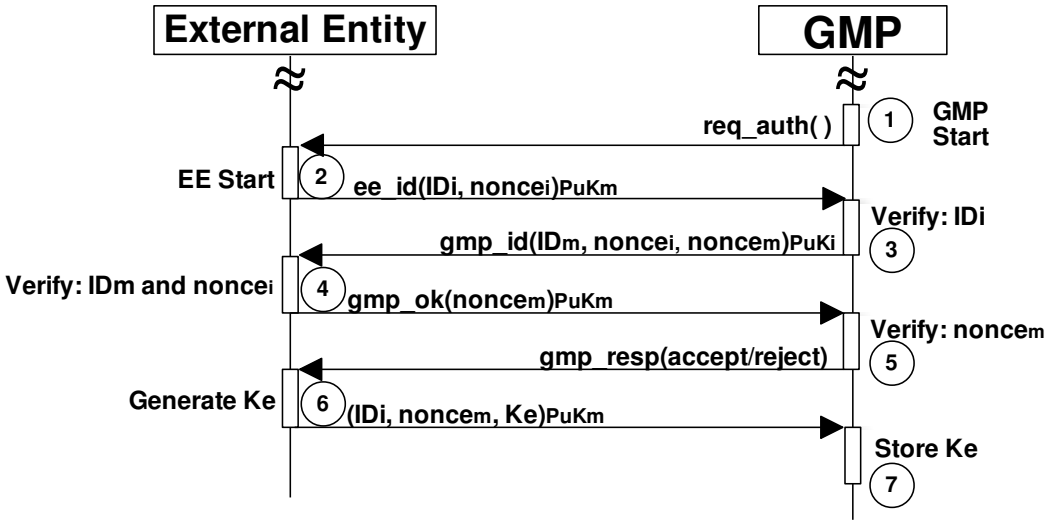


Figure 5.2: Sequence diagram of Mutual Authentication phase of protocol.

After verifying the correct reception of $nonce_i$, the entity seeks the PuK_m according to the received ID_m. The entity encrypts the received $nonce_m$ using PuK_m, and send this data to the *MCSoC* – step 4.

At the *MCSoC* side, the received $nonce_m$ is decrypted using PK_m. A correct $nonce_m$ authenticates the entity, once just the lawful pair {PuK_i, PK_i} can encrypt/decrypt the message with the $nonce_m$. This way, if the $nonce_m$ and the ID_i (received on first request message) are correct, the *MCSoC* sends an accept message to the entity. Otherwise, a reject message is sent – step 5.

Once finished the mutual authentication steps the entity generates a *session* key K_e, used during the Task Allocation phase. The entity uses this key to compute a Message Authentication Code (MAC) appended at the end of each task object code. The *MCSoC* uses this key to verify the integrity of each task deployed into the system.

The entity sends the tuple (ID_i, $nonce_m$, K_e) encrypted by the *MCSoC* public key (PuK_m), where ID_i and $nonce_m$ are used to authenticate the message – step 6.

The GMP keeps the pair {ID_i, K_e} to use it in future secure applications deployed by the entity, saving resource consumption and decreasing the latency to start the secure application – step 7.

5.2 Application admission and Task Allocation phases

As previously presented the application admission and task allocation have the following goals: (*i*) define the Opaque Secure Zone (*OSZ*) shape and location; (*ii*) transmit to the SPs that will receive the task's codes the K_e of the *AppInj* and peripherals used by application, and (*ii*) receive the task's codes from *AppInj* and verifies the integrity trough the MAC.

5.2.1 Application Admission

The Application Admission phase determines the location of the *OSZ* and maps the App_{sec}'s tasks. Figure 5.3 presents the application admission steps. The arrows indicate the messages and their contents using the data NoC.

The *AppInj* request to run an App_{sec} sending a `Run_app` message to the GMP. The `Run_app` message contains the application descriptor (number of tasks, inter-task dependencies, secure mode) (1). The GMP selects the cluster with enough resources to execute App_{sec} (2), sending to the LMP of the selected cluster the App_{sec} descriptor and the internal ID attached to the application by GMP(3).

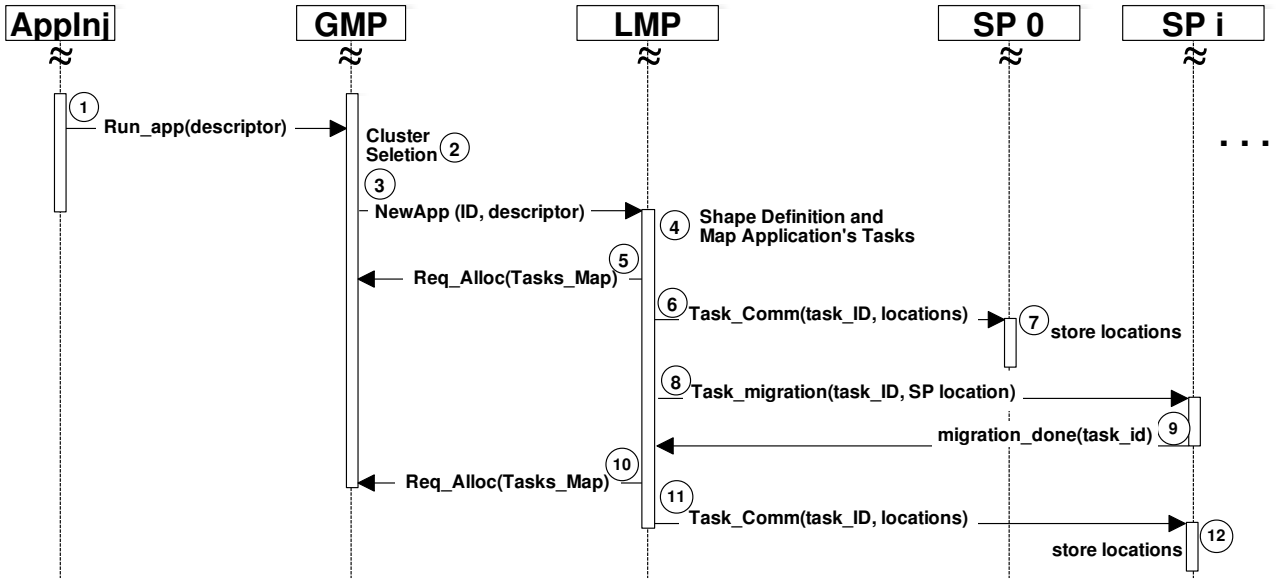


Figure 5.3: Application Admission phase protocol.

With the application descriptor, the LMP runs an algorithm to determine: (*i*) the number of SPs needs to run the application; (*ii*) the possibles *OSZ* shapes; (*iii*) the *OSZ* positioning inside the cluster; (*iv*) the number of task migrations needed to ensure exclusive SPs execution. The algorithm is a specific contribution of this Thesis and next subsection details it.

After defining the *OSZ*, LMP maps the tasks inside this region, considering as cost function the communication cost between the tasks (**4**) [Mandelli et al., 2011]. Note that mapping means definition of tuples *{task_ID, SP location }*, not the object code transfer.

Next, four actions from LMP occurs: (*i*) request to the GMP the tasks' object code (**5**) for the SPs that does not require task migration; (*ii*) transmission of the addresses of the communicating pairs to SPs that will receive the mapped tasks (steps **6** and **7** in Figure 5.3); (*iii*) the task migration request (**8**) to the SPs inside the selected *OSZ*, according to algorithm 5.1, that have other application's task running; (*iv*) after receiving the message with task migration finished (**9**), the LMP request the allocation to GMP (**10**) and sends the communicating pairs to SP (**11**). The task migration process occurs in parallel with request allocation and communication pairs actions, enabling the Task Allocation phase start to free SPs inside the *OSZ* while the task migration runs to release SPs of the *OSZ*.

LMP manages the task migration process since it knows the resources available and the task positioning in the cluster. The task migration runs at O.S level of the *MCSoC* and disposes a specific protocol to exchange the task' object code, data and heap memory area, enabling the runtime migration of the tasks. The task migration process is a auxiliary feature that allow creation of *OSZ* without resources sharing. A detailed explanation about the task migration mechanism can be obtained in [Ruaro and Moraes, 2017].

Beside the task migration the *MCSoC* supports reclustering, allowing a cluster to borrow an SP to run a task of an application running in another cluster. During the reclustering process, an LMP request a processor to others LMPs using the control NoC (broadcast mode) and then execute a task migration to obtain a free SP in their cluster.

Creating OSZ at Runtime

Algorithm 5.1 presents the pseudo-code to create at runtime an *OSZ*. The algorithm inputs are the number of App_{sec}'s tasks (`app.#tasks`); the number of tasks PEs may execute simultaneously (`#tasks.PE`); the cluster side size (`cluster_side`); and the manager processor position (`MP.pos`).

The calculated shape is a rectilinear polygon, enclosing a set of SPs, without any task executing inside it, avoiding computation and communication sharing. The *OSZ* shape, occasionally, results in an internal fragmentation, i.e. some SP belonging to the *OSZ* don't have any task to run (shape area minus the number of PEs to execute App_{sec}). The algorithm removes SPs from rectangular regions to avoid internal fragmentation, leading to rectilinear polygons.

The algorithm have the *MAX_MIGRATION* design time parameter, defining the maximum number of task migrations. Once a shape is defined, it may contain tasks belonging to other applications. The algorithms migrate the tasks running in the selected region to guarantee the exclusive use of the processors by the App_{sec}. Due to the cost of the task

migration, the algorithm limits the number of task migrations using the *MAX_MIGRATION* design time parameter.

The first step of the algorithm, lines 1–5, is the definition of the *shape_set*. The loop computes the number of SPs to execute App_{sec} according to `app.#tasks` and `#tasks.SP` (line 3). The function *shapes* returns a set of rectangular shapes. For example, consider *app.#tasks* = 7 and *#tasks.SP* = 2. The first iteration (t = 1) requires 7 SPs, returning shapes `{(3,3),(2,4),(4,2)}`, with a fragmentation equal to 2, 1 and 1 for each shape. The second iteration (t = 2) requires 4 SPs, returning shapes `{(2,2),(1,4),(4,1)}`. For these shapes, there is no fragmentation.

If the *shape_set* is empty (line 6–7), the algorithm returns *FALSE*, meaning that the cluster does not have a shape to execute App_{sec}. This happens when all calculated shapes exceed the cluster side, e.g., in a 3x3 cluster (`cluster_side` = 3) and *app.#tasks* = 10 the minimum shape side is 4, leading to a *FALSE* value return from the algorithm.

If the cluster receives App_{sec} (*else* block), the algorithm sorts the *shape_set* in ascending order, using the shape size as criteria. The rationale is to minimize the CPU sharing (one task per PE), maximizing the App_{sec} performance.

After the sorting step, the algorithm starts the search process. This process requires three loops: (*i*) outer loop (line 10), controls the number of tasks migrations; (*ii*) intermediate loop (line 14) traverses the shape set; (*iii*) internal loop (line 15) traverses the PE set of the cluster.

The search process adopts a Sliding Window Search (SWS) procedure (line 16), using as inputs the PE coordinates (*PE.xy*), the shape size (Δ.*xy*), the manager position (*MP.pos*), and the cluster side size (*cluster_side*). The conditions to obtain an appropriate shape location are: (*i*) the shape fits in the cluster; (*ii*) the shape does not overlap the manager processor; (*iii*) the shape does not overlap an active *OSZ* and have one row/column pad from adjacent active *OSZ*; (*iv*) the shapes boundaries does not touch any peripheral (share the same data NoC router). If the shape fits the conditions, the SWS procedure returns the number of processors executing tasks inside the region, otherwise -1, in the variable *usedPEs*.

The LMP needs access the PEs of the defined *OSZ* to send the communicating tasks location (step **6** on Figure 5.3) and the entity key (Subsection 5.2.2). The strategy to avoid unreachable PEs (in data NoC) during the SWS is start the search in the opposite cluster corner to the location of the cluster LMP.

The first iteration of the outer loop disables task migrations (*mig#* = 0). Thus, after executing the SWS procedure, all PEs must be available, i.e., there is no task executing in the PEs inside the region. If one of the shapes fills this condition (line 17), the bottom left coordinate of the selected shape receives the current PE position (lines 18-19). For this first iteration of the outer loop, defragmentation occurs only if necessary (lines 20-24). If the

Algorithm 5.1: Search resources to create the OSZ

```
   Input: app.#tasks, #tasks.PE, cluster_side, MP.pos
   Output: sh // selected shape
 1 shape_set ← ∅
 2 for t from 1 to #tasks.PE do
 3 |   PEs_needed ← ceil(app.#tasks / t)
 4 |   shape_set ← shape_set ∪ shapes(PEs_needed, cluster_side)
 5 end
 6 if shape_set = ∅ then
 7 |   return FALSE
 8 else
 9 |   sort(shape_set, largest) // sort shape set according to the area, largest first
10 |   for mig# from 0 to MAX_MIGRATION do
11 |   |   if mig# = 1 then
12 |   |   |   sort(shape_set, smallest) // sort shape set, smallest first
13 |   |   end
14 |   |   forall sh[i] in shape_set do
15 |   |   |   forall PE in (cluster_side × cluster_side) do
   |   |   |   |   // SWS: Sliding Window Search
16 |   |   |   |   usedPEs ← SWS(PE.xy, sh[i].Δxy, MP.pos, cluster_side)
17 |   |   |   |   if usedPEs = mig# then
18 |   |   |   |   |   sh[i].x ← PE.x
19 |   |   |   |   |   sh[i].y ← PE.y
20 |   |   |   |   |   if sh[i].fragmentation > 0 then
21 |   |   |   |   |   |   for y from 1 to sh[i].fragmentation do
22 |   |   |   |   |   |   |   PE.pos(sh[i].x, sh[i].y).invalid ← 1
23 |   |   |   |   |   |   end
24 |   |   |   |   |   end
25 |   |   |   |   |   if usedPEs ≠ 0 then
26 |   |   |   |   |   |   foreach PE in sh[i] do
27 |   |   |   |   |   |   |   if PE.used ≠ 0 then
28 |   |   |   |   |   |   |   |   migration_list(PE.xy, sh[i])
29 |   |   |   |   |   |   |   end
30 |   |   |   |   |   |   end
31 |   |   |   |   |   end
32 |   |   |   |   |   return sh[i]
33 |   |   |   |   end
34 |   |   |   end
35 |   |   end
36 |   end
37 |   return FALSE
38 end
```

shape has internal fragmentation, the leftmost shape column has some of the PEs marked as invalid (line 22), modifying the shape from a rectangular to a rectilinear format. After this process, the selected shape returns to the caller function.

Subsequent iterations of the outer loop enable task migrations, from 1 to *MAX_MIGRATIONs*. With the goal of reducing the number of task migrations, the shape set is reordered, with the smallest shapes evaluated first. The previous process is repeated, exe-

cuting the SWS procedure. When the number of used PEs inside the cluster is equal to the allowed number of task migrations, the shape is selected. Besides defragmentation, tasks in used processors are put in a migration list to be migrated to outside the region (lines 25-31).

5.2.2 Entity Key Exchange

This phase of the protocol transmits K_e of the *AppInj* and peripherals to the SPs belonging to *OSZ*, encrypted with a new key, K_m (internal *MCSoCs* key). The procedure works to distribute the K_e shared with the *AppInj* or shared with any peripheral used by App_{sec}.

Each task receives a MAC to guarantee the integrity of the object code. The MAC is created using the SipHash algorithm [Aumasson and Bernstein, 2012] and K_e. Therefore, the SPs need K_e to compute the MAC attached to each task, and for security reasons K_e must remain in the GMP.

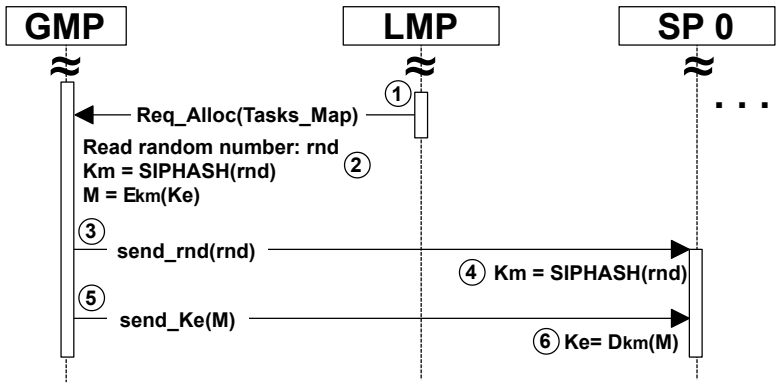


Figure 5.4: Entity key exchange.

Figure 5.4 presents the entity key exchange steps. LMP transmits the application's task map at the step **3** of the previous phase to GMP (`Req_Alloc` message, **1** in Figure 5.4).

In order to avoid broadcast transmission and prevent the reception of the random number by other SPs (**3**), GMP reads a random number from the PRNG module(**2**). GMP uses that number in the creation of the K_m key, transmitting it to each SP inside the *OSZ* using the data NoC. GMP and each SP that receive the random number create K_m using the SipHash algorithm and the transmitted random number (**4**).

GMP uses K_m to encrypt K_e, using an XOR operation between K_m and K_e. After encrypting K_e, GMP sends the encrypted value to each SP inside the *OSZ* using the data NoC (**5**). Then, each SP decrypt the received message using K_m (**6**), obtaining K_e, to be used in the next protocol phase.

5.2.3 Tasks Allocation and MAC verification

The goal of this phase of the protocol is to allocate the App_{sec}'s tasks in the SPs of the *OSZ*, guaranteeing their integrity.

This phase starts with GMP requesting to *AppInj* the object code of the App_{sec}'s tasks (1 in Figure 5.5). Each `req_task` message contains the mapping tuple $\{SP-location, taskID\}$, used by *AppInj* to send the task' object code to SP.

AppInj generates a MAC for each task, using the SipHash algorithm and K_e. Afterwards, the *AppInj* sends a `task_allocation` message with the task identification (*taskID*), the task' object code and the MAC (2). SP stores the object code in the memory and computes the MAC also using the SipHash algorithm and K_e (obtained in phase 5.2.2) (3). SP compares the received MAC with the computed MAC and sends the `task_allocated` message with the MAC comparison result to LPM (4) using the control NoC (red dashed arrows).

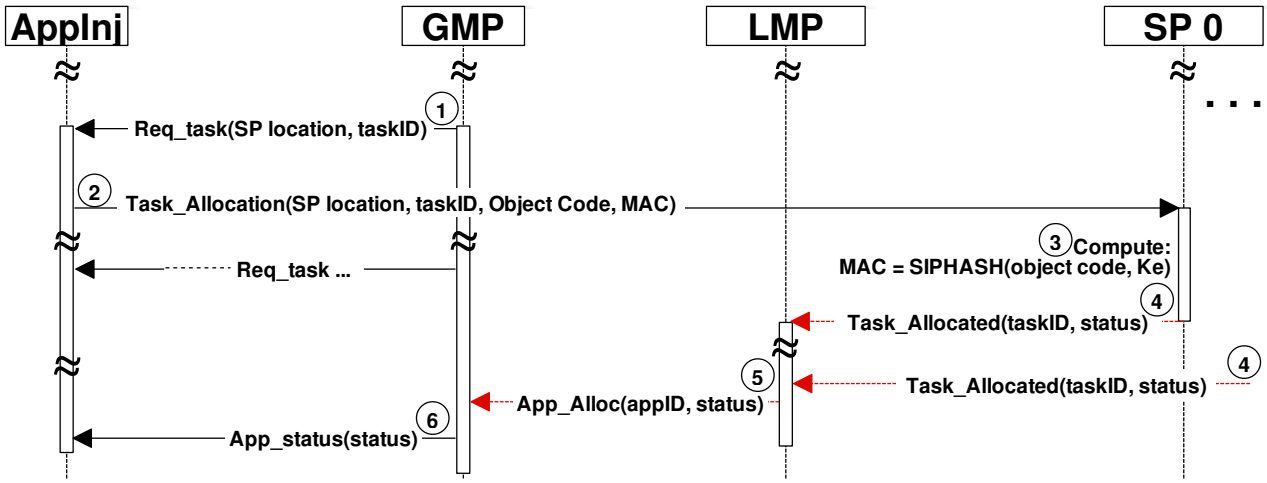


Figure 5.5: Tasks allocation and MAC verification phase. Red arrows: broadcast messages transmitted through the control NoC.

After mapping all tasks, LMP notifies the status of the allocation (5) to GMP. If all App_{sec}'s tasks were correctly received, App_{sec} might start its execution. Otherwise, the process is interrupted, and GMP notifies *AppInj* that App_{sec} was corrupted during its allocation (6).

5.3 Protocol phases evaluation

This section evaluates the protocol phases previously presented regarding the performance and the hardware (area) costs. To the performance evaluation, the steps responsible by the mutual authentication uses the OVPSim [OVP, 2018] description while an RTL SystemC description evaluate the others protocol phases. The hardware evaluation cost uses an RTL VHDL description.

5.3.1 Setup and Authentication phases evaluation

This section evaluates the first two phases of the protocol initially, corresponding to the system setup (Section 5.1.1) and mutual authentication (Section 5.1.2). Both phases do not impact directly in the latency to start a secure application but are required to run a secure application since the K_e common key defined at the end of the process is required when the application execution is required by *AppInj*.

The hardware architecture of the External Entity and GMP was described using OVPSim APIs [OVP, 2018], an accurate instruction simulator. The cryptographic functions of this steps use the TweetNaCl [Bernstein et al., 2015], a compact and self-contained public-domain C library. The library uses Curve25519 on the EDCH key exchange, the base of our proposed mutual authentication.

Table 5.1: Setup and Authentication phases evaluation.

Phase	Step	Actions	MIPS # Instructions (x10^6)	ARM # Instructions (x10^6)
Setup	1	generate Key pair	239.2	83.9
Authent.	1	start	0.001	0.001
	2	generate $nounce_i$ encrypt request	224.4	84.1
	3	decrypt request generate $nounce_m$ encrypt reply	449.0	168.3
	4	decrypt reply encrypt $nounce_m$	449.0	168.3
	5	decrypt $nounce_m$ encrypt A/R	449.0	168.3
	6	decrypt A/R generate Ke encrypt Ke	449.0	168.3
	7	decrypt Ke	224.5	84.1
TOTAL		**Setup + Authentication**	**2245.0**	**845.3**

Table 5.1 shows the two first phases of the protocol (1^{st} col.), the steps of the protocol (2^{nd} col.), the executed actions (3^{rd} col.), and the number of instructions of these actions in the MIPS-like OR1k (4^{th} col.) and ARM A9 (with SIMD) processors (5^{th} col.). The three main actions, generate a key pair, encrypt data and decrypt data, requires a similar number of instructions: 239.2M, 224.4M and 224.5M – MIPS processor, and 83.9M, 84.1M and 84.1M – ARM processor. The total amount of instructions of the two first phases of the protocol are 2,245.0M and 845.3M instructions, in processors MIPS and ARM, respectively. Considering the processors running at 500 MHz with a CPI=1, the time consumption relative to the mutual authentication of the External Entity and the *MCSoC* is 4.49s for MIPS and 1.68s for the ARM.

Although this overhead is relatively significant, in the order of seconds, only one execution of the mutual authentication process occurs for each entity to ensure the authen-

ticity of the parts and does not impact the latency to start the applications. The cost to compute the session key (K_e) corresponds to step 6 in Table 5.1, and it is smaller than half second. Thus, it is possible to consider to change this key periodically to increase the security of the system. Remember that the transmission of this key to the SPs of the selected *OSZ* occurs by using a lightweight cryptography method.

5.3.2 Admission to Allocation phases evaluation

This section evaluates the protocol phases that impact directly in the latency to start the App_{sec}, i. e. from Admission (5.2.1) to Allocation (5.2.3) phases. The evaluation was conducted in a clock-cycle RTL SystemC description of the *MCSoC* presented in Section 3.1. Operating system and applications are described in C language as presented in Section 3.2.

Admission phase evaluation

Four internal steps of the Admission phase impact on the delay to start the App_{sec}, (*i*) the cluster selection inside the GMP and the messages exchange of the protocol; (*ii*) the *OSZ* definition through the Algorithm 5.1 running in LMP; (*iii*) the application mapping by the LMP; and (*iv*) the task migration when the *OSZ* positioning needs release PEs to run the App_{sec}.

The cluster selection has a minor impact on the delay and it is a function of the number of the cluster in the *MCSoC*. The selection evaluates the free resources in each cluster to define where the App_{sec} will run, selecting the region with most resources available. Varying the number of clusters from 1 to 4, the selection take less than 500 clock cycles to define the cluster in all cases.

The Algorithm 5.1 defines the *OSZ* shape and positioning and its execution contribute to delay to start the App_{sec}. The Algorithm evaluation setup includes a 14x7 *MCSoC* instance (*cluster_side = 7*), a 9-task application (App_{sec}), up to 2 tasks may execute simultaneously per PE (*#tasks.PE = 2*).

With such configuration, the *shape_set* is equal to: $\{(3,3),(2,5)^*,(5,2)^*,(2,3)^*,(3,2)^*,(1,5),(5,1)\}$. Shapes marked with a '*' have internal fragmentation (equal to 1), requiring the removing of PEs from the shape.

Figure 5.6 presents the 14x7 *MCSoC*, with producer-consumer applications mapped when the system starts (pink PEs), with the goal to create obstacles to the *OSZ* creation, and three *OSZ*s. The creation of the *OSZ*s occurs according to the following sequence:

- *SZ*1: corresponds to the smallest execution time of Algorithm 5.1 because the execution of the outer loop finds a valid region in the first iteration;

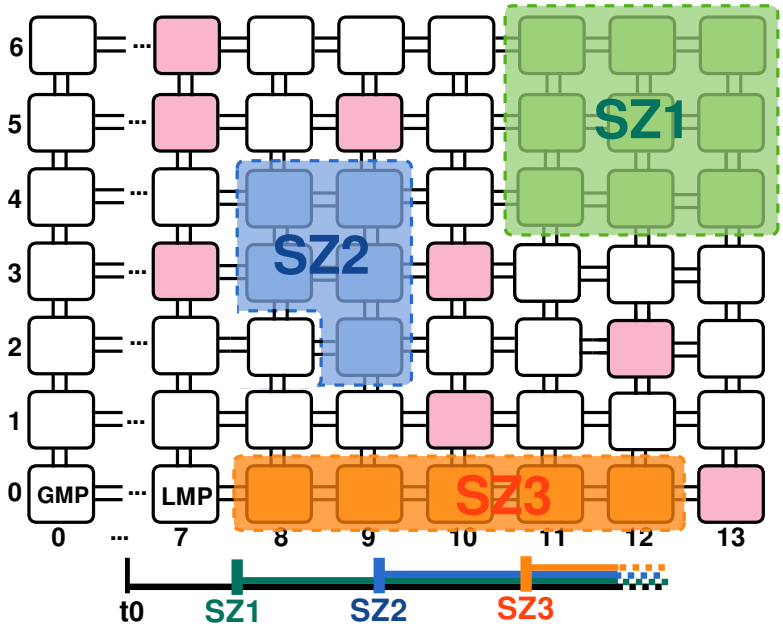


Figure 5.6: Algorithm 5.1 evaluation scenario with three OSZ positioning.

- *SZ*2: corresponds to an intermediate execution time due to the execution of the SWS several times, with the removal of a PE from the region to eliminates internal fragmentation;
- *SZ*3: corresponds to the largest execution time to find an *OSZ* in this scenario.

Note that the algorithm creates the three *OSZ*s at runtime, with App_{sec}s executing in parallel. Also, the blue OSZ (SZ2) requires internal rerouting when the botton-right PE send packets to PEs on the left side of the OSZ (if XY routing is used the packets would expose the communication). The experimental setup also has scenarios with tasks mapped inside the *SZ*s with the goal to activate the task migration, enabling to evaluate its cost (not presented in Figure 5.6).

Table 5.2 presents the scenario evaluation. The first column shows the Opaque Secure Zones created according to Figure 5.6. Results for execution of *Algorithm 5.1* (2nd column of table) are a function of the number of shapes and the position of selected shape in the cluster. The *Migration* results (3rd column) shows the cost to migrate one task of 4.3 KB at a distance of 4 hops. The *Cluster selection and message exchange* delay (4th column) is related to the initial cluster selection by GMP and the set of messages send to/from the PEs of the selected shape according with the protocol presented in Figure 5.3. The App_{sec} *mapping* delay (5th column) is the spend time to map the App_{sec} the into selected region. The 6th column is the sum of the previous columns that correspond to delay due to the Admission phase. All values in Table 5.2 are presented in clock cycles (cc - where K means 10^3).

The time spend by Algorithm 5.1 varies with the number of calculated shapes, the positioning of the shape, the previous tasks and *OSZ* found during the SWS and the peripheral positioning. Results show a low impact on executing Algorithm to create and

Table 5.2: Admission phase performance evaluation.

Secure Zone	Algorithm 5.1 (cc)	Migrations (cc)	Cluster selection and messages exchange (cc)	App_{sec} mapping (cc)	Total clock cycles (cc)
SZ1	3.9K	-	1.4K	4.3K	9.6K
SZ2	13.5K	-	1.2K	4.1K	18.8K
SZ2	27.2K	24.0K	1.9K	4.2K	58.3K
SZ3	21.1K	-	1.2K	4.1K	26.4K
SZ3	32.6K	23.4K	1.9K	4.2K	62.1K

search a suitable shape position. In the worst-case of the proposed evaluation scenario (SZ3), the delay was 32.6K clock cycles (cc). This corresponds to evaluate the SWS of seven calculated set shape from all valid initial positions of the cluster, then invert the set shape order (enabling 2 tasks by PE) to finally find a region starting in PE location (8,0).

As shown, the performance may be penalized if tasks migrations are required. The migrations can happen in parallel, i.e., more than one migration can perform at the same time, minimizing its cost. The cost of the *migrations* is a function of the number of migrations and the size of the task's object code. The proposed algorithm tries to minimize the number of migrations first looking for regions that not require migration and after looking by small shapes (and potentially fewer migrations).

The cost of the *messages exchange* varies with the distance of the PEs belonging to the *OSZ* relative to GMP and LMP. The *cluster selection and message exchange* has a insignificant impact on the delay to start the App_{sec}.

In summary, this set of results shows that the Application Admission phase is not time-consuming. Despite the exhaustive search made by Algorithm 5.1, the search space is small (few shapes to evaluate). The mapping algorithm is also fast because the search space is delimited by the *OSZ*, with enough resources to map the tasks.

The total time to execute this phase, including the cluster selection and the exchanged messages reached in the worst-case 62134 clock cycles, or 621.3μs@100MHz[1]. A usual case, without migration penalties uses less than 30K clock cycles, or 300μs@100MHz.

Key Exchange phase evaluation

The Key Exchange phase (Section 5.2.2) evaluation measures the delay time of each step after the GMP receives the `Req_alloc` message. Table 5.3 summarizes the elapsed time of each step of protocol according with Figure 5.4.

This phase of the protocol consumes, in average, 7324 clock cycles. The dominant cost is the SipHash time required to obtain the K_m key used to encrypt (step **2** in Figure 5.4) and decrypt (**6**) K_e. Other steps have a small impact on delay time of the key exchange phase.

[1] All MCSoC (routers, processor, DMNI, etc) runs at 100 MHz.

Table 5.3: Key Exchange phase evaluation.

SipHash and K_m at GMP (cc)	messages exchange (cc)	SipHash at SP (cc)	K_m at SP (cc)
3932	588	2270	534

Task Allocation phase evaluation

An experiment varying the object code size evaluates the Tasks Allocation and MAC verification phase - TAMV (Section 5.2.3). Table 5.4 presents in the 1st column the task object code size (in KBytes), in the 2nd column the number of clock cycles required to compute and verify the MAC, and in the last column the total time of this protocol phase.

Table 5.4: Task Allocation and MAC verification evaluation results.

Object Code size (KB)	MAC step (cc)	Total (cc)
2	67.2K	73.1K
3	100.1K	106.5K
4	133.0K	139.9K
5	165.9K	173.3K
10	330.4K	340.4K

Table 5.4 shows that the number of clock cycles (cc) to compute and verify the MAC is a function of the task object code size, being equal to 32.9 Kcc/KB. The equation below presents the number of clock cycles to compute and verify the MAC.

$$\text{TAMV}_{phase} \approx 2646 + (\textit{Object Code Size in KB} * 32.9\text{K})$$

where: 2646 cc is the average value between steps **3** and **5** of Figure 5.5.

Compared to the other protocol phases, this step dominates the latency to start secure applications. The reason to explain this larger delay is due to the MAC computation, which operates on 64-bit blocks, and the result of each block is used in the next block, thus being a sequential operation.

Figure 5.7 presents a timeline for the four phases impacting the latency to start an App_{sec} (the figure does not respect a scale related to each phase). The Key Exchange and MAC verification phases are executed in parallel, for each SP with App_{sec} tasks. The delay to start each of these phases corresponds to send the encrypted K_e and the tasks object code, respectively.

The Admission delay time of 18844 clock cycles presented in Figure 5.7 corresponds to the average case of Table 5.2, where half of the shape set is evaluated until finding an appropriate region to *OSZ*, and no migration is required.

As presented, the K_e key distribution is required to each SP belonging to the *OSZ*, i.e., with N SPs selected to run tasks of the App_{sec}, N set of interactions happens between

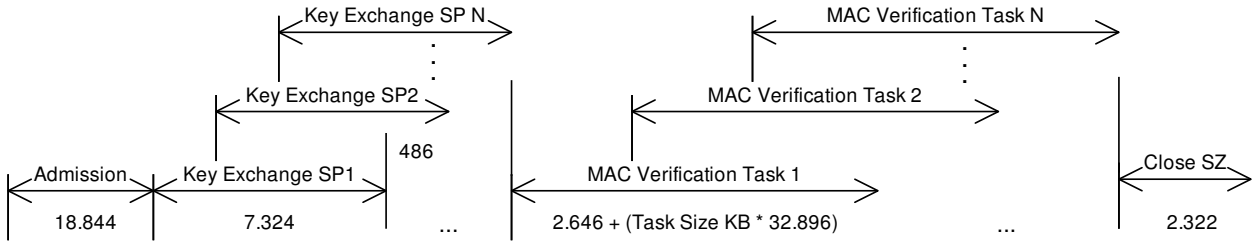


Figure 5.7: Timeline for the four phases impacting the latency to start an App_{sec}.

GMP and each SP. The average time spend to each interaction is 7324 clock cycles with a delay of 486 clock cycles to start the interaction with the next SP.

The Task Allocation step runs sequentially since the object code of each task is sent sequentially by the *AppInj*; however, the MAC verification runs in parallel once each SP calculate the MAC of the received message locally to check with the received MAC.

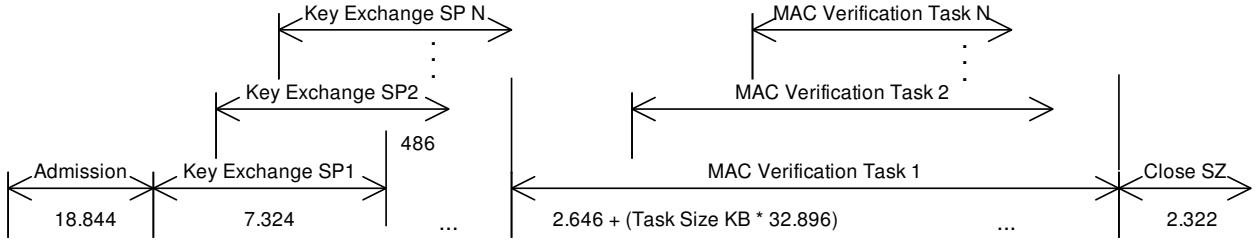


Figure 5.8: Timeline for the four phases impacting the latency to start an App_{sec} with task sorting.

Because the MAC calculation time is proportional to the object code size and occurs in parallel in different SPs, it is possible to reduce the total time spent to calculate the MAC sorting the sending tasks on *AppInj* from larger to smaller, taking advantage of parallel execution (Figure 5.8). Thus, in the best case, the time to calculate the MAC of all tasks corresponds to the time to calculate the MAC of the bigger size task.

Table 5.5 presents a comparative view of distinct applications running with and without security resources. The experiment also evaluates the task size sorting impact on delay to start the App_{sec}. The 1st and 2nd columns present the application and its corresponding number of tasks (# tasks). The 3rd column indicates the instant time in clock cycles when the request to run the application arrives the GMP. The 4th column presents the clock cycle number from the beginning of the application execution. The 5th column presents the delay time between the request and the beginning of the application execution, which correspond to the delay to start the application. The 6th column shows the cost due the security resources proposed.

Table 5.5 shows a small additional time to start the application in clock cicles (0.43 *ms* in the best case and 1.83 *ms* in the worst case, when running at 100 MHz). How expected the additional time is proportional with the object code size of tasks since the MAC calculation requires more time spend according to the object code size.

Table 5.5: Delay time to start different applications.

Application	# tasks	Request time (cc)	Start Execution (cc)	Delay to start (cc)	Cost (cc)
Prod/Cons	2	5.7K	35.3K	29.6K	-
Prod/Cons - OSZ	2	5.7K	78.2K	72.5K	42.9K
Prod/Cons - OSZ - sorting	2	5.7K	78.0K	72.3K	42.7K
MPEG	5	5.7K	74.4K	68.7K	-
MPEG - OSZ	5	5.7K	257.7K	252.0K	183.3K
MPEG - OSZ - sorting	5	5.7K	226.9K	221.2K	152.5K
DTW	7	5.7K	84.9K	79.2K	-
DTW - OSZ	7	5.7K	147.5K	141.8K	62.6K
DTW - OSZ - sorting	7	5.7K	137.2K	131.5K	52.3K

The advantage of task size ordering happens when the object code size difference is significant in such way that the time spend to compute the MAC of small object code sizes is contained by the time spend of bigger task code sizes. This occurs in the MPEG application were the PRINT task has 2.6 KB size, and the IDCT task has 6.7 KB size, resulting in a 12.2% small spent time to start the MPEG App_{sec}, when the task allocation occurs according to the task size.

5.4 Final Remarks

This chapter presented the security mechanisms used to guarantee the secure deployment of applications in *MCSoCs*. Three main phases were detailed: (*i*) the Mutual Authentication; (*ii*) the Application Admission and; (*iii*) the application's Tasks Allocation.

The mutual authentication method of entities [Caimi et al., 2018a] that deploy applications on *MCSoC* is a specific contribution of this Thesis. To the best of the Author's knowledge, there is no similar work for many-cores. The chapter detailed the Setup and the Mutual Authentication phases, and evaluates its costs for two processors architectures, with the costs being in the order of few seconds. Although this overhead seems high, only one execution of the mutual authentication process occurs for each *AppInj* to ensure the authenticity of the parts and does not impact the latency directly to start the applications. Besides that, only applications from trusty entities run in the many-core, reducing the probability of attacks from malicious applications.

The application's tasks allocation is another contribution of this Thesis ([Caimi et al., 2018b]). The relevant characteristics of the method include runtime execution (that allows the shape definition, *OSZ* positioning and tasks allocation - including task migration) and several *OSZ*s may co-exist in parallel.

In regards to the application admission, i.e. the latency to start a secure application, the dominant operation corresponds to the MAC computation and verification steps. The application is delayed by few milliseconds to start. Adding the delay of each protocol phase, the secure application is delayed for less than 4 ms (@100MHz) for 10KB tasks [Caimi et al., 2017b].

6. APPLICATION EXECUTION

After receiving all App_{sec}'s tasks correctly, the Application might start its execution in a runtime environment protected from attacks. This is obtained enabling the wrappers to create a Opaque Secure Zone (*OSZ*) into the *MCSoC*. The execution mechanism includes: *(i)* wrapper activation; *(ii)* launch application; *(iii)* retransmission of lost packets in and out the *OSZ* boundaries.

This chapter details the protocol steps to guarantee the execution of the entire secure application. Three main phases are detailed: (*i*) the OSZ closing at Section 6.1; (*ii*) the application execution at Section 6.2; (*iii*) the OSZ opening at Section 6.3. Section 6.4 presents the performance evaluation of these protocol phases. Finally, Section 6.5 presents the final remarks of the chapter.

6.1 Closing the Opaque Secure Zone

The goal of this phase is to close the wrappers surrounding the *OSZ* ("W" in Figure 3.1), and start the execution of the App_{sec}. Figure 6.1 shows the protocol steps related to *OSZ* closing and application execution steps.

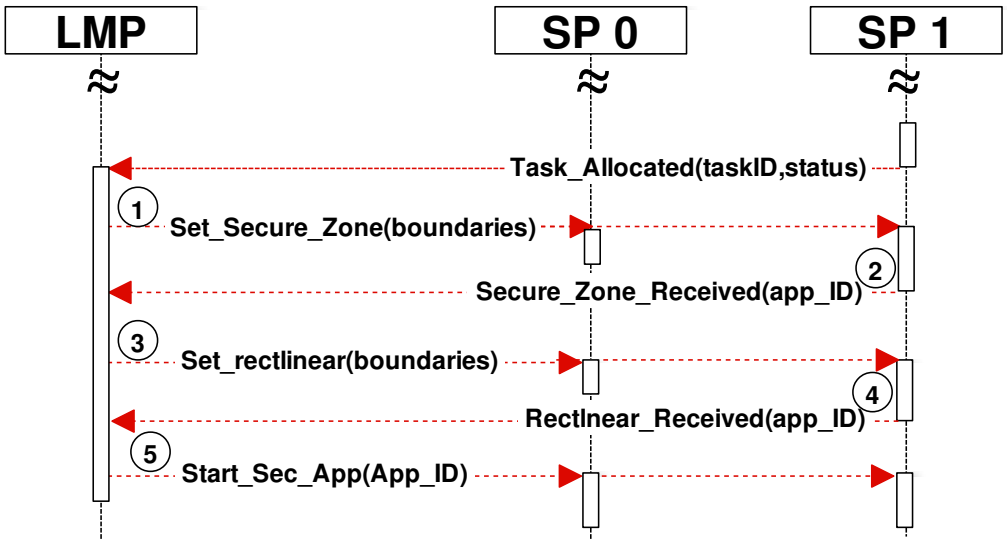


Figure 6.1: Close *OSZ* phase.

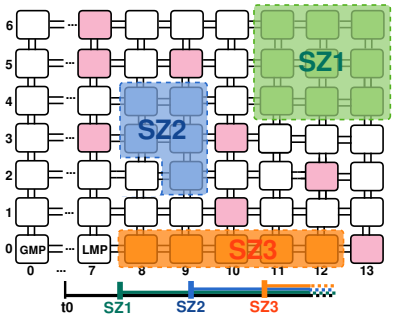
This phase starts with LMP sending a `Set_Secure_Zone` broadcast message (**1** in Figure 6.1) through the control NoC with the upper right (*UR*) and lower left (*LL*) corners of the *OSZ*. The operating system (OS) of All PEs receive this message, each one verifying if it is on the *OSZ* boundary. If the SP is on the *OSZ* boundary, the OS writes in a data structure the wrappers that must be closed.

The SP located at the *UR* corner of the *OSZ* transmits to the LMP a `Secure_Zone_Received` message (2). Once received this message, the LMP sends a `Set_rectilinear` broadcast message (3) through the control NoC with the *UR* and *LL* corners of SPs to remove from *OSZ*. The SPs belonging to this region and its neighborhood update the data structure of the wrappers to be closed. Two values identical and equal to the *LL* corner of the `Set_Secure_Zone` message indicates none cut must be done in the *OSZ*.

Table 6.1 details the values on `Set_Secure_Zone` and `Set_rectilinear` messages to SZ1 to SZ3, related to Figure 5.6 (reproduced at right side of the table).

Table 6.1: Parameters of the messages to close the OSZs.

	SZ1 (LL)(UR)	SZ2 (LL)(UR)	SZ3 (LL)(UR)
`Set_Secure_Zone`	(11,4)(13,6)	(8,2)(9,4)	(8,0)(12,0)
`Set_rectilinear`	(11,4)(11,4)	**(8,2)(8,3)**	(8,0)(8,0)



The SP located at the *UR* corner of the *OSZ* transmits to the LMP a `Rectlinear_Received` message (4). After receiving this message, the LMP can start the execution of App_{sec}, by broadcasting a `Start_Sec_App` message (5), using the control NoC in *global* mode. This message enables the SPs to activate the wrappers to block incoming/outcoming traffic and releases the tasks belonging to App_{sec} to execute.

6.2 Secure Application Execution

The `Start_Sec_App` message activates the wrappers surrounding the *OSZ* ("W" in Figure 3.1) starting the execution of the App_{sec}.

The wrapper activation occurs using a memory mapped register (*wrapper_reg*) at each PE. Each bit in the wrapper register enables/disables a given port wrapper of the PE. The wrapper acts over the control flow signals of each NoC port (8 ports for the data NoC and 4 ports for the Control NoC). Figure 6.2.a shows the wrapper logic for one port. Thus, the wrappers' area overhead is insignificant since its implementation requires a small number of gates, a register, and an Finite State Machine (FSM).

When the wrapper value is activated, the internal PE side has the control flow signals masked, disabling all external requests. If a request arrives, it is blocked, and due to the masking, the *ack* signal is high. This simple process results in discarding the packets arriving at the PE. The same process occurs when the PE tries to send a message, *req* blocked and *ack* enabled. The control NoC uses a single wrapper value, opening or blocking the

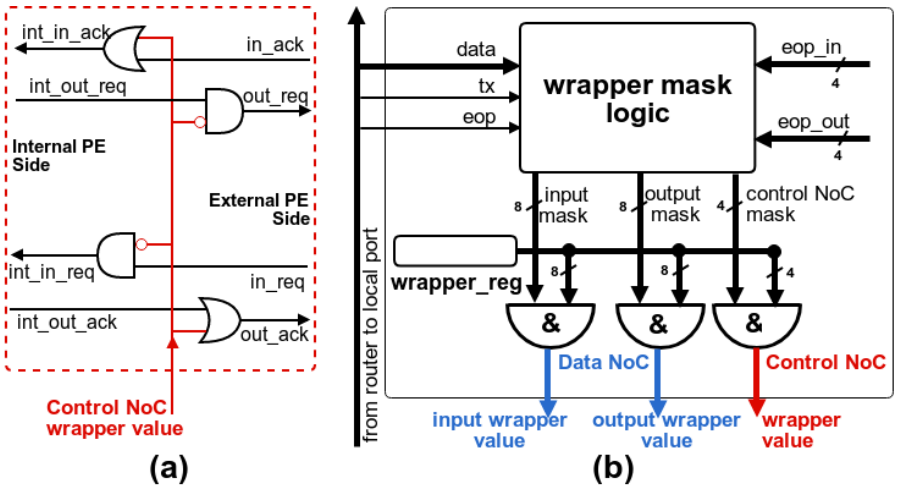


Figure 6.2: (a) Wrapper logic for one port; (b) Detailed Wrapper Control masking logic.

flows in both directions. The data NoC contains two wrapper values, enabling to selectively block the flow direction (required for I/O communication).

For security reasons, applications running at the SPs cannot access *wrapper_reg*, that only the OS has access to it. Figure 6.2.b details the "wrapper control" module. The value applied to the wrapper logic explained above is a result of a AND operation between the *wrapper_reg* value and a mask value coming from “wrapper mask logic" (WML) module. The default value of mask signals arriving the AND operation is ’1’, i.e., by default the wrappers values corresponds to *wrapper_reg* contents. Section 7.2 present detailed explanation about the wrapper control module.

When a message discarding occurs, the control NoC sends a retransmission request (using the brNoC) to the message source PE. When the retransmission request arrives at the source PE, the OS uses the control NoC to obtain a new path to the target PE, which avoids the *OSZ*. The control NoC returns a path, which is used to retransmit the message, and all subsequent packets through source routing.

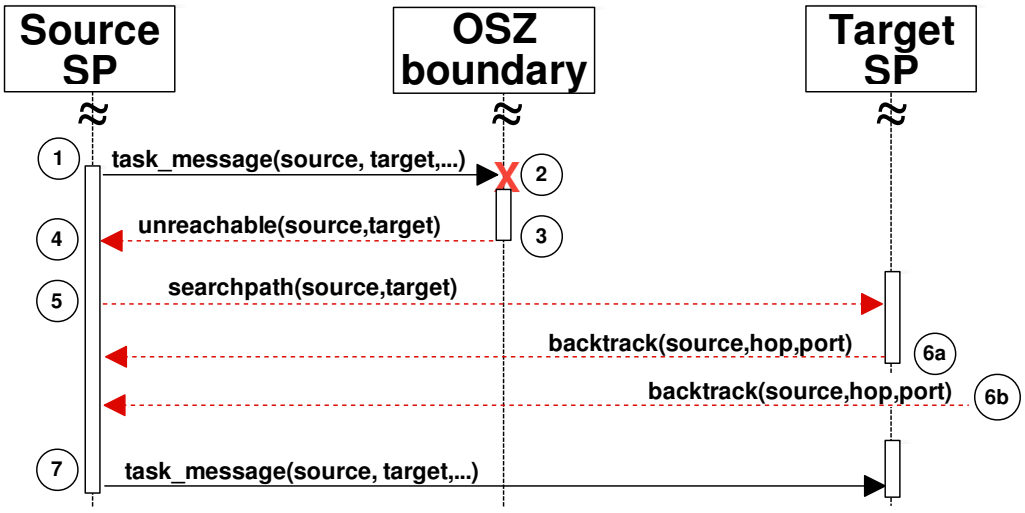


Figure 6.3: Rerouting sequence.

Figure 6.3 shows the sequence diagram of the search path and rerouting mechanisms. Considering an application running with its task exchanging messages (1) the *OSZ* activation discards the message (2) and the PE at the border sends a `unreachable` message (3) notifying the source SP. The source SP verifies if the message was sent previously (4) and request a path discovery (using the *brNOC*) through the `searchpath` message (5). When the `searchpath` message arrives the target PE the backtracking mechanism sends a sequence of `backtrack` messages, each one providing the hop number and the output port used by the hop in the SR mechanism (6a, 6b, ...). With all pairs `(hop, port)` received, the source SP process the path and resend the previously discarded message using source routing (SR) (7).

Figure 6.4.a shows two applications running ({T1,T2} and {T3,T4}) using XY routing. At some time, a App_{sec} creates a square *OSZ*, (3,2) to (4,3). After the *OSZ* activation, a message sent by task T1 (3,1) to task T2 (4,4) is discarded at PE (4,2) and an `unreachable` message is sent back to PE (3,1) notifying the message loss. At PE (3,1) a path discovery to PE (4,4) starts using a `searchpath` message. When this message arrives the target PE (4,4), the backtracking starts sending a unicast message to the source PE with the current hop (7) and the **incoming** port of the `searchpath` message (W). After the unicast message from (4,4) pass through PE (3,4) it also generates and sends a unicast message containing its hop (6) and the **outcoming** port to PE (4,4), i.e., port E (Figure 6.4.b). This process continues at each PE in the unicast path, with each PE sending a message with its hop and outcoming port to the PE (4,4).

Note that the first unicast message uses the incoming port, while other unicast messages use the outcoming port. This mechanism creates an invalid routing situation (incoming port equal to the outcoming port), which is used in the source routing as the mechanism to detect that path ended.

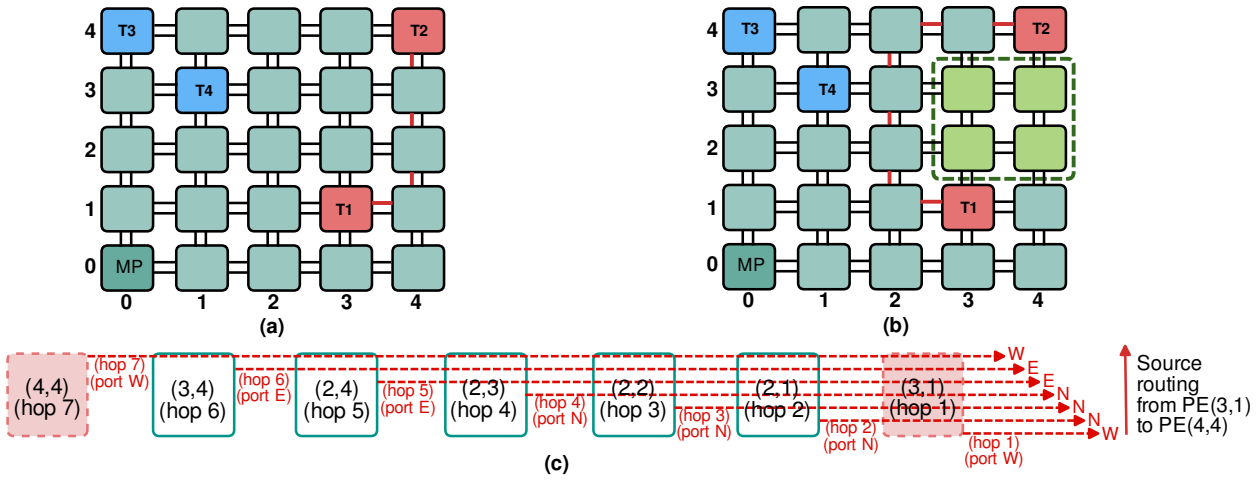


Figure 6.4: Backtrack example.

Figure 6.4.c shows the unicast messages sequence with respective hop and port. The resulting SR path from PE (3,1) to PE (4,4) circumventing the *OSZ* use the following directions at each PE: [W N N N E E **W**].

Note that the alternative path computation is executed once, only for the first lost message. Thus, the impact of closing an *OSZ* in the non-secure applications is minimal, as shown in the results section.

6.3 Protocol to Open an Opaque Secure Zone

At the end of App_{sec} execution, the "Open *OSZ*" phase clears the memory contents of the SPs inside it preventing any information leakage from being used by an attacker. In addition, the OS erase the K_e value (also K_ps if used) and release the wrapper opening the *OSZ*. Also, the LMP clears internal structures to release the cluster resources previously allocated to App_{sec} and sends a message to the GMP that releases its internal structures related to the cluster resources.

Figure 6.5 shows the protocol steps to open the *OSZ* at the end of the App_{sec}. When a App_{sec} task ends its execution (1), the PE sends a `End_Task` message with its task ID, using the control NoC in *global* mode.

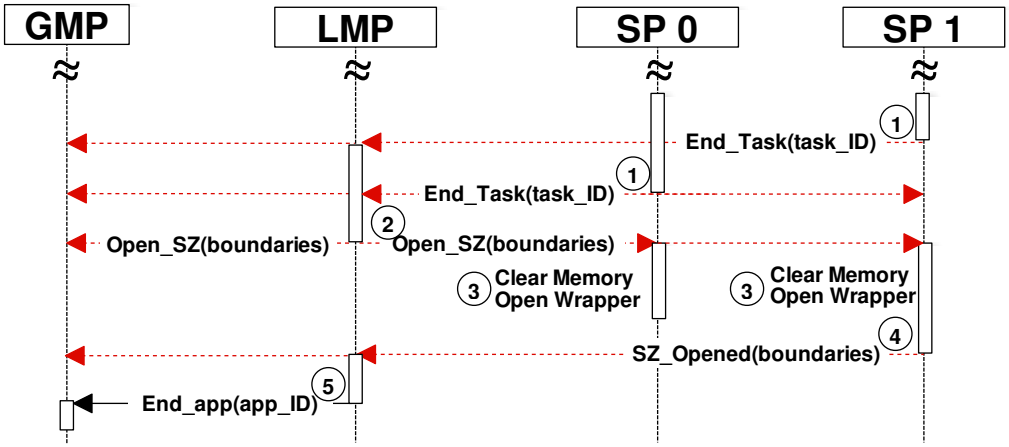


Figure 6.5: Open *OSZ* protocol steps.

When all App_{sec} tasks finish their execution, the LMP transmits an `Open_Secure_Zone` message (2). All PEs inside the *OSZ* clear their memory to prevent information leakage and then open the wrappers (3), releasing the locked resources. Then, the PE located at the upper right corner of the SZ transmits to the LMP a `Secure_Zone_Opened` message (4). Finally, the LMP clear the internal structures to release the cluster resources previously allocated to App_{sec} (5).

6.4 Application execution evaluation

This section evaluates the protocol phases previously presented and the impact of the *OSZ* creation in applications that use paths blocked by *OSZ*. The first experiment evaluates the steps presented in Section 6.1, corresponding to the impact to close *OSZ* in 4 scenarios, changing the *OSZ* size from 2x2 up to 3x3 SPs. Table 6.2 presents the evaluation, considering the *OSZ* size.

Table 6.2: *OSZ* Close phase evaluation.

OSZ Shape	2X2	2X3	3X2	3X3
Total (cc)	2302	2312	2316	2322

The number of clock cycles to close an *OSZ* starts when the LMP sends the `Set_Secure_Zone` message until the upper right PE in *OSZ* effectively activates their wrappers after the `Start_Sec_App` message. The *Close Secure Zone* results present a small increase in the number of the clock cycles, proportional to the distance, in hops, to the upper right PE of the *OSZ*. As presented in the previous chapter the Close *OSZ* protocol imply in a small impact on the latency to start the App_{sec} of just 2.3K clock cycles.

The second evaluation concerns the impact to open an *OSZ* in 5 scenarios, changing the *OSZ* size from 2x1 up to 5x5 PEs. Table 6.3 presents the evaluation, considering the *OSZ* size. The time to open a *OSZ* starts when the LMP sends the `Open_Secure_Zone` message until the reception of a `Secure_Zone_Opened` (Figure 6.5). The *Open Secure Zone* delay is related to the amount memory pages to erase and the size of each memory page. All memory pages (64 KB in the experiment) used by App_{sec} are erased at this protocol step.

Table 6.3: *OSZ* Open phase evaluation.

OSZ Shape	2x1	2x2	3x3	4x4	5x5
Total (cc)	5943	5,962	6006	6048	6081

The Open Secure Zone results present a small increase in the number of the clock cycles (around 6K clock cycles) to a 64 Kbytes memory page and proportional to the distance to the upper right PE of the *OSZ*.

The result disclosed by the table is the *scalability*. The overhead induced by the method comes from the *Close Secure Zone*, which corresponds to less than 2400 clock cycles to a large *OSZ* (2.4 μs@100MHz). The *Open Secure Zone* does not impact in the applications' execution time. The *Open Secure Zone* delay affects the amount of time to release the PEs, which is also small (around 6.0 μs).

The third evaluation presents the intrusiveness of non-secure applications in an App_{sec}. This experiment presents a scenario where an App_{sec} shares communication resources with malicious flows. Figure 6.6(a) presents the mapping configuration: an App_{sec}

mapped in the *OSZ* (surrounded by the doted line) and six other tasks mapped outside the *OSZ* generate malicious flows. The latency of the packets on the north port of the PE executing *TaskD* is measured according to 3 scenarios (Figure 6.6(b)): *(i)* App_{sec} executing alone in the system; *(ii)* all tasks running without the *OSZ*; *(iii)* all tasks running with the *OSZ* activated. Scenario *(i)* produces the baseline latency between taskD and taskF, without any disturbing traffic. In the second scenario *(ii)*, the malicious flows disturb the taskD latency. The 3rd scenario shows that the application running inside the *OSZ* the latency has the behavior of the baseline scenario *(i)*, *preventing* DoS or timing attacks.

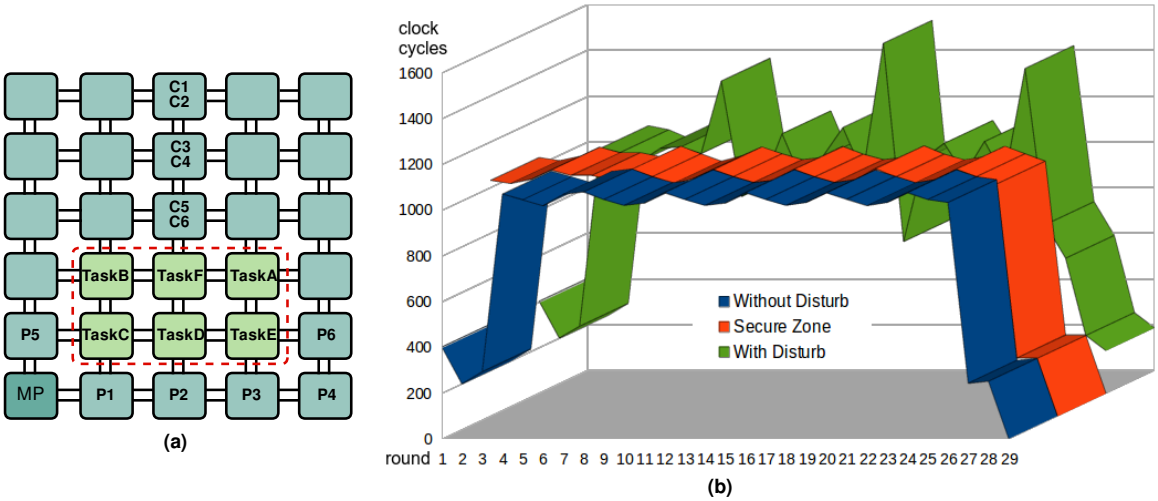


Figure 6.6: Communication latency: (a) App_{sec} in the *OSZ*, and malicious tasks outside the *OSZ*; (b) latency graph results [Caimi et al., 2017a].

The fourth evaluation presents the impact of the *OSZ* on non-secure applications. Close a region inside the *MCSoCs* at runtime implies that applications executing outside of *OSZ* must continue the execution, by reconfiguring and rerouting the source-target paths. Figure 6.7 presents the mapping of two non-secure applications. The dotted squares in the figure represent the boundary where the App_{sec} can be mapped and executed.

The non-secure applications start their execution, and at a given moment the App_{sec} is mapped, and then the *OSZ* closed. For each non-secure application, 6 secure zones scenarios are evaluated. Table 6.4 presents each scenario, the non-secure application execution time (AET) overhead (2nd and 4th columns), and the number of rerouted paths (3rd and 5th columns), for MPEG and DTW applications, respectively.

The execution time overhead of the applications outside of the *OSZ* is in the worst-case 2.56%. This overhead is related to the number of rerouted paths, which is a function of the *OSZ* location, and the number of interrupted flows. For example, the third DTW scenario (highlighted in gray) has a large *OSZ* (12 PEs), requiring the execution of 12 rerouting paths. This experiment showed that *OSZ* might be defined at runtime, with a insignificant impact on the performance of the applications outside of the region.

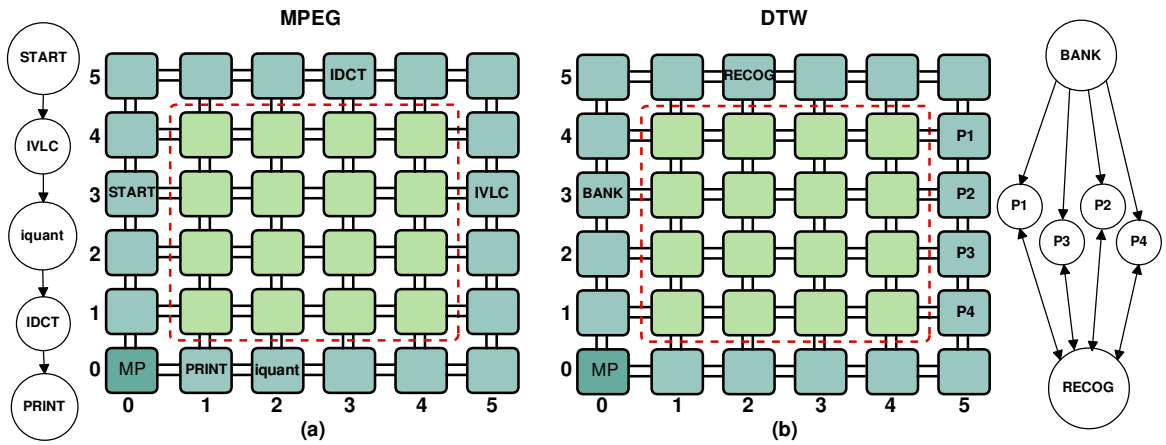


Figure 6.7: Task graphs and mapping. (a) MPEG and (b) DTW applications [Caimi et al., 2017a].

Table 6.4: Impact of the OSZ in the Non-Secure Applications [Caimi et al., 2017a].

OSZ Boundary	MPEG		DTW	
	AET overhead	# Paths Rerouted	AET overhead	# Paths Rerouted
((1,2),(3,4))	1.22%	7	2.53%	11
((2,2),(3,4))	1.13%	6	2.53%	11
((1,1),(3,4))	1.22%	7	2.56%	12
((3,2),(3,4))	1.13%	5	2.53%	10
((1,3),(4,3))	1.17%	7	0.98%	8
((1,2),(4,3))	1.17%	7	0.79%	9

The rectilinear shape of an *OSZ* can lead to an internal unreachable path when two communicating tasks are mapped with the XY communicating path crossing the *LL OSZ* corner. Next experiment evaluates the impact of internal unreachable paths on the performance of APP_{sec}. Figure 6.8 presents the mapping of MPEG application that results in the unreachable path between the *iquant* and *IDCT* tasks and the rerouting mechanism activation.

The message sent by *iquant* task to the *IDCT* task is blocked by wrappers and a searchpath message is fired. After get a new path the message is re-send using SR. The cost is related to one path discovery and reroute path and represent an overhead of just 0.26% considering the baseline MPEG application execution.

The opaque secure zone completely isolates the application, protecting computation and communication resources from attacks (integrity, confidentially, DoS, timing, and spoofing) from malicious applications, using wrappers, a low-cost hardware mechanism. Results show the effectiveness of the approach and the insignificant impact in the execution time of secure and non-secure applications.

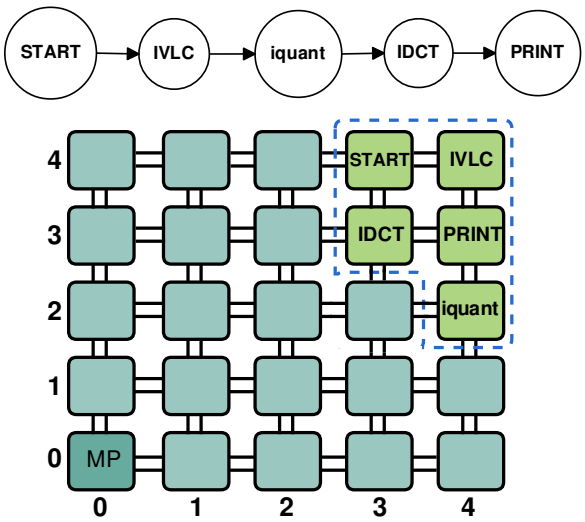


Figure 6.8: Task graphs and mapping of MPEG application with broken path between *iquant* and *IDCT* tasks.

6.5 Final Remarks

This chapter presented an original contribution of this Thesis, a method to set Opaque Secure Zones at runtime avoiding communication and computation sharing, including a dynamic rerouting mechanism to deal with broken paths of task messages ([Caimi et al., 2017a]. The proposed mechanism uses wrappers, a low-cost hardware mechanism due to the small area overhead. The *OSZ* isolates the application, without any communication and computation sharing, enabling tasks' messages exchange without cryptography, not penalizing the execution time of the secure application.

The evaluation shows a insignificant impact on the execution time of secure and non-secure applications, even in the presence of several broken paths and the respective rerouting and retransmission of messages. Two aspects explain this remarkable result: (*i*) the efficiency of *brNoC* to obtain new paths trough the `searchpath` and `backtrack` messages; (*ii*) the new SR path is used for all new messages to the same target PE.

7. PERIPHERAL ACCESS

Previous Chapters presented the approach to protect the admission and execution of applications with security constraints, including ECDH and MAC techniques and spatial reservation of computation and communication resources, resulting in an Opaque Secure Zone (*OSZ*) [Caimi and Moraes, 2019].

This chapter presents in Section 7.1 the communication model and the APIs used by the *MCSoC* to communicate with peripherals. Section 7.2 presents the mechanism that enables the access to peripherals using a protocol to open access points on the border of the *OSZ*s. Section 7.3 discusses communication issues related to open access points. Section 7.4 presents the mechanisms to protect the packet header and payload when a message goes outside the *OSZ*. Section 7.5 presents the evaluations and a discussion about the proposed mechanisms. Finally, Section 7.6 presents the final remarks of the chapter.

7.1 MCSoC communication model

The adopted communication model is message passing (MPI-like), with the API supporting non-blocking *Send*() and blocking *Receive*() primitives to communicate. The reason to adopt message passing comes from the adopted memory organization, distributed memories.

Figure 7.1 presents the flow to both send and receive a packet between two different PEs. The *Send*() primitive generates a system call, *send_packet*(). The kernel copies the message in a data structure named *pipe*, and the task continues its execution, performing a non-blocking *Send*(). At the consumer side, the task executes a *Receive*() primitive. The OS generates a request packet to the producer PE and changes the task status to *waiting*, performing a blocking *Receive*().

When the producer PE receives the request message, it programs the DMNI to send the packet, copying the data from memory and transmitting it to the NoC. At the consumer side, when the DMNI receives a packet, it interrupts the processor. The interruption handler calls the *read_packet*() at the OS level, which programs the DMNI to read the packet copying it from the NoC to the memory. Once the packet is received, the OS executes functions related to the contents of the packet. For example, if the packet contains a data to a *task_ID*, the packet is written in the *task_ID* memory space, the *Receive*() call is unlocked, and the *task_ID* is scheduled to execute.

Due to this handshake mechanism, the task messages remain into the OS structure called *pipe*. The OS has also a message buffer structure to store control messages that use the data NoC (such as `Req_task`, `Req_alloc`, `App_status`). These two structures, *pipe* and

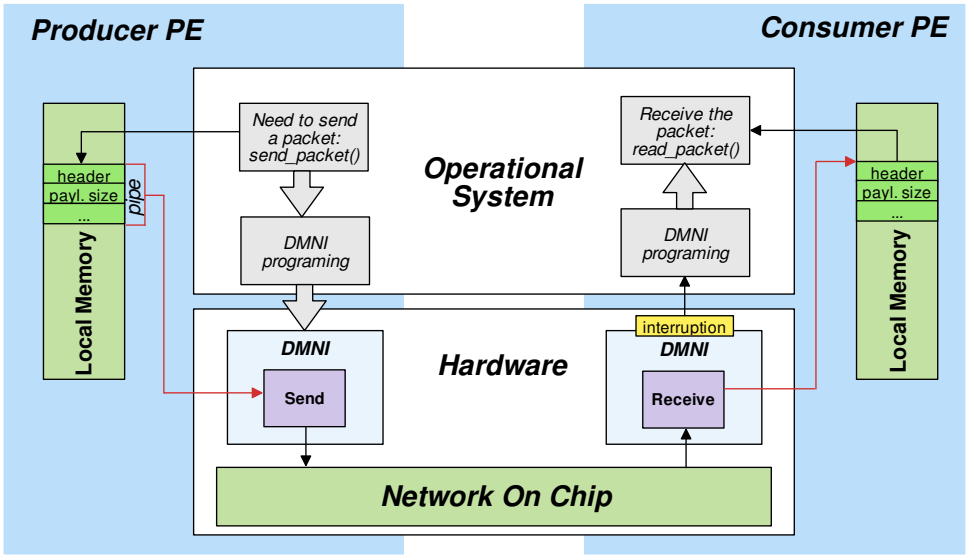


Figure 7.1: Inter-PE communication flow [Ruaro et al., 2016].

control messages buffer, enable the retransmission of packets when a discard occurs due to the presence of an *OSZ* boundary.

Communication with peripherals uses a second API, with primitives *IO_Send*() and *IO_Receive*(). Both primitives are blocking, that is, the task calls the primitive and remains blocked until an answer message arrives from the communicating peripheral. The I/O access uses a master/slave communication model where the PE is the communication master and peripherals the communication slaves. Regardless the I/O primitive used, the initial packet is sent from the PE to a peripheral, and the answer comes from a peripheral to the PE.

At the lower level, the OS communicates using the data NoC with *IO_request*, *IO_delivery*, and *IO_ack* packets. The *IO_Receive*() primitive uses the *IO_request* at the PE side and the *IO_delivery* at the peripheral side. The *IO_Send*() primitive uses *IO_delivery* at the PE side and the *IO_ack* at the peripheral side. Figures 7.2(a) and 7.2(b) show the API primitive `IO_Receive` and `IO_Send`, and the underlying message exchange between the Consumer PE and a peripheral (a) and the Producer PE and a peripheral (b).

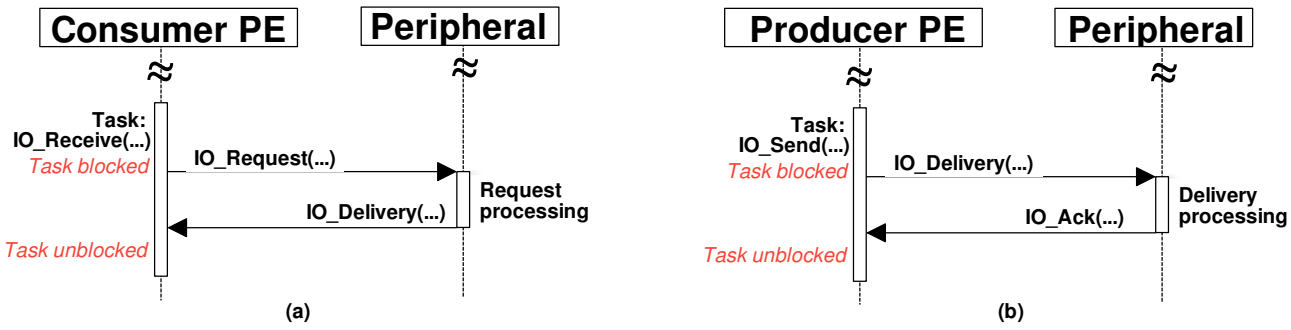


Figure 7.2: I/O API primitives. (a) Consumer task accessing a peripheral. (b) Producer task accessing a peripheral.

As presented in Figure 3.2 (on page 50), the data NoC differentiates the API's messages (MCSoC internal data or I/O data) with a flag into the header field (first flit) of the packet, enabling a distinct control of the packets when necessary. Specifically, this feature helps to block all data packets arriving at the boundary of the *OSZ* (in both directions) and to apply selective management of I/O packets.

This communication method with peripherals helps preventing attack attempts, either by PEs trying to forge I/O packets or even peripherals trying to carry out the attack. Communication is always initiated by the PE in the *OSZ*. Note that if a peripheral does not answer, a DoS attack may occur. A simple solution to avoid this problem is to adopt a timer at the OS level, which avoids long waiting times.

7.2 Enabling I/O access from OSZ

The incoming and outcoming packets arriving at a given port are discarded when the *OSZ*'s wrappers are enabled. By default, this mechanism prevents data exchange from a PE inside the *OSZ* to a peripheral and also prevent the data exchange from a peripheral to a PE inside the *OSZ*.

Figure 7.3.a details the PE with wrappers, and Figures 7.3.b and 7.3.c details the wrapper attached to the control NoC and the wrapper control module respectively.

When the wrapper is enabled, the internal PE side has the control flow signals masked, disabling all external requests. If a request arrives, it is blocked, and due to the masking, the *ack* signal is high. This simple process results in discarding the packets arriving at the PE. The same process occurs when the PE tries to send a packet, *req* signal is blocked and *ack* signal is enabled.

Besides this process, a dedicated circuit monitors the incoming requests in the data NoC when the wrappers are active. As mentioned previously, when a packet request arrives in an activated wrapper, it is discarded. This circuit stores the header flit (first flit) to recognize the source and target PEs, and create a notification message through the *brNoC* (`UNREACHABLE` message) to the source PE, enabling the retransmission mechanism. Note that this massage does not carry any information related to the position of the *OSZ*, preventing attackers from discovering where the *OSZ* is.

When the notification message arrives at the source SP (`UNREACHABLE` message), the OS uses the control NoC to obtain a new path to the target SP, which avoids the *OSZ*. The control NoC returns a path, which is used to retransmit the packet, and all subsequent packets through source routing.

As show in Figure 7.3.c, the control NoC uses a single wrapper value, opening or blocking the flows in both directions. The data NoC contains two wrapper values, enabling to selectively block the flow direction (required for I/O communication).

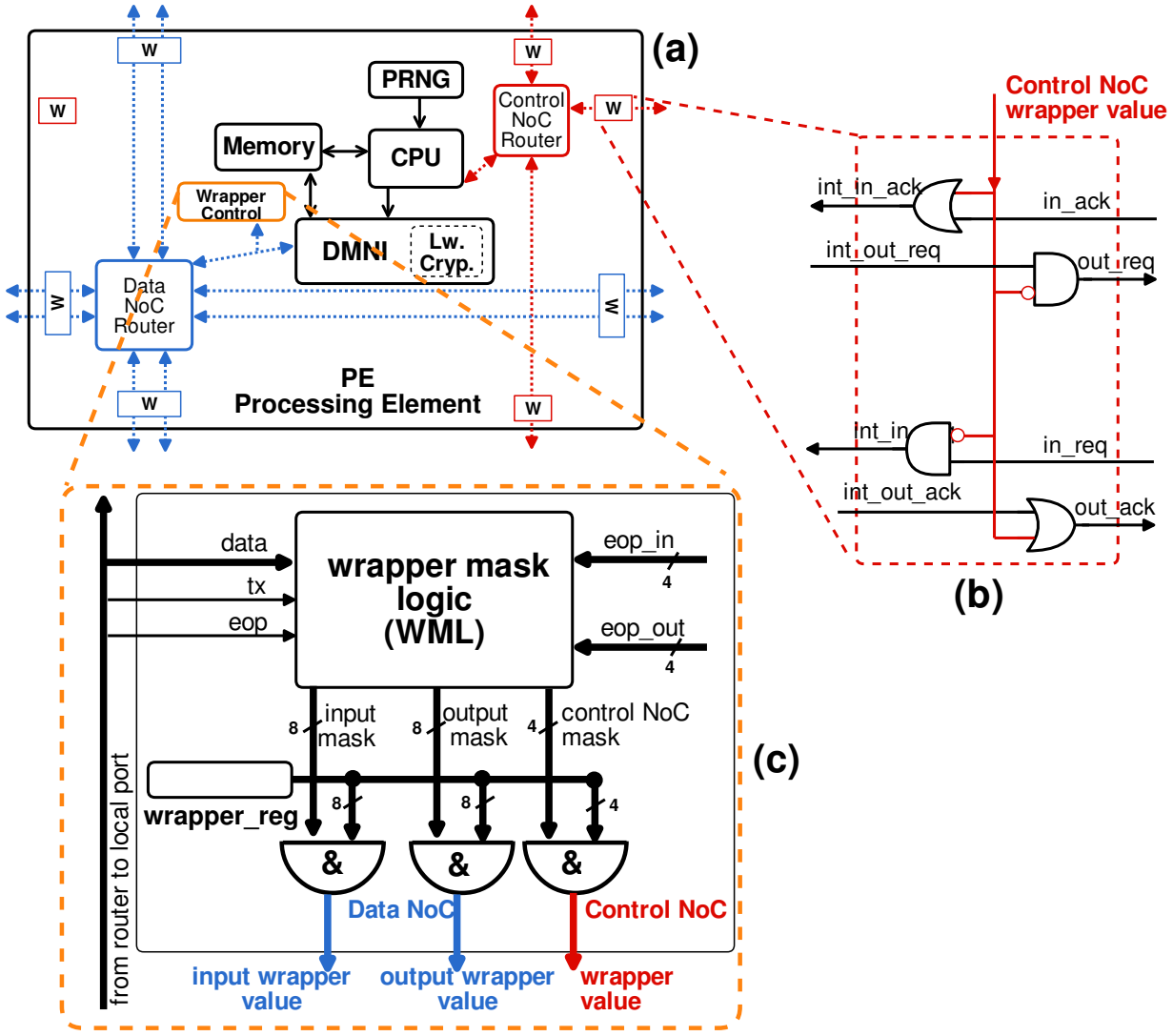


Figure 7.3: Wrappers modules. (a) PE architecture; (b) brNoC wrapper module; (c) wrapper control module

The wrapper activation occurs using a memory mapped register (*wrapper_reg*) at each PE. Each bit in the wrapper register enables/disables a given port wrapper of the PE. The wrapper acts over the control flow signals of each NoC port (8 ports for the data NoC and 4 ports for the Control NoC). Figure 7.3.b shows the wrapper logic for one port. Thus, the wrappers' area overhead is negligible since its implementation requires a small number of gates, a register, and an FSM.

For security reasons, the applications running at the SPs cannot access *wrapper_reg*, given that only the OS has access to it. Figure 7.3.c details the "wrapper control" module presented in Figure 7.3.a. The value applied to the wrapper logic explained above is a result of a AND operation between the *wrapper_reg* value and a mask value coming from "wrapper mask logic" (WML) module. The default value of mask signals arriving the AND operation is '1', i.e., by default the wrappers values corresponds to *wrapper_reg* contents.

As explained, an *OSZ* block all incoming and outcoming messages. However, I/O communication requires a selective control to send packets to peripherals and receive packets from peripherals. The wrapper control module performs the I/O packet control, enabling the communication with peripherals. Distinct wrapper values are provided to incoming and outcoming traffic in the data NoC (input wrapper value and output wrapper value in Figure 7.3.c).

According to the relative position of the SP with the peripheral location, the wrapper mask configuration occurs through configuration message or memory mapped register (`input_mask_reg` and `output_mask_reg`) inside the wrapper mask logic. Configuration message is required when the *OSZ* boundary is not connected to the communicating SP, otherwise the OS set the memory mapped register directly. Figure 7.4 presents the six configuration scenarios, according to the relative position of SP and peripheral. Due to the master/slave communication mode, each I/O interaction needs to open two wrappers on the data NoC to (a) send a packet with the request, or with the data and (b) receive the answer with the data or ack response.

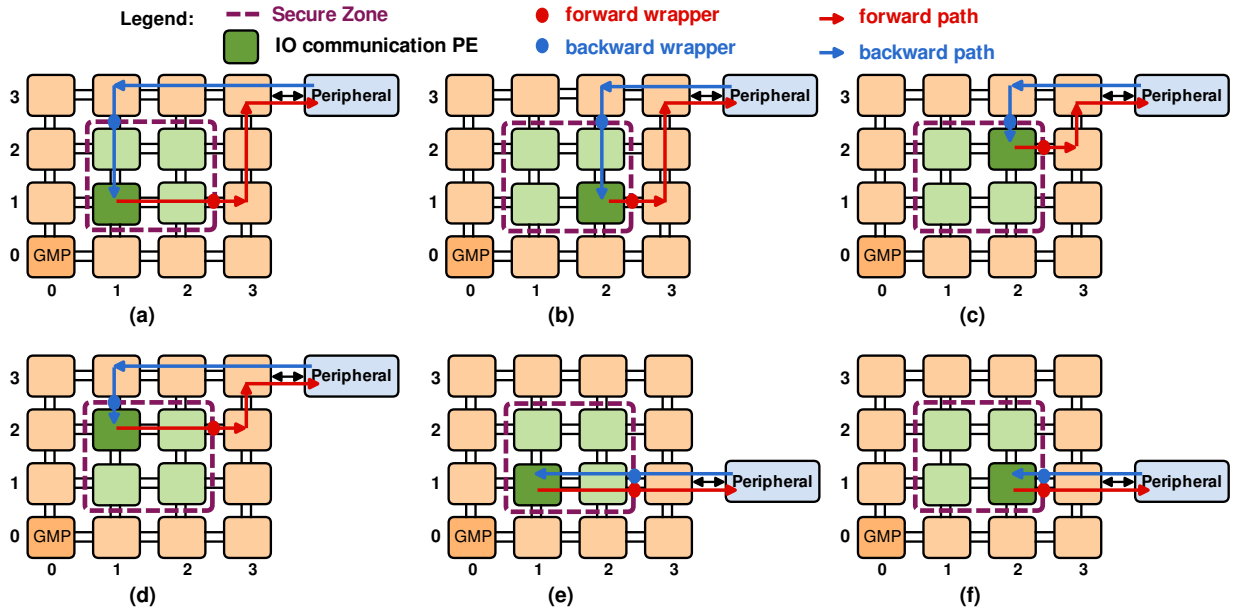


Figure 7.4: Configuration scenarios for I/O communication from *OSZ*.

Table 7.1 summarizes the wrapper configuration with respect to the method used (message or memory mapped register) and the direction of the data message (forward - output direction; backward - input direction). The difference between cases (a) and (e) refers to the configuration messages goes to distinct or same SP. The difference between cases (c) and (f) refers to the port masked, i.e. distinct or same port.

Before transmitting a message to a peripheral, the OS of the communicating SP first sends two configuration messages to the boundary of the *OSZ* (or set the respective mask register according to Table 7.1), to set the mask registers. The SP knows the boundaries of *OSZ*, obtained in the 'Close *OSZ*' step. When the mask configuration message arrives at

Table 7.1: Mask wrapper configuration cases.

Case	forward wrapper	backward wrapper
(a)	message	message
(b)	mapped register	message
(c)	mapped register	mapped register
(d)	message	mapped register
(e)	message	message
(f)	mapped register	mapped register

the target router, the wrapper control module (Figure 7.3.a) intercepts the message (i.e., this message is not consumed by SP) to set the mask value (input mask or output mask value) as shows in the Figure 7.3.c.

The mask configuration messages contain the direction (input or output) and the port side to mask (e.g., north). Note that the WML module (Figure 7.3.c) receive the EOP signals (End-Of-Packet) of each data router port. Thus, once an EOP is received in an opened port (input or output), this port is closed. This mechanism ensures that the secure zone will receive only one I/O packet for each request.

After App_{sec} has executed the IO_Receive or IO_Send functions, the API needs to adjust the message and configure the wrappers before send the I/O packet. Figure 7.5 show the procedures before sending the I/O message.

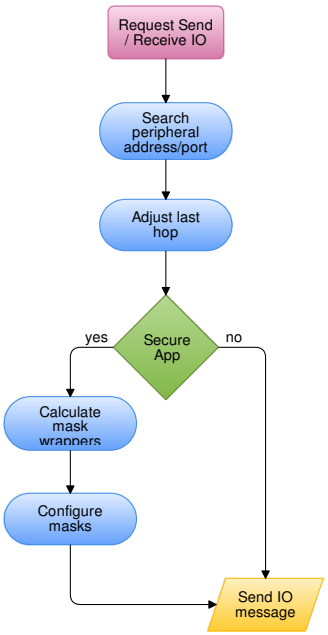


Figure 7.5: Procedures to adjust the I/O packet and wrappers configuration.

Initially the OS search by the address and the port where the peripheral is attached. At the application level, the peripheral is available from their name but, at the communication level the XY position and the port are used (e.g. peripheral called `raw_log` attached to the router (4,4), north port).

In our model, peripherals are attached directly to a router port (e.g., north port), and there is a PE connected to the local port of this router. Thus, we need to adjust the last hop to avoid that the packet be consumed by the local port, being forwarded to the peripheral.

Next, considering that the communication with the peripheral is requested by a secure application, before sending the I/O message, the mask values and the wrapper configuration are provided by the OS according to the procedure summarized in Table 7.1 and Figure 7.4, i.e., using configuration messages or mapped register configuration.

7.3 I/O communication issues

With multiple active *OSZ*s and other applications running into *MCSoC* the procedure of sending and receiving I/O messages opening specific ports, with specific directions is subject to four main issues:

(a) from *OSZ* to I/O – the message hit other *OSZ* in the path to the peripheral: this causes the discard of the message, and an unreachable message (using the control NoC on global mode) arrives at the source PE inside the *OSZ*. The PE sends mask configuration messages to reestablish the initial wrappers conditions and retransmit the message using source routing;

(b) from I/O to *OSZ* – the message hit other *OSZ* before arriving at the *OSZ* boundary: this cause a message discarding and an unreachable message arrives at the PE connected to the peripheral. This SP request a *searchpath* service and obtain the source routing path to the target PE inside the *OSZ*. Then the SP sends a configuration message to the peripheral to retransmit the message using the source routing path received. At *OSZ* side no action is needed;

(c) from external PE to an I/O – after opening the input port and before receiving the peripheral response, a distinct I/O message addressed to a peripheral arrives in the *OSZ*, resulting in it being discarded by the wrapper. With the target address present in the header flit is possible to discard the message at the wrapper and sends a unreachable message to the source PE. Then source PE request a searchpath to resend the discarding message. At *OSZ* side no action is needed;

(d) from distinct I/O to *OSZ* – after opening the input port and before receiving the peripheral response a distinct I/O message (addressed to a PE outside the *OSZ*) arrives in the *OSZ*, resulting in it being discarded by the wrapper. With the target address present in the header flit is possible to discard the message at the wrapper and sends a unreachable message to the source. The unreachable message arrives at the PE connected to the peripheral and the procedure is similar to (b). At *OSZ* side no action is needed.

As presented, each issue implies in specific actions taken by the OS to guarantee the App_{sec} execution. Figure 7.6.a shows the OS's procedure when a unreachable message arrives to the PE after an I/O message is discarded due to hitting another *OSZ*. Initially the PE verifies the message ID and if it belongs to an App_{sec}. If the discarded message belongs to an App_{sec}, before sending the searchpath message the OS must compute new values to the wrapper's masks (close the opened wrappers in data NoC) and reconfigure the wrappers to enable the searchpath message pass through control NoC wrapper.

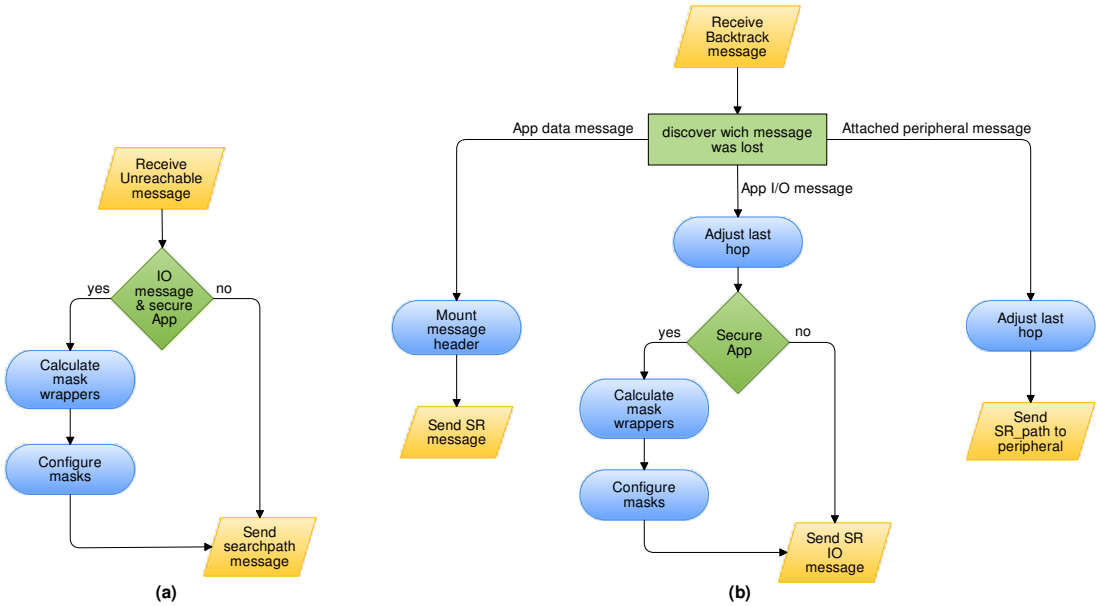


Figure 7.6: OS procedures at message received: (a) unreachable; (b) backtrack.

Figure 7.6.b shows the OS's procedure when a backtrack message arrives on a PE. The procedure's goal is to discover which condition resulted in the discard of the original message, to then resend it. The first case, (*app data message - request/delivery*) corresponds to regular messages sent by applications that need resend because reached an *OSZ*.

The second condition in Figure 7.6.b, *app I/O message*, means that an I/O message reached another *OSZ* and, therefore, the source PE must send the message again. First, the OS adjust the last hop (avoiding the message to be consumed by the local port at the target PE). Next, the OS verifies if the message belongs to an App_{sec} and, if this is true, calculate and configure the wrapper' mask to then resend the message.

Finally, the third condition (*peripheral message*), means that the an I/O message from an Peripheral attached to this router had a discarded message. In this case, the OS of the PE is responsible by dealing with the discarding, obtaining the SR path sending it to the peripheral. With the SR path the peripheral resend the I/O message to the target PE. If none peripheral is attached to the routers means that wrong unreachable/backtrack messages arrived to the PE.

A possible attack refers to a malicious peripheral that, even authenticated, sends a spoofing message during the period that the input data port is open to the I/O message. The prevention is associated to the lightweight cryptography, where the message received by the internal SP discards the message due to the wrong key used to encode the message and re-open the input port to receive the legitimate I/O message.

7.4 Protecting packet header and payload outside the OSZ

This section presents the mechanisms adopted to protect the header and the payload of the I/O messages outside the *OSZ* since they are exposed to data integrity and confidentiality attacks when leave the *OSZ*.

The *MCSoC* and peripheral share a secret key (K_e) obtained during the authentication phase (Section 5.1.2) and this key is transmitted to the PEs of the *OSZ* during the key exchange phase (Section 5.2.2). As presented below, four actions using the secret key are addressed to protect the header and the payload of I/O messages.

- Header protection using software SipHash: at the *MCSoC* side, before sending the I/O message, the OS applies the SipHash algorithm into the message header fields, excepting the second field, and insert the computed value in the last field of the message header (Figure 3.2). The second field is not considered by the SipHash because it indicates whether the message uses SR or XY routing. As the SR mechanism discard the hops already taken, the content of this flit changes in the path, and if it were included in the SipHash, the computation at the target router would be wrong.
- Payload protection using software AES: at the *MCSoC* side, before sending the I/O message, the OS applies the AES algorithm into the payload of the message;
- Payload protection using hardware AES: the hardware implementation is partially addressed into the architecture, in the context of other work in the research group [Oliveira et al., 2018], with an AES module attached between the network interface module and the local router port;
- Payload protection using hardware SIMON: the hardware implementation is partially addressed into architecture, in the context of other work in the research group [Silva, 2018];

The SipHash [Aumasson and Bernstein, 2012] and AES [NIST, 2001] software implementations are available as library modules of the OS. The execution of both algorithms occurs transparently for the App_{sec}. After assembling the message header and before sending the I/O message, the OS verifies whether it belongs to an App_{sec} or not to decide sending it without protection or to apply both protection mechanisms.

Although the AES cannot be considered a lightweight cryptography module, we use it for comparison purposes. The wrappers' configuration messages do not need SipHash/AES protection, because these messages do not leave the *OSZ* and therefore are not exposed to attacks.

7.5 Peripheral Access Evaluation

This section evaluates the peripheral access, specifically the wrapper configuration mechanism that enables the communication of PEs inside the *OSZ* with peripherals. The evaluation uses a clock-cycle accurate RTL SystemC description of the *MCSoC*.

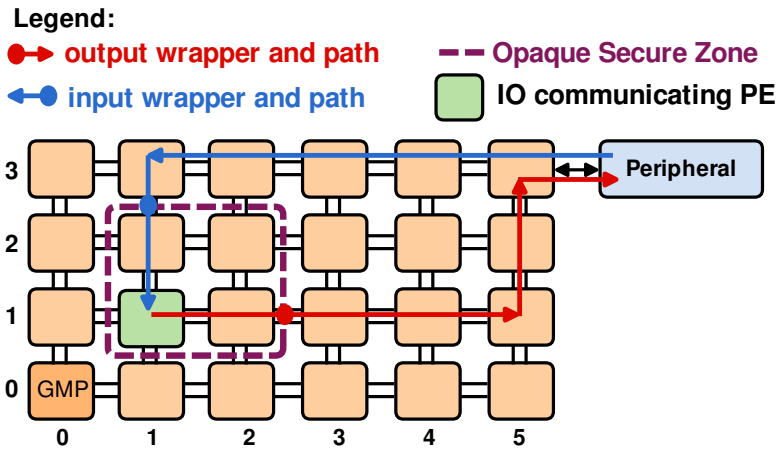


Figure 7.7: Example scenario for I/O evaluation.

The first experiment evaluates the performance impact using the proposed mechanism to communicate with I/O devices. The experiment uses the platform presented in Figure 7.7 and consider five scenarios:

(1) baseline: an application executing I/O communication without *OSZ*;

(2) *OSZ* activated, SPs {(1,1), (2,1), (1,2), (2,2)}, enabling to evaluate the impact of the wrappers' configuration through messages (worst-case);

(3) *OSZ* activated with a second *OSZ* (SPs {(4,1),(4,2)}), blocking the output I/O messages from SP (1,1), requiring the *searchpath* service and the reroute mechanism to retransmit the output message;

(4) *OSZ* activated with a second *OSZ* (SPs {(4,2),(4,3)}), blocking the input I/O messages from the peripheral, requiring the *searchpath* service and the reroute mechanism to enable the peripheral to retransmit the message;

(5) *OSZ* activated with a second *OSZ* (SPs {(4,1),(4,2),(4,3)}, blocking both output and input I/O messages, requiring two *searchpath* services and reroute mechanisms to correct delivery the I/O message.

As the goal of this experiment is to evaluate the performance of wrapper configuration and retransmission mechanism for I/O messages, the header and payload protection, with MAC and cryptography respectively, are disabled.

In all scenarios, SP (1,1) runs a task with 50 iterations, communicating with the peripheral at each iteration. Table 7.2 presents for each scenario the App_{sec} execution time in the second column (in clock cycles), and the average overhead per iteration in the third column (in clock cycles) according to the simulated scenario.

Table 7.2: Overhead to Communicate with I/O devices.

Scenario	Clock Cycles (CC)	Average Overhead CC/iteration
(1) I/O	283,7K	-
(2) I/O + *OSZ*	317,7K	680.7K
(3) I/O + *OSZ* + output rerouting	320,9K	744.4K
(4) I/O + *OSZ* + input rerouting	321,8K	763.0K
(5) I/O + *OSZ* + rerout. both dir.	326,5K	856.6K

The total execution time increases from 12% up to 15%, an expected result given that the secure application is a synthetic application, executing only I/O communication. The relevant result is the one presented in the third column, the communication overhead. The overhead to configure the wrappers and find new paths corresponds to less than 900 clock cycles per I/O access. Once the path is configured, it is used for the next packets, without incurring additional overhead. Two main reasons explain this remarkable result: (*i*) a simple wrapper configuration mechanism; (*ii*) the adoption of a dedicated NoC to find the paths when rerouting is required.

The second experiment evaluates the wrappers' behavior at the *OSZ* boundary, as shown in Figure 7.7, under a DoS attack campaign promoted by a malicious task located at SP (3,3), including several periods with the wrapper opened.

The experiment showed that no malicious task message traversed the *OZS* boundary. This happens due two mechanisms, (*i*) the selective opening of the wrapper, where the opening direction discard all messages in a contrary direction (i.e., if the wrapper is enabled to send an I/O message, it discards any attempt to inject a message); (*ii*) the flag in the header field (first flit of the packet) that differentiate the messages from I/O API from the task communication API.

The third experiment evaluates the cost to protect the message header using a MAC based on SipHash. Before sending an I/O message the SipHash value is computed over the message header (except the second field) and appended at the last two fields. During the I/O message reception, the SipHash value is computed locally and compared with the received one. If a mismatch occurs the message is discarded and a retransmission request is sent. As the SipHash support is not yet addressed at the peripheral side, for performance evaluation we use another PE to send authenticated message headers, emulating

the communication with peripheral behavior. Table 7.3 presents the overhead to apply the SipHash algorithm at the PE side.

Table 7.3: Overhead to compute the SipHash on I/O messages.

Scenario	Clock Cycles (CC)	Overhead CC
Input	209	-
Input + SipHash	3221	3012
Output	246	-
Output + SipHash	2969	2723

The Input time corresponds to the interval between the OS interruption by DMNI until releasing the message payload to the application (including the computation and MAC verification, when applicable). The Output time measure starts after the wrapper configuration until after the DMNI programming (including MAC computation, when applicable).

The Input overhead is slightly higher than the Output overhead due to the local MAC computation and comparison, that validates the received message. Even with the SipHash software implementation, the overhead corresponds to about 3000 clock cycles (3 μseg@ 100MHz).

The fourth experiment evaluates the cost to protect the I/O message' payload through a software implementation of AES using ECB mode and 128 bits of key size. The ECB mode presents a risk regarding the repetition of messages that could lead to plaintext attacks.

The I/O message encryption occurs splitting the message payload in blocks of 16 bytes and applying the AES algorithm before send the message to the peripheral. At the reception, after the DMNI interruption and before delivery the message to the application, the decryption occurs splitting the received payload and applying the AES algorithm.

The AES uses a self-contained implementation provided by B. Conte [Conte, 2018] without any cross-licensing. The time spent to encrypt or decrypt the I/O messages are similar and scale with the payload length. The average value obtained during the AES execution is 7000 clock cycles per block. Considering an I/O message with just 100 bytes result in an overhead of approximately 49K clock cycles.

The number of clock cycles per block represents a considerable overhead, showing the nonapplicability of a software implementation of the AES algorithm.

A hardware implementation of the AES [Hemanth, 2004] using the EBC operation mode was integrated to a similar *MCSoC* architecture (single 32 bits channel NoC, no rerouting and I/O support) [Oliveira et al., 2018]. The proposed experiment uses one AES module placed between the DMNI and the router ports, to process both operations, encryption and decryption. Once the data is loaded, after 13 clock cycles, the data (encrypted/decrypted) is available. The AES module area evaluation uses ST Microelectronics

standard-cells and CMOS 65nm technology, with a silicon area corresponding to twice the router area (8-flit buffer).

A serialized version of SIMON lightweight cryptography hardware module [Beaulieu et al., 2013] was also implemented by the research group [Silva, 2018]. The work shows a latency of 140 and 141 clock cycles to encrypt and decrypt a block, respectively. The SIMON module area evaluation uses ST Microelectronics standard-cells and CMOS 65nm technology in the synthesis, and the results present an area 5 times smaller than AES implementation, equivalent to less than half of the router area.

7.6 Final Remarks

This chapter presented the mechanisms and protocols to enable communication with peripherals from a secure application running inside an *OSZ*. We deal with two main issues: (*i*) how to permits I/O communication from/to *OSZ*, presenting the protocol to define access points in the border of the *OSZ* (original Thesis contribution) that selectively open specific ports to I/O messages pass through the *OSZ*' wrapper borders ([Caimi and Moraes, 2019]) and, (*ii*) the protection of I/O messages outside of *OSZ* through header protection using MAC and, payload protection, partially address, using AES (software and hardware implementation) and SIMON (hardware implementation).

The evaluation results show a low impact on the wrappers configuration in all scenarios, less than 900 clock cycles per I/O access. Even in the presence of blocked paths, there is a minimum effect over the I/O communication. This happens because the SP always use the last valid path to send the message. This way, after finding the new path using the searchpath mechanism, following messages make use of this SR path, decreasing the impact of a broken path.

The software mechanism using SipHash to protect the header' message outside the *OSZ* presents an acceptable cost of less than 3 μseg@ 100MHz. The software AES implementation shows an impracticable cost about 7K clock cycles to encrypt or decrypt a single block (128 bits). The hardware AES module implementation presents adequate performance but penalizes the silicon area overhead. Otherwise, the hardware SIMON module implementation shows a low area overhead and smaller performance when compared to AES implementation.

The cryptographic hardware modules implementation illustrates an open design space exploration to select a suitable encryption module that combines latency, throughput and area.

8. CONCLUSION AND FUTURE WORK

In this Thesis, we introduced the following statement: *It is feasible to develop a protocol to secure applications, including the application admission, runtime isolation of computation and communication resources, and access to external devices (memory and peripherals). Compared to other works available in the literature, the proposed protocol presents a smaller area, smaller communication latency, and execution time overhead.*

Regarding to the first part of the statement, this Thesis proposed a set of protocols and mechanisms to protect the execution of applications in *MCSoCs* during its three lifetime phases: Admission, Execution and Peripheral Access. This Thesis did not have the intention to cover all threats and attacks that the *MCSoCs* are exposed. Due to the *MCSoCs* complexity the entire protection, considering hardware and software, is a large task, without a silver bullet to solve the problem.

With respect to security in the *application admission*, the many-core must trust the entity transmitting the application and the integrity of the application must be verified to avoid the insertion of malicious code. The Thesis addressed these problems with two mechanisms: (*i*) the mutual authentication between external entities (as peripherals and applications' injectors) and the *MCSoC*, obtaining at the end of the process a shared key (K_e); (*ii*) the application' tasks integrity guarantee, using a software-based MAC verification with the SipHash algorithm.

At *execution time*, a malicious attacker may have access to sensitive computation or communication data and, therefore, a secure application that processes sensitive data will have its security harmed by a malicious process. The Thesis addressed the security problem during the application execution using Opaque Secure Zones (*OSZ*), a mechanism that avoids resource sharing by reserving computation and communication resources to exclusive use of the secure application. An algorithm was proposed to define the *OSZ* shape and location according to the resources required by the application and available at the *MCSoC*. The communication isolation occurs using wrappers at control signals of the NoC routers. To deal with lost messages and broken paths due to the *OSZ* activation, a search-path mechanism and message rerouting using Source Routing (SR) were provided.

Relative to the *communication with external devices*, the issues refers to enable the access to peripherals from the *OSZ* and the risk of unauthorized access to instructions and data during the exchange or, even the tampering of the message header during the message travel outside to the *OSZ*. The Thesis proposed a hardware module that manages the wrappers masking the access of I/O packets selectively to/from peripherals, maintaining the *OSZ* protection. The security of I/O messages outside of *OSZ* encompass header protection using MAC and payload protection, partially addressed the latter, using AES (software and hardware implementation) and SIMON (hardware implementation).

As presented in Chapter 1, resource sharing introduces vulnerabilities to the applications running on it. Regarding the security principles and the application lifetime phases, vulnerabilities include integrity, confidentiality, availability, authentication, and authorization.

It is possible to explore such vulnerabilities with attacks that compromise the system such as DoS, DDoS, timing attack, hardware trojan, spoofing, hijacking, man-in-the-middle, trojan horse, and backdoor.

As presented, the Thesis addressed these vulnerabilities and attacks at different phases of the application. In the following, we discuss the mechanisms adopted to prevent and mitigate these vulnerabilities, with Tables 1.1 and 1.2 as a guideline for the discussion.

At *application admission*, the vulnerabilities addressed by the Thesis are the integrity, authentication, authorization, and partially the availability. The mutual authentication (based on ECDH) and the MAC (based on SipHash) attached to the task's object code deal with these issues.

A MitM attack can occur if a malicious agent is placed between the *MCSoC* and an external entity. This attack is prevented by the proposed mutual authentication mechanism. Initially, the public keys of *MCSoC* and external entity are exchange using a secure channel and, during the authentication the keys and nonce values are used to guarantee the correct authentication of each one, preventing the MitM attack. At the end of the mutual authentication, a common key K_e is obtained.

To prevent integrity attacks (such as trojan horse, backdoor, hijacking, and spoofing), while transferring the task's source code to the *MCSoC*, the k_e is used as the key to the SipHash MAC algorithm. Thus, the external entity previously authenticated compute and attach a MAC to each task' object-code. At *MCSoC* side the MAC object-code is verified before proceed with the execution, preventing the insertion of malicious code to execute in *MCSoC*and, also guarantee that only external entities with authorization deploy the secure application in *MCSoC*.

The availability is partially addressed during the admission due to the communication model based on a master-slave mechanism where the external entity acts in slave mode. Thus, DoS or DDoS attacks originated from external entities are mitigated, although the traffic still exposed to other concurrent flows from others PEs into *MCSoC*.

At the *application execution*, the vulnerabilities addressed by the Thesis are the availability, confidentiality, integrity and authorization (Table 1.1). The application isolation using Opaque secure Zones (*OSZ*) is the mechanism used to prevent these issues. As presented, when an *OSZ* is active, data messages do not cross its borders. Only I/O data traffic can traverse, but just when the App_{sec} running inside the *OSZ* communicates with a peripheral. This I/O traffic use specif *OSZ* boundary ports, with a specific direction, and only one message in each direction is enabled.

Attacks to the availability (DoS and DDoS) are prevented due to the isolation provided by the *OSZ*, once none external data traffic crosses the internal limits of the *OSZ* what prevents explore router congestion or router starvation during the App_{sec} execution. In the same way, the prevention against confidentiality is guaranteed by the *OSZ* mechanism. Timing attacks are prevented since no time inferences can be taken from messages inside *OSZ* since none external data message can affect the router occupancy. Hardware trojan is prevented since none data traffic crosses the *OSZ* boundary due to the wrappers activation, avoiding data leakage.

The App_{sec} data integrity is also addressed by *OSZ* mechanism once, even that only applications deployed by authenticated entities runs on it, none data message outgoing the *OSZ* even when the application run in it tries to send data to a PE outside the *OSZ*. In other words, even if an application deployed by a legitimate entity tries to forge a target address outside the *OSZ* the data is not delivered due to the wrappers activation. In the same way, no external application has the authorization to send messages to the *OSZ* without previous authorization.

At the *I/O access*, the vulnerabilities addressed by the Thesis are the confidentiality, integrity, authentication, and partially the availability. The mutual authentication of peripherals with the *MCSoC*, the MAC attached to the header of I/O messages, the encryption of the payload in I/O messages, the master/slave communication model and the selective opening of the *OSZ* boundaries are the mechanisms implemented to deal with these issues.

The availability is partially addressed during the I/O access due to master/slave mechanism where the external entity acts in slave mode. Attacks like DoS or DDoS originated from external entities are mitigated, although the traffic still exposed to other flows from others PEs into *MCSoC*.

Confidentiality attacks that explore the data leakage are addressed by the encryption of the I/O message' payload, such way that only the external entity with the K_e key is able to obtain the plaintext of the original message. The integrity of the I/O message is obtained by the encryption of the payload and the MAC verification of header provided by SipHash algorithm. The authorization is addressed by selective opening of ports in *OSZ* bordering and the master/slave communication mode used on peripheral data exchange.

The second part of the Thesis statement present: *Compared to other works available in the literature, the proposed protocol presents a smaller area, smaller communication latency, and execution time overhead.*

The hardware support to deal with the vulnerabilities addressed by the Thesis includes the *brNoC*, the wrappers, the Wrapper Mask Logic (WML) module and the encryption modules (AES and Simon).

The *brNoC* and the baseline data NoC routers are synthesized using the ST Microelectronics standard-cells and CMOS 65nm technology. The baseline router has flit size

equal to 16 bits, two channels per link, buffer depth equal to 8, and support to XY and source routing. The 4 and 8 rows versions of *brNoC* internal CAMs represent 20.49% and 30.86% of the baseline router (data NoC), respectively. This is a remarkable value, indicating that the 4 row version is equivalent to one port of the baseline router (data NoC uses mesh topology supporting 5 ports).

The AES and SIMON encryption modules area evaluation uses the same ST technology in the synthesis. The AES module presented a silicon area corresponding to twice the data NoC router area (8-flit buffer). The SIMON module area evaluation presents an area 5 times smaller than AES implementation, equivalent to less than half of the data NoC router area.

The WML module and the wrappers were not synthesized, however the estimate area is smaller than the 4 row *brNoC* router due to the modules uses only three registers with 10 bits each one (1 wrapper and 2 mask registers), one 8 depth flit buffer and a control FSM unit.

Considering the 4-row *brNoC* router, the SIMON module, the WML with wrapper registers, the total area overhead is smaller than the area of the data NOC router. In perspective, the data NoC router corresponds roughly to 10% of the PE area in the baseline platform. In other words, the hardware area overhead of the mechanism implemented in the Thesis is equivalent to 10% of the baseline platform. The literature presents overheads between 9.9% and 233% to deal with just one application lifetime phase.

Regarding the communication latency overhead induced by the proposed mechanisms, two applications lifetime phases are affected: (*i*) the admission, due to MAC computation and verification; (ii) I/O access due to the wrapper mask configuration, header and payload protection by MAC and encryption respectively.

The first overhead induces only a delay to start the secure application and does not impact the application execution time. As shown, the impact is less than 33K clock cycles per Kbyte of the tasks' source code. This overhead occurs once and does not impact on the application execution time. In fact, due to the isolation mechanism promoted by the *OSZ*, the inter-task latency corresponds the best-case performance since there is no congestion inside the *OSZ*.

The second overhead affects the communication latency. The wrapper mask configuration requires about 680 clock cycles per I/O access. Evaluation with a synthetic application showed an overhead up to 15% due to the need to open the *OSZ* to the I/O communication. The header protection of I/O messages causes communication latency about 3K clock cycles per I/O message due to the MAC calculation and verification. The effective percentage overhead depends on the number of accesses performed by the application during its execution. The payload protection of I/O messages also causes communication overhead. The hardware implementation is partially addressed and shows small overhead

to encrypt/decrypt the message, about 13 clock cycles for AES and 140 clock cycles for SIMON.

Regarding the execution time overhead presented by the reviewed works in the Table 2.3, at the execution phase the execution time overhead is up to 35.8%, without taking into account the peripherals access (not supported in the state-of-the-art works). Our work presents an execution time overhead that is a function of the amount of I/O accesses. As applications run without interference inside *OSZ*, the execution time corresponds to the application running alone in the *MCSoC* (no overhead). The execution overhead occurs only due to delay induced by MAC and encryption applied to the I/O messages.

8.1 Future Work

As a guideline for future work, this Thesis has room for improvements as follows:

- incorporate to the baseline platform a RISC-V processor module enabling the use of standards benchmarks and delivering more computational power to the applications and O.S.;
- extent the authentication protocol to support the common key (K_e) renew between the peripherals and the *MCSoC* enabling full session key support;
- improve the Task Allocation and MAC verification phase making parallel the reception of the task object code with the MAC calculation;
- explore others MAC algorithms, selecting a suitable algorithm considering area and performance to implement a hardware module with the selected MAC;
- full hardware support to payload protection trough the implementation of a suitable cryptographic module in the baseline platform;
- provide a open standard I/O interface, such as AMBA or Wishbone, at the borders of *MCSoC* enabling the isolation of internal NoC signals to the peripheral. This feature increases the *MCSoC* security once hidden the internal signaling used, hindering attacks started on malicious peripherals;
- implement a peripheral request manager with arbitration functionality, enabling simultaneous requests to device and with capacity to deal with the unreachable messages previously sent, including the retransmission using SR;
- explore the use of Software Defined Networks (SDNs) in the *MCSoC*, as proposed by [Ruaro et al., 2019], to communication with peripherals. Potentially, this technique avoid the use of crytographic modules to peripheral communication since the traffic is isolated by dedicated paths between the source PE and the peripheral.

8.2 Summary of the publications produced during the Thesis

Table 8.1 presents the summary of the publications produced during the Thesis period, linking each one with the related Thesis Chapter .

Table 8.1: Summary of Publications.

Publication Reference	Relationship with the Thesis
Wachter, E.; Caimi, L.; Fochi, V.; Munhoz, D.; Moraes, F. **BrNoC: a Broadcast NoC for Control Messages in Many-core Systems**. Microelectronics Journal, Volume 68, October 2017, Pages 69–77.	Chapter 3 - development of the BrNoC
Ruaro, M.; Caimi, L.; Fochi, V.; Moraes, F. **A Framework for Heterogeneous Many-core SoCs Generation** In: LASCAS, 2019	Chapter 3 - Peripheral support
Ruaro, M.; Caimi, L.; Fochi, V.; Moraes, F. **Memphis: a Framework for Heterogeneous Many-core SoCs Generation and Validation**. Design Automation for Embedded Systems Journal, Accepted: 8 August 2019.	Chapter 3 - Peripheral support
Caimi, L.; Fochi, V.; Wachter, E.; Moraes, F. **Secure Admission and Execution of Applications in Many-core Systems** In: SBCCI, 2017.	Chapter 4 - General view of Thesis proposal
Caimi, L.; Fochi, V.; Wachter, E.; Moraes, F. **Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications** In: LASCAS, 2018	Chapter 5 - *OSZ* creation
Caimi, L.; Fochi, V.; Moraes, F. **Secure Admission of Applications in Many-Cores** In: ICECS, 2018	Chapter 5 - App_{sec} Admission
Caimi, L.; Fochi, V.; Wachter, E.; Munhoz, D.; Moraes, F. **Activation of Secure Zones in Many-Core Systems with Dynamic Rerouting** In: ISCAS, 2017, pp. 144-147	Chapter 6 - App_{sec} Execution
Caimi, L.; Moraes, F. **Activation of Secure Zones in Many-Core Systems with Dynamic Rerouting** In: ISVLSI, 2019	Chapter 7 - Peripheral access from 0*SZ*
Fochi, V.; Caimi, L.; Ruaro, M.; Wachter, E.; Moraes, F. **System Management Recovery Protocol for MPSoCs**. In: SOCC, 2017, pp. 367-374.	Publications not directly related to the Thesis subject - use of the BrNoC and peripheral support in the fault tolerance domain
Fochi, V.; Caimi, L.; Silva, M.; Wachter, E.; Moraes, F. **Fault-tolerance at the Management Level in Many-core Systems** In: SBCCI, 2018	

REFERENCES

[Ancajas et al., 2014] Ancajas, D. M., Chakraborty, K., and Roy, S. (2014). Fort-NoCs: Mitigating the threat of a compromised NoC. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[ARM, 2013] ARM (2013). big.LITTLE Technology: The Future of Mobile. Technical report, ARM Limited.

[ARM, 2018] ARM (2018). ARM Security Technology: Building a Secure System using TrustZone Technology . Source: http://infocenter.arm.com, Nov. 2018.

[Aumasson and Bernstein, 2012] Aumasson, J.-P. and Bernstein, D. J. (2012). SipHash: a fast short-input PRF. In *International Conference on Cryptology (INDOCRYPT)*, pages 489–508.

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

[Baron et al., 2013] Baron, S., Wangham, M. S., and Zeferino, C. A. (2013). Security mechanisms to improve the availability of a Network-on-Chip. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 609–612.

[Beaulieu et al., 2013] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. (2013). The simon and speck families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404.

[Beaulieu et al., 2015] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. (2015). The SIMON and SPECK Lightweight Block Ciphers. In *ACM/IEEE Design Automation Conference (DAC)*, pages 175:1–175:6.

[Benini and Micheli, 2002] Benini, L. and Micheli, G. (2002). Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78.

[Bernstein et al., 2015] Bernstein, D. J., van Gastel, B., Janssen, W., Lange, T., Schwabe, P., and Smetsers, S. (2015). TweetNaCl: A Crypto Library in 100 Tweets. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 64–83.

[Bjerregaard and Mahadevan, 2006] Bjerregaard, T. and Mahadevan, S. (2006). A Survey of Research and Practices of Network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1–51.

[Bogdanov et al., 2007] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J., Seurin, Y., and Vikkelsoe, C. (2007). Present: An ultra-lightweight block cipher. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 450–466.

[Bohnenstiehl et al., 2016] Bohnenstiehl, B., Stillmaker, A., Pimentel, J., Andreas, T., Liu, B., Tran, A., Adeagbo, E., and Bass, B. (2016). A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *IEEE Symposium on VLSI Circuits (VLSIC)*, pages 1–2.

[Boraten and Kodi, 2016] Boraten, T. and Kodi, A. K. (2016). Packet security with path sensitization for NoCs. In *Design, Automation Test in Europe Conference (DATE)*, pages 1136–1139.

[Caimi et al., 2018a] Caimi, L. L., Fochi, V., and Moraes, F. G. (2018a). Secure Admission of Applications in Many-Cores. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 761–764.

[Caimi et al., 2018b] Caimi, L. L., Fochi, V., Wachter, E., and Moraes, F. (2018b). Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4.

[Caimi et al., 2017a] Caimi, L. L., Fochi, V., Wachter, E., Munhoz, D., and Moraes, F. G. (2017a). Activation of Secure Zones in Many-core Systems with Dynamic Rerouting. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 144–147.

[Caimi et al., 2017b] Caimi, L. L., Fochi, V., Wachter, E., Munhoz, D., and Moraes, F. G. (2017b). Secure Admission and Execution of Applications in Many-core Systems. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 65–71.

[Caimi and Moraes, 2019] Caimi, L. L. and Moraes, F. (2019). Security in Many-Core SoCs Leveraged by Opaque Secure Zones. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 471–476.

[Canniere and Preneel, 2006] Canniere, C. D. and Preneel, B. (2006). Trivium. Source: http://www.ecrypt.eu.org/stream/e2-trivium.html, Nov. 2018.

[Carara et al., 2009] Carara, E. A., de Oliveira, R. P., Calazans, N. L. V., and Moraes, F. G. (2009). HeMPS - a framework for NoC-based MPSoC generation. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1345–1348.

[Castilhos et al., 2013] Castilhos, G., Mandelli, M., Madalozzo, G., and Moraes, F. (2013). Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 153–158.

[Certicom Research, 2009] Certicom Research (2009). SEC 1. Standards for Efficient Cryptography Group: Elliptic Curve Cryptography. Source: http://www.secg.org/sec1-v2.pdf, Dec. 2018.

[Churchhouse, 2002] Churchhouse, R. (2002). *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press, pages 240.

[Coburn et al., 2005] Coburn, J., Ravi, S., Raghunathan, A., and Chakradhar, S. (2005). SECA: Security-enhanced Communication Architecture. In *IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 78–89.

[Conte, 2018] Conte, B. (2018). Crypto-algorithms: AES. Source: https://github.com/B-Con/crypto-algorithms, Nov. 2018.

[Dinechin et al., 2014] Dinechin, B. D. D., Amstel, D. V., Poulhiès, M., and Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *Design, Automation Test in Europe Conference (DATE)*, pages 1–6.

[Dobraunig et al., 2018] Dobraunig, C., Mendel, F., and Schläffer, M. (2018). Differential Cryptanalysis of SipHash. Source: http://eprint.iacr.org/2014/722, Nov. 2018.

[Esmaeilzadeh et al., 2012] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2012). Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134.

[Evain and Diguet, 2005] Evain, S. and Diguet, J. P. (2005). From NoC security analysis to design solutions. In *IEEE Workshop on Signal Processing Systems Design and Implementation (SiPS)*, pages 166–171.

[Fernandes et al., 2016] Fernandes, R., Marcon, C., Cataldo, R., Silveira, J., Sigl, G., and Sepúlveda, J. (2016). A security aware routing approach for NoC-based MPSoCs. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

[Fletcher et al., 2012] Fletcher, C. W., Dijk, M., and Devadas, S. (2012). A secure processor architecture for encrypted computation on untrusted programs. In *ACM Workshop on Scalable Trusted Computing (WSTC)*, pages 3–8.

[GAPH, 2018] GAPH (2018). Hardware design support group. Source: www.inf.pucrs.br/gaph/, Nov. 2018.

[Gebotys and Gebotys, 2003] Gebotys, C. H. and Gebotys, R. J. (2003). A framework for security on NoC technologies. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 113–117.

[Girão et al., 2011] Girão, G., Barcelos, D., and Wagner, F. R. (2011). Performance and Energy Evaluation of Memory Organizations in NoC-Based MPSoCs under Latency and Task Migration . In *IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 56–80.

[Grammatikakis et al., 2014] Grammatikakis, M. D., Papadimitriou, K., Petrakis, P., Papagrigoriou, A., Kornaros, G., Christoforakis, I., and Coppola, M. (2014). Security Effectiveness and a Hardware Firewall for MPSoCs. In *IEEE High Performance Computing and Communications (HPCC)*, pages 1032–1039.

[Grammatikakis et al., 2015a] Grammatikakis, M. D., Papadimitriou, K., Petrakis, P., Papagrigoriou, A., Kornaros, G., Christoforakis, I., Tomoutzoglou, O., Tsamis, G., and Coppola, M. (2015a). Security in MPSoCs: A NoC Firewall and an Evaluation Framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1344–1357.

[Grammatikakis et al., 2015b] Grammatikakis, M. D., Petrakis, P., Papagrigoriou, A., Kornaros, G., and Coppola, M. (2015b). High-level security services based on a hardware NoC Firewall module. In *Workshop on Intelligent Solutions in Embedded Systems (WISES)*, pages 73–78.

[Gupta et al., 2002] Gupta, V., Gupta, S., Chang, S., and Stebila, D. (2002). Performance Analysis of Elliptic Curve Cryptography for SSL. In *ACM Workshop on Wireless Security (WiSe)*, pages 87–94.

[Hanka and Wippel, 2011] Hanka, O. and Wippel, H. (2011). Secure deployment of application-tailored protocols in future networks. In *International Conference on the Network of the Future (NoF)*, pages 10–14.

[Hankerson et al., 2003] Hankerson, D., Menezes, A. J., and Vanstone, S. (2003). *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., pages 332.

[He et al., 2012] He, D., Chen, J., and Chen, Y. (2012). A secure mutual authentication scheme for session initiation protocol using elliptic curve cryptography. *Security and Communication Networks*, 5(12):1423–1429.

[Hemani et al., 2000] Hemani, A., Jantsch, A., Kumar, S., Postula, A., Öberg, J., Millberg, M., and Lindqvist, D. (2000). Network on chip: An architecture for billion transistor era. In *Nordic Circuits and Systems Conference (NORCHIP)*, page 8.

[Hemanth, 2004] Hemanth (2004). AES crypto core. Source: https://opencores.org/projects/aes_crypto_core, Dec. 2018.

[Hoffman et al., 2015] Hoffman, C., Cortes, M., Aranha, D. F., and Araujo, G. (2015). Computer security by hardware-intrinsic authentication. In *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 143–152.

[Hu et al., 2015] Hu, Y., Müller-Gritschneder, D., Sepulveda, M. J., Gogniat, G., and Schlichtmann, U. (2015). Automatic ILP-based Firewall Insertion for Secure Application-Specific Networks-on-Chip. In *Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC)*, pages 9–12.

[Intel, 2018] Intel (2018). Intel core i9-7980xe extreme edition processor. Source: https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-7980xe.html, Nov. 2018.

[Isakovic and Wasicek, 2013] Isakovic, H. and Wasicek, A. (2013). Secure channels in an integrated MPSoC architecture. In *Industrial Electronics Society (IECON)*, pages 4488–4493.

[ISO/IEC , 2012] ISO/IEC (2012). Trivium standard ISO/IEC 29192-3. Source: http://www.ecrypt.eu.org/stream/e2-trivium.html, Nov. 2018.

[Jacob et al., 2017] Jacob, N., Heyszl, J., Zankl, A., Rolfes, C., and Sigl, G. (2017). How to break secure boot on FPGA socs through malicious hardware. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 425–442.

[Katz and Lindell, 2014] Katz, J. and Lindell, Y. (2014). *Introduction to Modern Cryptography*. Chapman & Hall/CRC, pages 512.

[Khernane et al., 2016] Khernane, N., Potop-Butucaru, M., and Chaudet, C. (2016). BANZKP: a Secure Authentication Scheme Using Zero Knowledge Proof for WBANs. In *International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 307–315.

[Kinsy et al., 2017] Kinsy, M. A., Khadka, S., Isakov, M., and Farrukh, A. (2017). Hermes: Secure heterogeneous multicore architecture design. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 14–20.

[Koblitz, 1987] Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209.

[Kuntze, 2013] Kuntze, N.; Rudolph, C. (2013). Secure deployment of SmartGrid equipment. In *IEEE Power Energy Society General Meeting (PESGM)*, pages 1–5.

[Linder and Harden, 1991] Linder, D. H. and Harden, J. C. (1991). An Adaptive and Fault Tolerant Wormhole Routing Strategy for k-ary n-cubes. *Transactions on Computer*, 40(1):2–12.

[Mandelli et al., 2011] Mandelli, M., Amory, A., Ost, L., and Moraes, F. G. (2011). Multi-task Dynamic Mapping Onto NoC-based MPSoCs . In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 191–196.

[Mellanox Tecnhlogies, 2018] Mellanox Tecnhlogies (2018). TILE-Gx72 Processor Overview. Source: http://www.mellanox.com/page/products_dyn?product_, Nov. 2018.

[Moraes et al., 2004] Moraes, F. G., Calazans, N., Mello, A., Möller, L., and Ost, L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69 – 93.

[Mouha et al., 2014] Mouha, N., Mennink, B., Herrewege, A. V., Watanabe, D., Preneel, B., and Verbauwhede, I. (2014). Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers. In *Selected Areas in Cryptography (SAC)*, pages 306–323.

[NaNoC, 2015] NaNoC (2015). The nanoc project. Source: http://www.nanoc-project.eu/, Nov. 2018.

[NASA, 2015] NASA (2015). Numerical aerodynamic simulation - nas. Source: http://www.nas.nasa.gov/publications/npb.htm, Nov. 2018.

[NIST, 2001] NIST (2001). Announcing the advanced encryption standard (aes). Source: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Nov. 2018.

[Oliveira et al., 2018] Oliveira, B., Reusch, R., Medina, H., and Moraes, F. G. (2018). Evaluating the Cost to Cipher the NoC Communication. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–4.

[Oracle, 2017] Oracle (2017). Oracle's SPARC T8 and SPARC M8 Server Architecture. Technical report, Oracle Corporation.

[Osvik et al., 2006] Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology*, pages 1–20.

[OVP, 2018] OVP (2018). Open virtual platform. Source: http://www.ovpworld.org/technology_ovpsim, Nov. 2018.

[O'donnell et al., 2004] O'donnell, C. W., Suh, G. E., and Devadas, S. (2004). Puf-based random number generation. Source: http://csg.csail.mit.edu/pubs/memos/Memo-481/Memo-481.pdf, Dec. 2018.

[Papadimitriou et al., 2015] Papadimitriou, K., Petrakis, P., Grammatikakis, M. D., and Coppola, M. (2015). Security Enhancements for building saturation-free, low-power NoC-based MPSoCs. In *IEEE Conference on Communications and Network Security (CNS)*, pages 594–600.

[Popovici et al., 2010] Popovici, K., Rousseau, F., Jerraya, A. A., and Wolf, M. (2010). *Embedded Software Design and Programming of Multiprocessor System-on-Chip: Simulink and System C Case Studies*. Springer Publishing Company, Incorporated, pages 290.

[Rajesh et al., 2015] Rajesh, J., Ancajas, D. M., Chakraborty, K., and Roy, S. (2015). Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 8:1–8:8.

[Ramachandran, 2002] Ramachandran, J. (2002). *Designing Security Architecture Solutions* . John Wiley & Sons, Inc., pages 483.

[Real et al., 2016a] Real, M. M., Migliore, V., Lapotre, V., and Gogniat, G. (2016a). ALMOS Many-Core Operating System Extension with New Secure-Enable Mechanisms for Dynamic Creation of Secure Zones. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 820–824.

[Real et al., 2018] Real, M. M., Wehner, P., Lapotre, V., Göhringer, D., and Gogniat, G. (2018). Application Deployment Strategies for Spatial Isolation on Many-Core Accelerators. *ACM Transaction on Embedded Computing Systems*, 17(2):55:1–55:31.

[Real et al., 2016b] Real, M. M., Wehner, P., Migliore, V., Lapotre, V., Göhringert, D., and Gogniat, G. (2016b). Dynamic spatially isolated secure zones for NoC-based many-core accelerators. In *Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–6.

[Reinbrecht et al., 2018] Reinbrecht, C., Forlin, B., Zankl, A., and Sepúlveda, J. (2018). Earthquake — A NoC-based optimized differential cache-collision attack for MPSoCs. In *Design, Automation Test in Europe Conference (DATE)*, pages 648–653.

[Reinbrecht et al., 2016a] Reinbrecht, C., Susin, A., Bossuet, L., and Sepúlveda, J. (2016a). Gossip NoC - Avoiding Timing Side-Channel Attacks through Traffic Management. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 601–606.

[Reinbrecht et al., 2016b] Reinbrecht, C., Susin, A., Bossuet, L., Sigl, G., and Sepúlveda, J. (2016b). Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

[Reinbrecht et al., 2017] Reinbrecht, C., Susin, A., Bossuet, L., Sigl, G., and Sepúlveda, J. (2017). Timing attack on NoC-based systems: Prime+Probe attack and NoC-based protection. *Microprocessors and Microsystems (MICPRO)*, 52(C):556–565.

[Rouget et al., 2017] Rouget, P., Badrignans, B., Benoit, P., and Torres, L. (2017). Secboot - lightweight secure boot mechanism for linux-based embedded systems on fpgas. In *Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–5.

[Ruaro et al., 2018] Ruaro, M., Caimi, L. L., Fochi, V., and Moraes, F. G. (2018). A Framework for Heterogeneous Many-core SoCs Generation. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 89–92.

[Ruaro et al., 2019] Ruaro, M., Jantsch, A., and Moraes, F. G. (2019). Self-adaptive qos management of computation and communication resources in many-core socs. *ACM Transactions on Embedded Computing Systems*, 18(4):37:1–37:21.

[Ruaro et al., 2016] Ruaro, M., Lazzarotto, F. B., Marcon, C. A., and Moraes, F. G. (2016). DMNI: A specialized network interface for NoC-based MPSoCs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1202–1205.

[Ruaro and Moraes, 2017] Ruaro, M. and Moraes, F. G. (2017). Demystifying the cost of task migration in distributed memory many-core systems. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4.

[Salloum et al., 2013] Salloum, C. E., Elshuber, M., Höftberger, O., Isakovic, H., and Wasicek", A. (2013). The ACROSS MPSoC – A new generation of multi-core processors designed for safety–critical embedded systems. *Microprocessors and Microsystems (MICPRO)*, 37(8):1020 – 1032.

[Schneier, 1996] Schneier, B. (1996). *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., pages 758.

[Sepúlveda et al., 2015a] Sepúlveda, J., Diguet, J. P., Strum, M., and Gogniat, G. (2015a). NoC-Based Protection for SoC Time-Driven Attacks. *IEEE Embedded Systems Letters (ESL)*, 7(1):7–10.

[Sepúlveda et al., 2015b] Sepúlveda, J., Flórez, D., and Gogniat, G. (2015b). Reconfigurable Group-Wise Security Architecture for NoC-Based MPSoCs Protection. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 26:1–26:6.

[Sepúlveda et al., 2015c] Sepúlveda, J., Flórez, D., and Gogniat, G. (2015c). Reconfigurable security architecture for disrupted protection zones in NoC-based MPSoCs. In *Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8.

[Sepúlveda et al., 2017] Sepúlveda, J., Flórez, D., Immler, V., G., G., and Sigl, G. (2017). Efficient security zones implementation through hierarchical group key management at NoC-based MPSoCs. *Microprocessors and Microsystems (MICPRO)*, 50:164 – 174.

[Sepúlveda et al., 2016] Sepúlveda, J., Flórez, D., Soeken, M., Diguet, J. P., and Gogniat, G. (2016). Dynamic NoC buffer allocation for MPSoC timing side channel attack protection. In *IEEE Latin American Symposium on Circuits Systems (LASCAS)*, pages 91–94.

[Sepúlveda et al., 2015d] Sepúlveda, J., Gogniat, G., Flórez, D., Diguet, J. P., Pires, R., and Strum, M. (2015d). TSV protection: Towards secure 3D-MPSoC. In *IEEE Latin American Symposium on Circuits Systems (LASCAS)*, pages 1–4.

[Sepúlveda et al., 2018] Sepúlveda, J., Willgerodt, F., and Pehl, M. (2018). SEPUFSoC: Using PUFs for Memory Integrity and Authentication in Multi-Processors System-on-Chip. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 39–44.

[Sharma et al., 2018] Sharma, G., Ellinidou, . S., Anand, R., Kuchta, V., Markowitch, O., and Dricot, J. (2018). Secure Communication on NoC based MPSoC. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*, page 12.

[Silva, 2018] Silva, C. L. (2018). Design of a lightweight cryptography module for FPGA based on Simon and Speck algorithms. Course Completion Assignment, PUCRS. Source: http://www.inf.pucrs.br/~moraes/docs/tcc/tcc_cristovam.pdf, Dec. 2018.

[Silva and Zeferino, 2017] Silva, M. R. and Zeferino, C. A. (2017). Confidentiality and Authenticity in a Platform Based on Network-on-Chip. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 225–230.

[Sodani et al., 2016] Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. C. (2016). Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46.

[Wachter et al., 2017] Wachter, E., Caimi, L. L., Fochi, V., Munhoz, D., and Moraes, F. G. (2017). BrNoC: A broadcast NoC for control messages in many-core systems. *Microelectronics Journal*, 68:69 – 77.

[Wang and Suh, 2012] Wang, Y. and Suh, G. E. (2012). Efficient Timing Channel Protection for On-Chip Networks. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 142–151.

[Wassel et al., 2013] Wassel, H. M. G., Gao, Y., Oberg, J. K., Huffmire, T., Kastner, R., Chong, F. T., and Sherwood, T. (2013). SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip. In *International Symposium on Computer Architecture (ISCA)*, pages 583–594.

[Wassel et al., 2014] Wassel, H. M. G., Gao, Y., Oberg, J. K., Huffmire, T., Kastner, R., Chong, F. T., and Sherwood, T. (2014). Networks on Chip with Provable Security Properties. *IEEE Micro*, 34(3):57–68.

[Wehner et al., 2015] Wehner, P., Rettkowski, J., Kleinschmidt, T., and Göhringer, D. (2015). MPSoCSim: An extended OVP simulator for modeling and evaluation of Network-on-Chip based heterogeneous MPSoCs. In *Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 390–395.

[Woszezenki, 2007] Woszezenki, C. (2007). Alocação de Tarefas e Comunicação entre Tarefas em MPSoCs. Master's thesis, Pontifical Catholic University of Rio Grande do Sul (PUCRS).

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br