PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER ENGINEERING

PROPOSAL OF A SECURE NETWORK INTERFACE FOR PROTECTING IO COMMUNICATION IN MANY-CORES

GUSTAVO COMARÚ RODRIGUES

Bachelor Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Fernando Gehm Moraes
Co-Advisor: Rafael Follmann Faccenda

Porto Alegre
2022

ACKNOWLEDGMENTS

I would like to use this space to express my deepest gratitude towards the people who made this work possible.

First of all, I thank my advisors, Prof. Fernando Moraes and Rafael Faccenda, for the incredible guidance they offered. Thank you for taking me into this project, and for all the help and patience during this last year. I know it wasn't easy at times, and I hugely appreciate the support you've been giving me.

I thank Prof. Luciano Caimi for also providing excellent guidance throughout this project. Thank you for the invaluable insights and expertise.

I also thank my family, specially my parents Raquel and Eduardo, for the constant support they gave me during my graduation. Without them, none of this would've been possible in the first place.

Finally, I thank FAPERGS for financing my participation in this research, thus enabling me to pursue this work.

PROPOSTA DE UMA INTERFACE DE REDE SEGURA PARA PROTEÇÃO DE COMUNICAÇÃO DE E/S EM MANY-CORES

RESUMO

Os sistemas de múltiplos núcleos em um único chip (*many-cores*) são plataformas projetadas para fornecer alto desempenho através do paralelismo, atendendo a demanda atual de dispositivos embarcados com restrições de consumo de energia e comunicação. Um *many-core* contém elementos de processamento (PEs – Processing Elements) interligados por infraestruturas de comunicação complexas, como redes intra-chip (NoC – *Networks-on-Chip*). Interfaces de rede (NI – *network interface*) conectam PEs aos roteadores da NoC. À medida que a adoção e a complexidade dos *many-cores* aumentam, a proteção de dados aparece como um requisito de projeto. Esses sistemas lidam com informações confidenciais. Assim, é necessário proteger esses dados contra acessos não autorizados. A literatura apresenta técnicas de segurança como: criptografia, códigos de autenticação, códigos de correção de erros, criação de um perfil da comunicação para detectar comportamentos anômalos. Tais mecanismos de defesa buscam proteger o *many-core* de algum ataque específico, carecendo de propostas que protejam o sistema contra um conjunto mais abrangente de ataques. Zona Segura Opaca (OSZ – *Opaque Secure Zone*) é um mecanismo de defesa realizado em tempo de execução que busca encontrar uma região retilínea com PEs livres para mapear uma aplicação com restrições de segurança. A OSZ impede ataques de fontes externas, como negação de serviço (DoS – *Denial-of-Service*), ataque de temporização, *spoofing*, *man-in-the-middle*. Embora o método seja robusto contra ataques externos, ele ainda apresenta vulnerabilidades quando a aplicação executado na OSZ precisa se comunicar com periféricos externos. Este trabalho complementa o mecanismo de segurança OSZ através da proposta de uma Interface de Rede Segura (SNI – *Secure Network Interface*) para proteger a comunicação entre aplicações e dispositivos de E/S. Impondo o modelo de comunicação mestre-escravo e implementando um protocolo de autenticação leve, a SNI defende o sistema de ataques *spoofing* e *denial-of-service* envolvendo periféricos.

Palavras-Chave: Sistemas multi-núcleos baseados em redes intra-chip, segurança, OSZ (zonas seguras opacas), comunicação segura, NI (network interface), periféricos.

PROPOSAL OF A SECURE NETWORK INTERFACE FOR PROTECTING IO COMMUNICATION IN MANY-CORES

ABSTRACT

Many-cores are platforms designed to provide high-performance through the use of parallelism, meeting the current demand of embedded devices with power consumption and communication constraints. A many-core contains PEs (Processing Elements) interconnected by complex communication infrastructures, such as NoCs (Networks-on-Chip). Network Interfaces (NI) connect PEs to the routers of the NoC. As the adoption and complexity of many-cores increase, data protection appears as a design requirement. These systems handle sensitive information. Thus, it is necessary to protect this data from unauthorized access. The literature presents security techniques, such as cryptography, authentication codes, error correction codes, creation of a communication flow profile to detect anomalous behavior. These defense mechanisms seek to protect the many-core from a given attack, lacking proposals protecting the system against the plethora of possible threats. The Opaque Secure Zone (OSZ) is a defense mechanism executed at runtime that focuses on finding a rectilinear region with free PEs to map an application with security constraints. OSZ prevent attacks from outside sources, such as Denial-of-Service (DoS), timing attack, spoofing, man-in-the-middle. Even though the method is robust against external attacks, it still presents vulnerabilities when the application running in the OSZ needs to communicate with external peripherals. This work complements the OSZ security mechanism by proposing a Secure Network Interface (SNI) to protect the communication between applications and IO devices. By enforcing a master-slave communication model and implementing a lightweight authentication protocol, the SNI protects the system from spoofing and flooding attacks involving the peripheral.

Keywords: NoC-based many-cores, security, OSZ (opaque secure zones), secure communication, NI (network interface), peripherals.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. INTRODUCTION

Many-cores are platforms designed to provide high-performance through the use of parallelism, meeting the current demand of embedded devices with power consumption and communication constraints. A many-core contains PEs (Processing Elements) interconnected by complex communication infrastructures, such as hierarchical buses or NoCs (Networks-on-Chip) [Popovici et al., 2010]. PEs may be processors, 3PIP (third-party intellectual property) modules, memory blocks, or dedicated hardware accelerators. Examples of modern architectures with a large number of processors interconnected by NoCs include the Mellanox family TILE-Gx72 (72 cores) [Tecnhlogies, 2018], Intel Knights Landing [Sodani et al., 2016], Oracle M8 (32 cores) [Oracle, 2017], Kalray array (256 cores) [Dinechin et al., 2014], KiloCore chip (1,000 cores) [Bohnenstiehl et al., 2016], and Esperanto (1,100 RISC-V cores) [Peckham, 2020].

An NoC consists of routers and links and is responsible for forwarding data and controlling messages between PEs. Network Interfaces (NI) connect PEs to the routers of the NoC. Whenever a PE sends a message, the NI transforms it into a packet and delivers it to the router. Then, the router sends the packet to a neighbor router through a link according to a path defined by the routing algorithm. The routers constitute the underlying communication infrastructure of the system, where multiple interconnected routers define the network topology [Hemani et al., 2000; Benini and Micheli, 2002].

As the adoption and complexity of many-cores increase, the concern for data protection appears as a new design requirement [Baron et al., 2013]. A many-core may be employed in scenarios where availability is critical and downtimes must be minimized. These systems may also handle sensitive information; thus, protecting this data from unauthorized access is necessary. According to [Ramachandran, 2002], not only data protection, unauthorized access, and availability are concerns on the many-core 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 the validity of a claimed identity;

- Authorization: the process of determining whether a validated entity is allowed to access a secured resource based on attributes, predicates, or context;
- Auditing: the property of logging the system activities at levels sufficient for the reconstruction of events;
- Nonrepudiation: the prevention of any participant denying his role in the interaction once it is completed.

A consequence of the increasing number of features and functionalities inside a single chip is the adoption of 3PIPs to meet time-to-market constraints and reduce design costs. Such IPs come from different vendors, raising the risk of having a Hardware Trojan (HT) insertion [Li et al., 2016]. Assuming HTs infecting the NoC, these can perform several attacks that threaten security principles [Ramachandran, 2002]. Such attacks may affect *confidentiality* by redirecting messages to malicious agents, *availability* by dropping messages or blocking a communication path, and *integrity* by corrupting the content of a packet traversing the NoC.

The literature presents several techniques, such as cryptography [Charles and Mishra, 2020], authentication codes [Sharma et al., 2019], error correction codes [Gondal et al., 2020], creation of a communication flow profile to detect anomalous behavior [Charles et al., 2020]. Adopting these techniques makes it possible to detect violations related to security or faults in the NoC. Moreover, the authors propose spatial isolation via Secure Zones (SZ), simultaneously protecting communication and computation. A particular case of SZ is the Opaque Secure Zone (OSZ) [Caimi and Moraes, 2019], which is a defense mechanism executed at runtime that focuses on finding a rectilinear region on the system with free PEs to map an application with security constraints. The OSZ activation occurs by setting wrappers at the boundaries of the rectilinear region, blocking all incoming and outgoing traffic trying to cross the OSZ.

OSZ prevents attacks from outside sources, such as Denial-of-Service (DoS), timing attacks, spoofing, and man-in-the-middle [Caimi and Moraes, 2019; Caimi et al., 2018a]. Even though the method is robust against external attacks, it still presents vulnerabilities when HTs infect routers inside the OSZ or when the application running in the OSZ needs to communicate with external peripherals.

1.1 Motivation

Literature related to many-core that present methods to secure communication with peripherals is scarce, with most of them focusing on shared memory protection [Grammatikakis et al., 2015; Reinbrecht et al., 2016]. On the other hand, several works present

many-cores with peripherals but without security concerns [Lee et al., 2021; Vaas et al., 2021; Jiang et al., 2021]. Therefore, there is a gap to fulfill: *how to protect the communication of applications with peripherals*?

This work focus on one of the communication endpoints: the IO device. To monitor and control the IO communication, we propose the insertion of a Secure Network Interface at the edge of the platform to enable IO devices to interact with secure applications.

1.2 Objectives

The strategic objective of this work is the creation of a Secure Network Interface that enables the connection of IO devices and the secure interaction between Peripherals and Applications. To reach the strategic objective, the following specific goals are set:

I **Analyze the Authentication protocol and Threat model**.
The first step is to understand the operation of the authentication protocol of the baseline platform alongside the threats that it must protect the system from.

II **Gathering Requirements**.
Based on the functionalities observed on objective I, define the requirements of the Secure Network Interface.

III **NI (network interface) for peripherals**.
Design and implement the Secure Network Interface, meeting the requirements from objective II.

IV **SNI validation**.
Elaborate and simulate scenarios illustrating the SNI operations.

1.3 Document Organization

This work is organized as follows. Chapter 2 presents a set of works related to the use of network interfaces as a security component. Chapter 3 presents concepts required to follow this work, including the baseline architecture, Secure Zones, and the methods used in the platform for protecting the execution of applications. Chapter 4 presents the main contribution of this work, the design of the Secure Network Interface – SNI. Chapter 5 evaluates the SNI, including the communication protocol and the answers to attacks. Chapter 6 concludes this work and points out directions for future work.

2. RELATED WORKS

This Chapter discusses works related to the use of network interfaces as a security component, aiming to protect NoC-based systems from attacks involving IO communication.

2.1 Network Adapter Architectures in Network on Chip: Comprehensive Literature Review

Aghaei et al. [Aghaei et al., 2020] present the Network Adapter (NA) as a major component of NoC-based systems, as it directly impacts critical parameters such as power, latency, throughput, and area. The Authors review the proposals of different NAs throughout the literature and evaluate the parameters which have an impact on the NA architectures.

The Network Adapter is responsible for manipulating the end-to-end flow control, encapsulating the messages or transactions generated by the cores, and relaying it to the network. It implements two interfaces – Figure 2.1: the Core Interface (CI) is a standardized point-to-point protocol allowing the IP to be reused across several platforms, and the Network Interface (NI), which encapsulates the packet, contains buffers, implements synchronization protocols and helps the router in terms of storage.

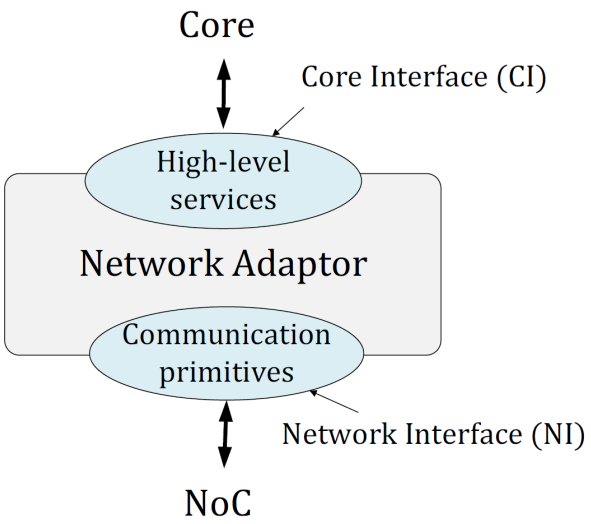


Figure 2.1 – The two interfaces implemented by the Network Adapter [Aghaei et al., 2020].

Aghaei et al. consider the security-aware design of communication architectures such as an NoC to be an increasing necessity, as their complexity may lead to new weaknesses. At the same time, the NoC itself may contribute to the security of the system by providing means for monitoring system behavior and detecting specific attacks. In their opinion, the NA is the ideal position to analyze incoming traffic and discard malicious requests.

2.2 A Security Framework for NoC Using Authenticated Encryption and Session Keys

Kapoor et al. [Kapoor et al., 2013] divide the PEs of a many-core system into two categories: secure cores, that store and process secret information that should remain uncompromised; and the non-secure cores, which may carry viruses or Hardware Trojans. The goal of their work is to protect the communication between secure and non-secure cores, avoiding the extraction of sensitive information.

To achieve this goal, the Authors propose an authentication method in which secure communication is established with a session key, issued by the secure core. This key is temporary and expires after a fixed duration of time.

The network interface of the cores, presented in Figure 2.2, has an Authenticated Encryption (AE) block. Every communication with the other IP cores passes through this block. On the sending side, the packet is encrypted using the session key, and a message authentication code (MAC) is generated. On the receiving side, the AE decrypts the message and asserts its authenticity through the MAC.

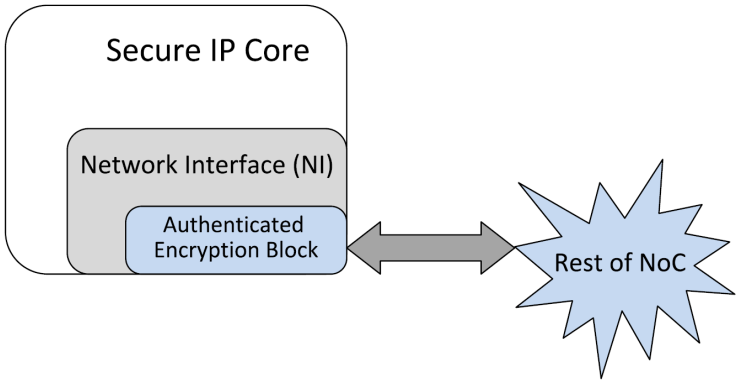


Figure 2.2 – The AE Block is embedded inside an IP core. It filters the communication with other cores [Kapoor et al., 2013].

Despite the added security mechanisms, the use of encryption and MAC increases communication latency and adds a significant area overhead. Storing a different key for each communicating pair increases the silicon area as well.

2.3 Security Mechanisms to Improve the Availability of a Network-on-Chip

Baron et al. [Baron et al., 2013] analyzed the security vulnerabilities of the SoCIN NoC-based system. They proposed a set of mechanisms to protect the Network-on-Chip

from attacks by malicious cores. After analyzing the SoCIN system, they presented a threat model composed of four attacks.

1. **Masquerade:** occurs when the malicious core sends a packet through the NoC using the source address belonging to another core. When the message reaches its destination, the receiving element answer to another (third) core. This unexpected response may cause a system to malfunction.
2. **DoS (Invalid Target):** when a packet is sent to an invalid destination, it is propagated through the routers until it reaches the NoC boundary, and there it remains blocked. As others packets attempt to use this path, the blockage propagates backward.
3. **DoS (Flooding):** the sending of multiple packets to the same destination may reduce the availability of the NoC by consuming excessive network bandwidth.
4. **DoS (Packet without Trailer):** the network protocol does not impose any limit to the packet length, thus a packet sent without a trailer properly identifying its end may result in the blockage of the NoC infrastructure.

To tackle these attacks, they designed the Security Wrapper (SEW), a hardware module inserted between the network interface and the NoC router. The SEW module filters malicious packets sent by an attacking core. It is composed of two separate wrappers.

- **Wrapper 1:** implements a countermeasure to attacks *1* and *2* by verifying if the packet addresses are valid. The source address has to be the same as the node address, while the target must not be out of the network boundaries. If these conditions are not met, the attack is detected, and the packet is discarded.
- **Wrapper 2:** offers solutions to the other two possible attacks. Attack *3* is solved by limiting the maximum bandwidth per core: after injecting a packet in the NoC, the next one is delayed for a period proportional to the allocated bandwidth. Attack *4* is dealt with by verifying if the packet size exceeds a predefined limit. If it does, the packet is broken, and a trailer is manually inserted. The rest of the packet is discarded.

2.4 What Can a Remote Access Hardware Trojan do to a Network-on-Chip?

Ahmed et al. [Ahmed et al., 2021] discuss a Remote Access Hardware Trojan (RAHT) attack in which an HT works in conjunction with an external malicious device to compromise an NoC-based system.

Figure 2.3 presents the attack model. In the proposed attack, an NoC router is infected by a hardware Trojan having a small area footprint. The HT counts the number of

packets traversing the router over a time window and periodically sends it to an external attacker. The attacker then analyses the leaked traffic information using a machine-learning algorithm. This mechanism can infer information such as architectural details or the application details running on the system.

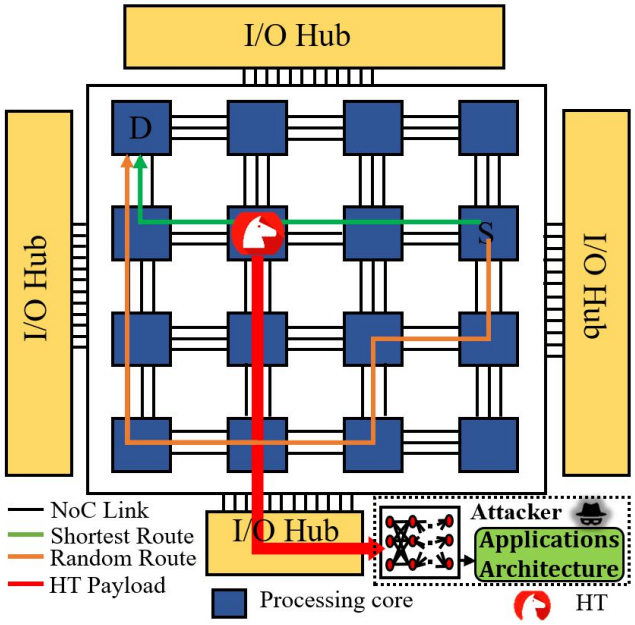


Figure 2.3 – Threat model of the proposed RAHT attack [Ahmed et al., 2021].

To protect the system from this RAHT attack, Ahmed et al. propose a security mechanism that uses controlled random routing to confuse the external attacker. The idea is that using random routing decisions reduces the correlation between the traffic information collected and the system architecture and applications.

This work corroborates the security concerns presented in Chapter 1 by proposing an attack that is enabled by both a hardware Trojan and a malicious IO device. Even though it offers a routing solution to the RAHT attack, it does not prevent the unauthorized communication between HT and peripherals from taking place.

2.5 Final Remarks

The reviewed works highlight the threats that peripherals can represent to system security. The solutions protect mainly the NoC [Baron et al., 2013] or add mechanisms that severely impact performance and the system area [Kapoor et al., 2013]. There is a gap related to mechanisms that effectively protect the many-core against malicious peripherals.

This work aims to fill these gaps by proposing a Secure Network Interface (SNI) with a lightweight authentication mechanism, protecting the communication between PEs and peripherals against DoS and spoofing attacks.

3. FUNDAMENTAL CONCEPTS

This Chapter presents concepts required to follow this work. Section 3.1 presents the baseline architecture, including the hardware and software model. Section 3.2 introduces the concepts of Secure Zones and Opaque Secure Zones (OSZ). Section 3.3 details SeMAP, which is the communication method used in the platform with security mechanisms.

3.1 Basic platform

The many-core baseline platform used in this work is the Hermes MultiProcessor System (HeMPS) [Carara et al., 2009]. The baseline platform and this work are both developed at the *Hardware Design Support Group* (GAPH) research group [GAPH, 2021]. The main HeMPS platform features are:

- NoC-based system: the HERMES NoC [Moraes et al., 2004] allows multiple communications between PEs while ensuring scalability. The NoC adopts 2D-mesh topology, one physical channel, flit width equal to 32 bits, input buffer, credit-based flow control, round-robin arbitration, and XY routing algorithm.
- Homogeneous system: all PEs have the same hardware architecture with a router, a private memory, a MIPS-like processor, and a DMNI (Direct Memory Network Interface) module.
- Distributed memory: each PE has a true dual-port scratchpad memory for instructions and data, while message-passing performs the communication between PEs.
- Applications are modeled as a Communication Task Graph (CTG). The CTG is a model to represent functional parallelism, where an application is composed of parts that are independent of each other and thus are divided into tasks [Rauber and Rünger, 2013]. A graph node represents each task in a CTG, and the graph edges represent the communication between these tasks.
- Distributed management: the system has support to clusterization. Every cluster contains a Local Manager PE (LMP), which manages the cluster, and a set of PEs that run the applications tasks. The Global Manager PE (GMP) works as an LMP and distributes the applications to clusters. The OS (Operating System) running on PEs defines their role.

Works [Fochi, 2019; Caimi, 2019] extended the baseline platform to meet fault tolerance and security restrictions.

3.1.1 Hardware Model

Figure 3.1 overviews the extended HeMPS many-core, with support to fault tolerance and security mechanisms. In Figure 3.1(b), two mesh NoCs interconnect PEs: *data* and *control* NoC. The *data NoC* is a standard wormhole packet switching NoC without virtual channels. It has two particular architectural features. The first one is the adoption of two physical channels, acting as two disjoint NoCs. To minimize the area overhead, the flit size is 16 bits (half of the word size), being the network interface (DMNI) responsible for serializing/deserializing the flits. The reason to adopt two physical NoC is to enable fully adaptive routing. The second feature is simultaneous support for XY (default routing algorithm) and source routing (SR). Source routing is a turn-based routing algorithm in which the packet carries in the header the turns that must be taken. The SR is required when, e.g., it is necessary to circumvent an OSZ or avoid a path with a faulty or infected router.

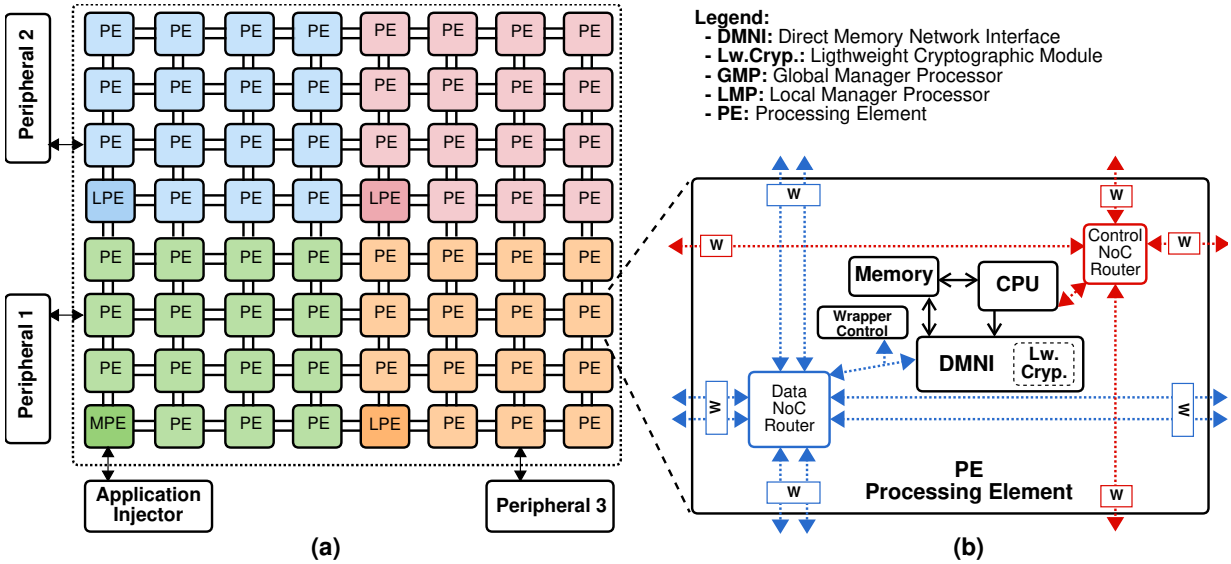


Figure 3.1 – NoC-based many-core. Wrappers (W) are added to the control signals of NoCs links, enabling to isolate ports individually [Caimi, 2019].

The *control NoC* [Wachter et al., 2017], named *BrNoC* is a lightweight network-on-chip, with all packets having one flit. When transmitting in broadcast (default transmission mode), packets reach all PEs of the system. Thus, this NoC can find a path from a source PE to a target PE if it exists, even in the presence of a fault or an HT (Hardware Trojan) in the data NoC. This NoC may also use the unicast transmission to create a path between a source and a target PE, using a backtracking procedure. For security reasons, only the OS accesses the control NoC, avoiding its use by malicious applications.

Both NoCs contain test wrappers, or simply *wrappers*, in the control flow signals. When activated, the wrapper enables to discard all incoming and outgoing packets of a given port. The data NoC observes and respects the status of the wrappers. A data message

arriving in an activated wrapper is always discarded, and the control NoC replies to the source of the message a new broadcast reporting that the message needs retransmission.

The control NoC has two operation modes: *global* and *restrict*. The *global* mode enables the control messages to pass through the wrappers, even if they are enabled. This mode enables the PEs inside the *SZ* to exchange messages with manager PEs. The *restrict* mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded, which is fundamental for searching paths without secure zones.

The platform is modeled at the RTL level, part in SystemC (memory, processor, DMNI) and part in VHDL (data and BrNoC routers).

BrNoC Control Network

Figure 3.2 details the BrNoC architecture. Its topology follows the same 2D mesh used by HERMES, with North, South, East, West, and Local ports. The BrNoC internal modules are: (*i*) an Input Arbiter and Input Finite-State Machine (I-FSM); (*ii*) a central Content-Addressable Memory (CAM); and *(iii)* an Output Arbiter and Output Finite-State Machine (O-FSM).

The round-robin Input arbiter selects the port with data to write into the CAM. To write a message to the CAM, the I-FSM must assert that the data is not in the memory and it has available space, marked by the *used* field.

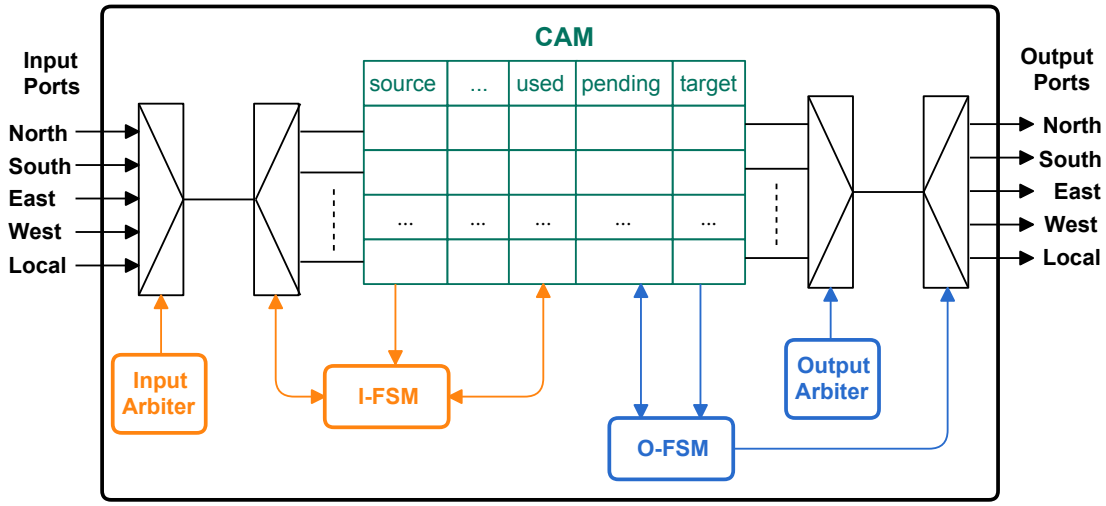


Figure 3.2 – BrNoC architecture. Source: [Wachter et al., 2017].

The BrNoC Input and Output logic are independent. A round-robin Output Arbiter selects a CAM line to propagate to the outputs (broadcast). The O-FSM searches the *pending* field for messages that need to be sent. The data is propagated to all ports except to the one where it came from.

The most relevant BrNoC feature is that all messages fit in one flit. The payload size is parameterizable according to the constraints of the design. The advantages of 1-flit messages are: (*i*) no buffers on local ports; (*ii*) simplified switching mode, which enables the broadcast; (*iii*) smaller router silicon area.

The BrNoC has four distinct services: (*i*) broadcasting to **all** PEs, which broadcasts a message to all processors; (*ii*) broadcasting to a **target**, which also broadcasts a message, but the only processor that receives the message is a defined *target*; (*iii*) broadcasting **without a target**, to send internal BrNoC control messages; and (*iv*) **unicast**, implemented through a backtracking mechanism.

Note that the O-FSM showed in Figure 3.2 only propagates a message to the local output port when it is a broadcast to **all** or when a broadcast to the **target** arrives at its destination. In this last case, the message is only sent to the local port and is not propagated to the remaining ports.

A broadcast **without a target** sent by the source of each propagated message erases each CAM line. This message releases the CAM line to receive a new broadcast.

3.1.2 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. The management of HeMPS occurs in the Manager PE, which has a different kernel from the other PEs.

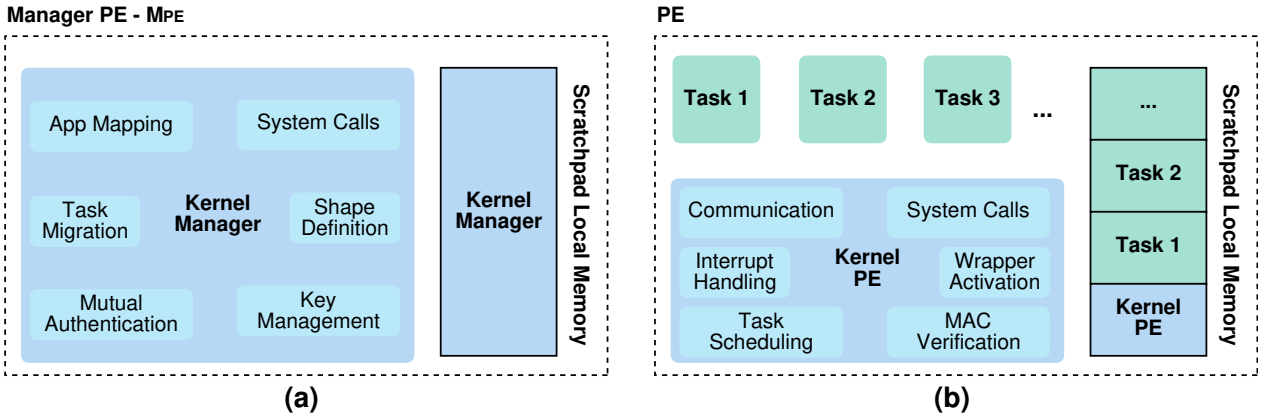


Figure 3.3 – Overview of the kernels: (a) Manager PE kernel controls the system and do not execute users' tasks; (b) Regular PE kernel manage users' tasks [Ruaro et al., 2019; Caimi, 2019].

At the Manager PE level, the local memory is reserved to the kernel, without executing user's tasks. The Manager PE executes heuristics as task mapping, task migration, monitoring, authentication and key management (Figure 3.3(a)).

At the regular PE level, 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 (Figure 3.3(b)).

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 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). Figure 3.4 presents an application following this model.

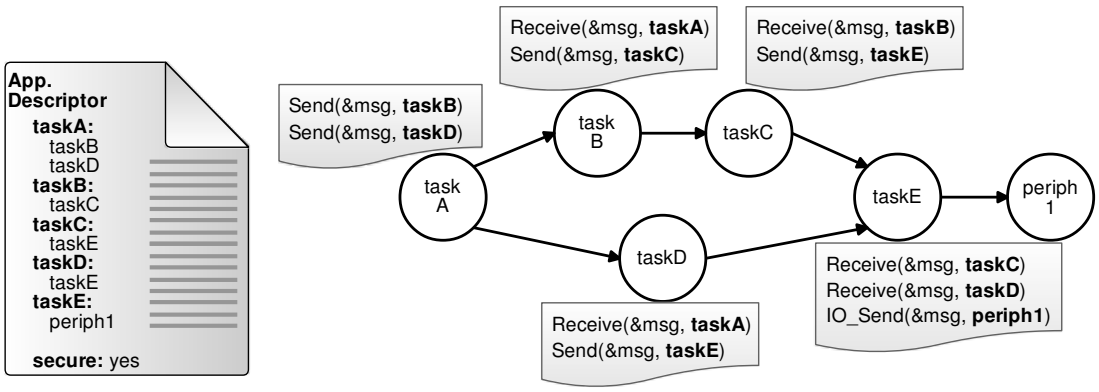


Figure 3.4 – Application task graph example [Caimi, 2019].

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 requests 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. 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 to 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 enable packet retransmission to inter-task communication and inter-manager communication respectively.

3.2 Opaque Secure Zones - OSZ

Resource sharing is an essential feature of many-cores. Different applications may execute in the same processor, share the NoC links, and shared memories. This feature, resource sharing, is the source of issues related to security.

Security 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 execution time of the heuristics. However, design-time methods do not apply to dynamic workload scenarios. Thus, these methods are limited to scenarios where the workload is known beforehand without changing during the system lifetime.

Secure Zone (SZ) is a runtime approach adopted to limit resource sharing. It is possible to classify such proposals using a set of orthogonal criteria [Caimi and Moraes, 2019]:

- **Creation time:** when SZ is defined, at design time or runtime.
- **Shape:** the SZ may be discontinuous or continuous, with a rectangular or rectilinear shape.
- **Communication sharing:** the SZ may allow flows belonging to sensitive applications to share NoC links or the flow inside the SZ is forbidden to other applications.
- **Computation sharing:** the SZ may allow tasks belonging to sensitive applications to share the same processor or applies resource reservation to sensitive applications.
- **Methods:** the methods used by the SZs include cryptography, routing algorithms, spatial and temporal isolation, and rerouting.

Figure 3.5 presents examples of SZs. Discontinuous SZs (*SZ*2) require more efforts to prevent attacks (encryption or routing schemes) due to the flows' exposure, while continuous SZs 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.

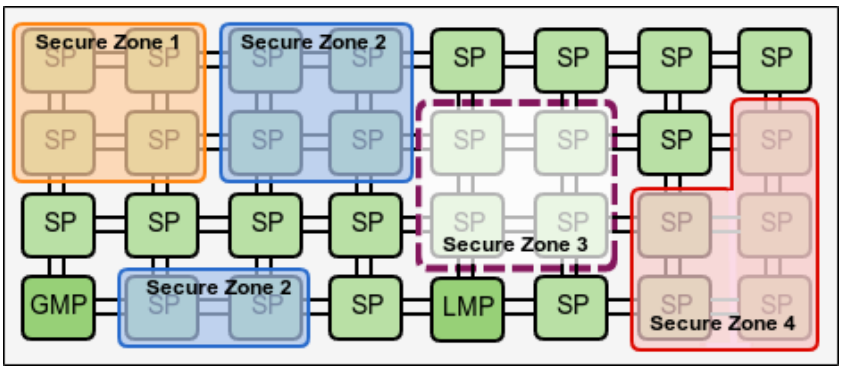


Figure 3.5 – *SZ*1: continuous and rectangular, *SZ*2: discontinuous, *SZ*3: continuous, rectangular, and opaque, *SZ*4: continuous and rectilinear. Source: [Caimi and Moraes, 2019].

According to the previous classification, **Opaque Secure Zones** (OSZs) are created at runtime, and have a rectilinear shape, without computation and communication resource sharing. The PEs of the *OSZ* are reserved for running a single secure application (*SZ*3, in Figure 3.5). The only resource-sharing exception is communication with I/O devices. The method that enables OSZ is the dynamic rerouting mechanism. The rerouting mechanism ensures that the secure application traffic stays inside the OSZ and deviates the traffic that should cross the OSZ.

The OSZ method, summarized in Figure 3.6, is a countermeasure protecting both communication and computation. The proposed method includes: *(i)* OSZ shape selection; *(ii)* wrapper activation; *(iii)* retransmission of lost packets in and out the OSZ boundaries; *(iv)* start the secure application (App_{sec}). In Figure 3.6(a) the many-core contains one application in execution, app_1. Next, the MP (manager PE) maps an App_{sec}, activating the wrappers at the boundary of the OSZ. At this moment (Figure 3.6(b)), the app_1 traffic is blocked by the OSZ. Figure 3.6(c) shows the App_{sec} executing in the OSZ, and the traffic of app_1 circumventing the region. During the App_{sec} execution, all communication and computation resources of the OSZ are reserved for the application.

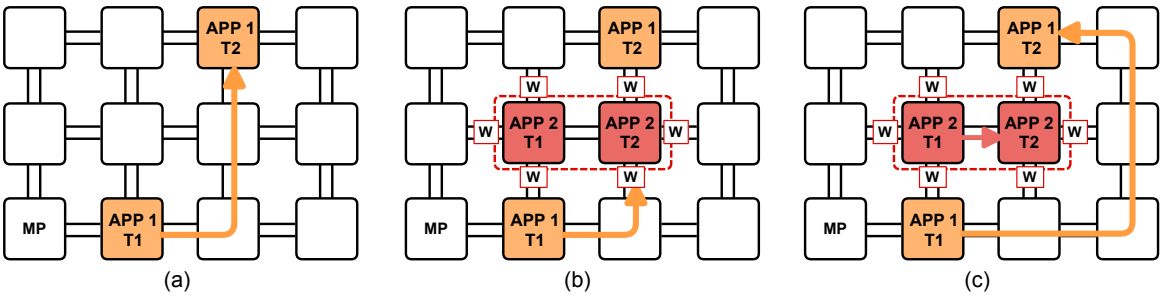


Figure 3.6 – Secure zone and dynamic reconfiguration of routing paths. Source: [Caimi et al., 2018b].

The OSZ is configured by the Manager PE, which runs an algorithm to determine: (*i*) the number of PEs to run the application; (*ii*) the possible OSZ shapes; (*iii*) the OSZ positioning inside the cluster; (*iv*) the number of task migrations needed to ensure exclusive PE execution.

3.3 SeMAP - IO Communication Model

As the peripherals are outside the OSZ, it is necessary to open the OSZ to enable incoming and outgoing messages with these devices. Opening the OSZ does not contradict the basic rule of the method: flows belonging to other applications must not cross the OSZ. However, we are opening the OSZ frontier to communication flows with peripherals, which can represent a risk to the App_{secs}, and it is necessary to create a set of mechanisms to guarantee this communication without incurring threats to the App_{secs}. Basic premises, defined in [Caimi and Moraes, 2019], include:

1. Differentiate the PE-PE communication from the PE-Peripheral. This API differentiation prevents malicious applications from trying to inject packets into OSZs.
2. Master-slave communication. The PEs inside the secure zone must initiate all transactions. Thus, the PEs of the OSZ discard all unexpected packets.
3. Packets must be signed to ensure their authenticity and ensure they come from the correct peripheral.
4. To minimize the attack surface, each OSZ has one input access point (AP) and one output AP.

Secure Mapping with Access Point (SeMAP) [Faccenda et al., 2022] (Appendix C) restricts the OSZs mapping to follow the above premises. Figure 3.7 illustrates a possible organization of the system in gray and secure areas. Gray areas run applications without security requirements and guarantee a path between App_{secs} and peripherals. In this example, there are three App_{secs} mapped in the secure areas. The mapping of App_{secs} requires at least one side juxtaposed to the gray area in such a way to have a path to the peripherals.

The SeMAP reduces complexity in choosing access points (APs), unifying the input and output APs in the same coordinate. The application knows the coordinate of the AP, and each PE computes the path to/from the peripheral using source routing (SR). The AP stays opened while the application is running. For output flows, the concern will be the monitoring of the output rate to prevent the App_{sec} itself from trying to carry out a DoS attack on a peripheral. For input flows, the security verification must occur at two points: (1) at the AP, verify if it is an IO packet; (2) at the target PE, verify the packet signature. Remember that only IO packets enter/leave the AP. Therefore, an App_{sec} could try to attack peripherals but not other PEs.

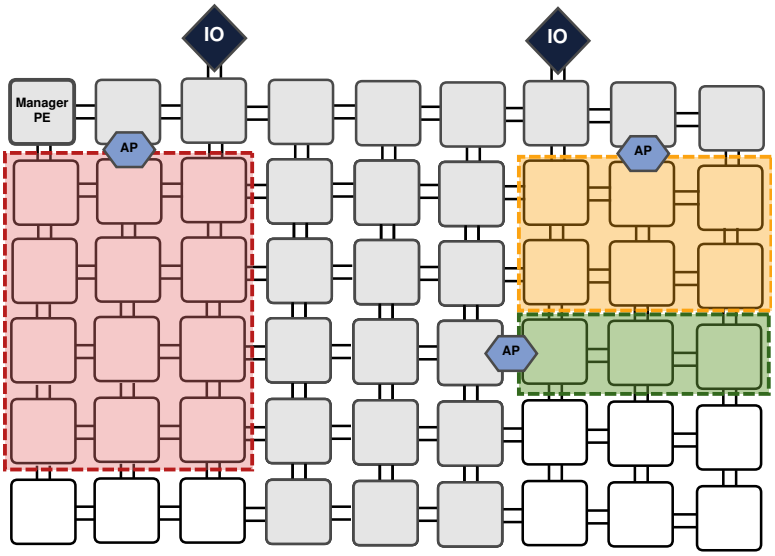


Figure 3.7 – Gray and secure areas. Three App_{secs} mapped on the secure area, each one with an Access Point (AP) [Faccenda et al., 2022].

3.3.1 Vulnerabilities

Although the Opaque Secure Zone is a robust method for ensuring the security of an application, there are vulnerabilities to be addressed.

- Malicious peripheral (3PIP) connected to the system;
- Hardware Trojans in NoC routers may disrupt communication between PE;
- Data packets traversing the “insecure” part of the network can be targeted;
- Malicious traffic may enter the secure zone through the AP and disrupt the application.

Therefore, it is necessary to create countermeasures to avoid attacks. The following Chapter details the main contribution of this work, the design of a Secure Network Interface responsible for managing the IO traffic.

4. SECURE NETWORK INTERFACE

The security functionalities available in the platform protect the application during two out of three phases of its lifetime [Caimi, 2019]. *Admission* is secured through the use of lightweight cryptography and authentication codes, while *Execution* is protected by isolating the application inside an OSZ.

Nevertheless, the *IO Access* phase is still an open issue to be addressed. This Chapter proposes a Secure Network Interface (SNI) to manage peripheral access, ensuring secure communication between an application and IO devices.

Figure 4.1 presents the main components of the many-core architecture. The two main system components are:

- **PE**: 32-bit RISC processor, a NI (Network Interface) with DMA capabilities, local scratch-pad memory, and two NoC routers;
- **Peripherals:** an SNI (Secure NI) makes the interface between the NoC and IO devices.

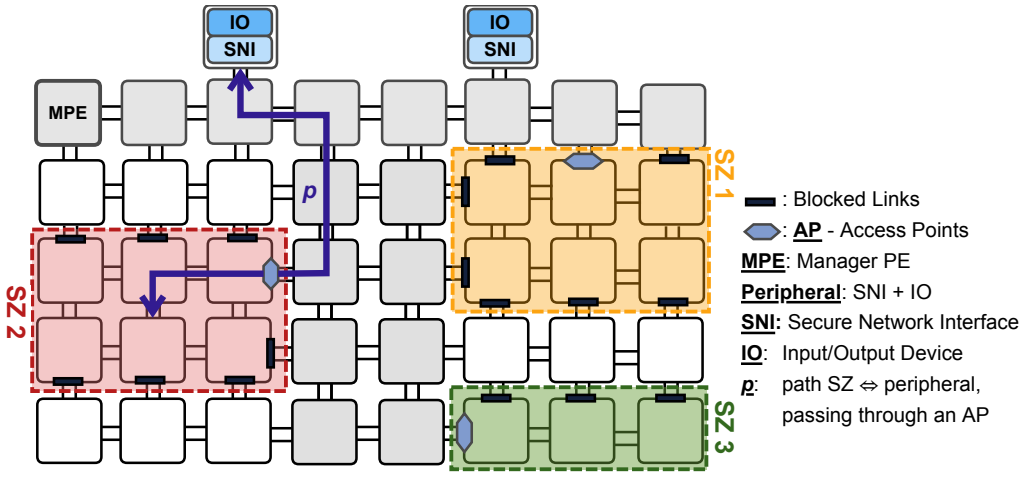


Figure 4.1 – Many-core partitioned on secure and gray areas (SA and GA). Three App_{secs} mapped on SZ1 to SZ3. GA is reserved for applications without security constraints. (Source: Author.)

This Figure shows the components involved in the communication between an OSZ and the proposed SNI. The path between OSZ and SNI (p) is subject to different attacks.

This Chapter contains five sections. Section 4.1 presents the threat model, describing a set of attacks that may harm the system security, Section 4.2 introduces the authentication protocol, the main security mechanism to prevent DoS and spoofing attacks. Using the threat model and the authentication protocol, Section 4.3 builds the set of requirements to develop the SNI. Section 4.4 is the core of this Chapter with the SNI hardware description. Section 4.5 concludes this Chapter by presenting the software services the SNI handles.

4.1 Threat Model

This section discusses the harmful behaviors from which this work aims to protect the system. These attacks can be performed by a malicious application (*MalApp*) and/or by a malicious peripheral (*MalPerph*).

As a 3PIP, the peripheral cannot be trusted and is considered a source of vulnerabilities. Embedding a peripheral into the system enables the following attacks:

- **Flooding:** *MalPerph* floods the network with packets, overloading the communication infrastructure and the system's components. This attack aims to affect the overall performance of the system or even bring it to a halt.
- **Misrouting:** *MalPerph* sends its messages to the wrong target, leaking sensitive information or disrupting the execution of applications.
- **Spoofing:** a malicious entity accesses a peripheral without authorization, stealing or corrupting sensitive data.

Applications must send and receive messages from outside the secure zone to communicate with an IO device. Therefore, the secure zone boundaries must open to let packets through. In the SeMAP method, this opening is called the Access Point (AP). This opening of the secure zone makes possible another set of attacks:

- **Spoofing of the AP:** a malicious entity sends forged packets that pass through the AP, pretending to be from a trustworthy peripheral.
- **Flooding of the AP:** a malicious entity floods the AP with packets, aiming to disrupt the application execution or the AP itself.

Furthermore, a packet traversing the non-isolated region of the network might have its information compromised by a Hardware Trojan infecting a router. This has to be taken into consideration when designing methods to approach the aforementioned attacks.

4.2 Authentication Protocol

To handle these security threats, it is necessary to provide a method of differentiating a genuine application (or peripheral) from a malicious one, making it possible to control which packets can safely enter the secure zone or reach the peripheral. This requirement is achieved through the use of a lightweight authentication protocol, which is proposed in another work (Appendix D).

This protocol is divided into four phases: *Initialization*, *Application Deploy*, *Communication*, and *Key Renewal*. It works by distributing authentication keys – *{k1, k2}* – to a group of communicating entities: the secure application and each SNI it needs to access. Every message between the application and the peripheral is sent with an authentication field, which can only be verified by someone who knows the keys.

To better understand the role played by the SNI in this process, the authentication protocol is disclosed briefly in the following subsections. The complete sequence diagram of the Authentication Protocol is presented in Appendix A.

4.2.1 Initialization

During the *initialization* phase, the Manager PE generates and sends unique keys, named ***k0***, to each communicating PE or SNI in the system. Since this action is performed before any application is admitted, there is no malicious traffic in the network. Thus the keys can be transmitted in plaintext, exempting the use of any complex key distribution algorithm. Each ***k0*** is used in the next phase to obfuscate sensitive information before sending it through the network.

4.2.2 Application Deploy and Authentication Keys Generation

When a new application is mapped into the system, the Manager PE must coordinate the process of generating *{k1, k2}* for that application. To do so, it randomly generates a tuple $\{appID, n, p\}$, where *appID* is the unique application identifier, and $\{n, p\}$ are integers values. This tuple must reach all the PEs and SNIs of the application but must remain secret to other entities. This can be done by obfuscating these values using the unique ***k0*** of each recipient ($k0_x$). Thus, the packets sent through the network contain initialization flits, shown in Equation 4.1.

$$\boldsymbol{i1} = appID \oplus k0_x \qquad \boldsymbol{i2} = (n \,\&\, p) \oplus k0_x \tag{4.1}$$

A SNI, upon receiving *{i1, i2}* retrieves the tuple $\{appID, n, p\}$ using its ***k0***, as shown in Equation 4.2.

$$appID = \boldsymbol{i1} \oplus k0_{SNI} \qquad n = MSB(\boldsymbol{i2} \oplus k0_{SNI}) \qquad p = LSB(\boldsymbol{i2} \oplus k0_{SNI}) \tag{4.2}$$

The authentication keys are derived from these parameters. This process consists in: (*i*) set the *appID* as the seed for a linear-feedback shift register (LFSR); (*ii*) shift the LFSR n times to obtain ***k1***; (*iii*) shift the LFSR p more times to obtain ***k2***. By the end of this phase, all communicating entities share the same pair of authenticating keys.

4.2.3 Communication

This phase includes the communication between the application and peripherals. The tasks are responsible for starting the communication by sending delivery or request data messages to/from the IO device. Any of these messages trying to access the peripheral must contain the tuple $\{appID, k1, k2\}$ encoded into its authentication flits, as in Equation 4.3.

$$\textbf{\textit{f1}} = k1_{PE} \oplus k2_{PE} \qquad \textbf{\textit{f2}} = appID \oplus k2_{PE} \tag{4.3}$$

When the corresponding SNI receives a packet, it must assert its authenticity by retrieving *appID* using the stored value of $k1$ (Equation 4.4). If the packet is authentic, the value encoded in ***f1*** and ***f2*** matches the original *appID* value, received during the *application deploy* phase.

$$(\textbf{\textit{f1}} \oplus k1_{SNI}) \oplus \textbf{\textit{f2}} == appID_{SNI} \tag{4.4}$$

If the authentication is successful, the SNI performs the service requested, replying with the requested data or acknowledging the data received. The outgoing response also contains the authentication flits, and a similar verification is performed before the packet enters the secure zone. Otherwise, if the authentication fails, the incoming packet is discarded, and the SNI takes no action.

4.2.4 Key Renewal

Using the same authentication flits for a long time is a security concern, making them more vulnerable to eavesdropping. The stealing of $\{f1, f2\}$ is not a sufficient condition for an attack. The attacker also needs to know the addresses of the devices and/or the time window a packet is expected. Nonetheless, to increase security, the authentication keys $\{k1, k2\}$ are renewed periodically, even if there is no threat detection.

During the key renewal process, one of the application's PEs randomly generates a pair of integers $\{n, p\}$ and sends it to all other communicating entities. In each device, the LFSR is initialized with the previous value of $k2$, shifted n times to obtain the new $k1$, and

p more times to get the new $k2$. All IO communications are frozen during the renewal to ensure synchronization between communicating parties.

4.3 Requirements

Based on the role played by the SNI in the authentication protocol, this subsection defines the set of requirements for constructing a network interface to tackle the security threats raised in Section 4.1.

Although the SNI is designed as a security module, it has the duties of a standard network interface. This is taken into consideration in the first three requirements.

1. Enable the communication between the peripheral and other devices in the network. This is done by implementing the system's IO communication API.
2. Abstract the data network protocol for the peripheral. This increases the number of compatible devices, as a simpler protocol can be implemented for interaction between SNI and peripherals. But it also has the security advantage of hiding the inner workings of the system, which makes attacks less likely.
3. Use the lowest amount of silicon area as possible. This module is instantiated many times in the system, which can impose a large area overhead.

Attacks performed by a malicious peripheral can happen in two ways: the sending of unrequested packets, and relaying the packet to the wrong recipient. The next two requirements are made to avoid these harmful behaviors.

4. Enforce the master-slave communication model, in which only an application can start the exchange of messages. This makes the peripheral unable to inject unwanted packets into the network.
5. The SNI only sends messages to authorized applications through source routing paths set by trusted entities. Thus, the peripheral cannot deliberately interact with unapproved parties.

Spoofing attacks, both targeting the peripheral or the application, are dealt by the authentication protocol. The next requirements regard the role of the SNI in this context.

6. Register the applications authorized to interact with the peripheral.
7. Grant the access to the peripheral only to authorized applications. Sensitive information provided by the peripheral cannot be directly stolen by a malicious application.

8. Be able to execute the authentication protocol procedures, such as asserting the authenticity of a packet, and performing the keys derivation procedure.

The final security constraints and observations are disclosed in the last requirements.

9. Quickly discard packets that fail the authentication, avoiding flooding attacks
10. Build packets that follow the Source Routing algorithm, which obfuscates the addresses of the communicating devices, making attacks more difficult

4.4 Hardware Architecture

The designed Secure Network Interface contains five components, presented in Figure 4.2. Communication to and from the data-NoC is driven simultaneously through two independent modules: Packet Handler and Packet Builder, respectively. The information necessary to communicate with each application is stored in the Application Table by the Packet Handler and retrieved by the Packet Builder when needed. The data received or sent to the peripheral is kept in buffers until consumed. The following subsections present these components in detail.

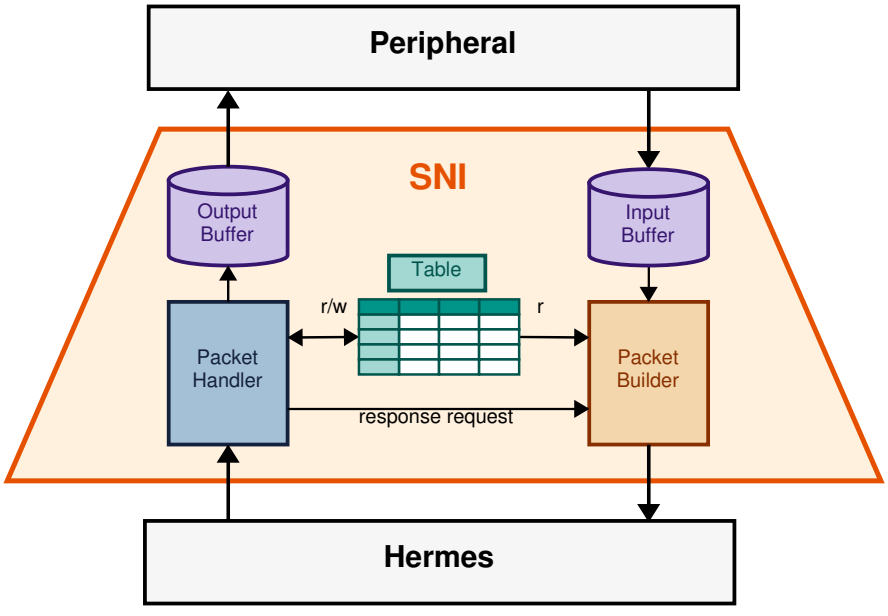


Figure 4.2 – Overview of the Secure Network Interface and its components. (Source: Author.)

4.4.1 Application Table

The main function of the Application Table is to record the applications that are allowed to access the peripheral, as well as the information needed to authenticate and answer packets sent by these applications. Each line of the table corresponds to a different communicating application. This application granularity reduces the table size (and, consequently, area) compared to a table with task granularity. Figure 4.3 illustrates the fields available in the table.

line	valid	appID	k1	k2	path_to_SZ	path_size
#1						
#2						

Figure 4.3 – Application Table with two lines, each corresponding to a different application allowed to interact with the peripheral. (Source: Author.)

- **Valid**: flag used to signal whether the table slot is being used to store an application's information or not.
- **AppID**: ID of the application allowed to access the peripheral.
- **K1 and K2**: keys used by the authentication protocol to assert if a given packet was really sent by the application.
- **Path_to_SZ**: sequence of turns a packet has to take in the network to reach the OSZ. This field is required to send messages to the application through source routing.
- **Path_Size**: contains the size of the **Path_to_SZ**, which may have up to 6 flits.

The SNI handles paths from one up to six flits, making the *Path_to_SZ* the largest field on the table. To avoid passing large busses through the table interface, *Path_to_SZ* is broken into six segments, each one corresponding to a flit in the path. Only one segment can be read or written at a time.

The Application Table works as a Content-Addressable Memory (CAM). To access the desired line, we must inform the *appID* of the application we are looking for. The table itself searches the correct line and makes it available for reading or writing. In a CAM, the information used to distinguish each line (here the *appID*) is called a *tag*.

Furthermore, the table offers two separate interfaces, depicted in Figure 4.4. The primary (read-write) interface is connected to the Packet Handler, while the secondary (read-only) interface is connected to the Packet Builder. This separation enables the SNI to send and receive messages simultaneously.

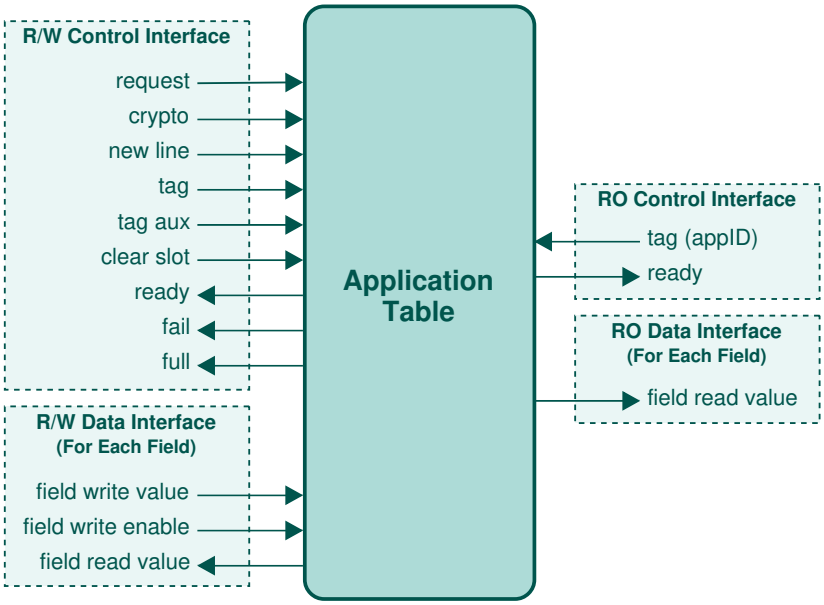


Figure 4.4 – Application Table interfaces: primary (Read/Write) and secondary (Read Only). (Source: Author.)

Using the read-only interface, the Packet Builder can access the table by simply providing the *appID* of desired application. The table uses a combinational circuit to find the matching line, outputting its fields for the Packet Builder to read. If a matching application is indeed found, the signal *ready* is raised to indicate success.

The table's primary interface is more complex. Besides being able to search for an application using its *appID*, it can also allocate an empty line for usage, and perform what is called a Crypto Search.

During the Crypto Search, the table receives not the *appID*, but the authentication flits *f1* and *f2*. To find the right line, the table must perform the authentication process (Figure 4.5) to each row of the table. The line which is successfully authenticated can be accessed through the interface.

The three ways to search for an application using the primary interface are listed in Table 4.1. One bit signals ***request***, ***crypto***, and ***new_line*** defines the search method. It also specifies the values needed as tag for each type of search.

Implementing this interface using a purely combinational circuit would result in a lot of hardware replication, as all the lines would be searched in parallel. To avoid this area

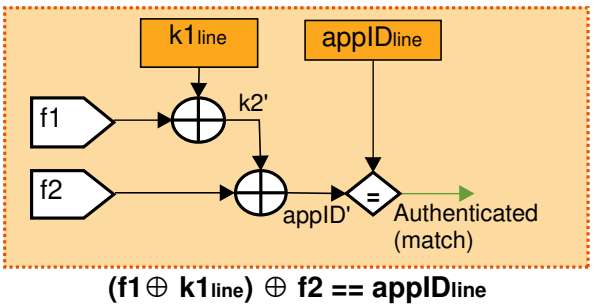


Figure 4.5 – Authentication process, used to find a match during Crypto Search. (Source: Author.)

Table 4.1 – The different ways of searching for a line in the Application Table.

Search Type	Tag	Tag Aux	Description
New Line	–	–	Allocates an unused line.
Regular Search	*appID*	–	Find the line which contains the same *appID*.
Crypto Search	*f2*	*f1*	Find the line for which *f1* and *f2* are valid.

overhead, the search process is done sequentially and is managed by the FSM depicted in Figure 4.6.

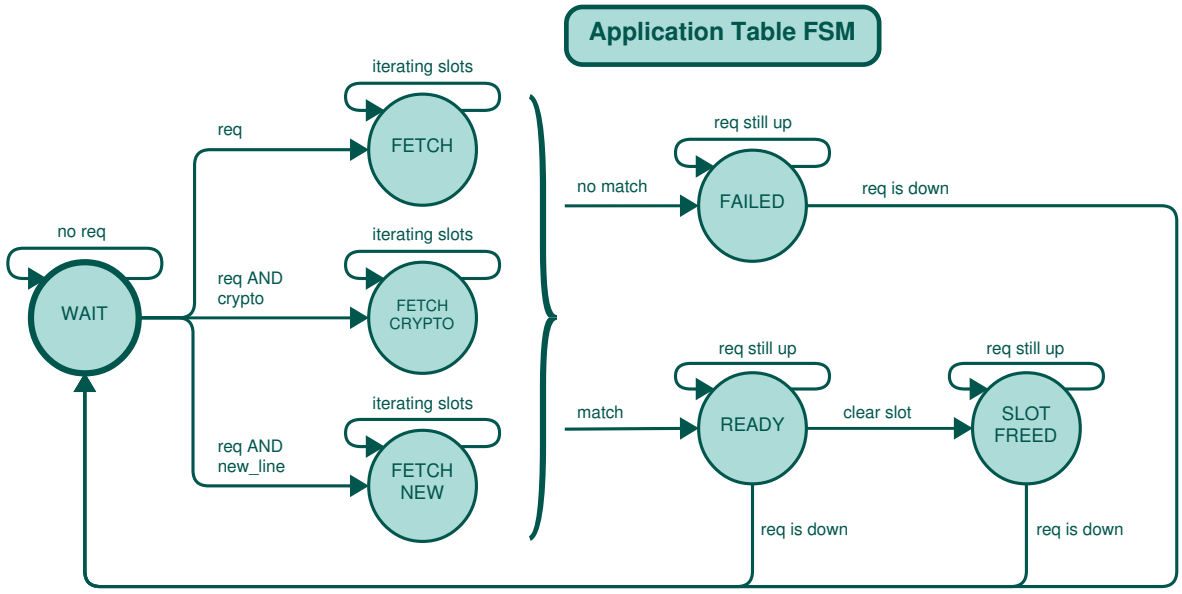


Figure 4.6 – Application table FSM. (Source: Author.)

To access an application, the Packet Handler has to make a request to the Application Table by raising the signal ***request*** and informing the desired ***tag***. It can choose between the different search types by using the signals ***crypto*** and ***new_line***. The FSM, then, moves from *WAIT* into the corresponding *FETCH* state (*FETCH_CRYPTO* for Crypto Search, and *FETCH_NEW* for searching a new line).

During fetching, the table iterates over all its lines looking of a matching application. When the correct line is found, the FSM transitions to *READY*, making the line available for reading/writing, and raising the signal ***ready***. At this point, the line can also be deallocated by raising ***clear_slot***. If no match is found, the FSM transitions to *FAILED* and raises ***fail***. The table only returns to is waiting state once the ***request*** signal is dropped.

4.4.2 Packet Builder

For the SNI to communicate with applications, it needs to be able to answer to incoming messages. The Packet Builder module is responsible for assembling those answers and sending them through the data NoC.

Once the Packet Handler decides to send a message to an application, it notifies the Packet Builder through the Response Request Interface (Figure 4.7). It raises the signal ***response_req*** and informs the packet parameters: ***service*** defines the type of message to build, while ***appID*** specifies which application to send it to.

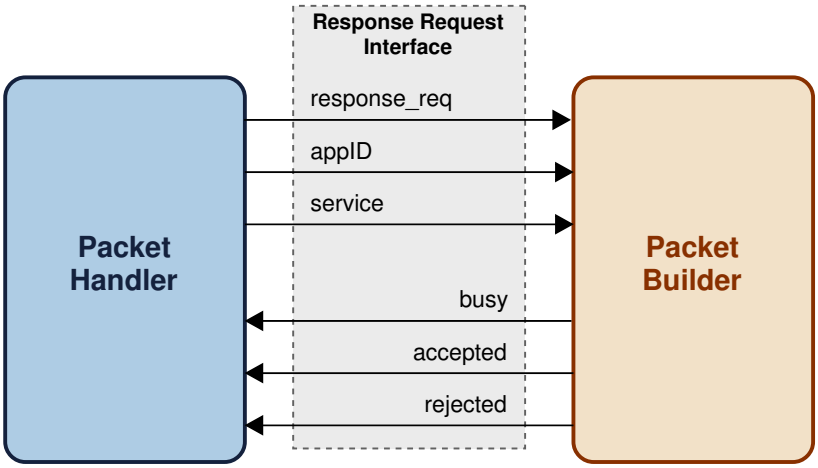


Figure 4.7 – Interface between the packet handler and the packet builder blocks. (Source: Author.)

The process of building the packet is managed by the FSM depicted in Figure 4.8. Upon receiving a request and registering the parameters, the Packet Builder moves to the *CHECK_TABLE* state, where it verifies if the specified application can be accessed through the table. If the application ID is not found, the FSM transitions to the state *REJECT_REQ*, raising the signal ***rejected*** and waiting for the ***request*** signal to be dropped. Otherwise, if the application exists, the Packet Builder raises the signal ***accepted*** and proceeds to actually building the packet.

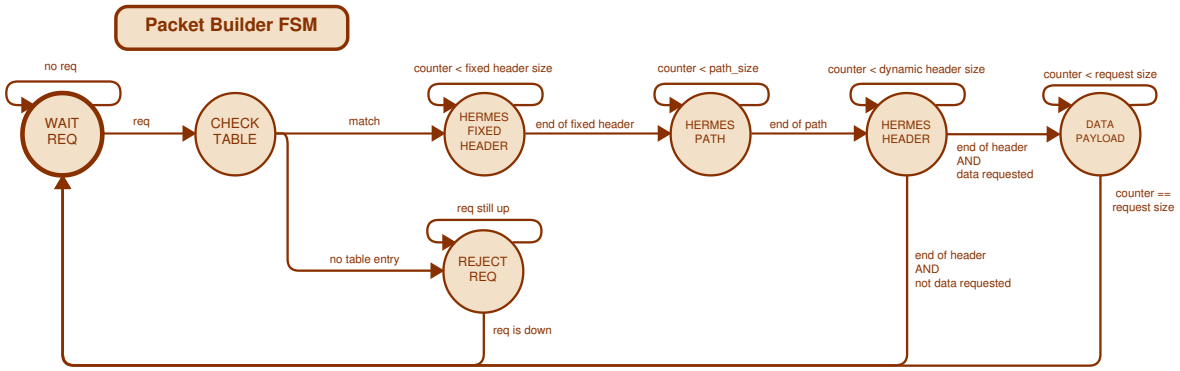


Figure 4.8 – Packet builder FSM. (Source: Author.)

The states *HERMES_FIXED_HEADER*, *HERMES_PATH* and *HERMES_HEADER* are all responsible for transmitting different parts of the packet's header. During this process, the message's flits are generated one by one, following the structure used by the data NoC. Once one flit is successfully injected into the network, the counter is updated and the next flit is generated. The information required to fill the flits is mostly retrieved from the Application Table. This process goes on until the entire header has been transmitted.

If the ***service*** requested by Packet Handler does not require a payload, the transmission is now over, and the FSM returns to the *WAIT_REQ* state.

There is also the scenario in which the SNI must send the peripheral's data to the application. In this case, the FSM transitions to *DATA_PAYLOAD*. During this state, the Packet Builder reads the data written by the peripheral in the Input Buffer and uses it to fill the packet's payload. Once the desired amount of flits has been reached, the Packet Handler returns to its idle state and awaits a new request.

4.4.3 Packet Handler

The SNI is a passive module in the sense that it waits for an incoming message to tell it what to do. The Packet Handler is the component responsible for receiving packets from the data network and carrying out the appropriate action. It can be considered the most important of the SNI's components, as it does all of the decision-making, acting as the manager of the other components.

The handling process is divided into five distinct phases, and is controlled by the FSM depicted in Figure 4.9. Aiming for ease of comprehension and maintainability, the FSM was divided into blocks of states, each block corresponding to a different phase. This section

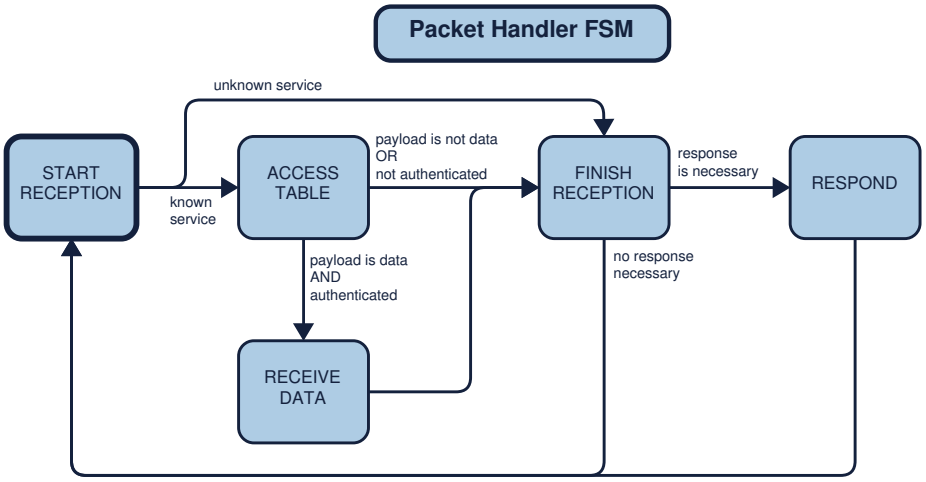


Figure 4.9 – The simplified Packet Handler's FSM. Each blocks of states corresponds to a handling phase. (Source: Author.)

goes through all the five handling phases, explaining them individually. Appendix B shows the complete state diagram of the FSM, opening up all the blocks simultaneously.

The ***START_RECEPTION*** phase (Figure 4.10) is responsible for receiving the incoming packet header. Once the first flit arrives through the data NoC, the FSM leaves the *WAIT_REQ* state and transitions to *PARSE_HERMES_HEADER*.

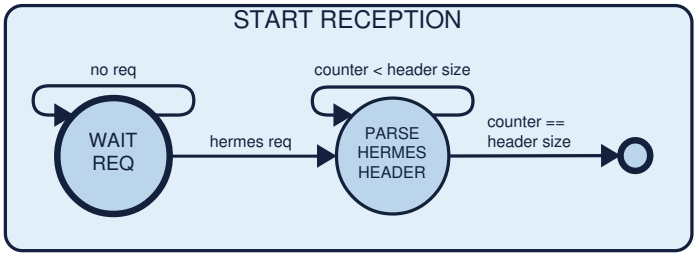


Figure 4.10 – ***START_RECEPTION*** phase of handling. Small circle used to symbolize exit. (Source: Author.)

During this state, a counter keeps track of the flits being received. When the incoming flit corresponds to a relevant information (e.g. *service*, *appID*, *f1* or *f2*), it is saved in the correspondent register. This flit-by-flit parsing was adopted to avoid wasting area by buffering the whole packet header.

The Application Table's search mechanism is also started at this state: as soon as the packet's *service* is received and the corresponding *tag* (e.g. *appID*) is written to a register. This is done to save time, avoiding having to stop the handling flow to wait for the table to find a match.

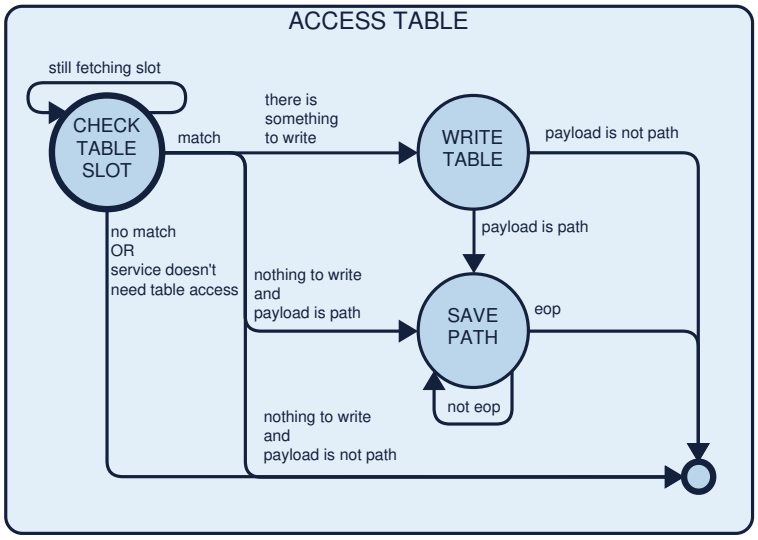


Figure 4.11 – ***TABLE_ACCESS*** phase of handling. Small circle used to symbolize exit. (Source: Author.)

After parsing the packet's header, the FSM moves into the ***TABLE_ACCESS*** phase, depicted in Figure 4.11. The *CHECK_TABLE_SLOT* state starts by verifying if the Application Table found the line requested. This checking serves the double purpose of obtaining access to the table, as well as authenticating a packet. If we provided an application's credential (i.e. *appID* or $\{f1, f2\}$) and the table was able to find it, this means the incoming packet is authentic.

Once we have access to the requested line, we can read and write its fields. The packet's ***service*** determines which fields are to be written, if any. For instance, an application configuration service would fill out every field of the line; while key renewal might just update the values of *k1* and *k2*.

Almost all fields are written in the *WRITE_TABLE* state, using the information registered during the parsing of the packet's header. The only exception is the *Path_to_SZ* field, which is retrieved from the packet's payload and written flit-by-flit during the *SAVE_PATH* state. The value of the the *Path_Size* field is determined by counting how many path flits are received.

If the incoming packet is authenticated and contains data to the peripheral, the FSM moves to the ***RECEIVE_DATA*** phase. During this stage, each flit of the packet's payload is received from the data NoC and written into the Output Buffer, to be consumed by the peripheral.

The ***FINISH_RECEPTION*** stage is simply responsible to discard the remaining of the packet if there are still flits to be received. This is used to drop packets with unknown services or unauthenticated applications. It also serves the purpose of making sure the incoming packet is fully received, may any packet be sent with an unexpected payload.

The last possible step in handling the incoming packet, is to send an answer. The ***RESPONSE*** phase is responsible for communicating with the Packet Handler, sending a response request through the interface aforementioned in Section 4.4.2. If the Packet Builder is busy with the construction of another message, the FSM waits for it to be available in the state *WAIT_TX_AVAIL*. When the module becomes free, the FSM transitions to *REQ_RESPONSE*. Once the acknowledge is received, the Packet Handler returns to its idle state and awaits a new packet to arrive.

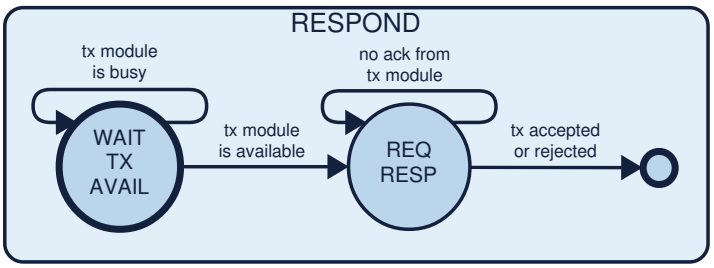


Figure 4.12 – Respond stage of handling. Small circle used to symbolize end of stage. (Source: Author.)

4.5 Services Treated by the SNI

Now that the structure of the SNI as a hardware device was presented, this section presents how the SNI interacts with the other components of the system.

There are four different services the SNI performs: two for managing the authentication protocol parameters; and another two for communicating with applications. The first two services correspond to the *Initialization* and *Application Deploy* phases of the authentication protocol, respectively.

IO_INIT: this service is used by the Manager PE to set the ***k0*** value of the SNI, during the initialization phase of the authentication protocol. This operation can only be performed once.

IO_CONFIG: used to register a new application in the SNI table, thus granting it authorization to access the peripheral. The received packet contains the ***appID***, the pair of authentication keys **{*k1,k2*}** and the ***path_to_SZ***. Following the authentication protocol, *appID* is obfuscated using *k0*. By the end of handling, a new line has been allocated in the Application Table, where all values are stored.

Remark: the LFSR is not yet integrated into the SNI. Thus, instead of sending the tuple {*appID,n,p*}, the system manager sends {*appID,k1,k2*}.

The SNI is accessible to the applications through the IO Communication API, discussed in Section 3.3. The packets themselves were modified to contain the information needed for the authentication process.

IO_REQUEST: is sent by the application to request data from the peripheral. The packet encodes the tuple **{*appID,k1,k2*}** into the authentication flits $f1$ and $f2$. When the packet is received by the SNI, the Application Table performs a crypto search. If a matching application is found, the packet is said to be authentic.

Only if the incoming message is successfully authenticated, the SNI answers the application with an IO_DELIVERY packet, containing the data requested. This outgoing packet is sent through the path configured earlier and also contains the authentication flits, for verification on the application side.

IO_DELIVERY: carries data the application wants to convey to the peripheral. It also contains the tuple **{*appID,k1,k2*}** embedded into $f1$ and $f2$. The authentication process is the same as the performed for the IO_REQUEST service.

If the authentication succeeds, the data contained in the packet is relayed to the peripheral, and the SNI answers the application with an acknowledge message (IO_ACK service).

5. RESULTS

The proposed SNI was implemented using VHDL, integrated into the HeMPS system, and validated through RTL simulations. Section 5.1 details a contribution of this work, corresponding to the creation of a benchmark suite with IO operations. These benchmarks are required to validate the SNI design. Next, Section 5.2 evaluates the SNI, in terms of communication protocol and the behavior when a malicious task tries to attack the SNI.

5.1 Benchmark for IO Applications

To validate the implementation of the proposed security mechanisms, a set of applications was modified to perform IO communication. Figure 5.1 shows the cyclic task graphs (CTGs) representing the applications used as benchmarks. The design of these modifications takes into account the nature of the application, implementing IO transactions coherent with the function of each task. For example, tasks with MEM or RAM have READ and WRITE operations as it is interacting with shared memory. But it also aims to create a collection of multiple communication profiles, making it possible to study how the system behaves in different scenarios.

- **MPEG and DTW** are examples of simple communication transactions. Each task communicates with only one peripheral and vice versa.
- **AES and Dijkstra** shows the same peripheral communicating with multiple tasks.
- **Fixe_base_test_16** presents a scenario in which one peripheral has to interact with a large number of different tasks.
- **MWD** has one peripheral responsible for dealing with multiple tasks performing both READ and WRITE operations.
- **Synthetic** contains a task accessing multiple peripherals.
- **VOPD and MPEG4** are based on real case scenarios and communicate with a variety of IO devices.

Table 5.1 evaluates the benchmarks when executing IO operations. The second column presents the total number of IO transactions executed by the application, and the third column the number of IO operations per task. The fourth and fifth columns present the absolute execution time, and the remaining columns the execution time overhead. Most applications present a small relative overhead (less than 5%), except synthetic and MWD

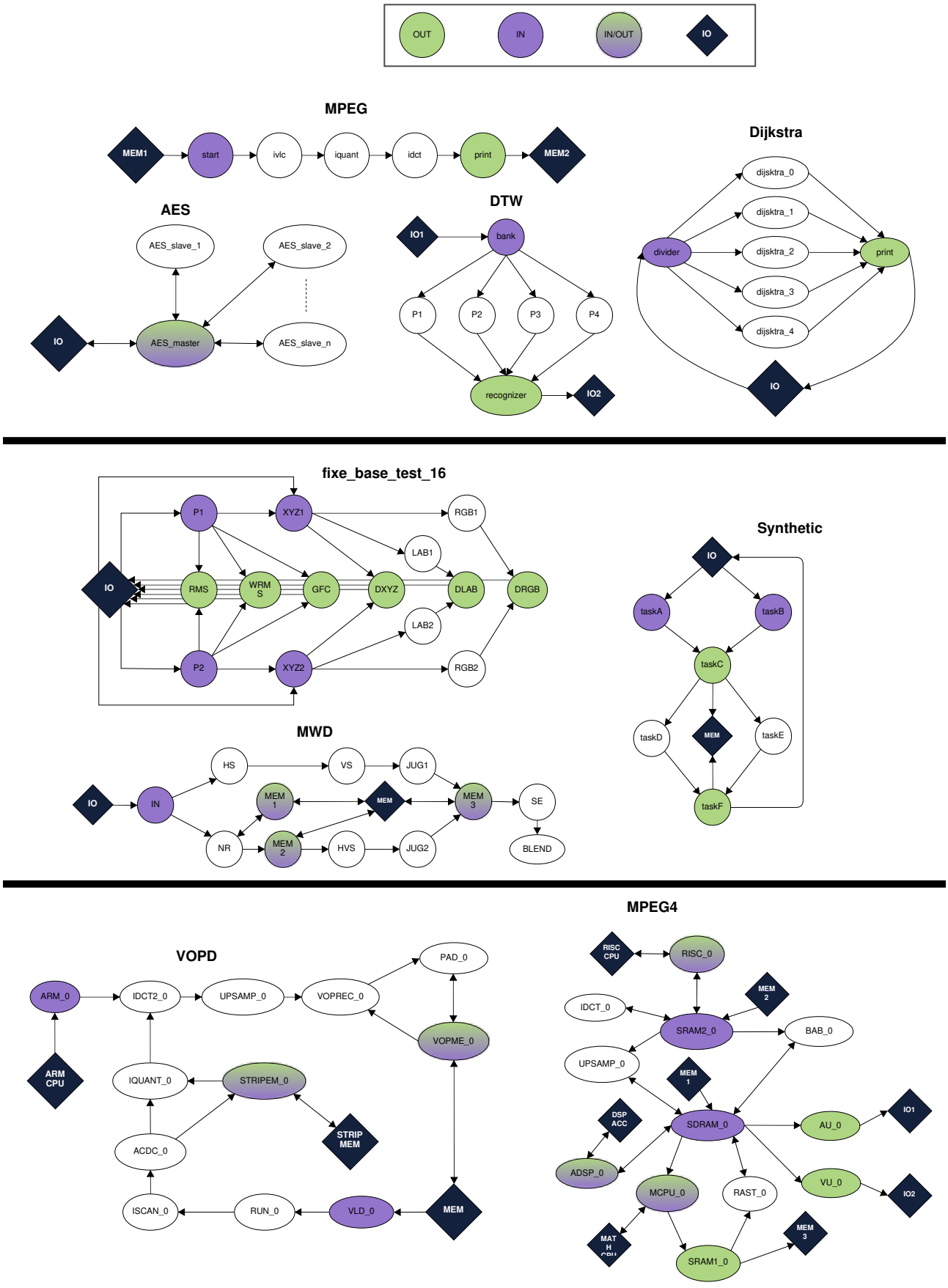


Figure 5.1 – CTGs for the applications adapted to execute IO communication. (Source: Author.)

benchmarks. The large overhead observed in these two applications is due to the fact that they do not execute a real computation, being both applications written only with send-receive primitives. *The main achieved result was the creation of a set of benchmarks to validate the SNI design.*

Table 5.1 – Evaluation of the applications executing IO operations.

Application	IO transactions	IO per task	Execution Time (ms)		Variation (ms)	Percentage
			Baseline	w/ IO		
AES (11 slaves)	5	AESmaster(5)	5.13	5.36	0.23	4.43%
Dijkstra	3	Divider(2), Print(1)	6.29	6.36	0.07	1.08%
DTW (40 iterations)	80	Bank(40), Recog(40)	4.12	4.21	0.08	1.99%
Fixe Base Test 16	10	P1(1), P2(1), RMS(1), WRMS(1), GFC(1), XYZ1(1), XYZ2(1), dXYZ(1), dLAB(1), dRGB(1)	5.25	5.38	0.12	2.33%
MPEG (10 iterations)	20	Start(10), Print(10)	5.06	5.13	0.08	1.54%
MPEG4	11	SDRAM(1), SRAM2(1), SRAM1(1), AU(1), VU(1), RISC(2), MCPU(2), ADSP(2)	24.42	24.78	0.36	1.47%
Synthetic	50	A(1), B(1), C(1), F(2)	1.20	1.49	0.29	23.83%
VOPD	6	ARM(1), VLD(1), STRIPEM(2), VOPME(2)	3.42	3.56	0.13	3.92%
MWD	62	IN(15), MEM1(10), MEM2(16), MEM3(21)	2.71	3.78	1.06	39.18%

5.2 SNI Evaluation

This section shows simulations to exemplify the SNI behavior. It describes each stage of the communication protocol and demonstrates how the SNI reacts to incoming packets of different services. The waveforms depicted in this section were simplified for the sake of clarity.

5.2.1 Initialization

During the *Initialization* phase, the Manager PE sets the value of ***k0*** for the SNI using the **IO_INIT** service. The waveform in Figure 5.2 shows the packet arriving at the SNI and being parsed by the Packet Handler.

The packet is received through the data-NoC interface, where the ***rx*** signal indicates the receiving of a flit via the ***data_in*** bus. The last flit of the packet is signaled by

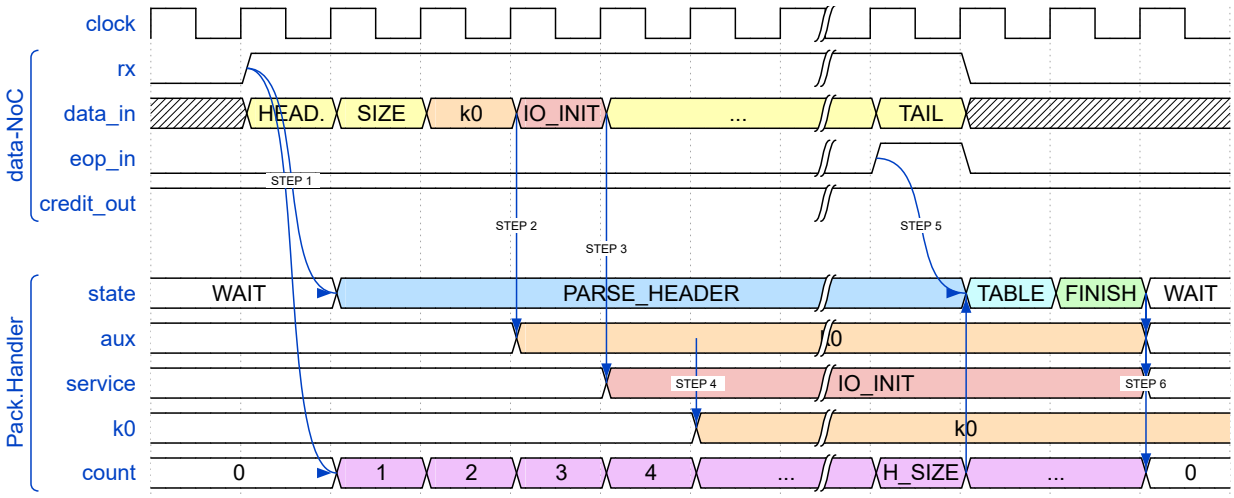


Figure 5.2 – Waveform illustrating the handling of the **IO_INIT** service. (Source: Author.)

eop_in. The signal ***credit_out*** enables or disables incoming communication; while this signal is down, the router will not attempt to send any flit.

The handling flow can be followed by seeing the Packet Handler's ***state***. Upon the packet arrival (step 1) the FSM switches to the *PARSE_HEADER* state, and the incoming flits start to be counted. During this state, relevant fields of the header are saved on their corresponding registers.

The received packet contains two important fields: the ***service*** identifier and ***k0***. As ***k0*** is received before the service is known, its value is stored in an intermediate register until its meaning can be understood (2). Upon the reception of ***service*** (3), the ***k0*** value is transferred to its rightful register (4), where it remains unchanged until the system resets.

After receiving the entire header, no other action is taken, and the SNI moves to the next states (5), converging to *WAIT*. When the service handling is ended, the FSM resets its registers and counters (6), and waits for the next packet to arrive.

5.2.2 Application Configuration

The application configuration process is responsible for registering a new application in the SNI table via **IO_CONFIG** service, thus granting it permission to access the peripheral. This section details and exemplifies this process. First, let's consider an SNI with the Application Table shown in Figure 5.3, where one application already registered.

To configure another application, the MPE must send an **IO_CONFIG** packet to the SNI. Figure 5.4 displays the handling of this packet.

Following the authentication protocol, this packet carries the pair of initialization flits {***i1***,***i2***} used to obtain the credentials of the application. They contain, respectively, the

line	valid	appID	k1	k2	path_to_SZ	path_size
#0	'1'	0x1234	0x62C8	0xA2D4	{ 0x7113, 0x7330, 0x71EE }	3
#1	'0'	--	--	--	--	--
#2	'0'	--	--	--	--	--
#3	'0'	--	--	--	--	--

Figure 5.3 – State of the Application Table before the **IO_CONFIG** service, one application is already registered. (Source: Author.)

obfuscated values of ***appID*** and the pair {***n***,***p***} used to derive the authentication keys. The payload of this packet is loaded with the ***path_to_SZ*** to be configured for the application.

When the packet arrives, the Packet Handler goes to the *PARSE_HEADER* state (step 1). During this state, the authentication flits are received (2, 3) and decrypted using ***k0***, thus retrieving the values of ***appID*** and {***n***,***p***}.

When the ***service*** is received (4), the Packet Handler can make a request for the Application Table to allocate a new line for the application. This is done by raising the ***req*** and ***new_l*** signals in the table interface (5).

While the Packet Handler finishes receiving the header, the Application Table processes the request for a new line. In the state *FETCH_NEW*, the ***line*** counter is incremented each cycle until the table finds an unused line, and then raises the signal ***match***. Then, the FSM transitions to *READY* (6), enabling reading and writing to this line.

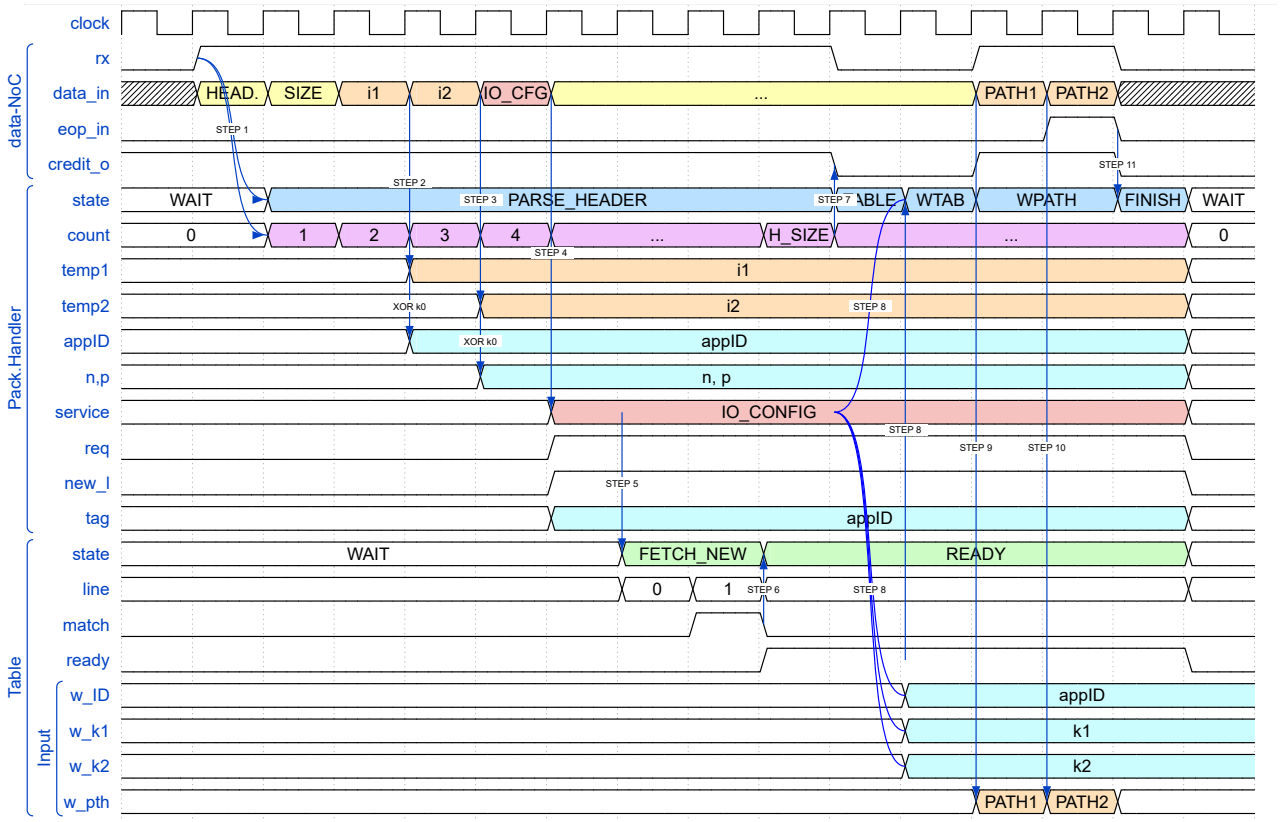


Figure 5.4 – Waveform illustrating the handling of the **IO_CONFIG** service. (Source: Author.)

During this interval, the process of generating {***k1***,***k2***} from ***appID*** and {***n***,***p***} should take place, but this mechanism is not yet implemented in the current version of the SNI.

After receiving the header, the Packet Handler moves to the ***CHECK_TABLE*** state, pausing the reception of incoming flits (7). This state verifies if the table succeeded in allocating the desired line (8).

The actual configuration of the application happens in the next states. During the *WTAB* state, the fields ***appID***, ***k1*** and ***k2*** are written into the table (8). Then, the Packet Handler moves to the *WPATH* state, in which it receives ***path_to_SZ*** from the packet payload, writing it flit-by-flit into the table (9, 10). Once the EOP is received, the Handler returns to its idle state (11), resetting the table request.

After handling this packet, the new Application Table content is the one presented Figure 5.5.

line	valid	appID	k1	k2	path_to_SZ	path_size
#0	'1'	0x1234	0x62C8	0xA2D4	{ 0x7113, 0x7330, 0x71EE }	3
#1	***'1'***	***appID***	***k1***	***k2***	***{ PATH1, PATH2 }***	***2***
#2	'0'	--	--	--	--	--
#3	'0'	--	--	--	--	--

Figure 5.5 – State of the Application Table after the IO_CONFIG service, with two applications registered. (Source: Author.)

5.2.3 Successful Communication

This section provides an example of successful communication between an application and an IO device. The application registered in the previous section sends an *IO_REQUEST* message, asking for data to the peripheral. The SNI is responsible for authenticating the incoming packet and answering the application with the requested data. Figure 5.6 shows the Packet Handler receiving the *IO_REQUEST* message.

During the *PARSE_HEADER* state (step 1), the authentication flits ***f1*** and ***f2*** are saved in registers (steps 2 and 3). When the *service* is received (4), the Packet Handler makes a request for the Application Table to search the corresponding line (5): signals ***req*** and ***crypto*** are raised, and the authentication flits passed through ***tag*** and ***tagAux***.

While the Packet Handler parses the remaining flits of the header, the Application Table process the request. It moves from its idle state to *F_CRYPTO*, to perform the requested Crypto Search. Iterating over all lines, it searches for the application corresponding to the specified {***f1***,***f2***}. When the table locates the matching application, it moves to the *READY* state (6), raising the signal ***ready***. Henceforth, we can say that the request is authenticated.

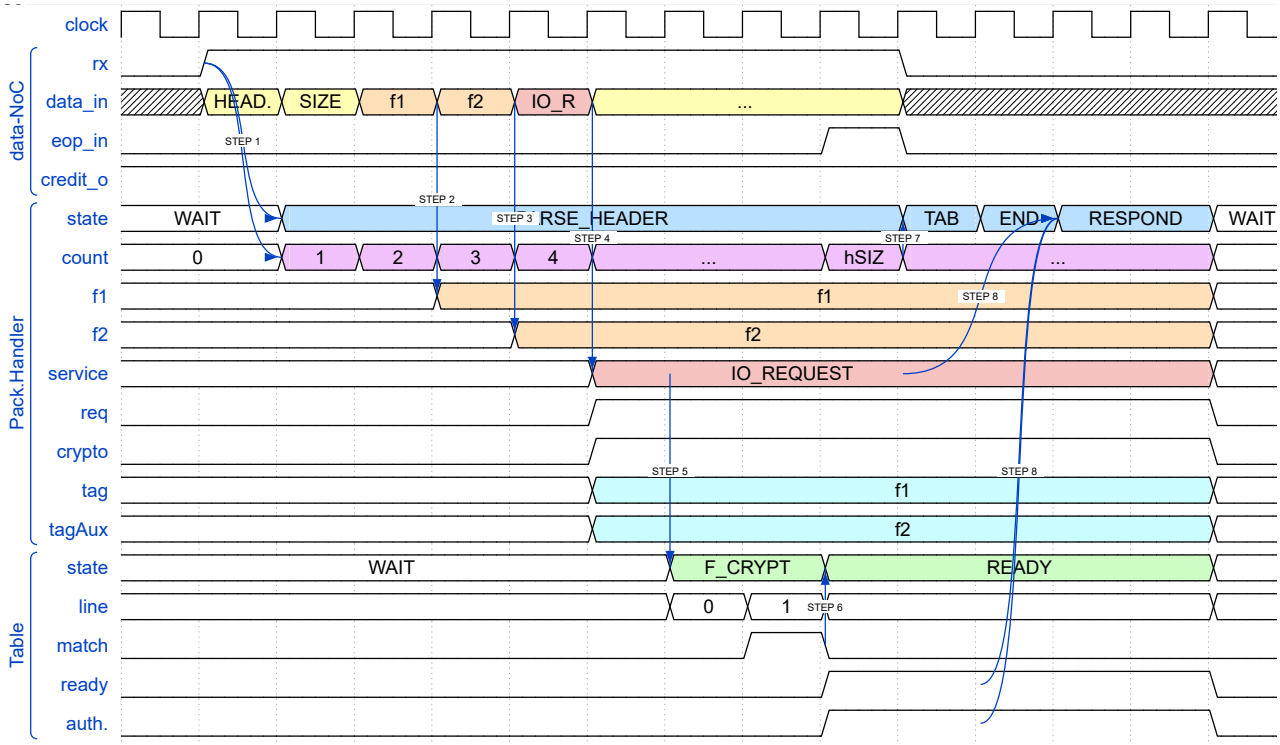


Figure 5.6 – Reception and handling of the **IO_REQUEST** service by the Packet Handler. (Source: Author.)

When the header reception finishes (7), the Packet Handler moves to the *TABLE_CHECK* state and asserts that the packet is indeed authenticated (8). It may now proceed to request the building of a response message, this happens during *RESPOND* state. Once the Packet Builder accepts the request, the Handler may return to the *WAIT* state.

This process of building and transmitting a response message is performed by the Packet Builder, and is depicted in Figure 5.7.

The requesting for the construction of an outgoing packet takes place by the Packet Handler raising the signal ***req_resp*** (step 1), while providing the parameters ***appID*** and ***service***. Upon receiving a request, the Packet Builder uses the *appID* to access the application's line in the table (2). The *CHECK_TABLE* state verifies if the line was successfully found. Since there was a match, the Packet Builder accepts the request by raising ***ack_resp*** and proceeds with the packet assembly.

During the *FIRST* state, the Packet Builder sends the first flits of the header. Then, the *PATH* state configures the source-routing path by reading the contents of ***path_to_SZ*** from the table and injecting it into the packet header (4, 5). The *HEADER* state is responsible for the rest of the header, encoding the credentials of the application {***appID***,***k1***,***k2***} into the authentication flits {***f1***,***f2***} (6, 7).

After the header transmission, the last step is to build the packet payload. During the *DATA_PAYLOAD* state, the Packet Builder reads the peripheral data from the Input Buffer,

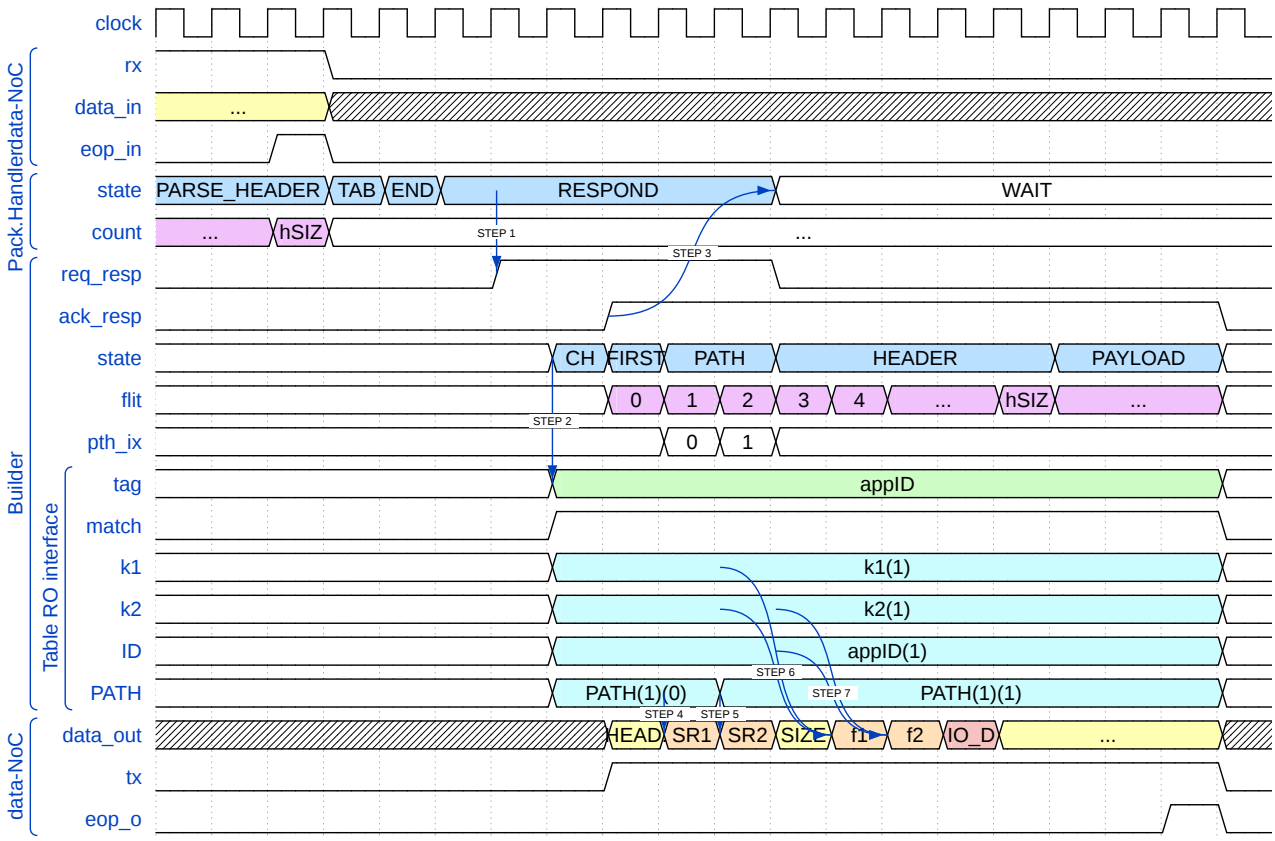


Figure 5.7 – The transmission of a **IO_DELIVERY** message by the Packet Builder. (Source: Author.)

forwarding it to the data network. Upon finishing, it transitions back to the idle state and awaits a new request.

5.2.4 Spoofing Attack

This example shows the reception of a malicious packet trying to steal sensitive information from the peripheral This corresponds to a spoofing attack since the malicious task is trying to forge an application identifier. The incoming *IO_REQUEST* message contains invalid authentication flits and must be ignored by the SNI. Figure 5.8 illustrates the handling of the malicious packet.

The packet handling starts as in the previous section: the packet header is received during *PARSE_HEADER* (step 1); the authentication flits ***f1*** and ***f2*** are saved in registers (2, 3); and, when the ***service*** is received (4), the Application Table is requested to perform a Crypto Search (5).

In this simulation, there is no line in the Application Table matching the specified {***f1***,***f2***} flits. After searching all lines of the Application Table, the FSM goes to the *FAILED* state and raises the signal ***fail***. When the Packet Handler reaches the *TABLE_CHECK* state,

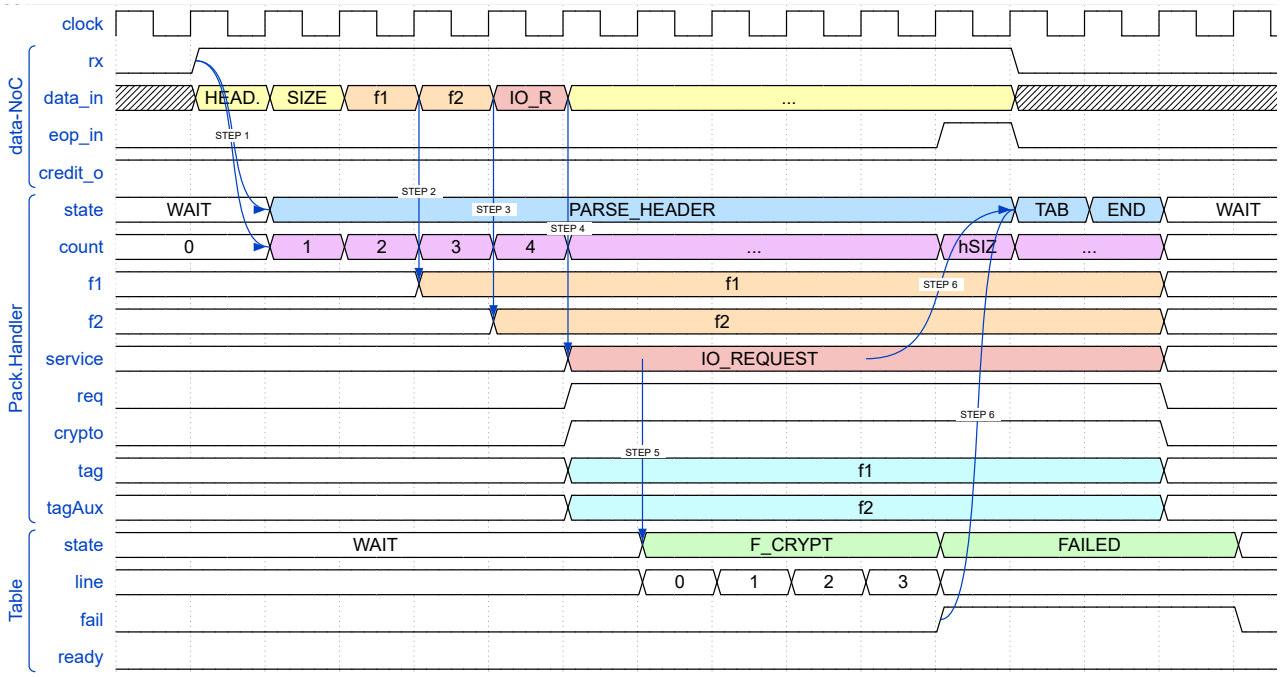


Figure 5.8 – Execution of a spoofing attack. (Source: Author.)

it verifies that the application was not authenticated (6). It moves to the *END* state, dropping the remainder of the packet. The handling is over, and no sensitive data was leaked.

6. CONCLUSION

By embedding an untrusted 3PIP, an NoC-based system becomes exposed to several security threats, such as the leakage of sensitive information. This work complements the OSZ and SeMAP security mechanisms, tackling these vulnerabilities by proposing a Secure Network Interface (SNI) to protect the communication between applications and IO devices.

The set of requirements established for the SNI protects the system from spoofing and flooding attacks performed by (or targeting) the peripheral. The lightweight authentication protocol ensures that only authorized applications can access the device, while the master-slave communication model prevents the peripheral from inserting undesired packets into the network. The SNI module was designed and then implemented using VHDL. The verification was performed through RTL simulations, in which the system runs the applications from the proposed benchmark.

During the development of this work, the author participated in two papers. The first one [Faccenda et al., 2022] (Appendix C) proposes the SeMAP method for mapping OSZs while enabling secure IO communication, which is explained in Section 3.3. The second one (Appendix D), submitted to ISCAS'23, introduces the lightweight authentication protocol described in Section 4.2. Furthermore, a third paper is being written for IEEE Design&Test, detailing the SeMAP approach.

Although this work addresses some of the IO vulnerabilities presented by the baseline platform, there are still others left open for future work. When a secure application communicates with a peripheral, its messages must leave the OSZ, exposing them to attacks that can compromise or corrupt sensitive data. A high-level protocol that monitors the attacks and countermeasures to avoid or mitigate them is a direction to follow.

REFERENCES

Aghaei, B., Reshadi, M., Masdari, M., Sajadi, S., Hosseinzadeh, M., and Darwesh, A. (2020). Network adapter architectures in network on chip: comprehensive literature review. *Cluster Computing*, 23(1):321–346.

Ahmed, M. M., Dhavlley, A., Mansoorz, N., and Dinakarraoy, S. M. P. (2021). What Can a Remote Access Hardware Trojan do to a Network-on-Chip? In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

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.

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

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.

Caimi, L. L. (2019). *Secure Admission and Execution of Applications in NoC-based Many-cores Systems*. PhD thesis, PPGCC-PUCRS. 121p.

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, 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, 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.

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.

Charles, S., Lyu, Y., and Mishra, P. (2020). Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4510–4523.

Charles, S. and Mishra, P. (2020). Securing Network-on-Chip Using Incremental Cryptography. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 168–175.

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.

Faccenda, R. F., Comaru, G., Caimi, L. L., and Moraes, F. G. (2022). Secure Communication with Peripherals in NoC-based Many-cores . In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6.

Fochi, V. M. (2019). *Fault-tolerance at the Management Level in Many-core Systems*. PhD thesis, PPGCC-PUCRS. 108p.

GAPH (2021). Hardware Design Support Group. Source: www.inf.pucrs.br/gaph/, Dec. 2021.

Gondal, H., Fayyaz, S., Aftab, A., Nokhaiz, S., Arshad, M., and Saleem, W. (2020). A method to detect and avoid hardware trojan for network-on-chip architecture based on error correction code and junction router (ECCJR). *International Journal of Advanced Computer Science and Applications*, 11(4):581–586.

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

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)*, pages 166–173.

Jiang, Z., Yang, K., Ma, Y., Fisher, N., Audsley, N. C., and Dong, Z. (2021). I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance. In *Design Automation Conferenc (DAC)*, pages 1159–1164.

Kapoor, H. K., Rao, G. B., Arshi, S., and Trivedi, G. (2013). A Security Framework for NoC Using Authenticated Encryption and Session Keys. *Circuits Syst Signal Process*, 32(6):2605–2622.

Lee, C., Cho, J., Kim, J., and Jin, H. (2021). Transparent many-core partitioning for high-performance big data I/O. *Concurr. Comput. Pract. Exp*, 33(18):1–11.

Li, H., Liu, Q., and Zhang, J. (2016). A survey of hardware Trojan threat and defense. *Integration, the VLSI Journal*, 55:426–437.

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.

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

Peckham, O. (2020). Esperanto Unveils ML Chip with Nearly 1,100 RISC-V Cores. https://www.hpcwire.com/2020/12/08/esperanto-unveils-ml-chip-with-nearly-1100-risc-v-cores.

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, 290p.

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

Rauber, T. and Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. Springer, 2nd edition.

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

Ruaro, M., Caimi, L. L., Fochi, V., and Moraes, F. G. (2019). Memphis: a Framework for Heterogeneous Many-core SoCs Generation and Validation. *Design Automation for Embedded Systems*, 23(3-4):103–122.

Sharma, G., Kuchta, V., Sahu, R. A., Ellinidou, S., Bala, S., Markowitch, O., and Dricot, J. (2019). A twofold group key agreement protocol for NoC-based MPSoCs. *Transactions on Emerging Telecommunications Technologies*, 30(6):1–18.

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.

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

Vaas, S., Ulbrich, P., Eichler, C., Wägemann, P., Reichenbach, M., and Fey, D. (2021). Taming Non-Deterministic Low-Level I/O: Predictable Multi-Core Real-Time Systems by SoC Co-Design. In *ISORC*, pages 43–52.

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.

APPENDIX A – COMPLETE AUTHENTICATION PROTOCOL

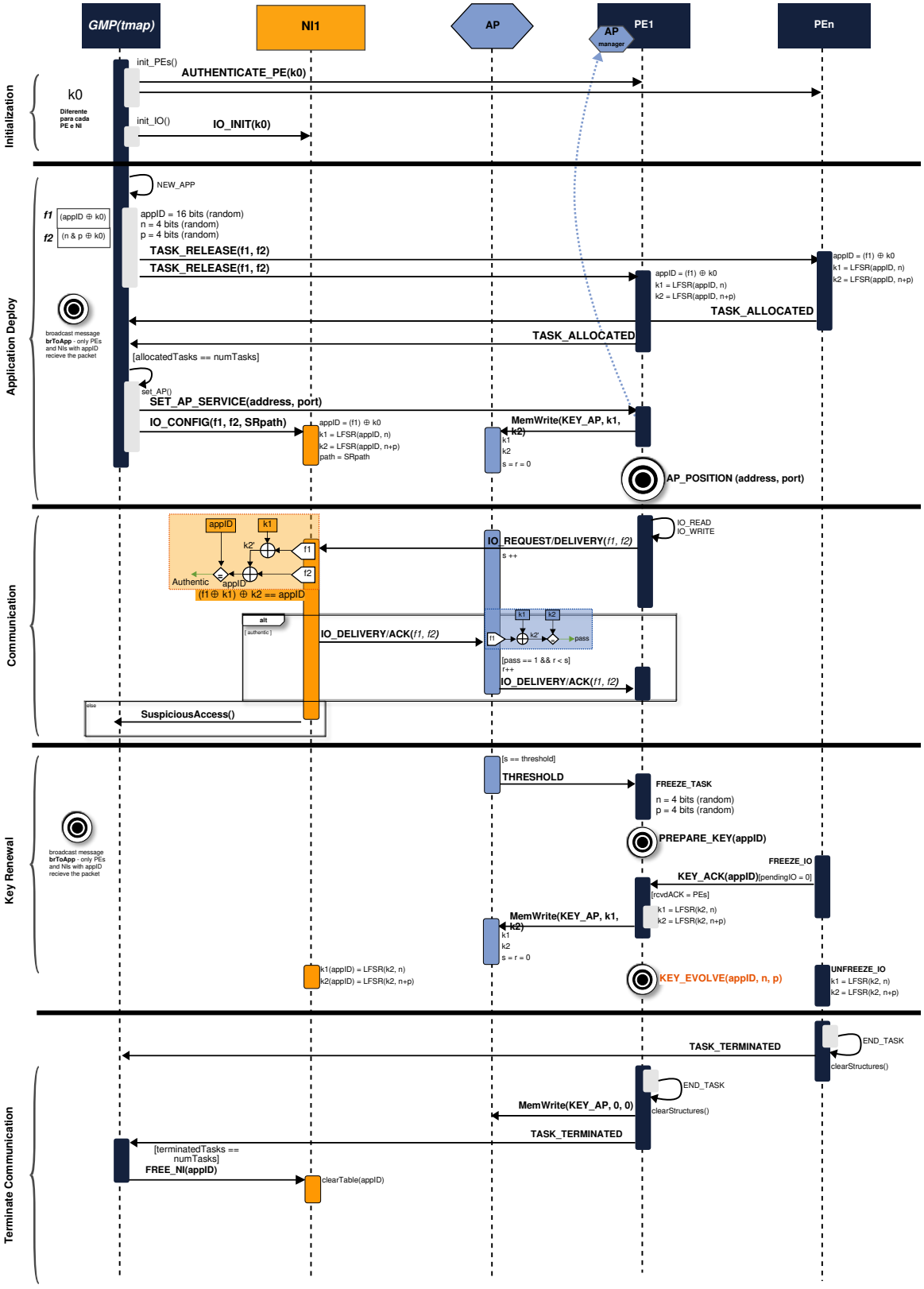


Figure A.1 – Sequence diagram of the entire authentication protocol. (Source: Author.)

APPENDIX B – PACKET HANDLER'S FSM

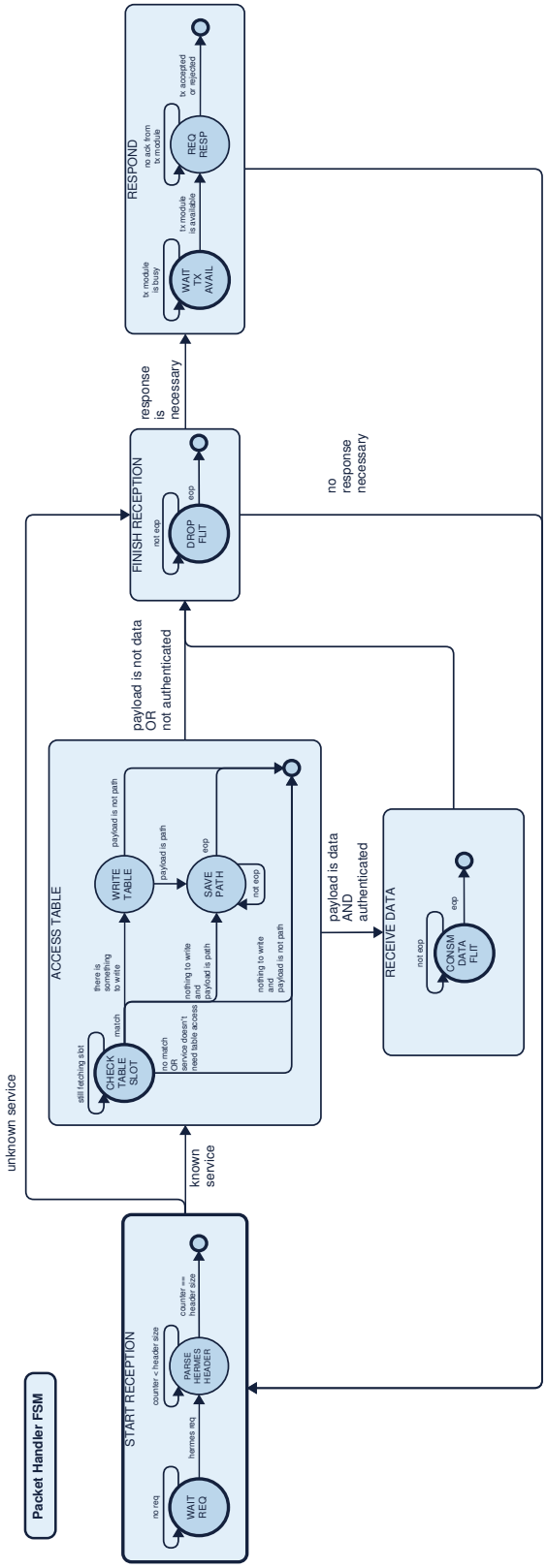


Figure B.1 – The Packet Handler's state diagram, comprising all states. (Source: Author.)

APPENDIX C – SECURE COMMUNICATION WITH PERIPHERALS IN NOC-BASED MANY-CORES

Paper published at SBCCI'22:

Secure Communication with Peripherals in NoC-based Many-cores
FACCENDA, RAFAEL FOLLMANN; COMARU, Gustavo; CAIMI, LUCIANO L.; MORAES, Fernando Gehm
In: SBCCI, 2022
https://ieeexplore.ieee.org/document/9893244

Secure Communication with Peripherals in NoC-based Many-cores

Rafael Follmann Faccenda*, Gustavo Comarú*, Luciano Lores Caimi†, Fernando Gehm Moraes*
*School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
{rafael.faccenda, gustavo.comaru}@edu.pucrs.br, fernando.moraes@pucrs.br
†UFFS, Federal University of Fronteira Sul, Chapecó, Brazil – lcaimi@uffs.edu.br

***Abstract*—Many-core systems-on-chip (MCSoCs) contain processing elements (PEs), peripherals attached to the system, and an NoC connecting them. These systems have different flows traversing the NoC: PE-PE and PE-peripheral flows. Malicious hardware or software can hinder system security due to the resource sharing feature, such as CPU sharing for multitasking or sharing NoC links for flows belonging to different applications. Methods that isolate applications with security constraints, such as Secure Zones (SZs), protect PE-PE flows against most of the attacks reported in the literature. Proposals with methods to secure the communication with peripherals in the literature are scarce, with most of them focusing on shared memory protection. This paper presents an original approach, Secure Mapping with Access Point - SeMAP, which creates mapping policies for SZs, and communication strategies with IO devices, to protect PE-peripheral flows. Results show that the application execution time is not penalized by applying SeMAP, presenting advantages compared to a state-of-the-art approach. In terms of security, SeMAP successfully resisted an attack campaign, blocking malicious packets attempting to enter the SZ.**

***Index Terms*—Security, NoC-based Many-cores, Secure Zones, Peripherals.**

I. Introduction

Many-core systems on chip (MCSoCs) provide high computing performance due to the parallelism offered by the numerous resources inside the chip. Current applications have increasing demands on dedicated resources, such as shared memories, hardware accelerators (e.g., neural engines), and communication interfaces [1]. Thus, MCSoCs should contain, besides the set of processing elements (PEs), support for peripherals leading to the adoption of *heterogeneous architectures* instead of homogeneous ones (e.g., MPSoCs). Figure 1 presents a 4x4 MCSoC instance, with four peripherals attached to the NoC borders.

A consequence of the increasing number of features and functionalities in MCSoCs is the adoption of third-party IPs (3PIPs) to meet time-to-market constraints and reduce design costs. Such IPs come from different vendors, raising the risk of having malicious hardware and/or software inserted in the design [2]. Thus, *security* is a major design constraint.

Malicious hardware/software can hinder system security due to the resource sharing feature, such as CPU sharing for multitasking or sharing NoC links among flows belonging to different applications. Thus, methods that isolate applications with security constraints (App_{sec}) [3, 4] protect applications against most of the attacks reported in the literature. Secure Zones (SZ, in Figure 1) [5, 6] is an example of a defense mechanism based on spatial isolation. SZs reserve PEs and links to execute an App_{sec} without sharing the resources inside the SZ with other applications.

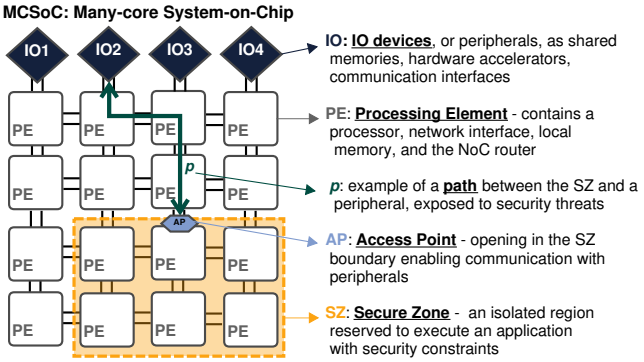


Fig. 1. MCSoC and terminology adopted in this work.

Literature related to many-core that present methods to secure the communication with peripheral are scarce, with most of them focusing on shared memory protection [7, 8]. On the other hand, several works present MCSoCs with peripherals but without security concerns [9, 10]. Therefore, there is a gap to fulfill: *how to protect the communication of applications with peripherals*?

A possible solution is to encrypt data in the communication path between the SZ and peripherals. However, cryptography is a partial solution, only ensuring confidentiality. Other security threats may compromise the communication path SZ↔IO (***p*** in Figure 1):

1) Denial of service (DOS) and side-channel attacks in the SZ↔IO path;
2) Unauthorized access to the IO. IO devices must be aware of applications with access rights to avoid attacks by malicious entities;
3) Unauthorized access to the App_{sec} running into the SZ. Communication with IO devices requires opening access points (AP, in Figure 1) at the SZ border. The AP is a vulnerability that malicious applications can explore to access the App_{sec};
4) DOS due to the lack of paths to the IO devices. A given application or SZ may isolate IO devices. Thus, reachability is a design concern, ensuring a path between SZs↔IOs.

978-1-6654-8128-1/22/$31.00 ©2022 IEEE

The *goal* of this work is to define SZ mapping policies and communication strategies with IO devices, addressing issues 3 (APs) and 4 above (ensure reachability to IO devices).

The *original contribution* is the Secure Mapping with Access Point ($SeMAP$) approach, which enables the mapping of multiple SZs simultaneously, protecting App_{sec}s against unauthorized accesses, ensuring the availability of paths to the IO devices.

This paper is organized as follows. Section II presents the related work regarding peripherals in many-cores. Section III discusses the threat model assumed in this work. Section IV presents two methods to protect the communication with peripherals, DSZ and $SeMAP$, being $SeMAP$ the original contribution of this paper. Section V evaluates both methods in terms of performance and security. Section VI presents the conclusions and directions for future work.

II. Related Work

Recent works present many-cores with peripherals attached to them, addressing communication performance improvement [9, 10] or timing predictability [11, 12, 13].

Lee et al. [10] propose a message-based system calls to enhance the performance of storage IO for the MapReduce application model in many-cores. In addition, the Authors explore the intracluster locality of task allocation in the cores. As a result, the execution time of MapReduce was reduced by 29%.

Jiang et al. [11] optimize IO operations in safety-critical systems with Virtual Machines. The proposal is a hardware hypervisor to shorten the overhead of the IO communication in an application inside a VM, called *IO-GUARD*. The main objective is to enhance the IO path and resource management.

Vaas et al. [12] also focus on safety-critical systems, proposing the LOW_{IO}, an interface that reduces the interference of low-level non-deterministic IO operations on critical tasks. Hardware units control the access to peripherals, giving priority to critical transactions.

Zhao et al. [13] propose dedicated IO co-processing units and a scheduling model to provide predictability for hard real-time systems. A module named IO Processing Unit (IOPU) controls the IO tasks. The objective is to make the communications predictable.

In terms of security, Ehret et al. [14] focus on securing edge devices against attacks on their IO ports. This approach considers that the devices are installed away, making it possible for a malicious user to access the system from its ports manually. Even though this work does not consider a many-core, it is possible to apply the adopted threat model to a many-core.

Grammatikakis et al. [7] propose an NoC firewall to protect the access of a shared memory accessed through the NoC, avoiding sensitive data corruption or access of an unauthorized element. The firewall isolates the NoC, only allowing an authorized process of the MCSoC to access the memory.

Reinbrecht et. al. [8] also focus on the security of MCSoCs with shared memories. The authors propose two new attack types targeting shared memories, called Prime+Probe Arrow and Prime+Probe Firework, that can affect systems with Secure Zones when the application running inside it needs to use the shared memory. The Authors propose the *Gossip-NoC*, which includes a traffic monitor. When an anomalous behavior is detected, the monitor sends an alert message to a system manager, which changes the routing algorithm from XY to YX, avoiding malicious traffic.

In summary, proposals [9, 10, 11, 12, 13, 15] optimize the communication performance with peripherals, or systems with real-time constraints. On the other hand, the concern of [7, 8] is to protect access to shared memory. General security methods to protect access to peripherals other than shared memories are a gap in the literature. Actual many-cores have a rich set of accelerators besides shared memories, requiring the availability of security mechanisms to protect the communication.

III. Threat Model

Resource sharing in PEs and NoC links introduces vulnerabilities to the applications. Considering the reference MPSoC architecture (Figure 1), the attack surface includes the access point (*AP*) and the exposed path (*p*). Malicious entities (tasks or peripherals) may explore this surface in attacks such as:

i Spoofing: falsification of identity – a malicious entity could try to pass through the AP, pretending to be a trustworthy peripheral;
ii DoS (flooding): a malicious entity could attempt to flood the SZ by injecting packets through the AP;
iii DoS (blocking): Hardware Trojans (HTs) may block, drop, or misroute flows to/from the peripherals;
iv Snooping: once the packet leaves the SZ, it is exposed to malicious entities, being vulnerable to snooping attacks.
v SCA: a malicious entity could monitor the exposed flows to execute, e.g., timing attacks [8].

Our proposal addresses threats (i) and (ii) using a key shared by the App_{sec} and the IO. This key ensures that a given packet can only enter or leave the SZ if it has the correct key. The communication protocol mitigates the threat (iii), which can detect when an expected answer to a transaction does not reach the SZ. Encrypting data in the exposed path mitigates snooping attacks. We assume in this work that packets in the exposed path are encrypted to avoid such attacks. SCA mitigation is out of the scope of the current work. Key exchange between the peripheral Network Interface (NI) and the SZ is discussed in [16].

The focus of this work is the proposal of methods to enable secure communication of applications executing in isolated resources (Secure Zone) with peripherals (outside the Secure Zone), *not* on proposing countermeasures. Examples of countermeasures include the dynamic changing of the AP location when detecting an attack or using methods to locate the attack source [17].

IV. Safe Communication with Peripherals

This work adopts the opaque SZ model [18]. Once the SZ is closed, all traffic trying to cross it is re-routed. The opaque SZ method prevents the attacks described in the threat

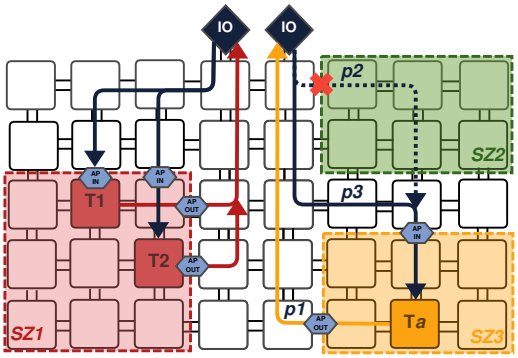


Fig. 2. Example of Dynamic SZ method, adapted from [18]

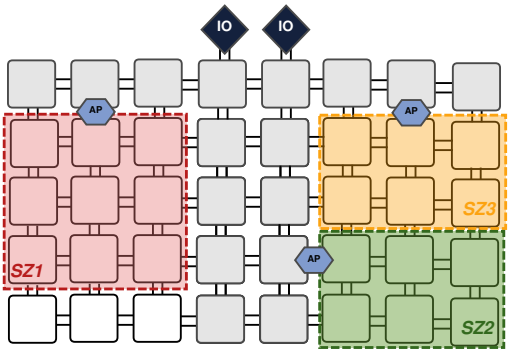


Fig. 3. Example of gray and secure areas. Three App_{sec}s mapped on the secure area, each one with an Access Point (AP).

model, including timing and logical SCA [19]. However, to allow communication between peripherals and App_{sec}, it is necessary to open the SZ boundary. The controlled opening at the boundary of the SZ is called *access point* (AP).

Opening the SZ does not violate the fundamental rule governing the SZ method: flows belonging to applications other than App_{sec} must not cross the SZ. Opening the SZ to peripherals requires a set of rules to ensure the security of App_{sec}. Work [18] presents a set of rules to meet security constraints:

1) Differentiate PE↔PE from PE↔IO communication. This differentiation prevents malicious applications from trying to inject packets into SZs.
2) Master-slave communication. PEs inside the SZ initiate all transactions with peripherals. Any unexpected packet arriving in an AP is discarded.
3) Add a key in IO packets, ensuring their authenticity and source.
4) Avoid unreachable resources, i.e., an SZ may not block the access to peripherals.

We present below two methods for communicating with peripherals: (i) *Dynamic SZ*, initially presented in [18]; (ii) Secure Mapping with Access Point ($SeMAP$), herein proposed.

A. Dynamic SZ – DSZ

The DSZ method is flexible in terms of App_{sec} mapping. SZs can be mapped at any region of the MCSoC, respecting the restriction of not blocking paths to peripherals. Figure 2 illustrates a system with 3 SZs.

APs are established for each communication transaction, using XY routing by default. The task that starts an IO communication opens two unidirectional APs (AP IN and AP OUT in Figure 2), transmitting two control packets to the SZ border. Each AP is closed when a packet traverses it. This method ensures that only one packet traverses the AP per transaction, minimizing attack attempts. On the other hand, if multiple tasks communicate with peripherals, several APs can be opened simultaneously ($SZ1$ in the Figure), increasing the attack surface. Packets to traverse the APs must meet two conditions: (i) be IO packets; (ii) match the key shared by the peripheral and the App_{sec}. The packet is discarded otherwise.

The mapping flexibility brings the *masking effect*. Consider Figure 2 having the $SZ1$ and $SZ3$ mapped and running in the system. When $SZ2$ enters in the system, it blocks path $p2$, from the IO device to $SZ3$. Thus it is necessary to compute a new path using source routing. The PE closest to the IO device computes the new path ($p3$), transmitting it to the IO device to be used in the subsequent data transmissions. This approach adds a security threat, as it involves a PE not related to App_{sec}, allowing it to know the location of the AP and use this information to initiate an attack.

Despite the DSZ application mapping flexibility, there is a restriction related to the task mapping inside the SZ, named *alignment effect*. The DSZ does not allow two or more tasks on the same X or Y coordinate to communicate with peripherals because the DSZ authorizes only one transaction per AP. If two tasks are aligned, both activate the same AP, but only one packet passes through it, thus blocking one of the tasks.

B. Secure Mapping with Access Point – SeMAP

Our proposal, named $SeMAP$, restricts the App_{sec} mapping and allows only one bidirectional AP per SZ. The goal is to have a single aperture for all the IO transactions. Our mechanism creates two logical regions, at system startup: *Gray* areas (GA) run applications without security requirements and guarantee a path between App_{sec} and peripherals, i.e., reachability. *Secure* areas only run App_{sec}s. Figure 3 illustrates an example of these two regions, with three App_{sec} mapped in the secure areas. The peripherals are attached to the North side of the system for the sake of simplicity. The approach does not restrict peripherals attached to a given system side. The mapping of App_{sec}s requires at least one side juxtaposed to a GA in such a way to have a path to the peripherals.

The process to deploy an App_{sec} into the secure area requires four steps, detailed below.

1) SZ Shape and Location: The definition of the SZ shape prioritizes shapes having the width of the secure area. This method improves system utilization, avoiding PEs without access to the gray areas. The SZ shape and coordinates selection follow a Sliding Search Window (SSW) algorithm. The starting point of the SSW is the row nearest to the peripherals. The result of this step is a set of PEs reserved to execute the App_{sec}.

2) AP Definition: Any port of any frontier router next to a GA may receive the AP. The default location is the top-left or top-right router according to the gray area position. The second case is the north port of the top-middle router if the SZ is near to the top of the GA. Figure 3 presents both cases: the AP of the $SZ1$ and $SZ3$ are in the middle-top position, while the $SZ2$ is the default case, with the AP at the top-left position. Since any port can become an AP, there is the possibility to change the AP location periodically or whenever suspicious behavior is detected.

3) Task Mapping in the Selected Shape: The system manager maps tasks that communicate with peripherals near the AP and the remaining tasks according to the hop number between communicating pairs. After the mapping execution, the SZ borders are "closed", isolating the SZ. Only packets to/from peripherals can cross the SZ through the AP.

4) Path configuration: PEs inside the SZ does not use the XY routing algorithm to reach the peripheral. The first communication of a given task with a peripheral fires a path configuration heuristic. First, the OS computes the path PE→AP, then the path AP→peripheral, according to the gray area shape. The path from the peripheral to the PE is generated using the opposite ports in the reverse order. Figure 4 illustrates an example of a path computation between "Task" and IO. There are two paths: path A, the orange arrow from Task→IO, and B, the blue arrow from IO→Task. The circles show each of the ports taken for each of the paths. After computing the path to the peripheral, the next step is to send the reverse path to the peripheral. The OS sends the path to the peripheral, which stores the path and uses it for every communication with that specific task.

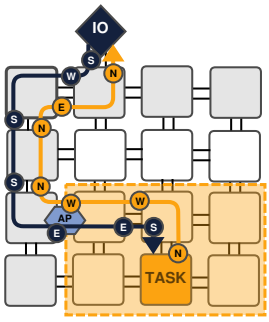


Fig. 4. Example of source routing paths from/to task to/from peripheral through the AP.

$SeMAP$ contains *security mechanisms* implemented in the APs. Packets to/from a peripheral only traverse the AP if the key embedded in the packet header matches the shared key. In addition to key verification, the AP monitors the frequency of incoming and outgoing packets, generating a suspicious alert if a threshold rate is reached. Such behavior may signalize an external peripheral (incoming packets) or App_{sec} is attempting to execute an attack.

V. Results

This section presents the performance of applications considering the two methods for communicating with peripherals. The second part of the results discusses the security aspects of the communication with peripherals. Section V-C compares both approaches.

Experiments use as baseline system the Memphis MCSoC [20], modeled at the RTL level (SystemC and VHDL). The MCSoC uses two NoCs: one for data and one for control. The data-NoC is a packet switching network with two physical channels, supporting XY (default) and source routing (when rerouting is necessary). The control-NoC is a broadcast NoC with single-flit packets and a search path mechanism, which allows path discovery for source routing [21].

A. Performance of the Communication with Peripherals

Figure 5 presents the applications mapping to evaluate the methods of communication with peripherals. The system receives the DTW (Dynamic Time Warping) application at startup. At *5 ms*, a new application enters the system (MPEG decoder). Note that the DTW DSZ (Figure 5(a)) has two paths broken by the MPEG, firing two path search computations (due to the "masking effect"). At *9 ms*, a PC (Producer-Consumer) is mapped, also blocking two DTW paths. A second scenario is evaluated, swapping the DTW with MPEG (the reference application is always mapped in the bottom-left corner of the system). In the first scenario (DTW in the left corner), the IO communication volume (number of messages exchanged with the peripherals) is higher.

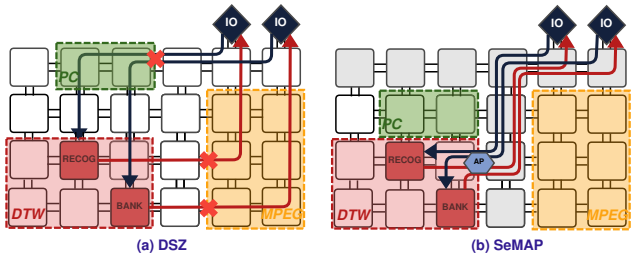
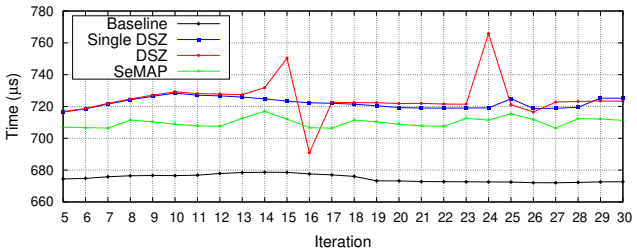


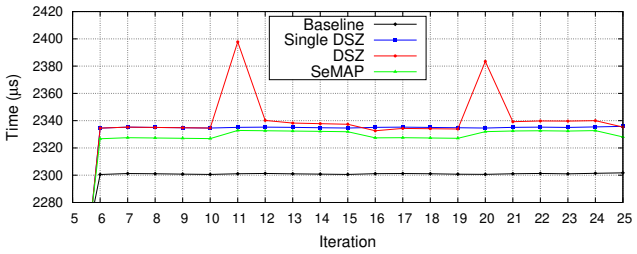
Fig. 5. Application mapping to evaluate the methods to communicate with peripherals.

Figure 6 presents the iteration latency for DTW and MPEG applications. The y-axis is the time required to execute each application iteration (in μs), and the x-axis is the iteration number. Graphs omit the first five iterations, considering these as the warm-up period. Each graph has 4 curves:

- **Baseline** (black line): execution of the applications without communication with peripherals. Input data is assumed to be stored in the local memories, and results are also stored in the local memories. Simulating the baseline MCSoC aims to evaluate the overhead due to the communication with peripherals.
- **Single DSZ** (blue line): only the reference application (DTW or MPEG) executes in the system. The goal of simulating the DSZ approach without other applications is to evaluate the DSZ method in the absence of the masking effect.
- **DSZ** approach (red line) using evaluation scenario presented on Figure 5(a).
- **SeMAP** approach (green line) using evaluation scenario presented on Figure 5(b).



(a) DTW iteration latency. Execution time (ms): 18.26 (baseline), 18.44 (single DSZ), 18.51 (DSZ), 18.37 ($SeMAP$).



(b) MPEG iteration latency. Execution time (ms): 13.82 (baseline), 13.91 (single DSZ), 13.94 (DSZ), 13.98 ($SeMAP$).

Fig. 6. Iteration latency using the baseline MCSoC, DSZ and $SeMAP$.

Figure 6 shows that:

- Comparing **SeMAP** and **Single DSZ** versus **Baseline**, the latency per iteration increases 1.2% (MPEG) to 7.3% (DTW) when there is IO communication (average values). The latency increases due to the: (i) non-minimum paths; (ii) master-slave communication protocol, i. e., all transactions started by the App_{sec}; (iii) management of APs in the DSZ method (opening and closing of APs at each transaction).
- Total execution time has a minimal overhead - 1.3% for DTW (**Baseline** versus **DSZ**) and 0.5% for MPEG (**Baseline** versus **SeMAP**).
- **SeMAP** reduces the latency per iteration (0.33% for MPEG and 2.7% for DTW, best cases) compared to **Single DSZ** because it does not need to manage APs. On the other hand, there is a latency per AP to start the application execution, as it is necessary to compute the paths for each AP (25μs@100MHz - average value per path).
- The masking effect, observed in **DSZ** (red curves), increases the iteration latency when a new application enters the system, blocking the PE→Peripheral communication, requiring to reroute the broken paths. In both scenarios, the masking effect only affects few iterations, since this mechanism is activated once, being the alternative path taken for the subsequent communications. The valley observed in the first scenario is due to the application pipeline behavior, i.e., while a task is blocked waiting for a new path, the other tasks continue to run. The performance degradation increases in scenarios with a larger number of broken paths,
- **SeMAP** is immune to the masking effect, presenting a small latency increase (0.2% to 1.8%) when a new application enters the system. The network traffic increases when a new application is admitted due to the transmission of the object code of the tasks. This increase in network traffic explains the slight increase observed in latency.

B. AP Security Evaluation

We executed an attack campaign with three different packet types arriving on APs: (i) application packets (PE-PE); (ii) IO packets with incorrect key; (iii) IO packets with a forged key. In scenarios i and ii, the APs correctly dropped the packets and notified the arrival of a suspicious packet to the system manager.

The master-slave protocol adds a random sequence number in the request packet (for read or write operations). The IO must answer with this number. Consider scenario iii, where the malicious peripheral forges the key and the AP address. In both methods, the packet reaches the PE, which notifies the system manager to isolate the malicious peripheral upon the reception of an unexpected packet or a packet with a wrong sequence number (it would be costly in terms of silicon area to have registers in the AP to store a list of malicious peripherals). Thus, it is necessary to meet four conditions to execute a successful attack (i) correctly forge the key; (ii) send the packet to the AP address; (iii) insert the packet into the SZ when a task is waiting for a peripheral answer; (iv) generate the correct sequence number. We consider that the fulfillment of these four conditions has a minimal probability of occurring, being sufficient to guarantee a secure communication between App_{secs} and peripherals.

C. Discussion

According to the threat model (Section III), both DZS and $SeMAP$ methods avoid spoofing and DoS (flooding) by detecting malicious packets arriving at an AP, blocking them, and notifying the address of the malicious entity (PE ou peripheral) to the system manager. The communication API avoids DoS (blocking) attacks when the communication started in the SZ does not receive an answer after a given period (watch-dog timer).

Table I compares the DSZ and $SeMAP$ methods qualitatively. The DSZ is recommended for scenarios with few SZs coexisting simultaneously (due to the masking effect) and a few tasks communicating with IO (due to the alignment effect). *The proposed SeMAP is a generic approach to map* App_{secs}, without the restrictions observed in the DSZ method. Despite the advantages, $SeMAP$ has limitations related to the use of resources and possible congestion in the AP. It is possible to mitigate the first limitation with defragmentation techniques, and the second limitation would only occur in cases of very intense communication with peripherals.

VI. Conclusion

The Introduction raised the following question: *how to protect the communication of applications with peripherals*? The answer is to use the DSZ or $SeMAP$ methods, which secure the SZ↔IO communication against spoofing, DoS,

TABLE I
DSZ AND SEMAP QUALITATIVE COMPARISON.

	DSZ	SeMAP
PROS	Better resource utilization due to the mapping flexibility	Single bidirectional AP per App_{sec}, reducing the attack surface
	Communication with peripherals is not concentrated in a single AP (better traffic distribution)	The AP stays opened during the App_{sec} execution, avoiding configuration messages per IO transaction, reducing its management cost
		No alignment effect
		No masking effect
		The nearest PE to the peripheral does not need to be interrupted to compute a path to the SZ and is unaware of the AP position
CONS	Masking effect (Section IV-A)	Smaller resource utilization than DSZ due to the partition of the system into *gray* and *secure* areas
	Alignment effect (Section IV-A)	Fragmentation of the secure areas at runtime. It is possible to defragment the system using task migration
	Several APs opened in SZ simultaneously, increasing the attack surface	Peripheral traffic concentrated in a single AP may lead to NoC congestion

and snooping attacks. DSZ is flexible in mapping the SZ at any place of the MCSoC but presents limitations related to the SZ↔IO paths and APs (larger attack surface). $SeMAP$ adopts a restrictive mapping (secure and gray areas) and one bidirectional AP per SZ. Results show that the iteration latency increases up to 7.3% when communicating with peripherals, but the overhead is minimal considering the total execution time (worst-case 1.3%).

$SeMAP$ is the architecture to adopt due to the absence of the limitations compromising the DSZ method.

Future work includes: (i) define a Secure NI to be inserted between the MCSoC and an IO device; (ii) add a reservation protocol in the IO communication API to avoid a peripheral answering to a malicious request; (iii) study defragmentation techniques to be deployed at runtime.

ACKNOWLEDGMENT

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES (Finance Code 001), and CNPq (grant 309605/2020-2).

REFERENCES

[1] A. Kamaleldin and D. Göhringer, "AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-V Processors," *IEEE Access*, vol. 10, pp. 43 895–43 913, 2022.
[2] H. Li, Q. Liu, and J. Zhang, "A survey of hardware Trojan threat and defense," *Integration, the VLSI Journal*, vol. 55, pp. 426–437, 2016.
[3] L. Caimi, R. Faccenda, and F. G. Moraes, "A Survey on Security Mechanisms for NoC-based Many-Core SoCs," *Journal of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 1–15, 2021.
[4] S. P. Azad, G. Jervan, M. Tempelmeier, and J. Sepúlveda, "CAESAR-MPSoC: Dynamic and Efficient MPSoC Security Zones," in *ISVLSI*, 2019, pp. 477–482.
[5] S. Pinto, P. Machado, D. Oliveira, D. Cerdeira, and T. Gomes, "Self-secured devices: High Performance and Secure I/O Access in TrustZone-based Systems," *J. Syst. Archit*, vol. 119, p. 102238, 2021.
[6] E. M. Benhani, C. M. López, and L. Bossuet, "Secure Internal Communication of a Trustzone-Enabled Heterogeneous SoC Lightweight Encryption," in *FPT*, 2019, pp. 239–242.
[7] M. D. Grammatikakis, P. Petrakis, A. Papagrigoriou, G. Kornaros, and M. Coppola, "High-level Security Services based on a Hardware NoC Firewall Module," in *WISES*, 2015, pp. 73–78.
[8] C. Reinbrecht, A. A. Susin, L. Bossuet, G. Sigl, and J. Sepúlveda, "Timing attack on NoC-based systems: Prime+Probe attack and NoC-based protection," *Microprocess. Microsystems*, vol. 52, pp. 556–565, 2017.
[9] C. Lee, J. Lee, D. Koo, C. Kim, J. Bang, E.-K. Byun, and H. Eom, "Towards enhanced I/O performance of a highly integrated many-core processor by empirical analysis," *Cluster Computing*, pp. 1–13, 2021.
[10] C. Lee, J. Cho, J. Kim, and H. Jin, "Transparent many-core partitioning for high-performance big data I/O," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 18, 2021.
[11] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. C. Audsley, and Z. Dong, "I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance," in *DAC*, 2021, pp. 1159–1164.
[12] S. Vaas, P. Ulbrich, C. Eichler, P. Wägemann, M. Reichenbach, and D. Fey, "Taming Non-Deterministic Low-Level I/O: Predictable Multi-Core Real-Time Systems by SoC Co-Design," in *ISORC*, 2021, pp. 43–52.
[13] S. Zhao, Z. Jiang, X. Dai, I. Bate, I. Habli, and W. Chang, "Timing-Accurate General-Purpose I/O for Multi- and Many-Core Systems: Scheduling and Hardware Support," in *DAC*. IEEE, 2020, pp. 1–6.
[14] A. Ehret, E. D. Rosario, C. Schwicking, K. Gettings, and M. A. Kinsy, "Reconfigurable Hardware Root-of-Trust for Secure Edge Processing," in *HPEC*, 2021, pp. 1–7.
[15] A. Suyyagh and Z. Zilic, "Energy and Task-Aware Partitioning on Single-ISA Clustered Heterogeneous Processors," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 2, pp. 306–317, 2020.
[16] L. L. Caimi, V. Fochi, E. Wächter, D. Munhoz, and F. G. Moraes, "Secure Admission and Execution of Applications in Many-core Systems," in *SBCCI*, 2017, pp. 65–71.
[17] S. Charles, Y. Lyu, and P. Mishra, "Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4510–4523, 2020.
[18] L. L. Caimi and F. G. Moraes, "Security in Many-Core SoCs Leveraged by Opaque Secure Zones," in *ISVLSI*, 2019, pp. 471–476.
[19] C. Reinbrecht, A. Aljuffri, S. Hamdioui, M. Taouil, B. Forlin, and J. Sepúlveda, "Guard-NoC: A Protection Against Side-Channel Attacks for MPSoCs," in *ISVLSI*, 2020, pp. 536–541.
[20] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, "Memphis: a framework for heterogeneous many-core socs generation and validation," *Design Automation for Embedded Systems*, vol. 23, no. 3, pp. 103–122, 2019.
[21] E. Wachter, L. L. Caimi, V. Fochi, D. Munhoz, and F. G. Moraes, "BrNoC: A broadcast NoC for control messages in many-core systems," *Elsevier Microelectronics Journal*, vol. 68, pp. 69–77, 2017.

APPENDIX D – LIGHTWEIGHT AUTHENTICATION FOR SECURE IO COMMUNICATION IN NOC-BASED MANY-CORES

Paper submited to ISCAS'23:

Lightweight Authentication for Secure IO Communication in NoC-based Many-cores
FACCENDA, RAFAEL FOLLMANN; COMARU, Gustavo; CAIMI, LUCIANO L.; MORAES, Fernando Gehm
In: ISCAS, 2023

Lightweight Authentication for Secure IO Communication in NoC-based Many-cores

Rafael Follmann Faccenda*, Gustavo Comarú*, Luciano Lores Caimi†, Fernando Gehm Moraes*
*School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
{rafael.faccenda, gustavo.comaru}@edu.pucrs.br, fernando.moraes@pucrs.br
†UFFS, Federal University of Fronteira Sul, Chapecó, Brazil – lcaimi@uffs.edu.br

***Abstract*—NoC-based many-cores, with hundreds of IPs, are the current standard in the high-performance electronic industry. The attack surface on these systems increases at the same pace the complexity increases. Delegating security to software mechanisms does not guarantee system integrity, as it leaves the hardware exposed. Thus, adding hardware mechanisms in the many-core design is a requirement to execute applications safely. Proposals available in the literature include firewalls, spatial isolation, crypto cores, and PUFs, neglecting that applications communicate with peripherals, such as hardware accelerators and shared memories. This work presents a method to protect the communication between processing elements and peripherals by using a lightweight authentication process associated with hardware mechanisms for key generation and renewal. Our protocol protects the communication of applications with peripherals and simultaneously detects attacks such as DoS, spoofing, and eavesdropping. Attack campaigns show the method's effectiveness in blocking such attacks without impairing the application's performance.**

***Index Terms*—Security, NoC-based Many-cores, Secure Zones, Authentication, Peripherals.**

I. Introduction

Many-core systems-on-chip (MCSoCs) complexity makes security a design requirement as relevant as conventional metrics such as power, performance, and area. MCSoCs contain processing elements (PEs), peripherals attached to the system, and an NoC connecting them. These systems have different flows traversing the NoC: PE-PE and PE-peripheral flows. Malicious hardware or software can hinder system security due to resource-sharing, such as multitasking (CPU sharing) or shared NoC links between flows from different applications.

Proposals [1]–[3] optimize the communication performance with peripherals, or systems with real-time constraints, without considering security issues. On the other hand, the concern of [4, 5] is to protect access to shared memory. Grammatikakis et al. [4] propose an NoC firewall to protect a shared memory accessed through the NoC, avoiding sensitive data corruption or access from an unauthorized element. The firewall isolates the NoC, only allowing an authorized process of the MCSoC to access the memory. Reinbrecht et al. [5] also focus on the security of MCSoCs with shared memories. The Authors propose the Gossip-NoC, which includes a traffic monitor. When an anomalous behavior is detected, the monitor sends an alert message to a system manager, which changes the routing algorithm from XY to YX, avoiding malicious traffic. General security methods to protect access to peripherals other than shared memories are a gap in the literature. Actual many-cores have a rich set of accelerators besides shared memories, requiring the availability of security mechanisms to protect the communication.

The literature presents proposals for protecting applications running in many-cores, such as firewalls [6, 7], encryption [6, 8], routing obfuscation [9, 10], secure zones (SZ) [11]–[14], among others. However, applications communicate with peripherals, where efficient methods to secure this communication are a gap to fulfill. The *objective* of this work is to present a mechanism to protect the communication with peripherals based on a lightweight authentication process associated with hardware mechanisms for key generation and renewal. Application protection uses SZ mechanisms due to its effectiveness against attacks from malicious flows and tasks.

Our *original contribution* is a lightweight authentication protocol that protects the communication of applications with peripherals and simultaneously detects attacks such as DoS, spoofing, and eavesdropping. Attack campaigns show the capability of the method to block such attacks with minimal effect on the application's performance (smaller than $< 0.3\%$ on the application execution time).

II. Secure Architecture Model

Figure 1 presents the MCSoC architecture. The two main system components are:

- **PE**: 32-bit RISC processor, a NI (Network Interface) with DMA capabilities, local scratchpad memory, and two NoC routers;
- **Peripherals:** an SNI (Secure NI) makes the interface between the NoC and IO devices.

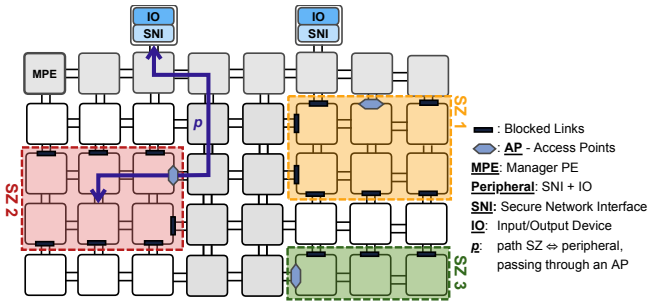


Fig. 1. MCSoC partitioned on secure and gray areas (SA and GA). Three App_{sec} mapped on SZ1 to SZ3. GA is reserved for applications without security constraints.

The MCSoC uses two NoCs: *data* and *control* NoCs. The *data* NoC transmits messages from tasks running on PEs and

data from/to IO devices (as shared memories and hardware accelerators). The data NoC has duplicated 16-bit links, uses wormhole packet switching, XY and source routing support. The *control* NoC transmits control messages using broadcast.

Secure Zones (SZ) [15, 16] is a protection mechanism adopted in MCSoC, which uses spatial isolation to protect applications. We adopt the Opaque Secure Zone (OSZ) method, proposed in [13], to protect applications with security constraints – App_{sec}. OSZ is a runtime defense mechanism that finds a region with available PEs to map an App_{sec}. The secure zone activation occurs by blocking ***all*** links at the boundaries of the secure zone.

This work extends the OSZ method by proposing a secure communication method between App_{sec} and IO devices. The two main actors involved in the communication are: Access Point (AP) and SNI. The AP is a hardware module that manages the traffic at the SZ boundary. Each OSZ has only one AP to reduce the surface attack, and its location may change at runtime when detecting suspicious behavior. The SNI controls access to the IO device by authenticating flows.

The system has secure and gray areas (SA and GA), as shown in Figure 1. SA receives App_{secs}, and GA applications without security requirements. The definition of the shape and location of these areas occurs at the system startup and cannot change at runtime. The System Manager (MPE) maps App_{secs} on secure areas. Internal flows in the SZ use XY routing. Flows to/from peripherals use XY routing to/from the AP, and the exposed path p is routed using source routing. The adoption of source routing in the exposed path obfuscates the source and target addresses [17, 18], being an important security mechanism of the proposed approach.

III. Threat Model and Security Mechanisms

At the same time that the AP enables the communication of App_{secs} with IO devices, it introduces vulnerabilities.

Packets entering SZ through the AP can cause attacks such as DoS (denial-of-service) [19], spoofing [20], and data corruption. The effects of these attacks include performance degradation up to the complete application hang.

Malicious packets to peripherals may execute DoS, spoofing, eavesdropping, and data corruption. Besides modifying the application data, with unpredictable effects, the intruder may steal sensitive information stored in the IO device.

The exposed path peripheral-SZ is prone to Hardware Trojans (HTs) and side-channel attacks (SCAs). The HT may access the packet content, corrupt the original data, execute an eavesdropping attack, misroute, or block packets [21]–[23].

Finally, the IO device may be malicious and execute DoS attacks or transmit the application data to an intruder.

Hashing and CRC mechanisms [24] may address data integrity, while lightweight encryption [25] provides confidentiality. SCAs are out of the scope of this work. Our *goal* is to prevent DoS attacks, spoofing, and eavesdropping, not allowing malicious packets to enter SZs or SNIs, nor malicious IO devices performing unauthorized data injection. To mitigate these attacks, we adopt the following **security mechanisms**:

1) master-slave communication, where all accesses to peripheral starts by the App_{sec} running in the SZ;
2) differentiated communication APIs for PE-PE and PE-peripheral communication;
3) transaction counters on the AP, with key renewal when reaching a pre-defined number of transactions or when suspicious behavior is detected;
4) obfuscation of packet source and destination addresses through the use of source routing;
5) lightweight authentication mechanism, described in the next Section.

The assumed trusty components are the MPE and the control NoC. The control NoC is considered trusty because tasks do not have access to it. Only the operating system (OS) of the PEs may access the control NoC.

IV. Authentication Protocol

This Section presents the main contribution of this work: the lightweight authentication protocol, divided in four phases: *Initialization*, *Application Deploy*, *Communication*, and *Key Renewal*.

A. Initialization

The *initialization* phase occurs at system startup. The MPE generates unique keys, named ***k0***, for each PE and SNI in the system and sends them to their respective PEs and SNIs. Since this action occurs when there is no other application or traffic in the system, these values can be transmitted without encryption, exempting the use of complex key distribution mechanisms such as Diffie-Hellmann [26], which would result in software and hardware overheads. Applications and IO devices do not have access to ***k0***, guaranteeing the confidentiality and integrity of these keys.

B. Application Deploy

The MCSoC has a peripheral named "Application Injector" – App_{inj}, responsible for deploying new applications into the system. Only the MPE receives requests from the App_{inj} (`New_App` messages), which executes the mapping heuristic, and manages the SZs. Figure 2 presents the diagram of the Application Deploy phase, fired when the App_{inj} requests the execution of an App_{sec}.

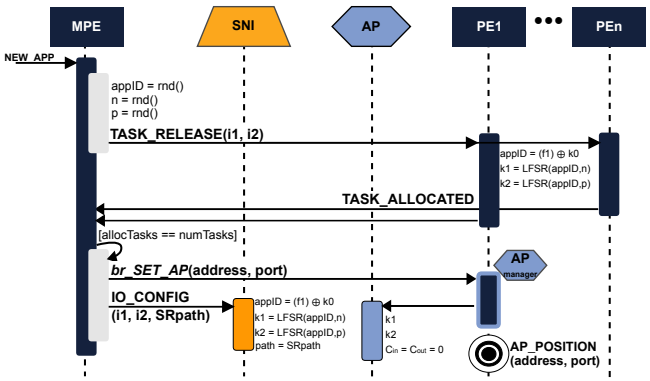


Fig. 2. Sequence diagram of the Application Deploy phase.

The MPE maps the App_{sec} and transmits the mapping result to the App_{inj}, which transmits the application object

code (protected by a Message Authentication Code mechanism [13]) to the selected PEs in the secure area. In parallel, the MPE randomly generates the tuple $\{appID, n, p\}$, where $appID$ is a unique application identifier and $\{n, p\}$ integer values. The MPE transmits `Task_Release` messages to all App_{sec} PEs with two initialization flits, $i1$ and $i2$ (Equation 1), with the tuple obfuscated by ***k0***.

$$\mathbf{i1} = appID \oplus k0_{PE_x} \qquad \mathbf{i2} = (n \;\&\; p) \oplus k0_{PE_x} \tag{1}$$

PEs restore $\{appID, n, p\}$ upon receiving the initialization flits: $appID = \mathbf{i1} \oplus k0_{PE_x}$; $n = MSB(\mathbf{i2} \oplus k0_{PE_x})$; $p = LSB(\mathbf{i2} \oplus k0_{PE_x})$. The $appID$ is the seed for a Linear-feedback shift register (LFSR). The $\{n, p\}$ values correspond to the number of shifts in the LFSR to generate the authentication keys $\{k1, k2\}$. The reason to use an LFSR for key generation comes from its simple hardware implementation and linearity. At the end of this step, all PEs have the same $\{k1, k2\}$ keys, without transmitting them through the NoC. The PEs notify the MPE through a `Task_Allocated` message the correct object code reception and keys generation.

The MPE executes four actions after receiving all `Task_Allocated` messages: (i) elects a PE as "AP manager", transmitting to it the AP coordinate (`br_Set_AP` via control NoC); (ii) configures the SNIs with $\{appID, n, p\}$ and the path from the SNI to the AP (`IO_config`); (iii) sends through the control NoC a message to block the SZ links (not included in the figure); (iv) sends through the control NoC a message to start App_{sec} (not included in the figure).

The "AP manager" configures the AP, transmits $\{k1, k2\}$ to it, and resets the transaction counters $\{Cin, Cout\}$. The "AP manager" also broadcasts the AP address for all PEs executing App_{sec} (`AP_position`).

At the end of the *Application Deploy* phase, all PEs and peripherals of App_{sec} have the authentication keys $\{k1, k2\}$, all links of the SZ boundary block the traffic, except at the AP, as presented in Figure 1.

C. Communication

This phase corresponds to authenticated communication between App_{sec} tasks and peripherals. Due to the master-slave communication method, tasks are responsible for starting the communication. Tasks may execute two services: `IO_delivery`, corresponding to sending data to a peripheral; `IO_request`, corresponding to reading data from a peripheral. Both services must include the tuple $\{appID, k1, k2\}$ encoded in two flits (Equation 2).

$$\mathbf{f1} = k1_{PE} \oplus k2_{PE} \qquad \mathbf{f2} = appID \oplus k2_{PE} \tag{2}$$

The SNI authenticates the received packet by retrieving the $appID$ with the $k1$ value stored at the SNI, as presented in Figure 3(a) (Equation 3).

$$(\mathbf{f1} \oplus k1_{SNI}) \oplus \mathbf{f2} == appID_{SNI} \tag{3}$$

The SNI discards the packet if the authentication fails, avoiding DoS and spoofing attacks.

The answer packet, SNI→SZ, has two authentication locations, at the AP and the target PE. The AP extracts $k2$ from **f1** (Figure 3(b)). If its equal to $k2_{AP}$, the flit enters the SZ. Otherwise, the AP discards the packet. This lightweight verification avoids attacks such as spoofing and DoS. The operating system (OS) extracts the AppID ($\mathbf{f2}_{SNI} \oplus k2_{PE}$) when the packet arrives at the PE. Then it is valid if the retrieved AppID matches the stored AppID.

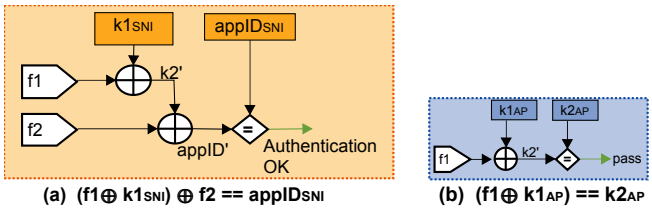


Fig. 3. Lightweight authentication modules.

Using flits {f1,f2} with the same values for a long time is an attack opportunity for a malicious entity, e.g., an eavesdropping attack executed by an HT. Note that stealing {f1,f2} is not a sufficient condition for making an attack. Due to source routing, the attacker does not have access to the SNI and AP addresses, and there is control related to the number of packets received at the AP (transaction counters). To increase the security of the method, the authentication mechanism renews periodically {k1,k2}, even if there is no threat detection.

D. Key Renewal

The transaction counters $\{Cin, Cout\}$ complement the authentication process. Condition $Cin < Cout$ must always be satisfied to accept a packet, due to the master-slave communication protocol. Two events start the key renewal process: (i) $Cout$ reaches a threshold value defined at design time (64 in our current implementation); (ii) malicious packet detection: a packet without an IO flag, Cin indicating an unexpected packet, or authentication fails. The AP notifies the MPE upon receiving a malicious packet.

The key renewal process starts with the AP notifying the "AP manager" (APM) to generate new keys. For synchronization reasons, the APM notifies all PEs in the SZ to complete any pending IO transaction, freezing the following IO communications. When all PEs in the SZ notify the APM, it generates two new random numbers $\{n, p\}$, transmitted to all the PEs in the SZ to generate new keys $\{k1, k2\}$ using the LFRS (the seed is the previous k2 key). The APM also transmits the new $\{k1, k2\}$ keys to the AP and resets $\{Cin, Cout\}$. This first part of the key renewal takes place inside the SZ, with no risk related to their security.

However, the key renewal process on peripherals cannot use the data NoC due to the probability of attacks. For this reason, the APM sends through the control NoC the tuple $\{AppID, n, p\}$ to the SNI, which locates the stored k2 in its table, deriving new keys using its LFSR.

The last step in the key renewal step is the transmission to the PEs in the SZ to unfreeze the IO transactions.

V. Results

Experiments use a 4x4 MCSoC running a DTW application (6 tasks) in an SZ (3x2), communicating with two peripherals.

The system is modeled at the RTL level, being the routers in VHDL and the remaining modules in SystemC.

A. Software and Hardware Overheads

Table I evaluates the *Application Deploy* and *Key Renewal* protocol phases. The row *Application Deploy* corresponds to the time (in clock cycles, *cc*) the application takes to start its execution with the SZ closed. Using the protocol detailed in Figure 2, the average overhead was 3.19%. The *Key Renewal* process took 5,921 *cc*s. This corresponds to a *best-case scenario* since no PEs have pending IO transactions, speeding up the renewal process.

TABLE I
EXECUTION TIME OF PROTOCOL PHASES.

Phase	Time (*clock cycles*)		
	SZ	w/ Auth. Protocol	%
Application Deploy	30,118	31,080	3.19%
Key Renewal	-	5,921	

The hardware overhead in the PE corresponds to the AP and the control NoC. The logic synthesis (Cadence Genus, 28 nm technology library) resulted in: data router: 6,880 gates, 12,789 μm^2; AP: 228 gates, 433μm^2; control router: 3,196 gates, 4,652 μm^2. The area overhead, compared to the data Router, of 4 APs and control router is 13.4% and 36.4%, respectively.

B. Security Evaluation

Table II presents the attack scenarios, considering the threat model (Section III). The attack campaign targets the AP and the SNIs (first column). The second column presents the attack scenarios: unprotected system, and the system using the security methods under attacks with increasing complexity. The third column details the effects and countermeasures considering the security mechanisms proposed in this work.

TABLE II
SIMULATED SCENARIOS TO EVALUATE THE AUTHENTICATION METHOD.

Attack target	Scenario	Effect & Countermeasure
AP	Unprotected (DoS & Spoofing)	Latency increases, deadline misses, App_{sec} hang
	DoS	AP discards packet (incorrect *f1* or *f2*)
	Spoofing - Unexpected IO packet	AP discards packet ($Cin = Cout$) **Countermeasure**: key renewal
	Spoofing - Expected IO packet - Incorrect PE address	Packet enters the SZ PE discards the malicious packet **Countermeasure**: key renewal
	Spoofing - Expected IO packet - Correct PE - PE expecting the packet	**Successful attack** Detection: correct answer packet from the SNI starts key renewal
SNI	Unprotected	Intruder can read and write data to/from the peripheral
	DoS	SNI discards packet
	Spoofing	**Successful attack** Detection: answer packet to App_{sec} starts key renewal

The DoS attack assumes that the intruder knows the AP and SNI locations, being able to flood the network with packets targeting these components. The spoofing attack assumes that a malicious entity (e.g., HT in the exposed path) executed an eavesdropping attack, discovering flits {f1,f2} (Eq. 2).

A successful attack in the SZ is unlikely to occur, given the number of conditions to meet. To attack the SNI it is necessary to have access to flits {f1,f2}. However, in both cases, the attack is detected, and countermeasures are executed.

Figure 4 presents the iteration latency for the unprotected and secure systems under DoS (red background) and spoofing (purple background) attacks. The x-axis corresponds to the application iteration, and the y-axis the iteration latency.

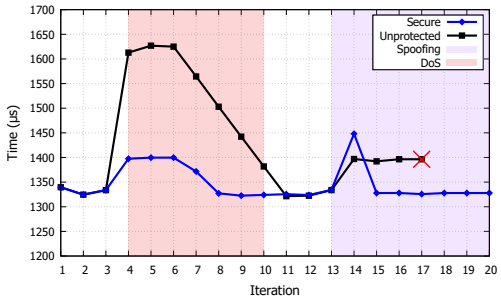


Fig. 4. Application latency for the unprotected and secure systems.

The DoS attack floods a random PE inside the SZ with invalid packets. In the unprotected system, even though these packets are discarded for having invalid content, they constantly interrupt the PE affecting the App_{sec} execution time. This attack increases the iteration time of the unprotected system up to 22%. In the protected system, the DoS does not directly affect the App_{sec} because the AP discards the packets. However, a slight increase in the iteration time (up to 4.9%) occurs due to the increased traffic attempting to enter the AP.

The spoofing attack sends packets with correct {f1,f2} values, varying the target PE address. In the unprotected system, the latency increases after the 13^{th} iteration, representing a packet being discarded. In the 17^{th} iteration, a malicious packet hits a PE waiting for an IO packet, causing the application to hang.

In the secure system, the attack in the 13^{th} iteration arrives at the AP with correct {f1,f2} values, but at the wrong moment (Cin == Cout). Thus, the AP discards the packet, and the attack is detected, firing the key renewal process, explaining the latency increase. The next malicious packets are discarded at the AP, not impacting the latency.

VI. CONCLUSION

This work adopted the OSZ method to protect App_{sec}, and five mechanisms to protect communication with IO devices: (i) master-slave communication; (ii) differentiated communication APIs; (iii) transaction counters; (iv) address obfuscation; (v) lightweight authentication protocol. Results showed the effectiveness of the method. Besides the reduced software and hardware overheads, to method protected applications against DoS and Spoofing attack campaigns.

The proposal provides countermeasures to mitigate possible attacks, with key renewal being the most important. Future work includes the periodic modification of the AP position and the path between the AP and the SNI.

REFERENCES

[1] C. Lee, J. Cho, J. Kim, and H. Jin, "Transparent many-core partitioning for high-performance big data I/O," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 18, 2021.

[2] S. Vaas, P. Ulbrich, C. Eichler, P. Wägemann, M. Reichenbach, and D. Fey, "Taming Non-Deterministic Low-Level I/O: Predictable Multi-Core Real-Time Systems by SoC Co-Design," in *ISORC*, 2021, pp. 43–52.

[3] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. C. Audsley, and Z. Dong, "I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance," in *DAC*, 2021, pp. 1159–1164.

[4] M. D. Grammatikakis, P. Petrakis, A. Papagrigoriou, G. Kornaros, and M. Coppola, "High-level Security Services based on a Hardware NoC Firewall Module," in *WISES*, 2015, pp. 73–78.

[5] C. Reinbrecht, A. A. Susin, L. Bossuet, G. Sigl, and J. Sepúlveda, "Timing attack on NoC-based systems: Prime+Probe attack and NoC-based protection," *Microprocess. Microsystems*, vol. 52, pp. 556–565, 2017.

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

[7] S. P. Azad, G. Jervan, M. Tempelmeier, and J. Sepúlveda, "CAESAR-MPSoC: Dynamic and Efficient MPSoC Security Zones," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 477–482.

[8] S. Charles and P. Mishra, "Securing Network-on-Chip Using Incremental Cryptography," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 168–175.

[9] L. S. Indrusiak, J. Harbin, C. Reinbrecht, and J. Sepúlveda, "Side-channel protected MPSoC through secure real-time networks-on-chip," *Microprocessors and Microsystems*, vol. 68, pp. 34–46, 2019.

[10] S. Charles, M. Logan, and P. Mishra, "Lightweight Anonymous Routing in NoC based SoCs," in *Design, Automation Test in Europe Conference (DATE)*. IEEE, 2020, pp. 334–337.

[11] J. Sepúlveda, D. Flórez, V. Immler, G. Gogniat, and G. Sigl, "Hierarchical Group-key Management for NoC-Based MPSoCs Protection," *Integrated Circuits and Systems (JICS)*, vol. 11, no. 1, pp. 38 – 48, 2016.

[12] M. M. Real, P. Wehner, V. Lapotre, D. Göhringer, and G. Gogniat, "Application Deployment Strategies for Spatial Isolation on Many-Core Accelerators," *ACM Transaction on Embedded Computing Systems*, vol. 17, no. 2, pp. 55:1–55:31, 2018.

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

[14] M. Ruaro, L. L. Caimi, and F. G. Moraes, "A Systemic and Secure SDN Framework for NoC-Based Many-Cores," *IEEE Access*, vol. 8, pp. 105 997–106 008, 2020.

[15] M. M. Real, V. Migliore, V. Lapotre, and G. Gogniat, "ALMOS Many-Core Operating System Extension with New Secure-Enable Mechanisms for Dynamic Creation of Secure Zones," in *PDP*, 2016, pp. 820–824.

[16] J. Sepulveda, R. Fernandes, C. Marcon, D. Florez, and G. Sigl, "A security-aware routing implementation for dynamic data protection in zone-based MPSoC," in *SBCCI*, 2017, pp. 59–64.

[17] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-NoCs: Mitigating the threat of a compromised NoC," in *DAC*, 2014, pp. 1–6.

[18] A. Sarihi, A. Patooghy, M. Hasanzadeh, M. Abdelrehim, and A.-H. A. Badawy, "Securing Network-on-Chips via Novel Anonymous Routing," in *NOCS*, 2021, pp. 29–34.

[19] S. Charles, Y. Lyu, and P. Mishra, "Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4510–4523, 2020.

[20] S. S. Rout, A. Singh, S. B. Patil, M. Sinha, and S. Deb, "Security Threats in Channel Access Mechanism of Wireless NoC and Efficient Countermeasures," in *ISCAS*, 2020, pp. 1–5.

[21] L. Daoud and N. Rafla, "Detection and Prevention Protocol for Black Hole Attack in Network-on-Chip," in *NOCS*, 2019, pp. 22:1–22:2.

[22] R. Manju, A. Das, J. Jose, and P. Mishra, "SECTAR: Secure NoC using Trojan Aware Routing," in *NOCS*, 2020, pp. 1–8.

[23] M. M. Ahmed, A. Dhavlle, N. Mansoor, S. M. P. Dinakarrao, K. Basu, and A. Ganguly, "What Can a Remote Access Hardware Trojan do to a Network-on-Chip?" in *ISCAS*, 2021, pp. 1–5.

[24] T. Boraten and A. K. Kodi, "Packet Security with Path Sensitization for NoCs," in *DATE*, 2016, pp. 1136–1139.

[25] A. Sarihi, A. Patooghy, M. Hasanzadeh, M. Abdelrehim, and A.-H. A. Badawy, "Securing on-Chip Communications: An On-The-Fly Encryption Architecture for SoCs," in *CSCI*, 2021, pp. 741–746.

[26] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.