





Integration of Monitoring Mechanisms in Secure Network Interfaces for Peripherals to Protect IO Communication in NoC-based Many-cores

Gustavo Comarú¹, Rafael Follmann Faccenda¹, Luciano Lores Caimi², Fernando Gehm Moraes¹

¹School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil

²UFFS, Federal University of Fronteira Sul, Chapecó, Brazil

{gustavo.comaru, rafael.faccenda}@edu.pucrs.br, lcaimi@uffs.edu.br, fernando.moraes@pucrs.br

Abstract— Many-core systems have gained popularity due to their high performance and parallelism. However, they also present significant security challenges. One critical issue that remains insufficiently addressed in current research is protecting Input/Output (IO) communication within these systems. This paper proposes a Secure Network Interface with Peripherals (SNIP) design, incorporating security mechanisms to safeguard communication between internal components in many-core systems. The proposed SNIP makes an original contribution by integrating security features into its network interface design, thereby addressing a research gap in the communication between many-core systems and peripherals. Furthermore, the paper proposes a monitoring mechanism within the SNIP to detect suspicious behavior during IO communication. The SNIP design focuses on the trade-offs between security, performance, and area overheads. Results demonstrate that the SNIP effectively protects IO communication in many-core systems from potential attacks while imposing minimal overhead, and it enables system-level countermeasures through the monitoring information provided. The proposed SNIP design offers a promising solution to address the security challenges of many-core systems, particularly in safeguarding IO communication.

Index Terms— Security, Monitoring, NoC-based Many-cores, Network Interface, Peripherals, IO devices.

I. INTRODUCTION

The demand for embedded devices with reduced power consumption and specific functionalities has increased the utilization of many-core systems. A many-core system consists of Processing Elements (PEs) interconnected by a communication structure, such as Networks-on-Chip (NoCs). An NoC employs routers and links to transmit data and control messages between PEs.

The Network Interface (NI) in an NoC-based system acts as the communication link between on-chip components, such as processors, memory modules, and dedicated hardware modules, and the NoC [1]. Its primary function is to convert incoming and outgoing data into a format compatible with the NoC infrastructure. To achieve this, the NI provides flow control, buffering, routing, and protocol conversion, facilitating seamless communication within NoC-based systems and enhancing their efficiency and scalability.

In addition to serving as a bridge between on-chip components and the communication structure, the NI can also connect input and output modules, such as accelerators or shared memories, referred to herein as *IO devices*. These IO devices often utilize standard protocols for data exchange. The

NI integrates these modules into the NoC infrastructure by adapting to their requirements and ensuring proper data encoding and decoding (e.g., an IO device using the AXI protocol [2]). This process may involve protocol converters, specialized buffer management, and tailored Quality of Service (QoS) policies to maintain high-performance data exchange with IO devices. Furthermore, NIs provide configurable interfaces that facilitate the integration of new hardware modules, promoting adaptability and extensibility in NoC-based systems.

Allowing the connection of external modules to the system via NoC introduces several security concerns, as third-party intellectual property (3PIP) cores may originate from untrustworthy sources. Therefore, incorporating security as a fundamental requirement in the design of NIs is crucial to protect against threats and ensure the integrity of communication between on-chip components and IO devices [3, 4].

Literature on NIs in NoC-based many-core systems primarily focuses on enhancing the NIs that connect already trusted PEs to the NoC [3, 5]. However, a gap exists in the literature regarding the protection and monitoring of open NoC ports that connect IO devices, which may not be fully trusted. Proposals to secure communication with IO devices are scarce, with most efforts concentrating on shared memory protection [6]. Conversely, several works present many-core systems with IO devices without addressing security concerns [7–9].

The goal of this work is twofold. First, to present the design of an NI, named SNIP, for communication with IO devices, integrating security mechanisms to safeguard communication with internal components in many-core systems. Second, propose a monitoring infrastructure that generates warnings to a system manager, enabling system-level countermeasures upon detecting an attack.

This paper extends the SBCCI’23 paper [10], “Secure Network Interface for Protecting IO Communication in Many-cores”, by incorporating the monitoring infrastructure into the SNIP. The fulfillment of the second goal is original to this paper. Results also include new attack campaigns to validate the security mechanism.

The paper is structured as follows: related work in many-core IO security (Section II.), the reference platform and security mechanisms (Section III.), threat model and countermeasures (Section IV.), SNIP design (Section V.), attack detection (Section VI.), results (Section VII.), and conclusion (Section VIII.).

II. RELATED WORK

Aghaei et al. [1] refer to the NI as the Network Adapter (NA), noting that it directly affects system power, latency, throughput, and area. The Authors review various NA architectures and evaluate the parameters influencing their design. They also consider the security-aware design of communication architectures, such as NoCs, increasingly necessary, as their complexity may introduce new vulnerabilities. At the same time, the NoC itself can contribute to system security by enabling the monitoring of system behavior and the detection of specific attacks. In their view, the NA is ideally positioned to analyze incoming traffic and discard malicious requests.

Kapoor et al. [11] divide the PEs of a many-core system into two categories: (i) secure cores, which store and process secret information; (ii) non-secure cores, which may carry viruses or Hardware Trojans (HT). The Authors propose an authentication method to protect the communication between secure and non-secure cores. The secure core issues a session key for each transaction. On the sending side, the NI encrypts the packet using the session key and generates a message authentication code (MAC). On the receiving side, the NI decrypts the message and asserts its authenticity through the MAC. Despite the added security mechanisms, encryption, and MAC increase communication latency by an average of 23% and add a significant area overhead.

Baron et al. [12] analyze the vulnerabilities of the SoCIN NoC-based system and propose mechanisms to protect the NoC from attacks by malicious cores. The Authors consider four attacks: (i) masquerade; (ii) DoS induced by an invalid target; (iii) DoS issued by flooding the NoC; (iv) DoS induced by a packet without a trailer. The Authors designed a Security Wrapper (SEW), a hardware module inserted between the NI and the NoC router. The SEW module filters malicious packets sent by an attacking core without impacting the latency, with an area overhead of 4.1%.

Ahmed et al. [13] discuss a Remote Access Hardware Trojan (RAHT) attack in which an NoC router is infected by an HT that periodically sends traffic information to a malicious IO device. This device analyses the leaked information using a machine-learning algorithm to infer information such as architectural details or the applications running on the system. The Authors propose a security mechanism that uses controlled random routing to confuse the external attacker. Random routing decisions reduce the correlation between the collected traffic information and the system architecture and applications. The Authors presented other work on this topic in [14], focusing on more complex traffic analysis attacks that the remote threat could execute. Even though the proposals offer a routing solution to the RAHT attack, it does not prevent unauthorized communication between HT and IO devices.

Restuccia et al. [15] propose the AKER framework, which aims to prevent unauthorized access to shared resources in a many-core. Hardware wrappers enforce the access control policy (which defines the allowed communication pairs) between the cores and the interconnect; illegal transactions are filtered before entering the network. These wrappers are configured and managed by a centralized root of trust. The

Authors state that though this work targets bus-based systems, its methodology can be extended to NoC architectures with some modifications. The Authors state that AKER has a limited impact on performance while using minimal resources.

Sankar et al. [16] propose the Sec-NoC, which utilizes authenticated encryption in the NI to enhance confidentiality and integrity of communication, and a packet retransmission mechanism based on ACK-NACK. The NI checks the packet integrity on the receiver side, and in case of failure, it sends a NACK packet back. Since an attacker can target an ACK or NACK packet, a timer can also trigger retransmission. This work aggregates the encryption and decryption mechanisms and retransmission buffers inside the NI, at a cost of 10% area overhead to a 16-core MPSoC and 1.3% increase in packet latency.

The reviewed works highlight the security threats posed by IO devices. Existing solutions primarily focus on protecting the NoC [12] or introduce mechanisms that significantly impact performance and system area [11, 16]. There is a lack of mechanisms that effectively safeguard many-core systems from malicious IO devices. This work addresses these gaps by proposing SNIP, incorporating a lightweight authentication mechanism to protect communication between PEs and IO devices against the attacks described in the threat model section.

III. BASELINE PLATFORM WITH SECURITY MECHANISMS

This work adopts as the reference architecture an NoC-based many-core platform, with IO devices connected to the borders of the NoC. Figure 1 presents a 4x4 system with four peripherals connected at the north side of the NoC. The system size and the position of the peripherals are defined at design time. In the context of this work, we highlight five elements:

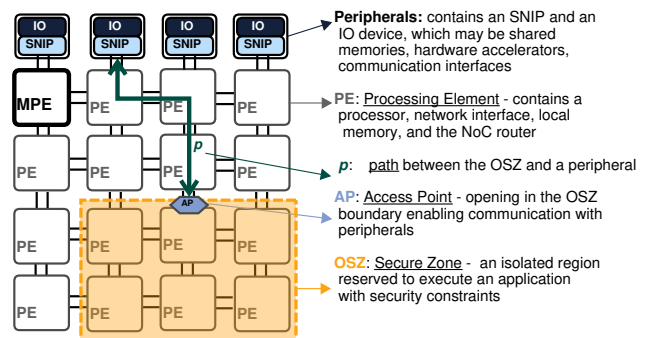


Fig. 1: NoC-based many-core and terminology adopted in this work (adapted from [10]).

- Opaque Secure Zone (OSZ [17]) – a region that executes an application with security requirements, blocking the traffic from other applications. The spatial isolation prevents attacks from other flows or tasks.
- Access Point (AP [18]) – opening in an OSZ border that controls the entry and exit of packets.
- Path p – the path between the AP and the SNIP. The path is defined by source routing (SR).

- Secure Network Interface with Peripherals (SNIP) - *our proposal* to secure the communication between the IO device with the application with security requirements.
- System Manager PE (MPE) - processing element reserved to execute management operations, such as application allocation and mapping, PE and SNIP configuration.

A. IO Communication

Packet exchange between IO devices and applications is based on the host-device model, where the PE acts as the host and the SNIPs function as the devices. Thus, communication is always initiated by the host (PE), and the device (SNIP) must send a response packet to confirm the operation.

To receive data from an IO device, the task sends an IO_READ packet, and waits for an IO_DELIVERY packet with the requested data. To send data to the IO device, the task transmits an IO_WRITE packet and waits for an IO_ACK to confirm the operation's success.

Regarding security, packets carry a key checked both in the PE and the SNIP. The PEs can manage the keys after receiving the generation parameters from the MPE during the application deployment. However, the SNIP depends on MPE commands to handle the keys correctly. The first command is the IO_CONFIG, which enables the SNIP to interact with a specific application, informing the corresponding key generation parameters and application identifier. In addition, the MPE sends IO_RENEW commands periodically to renew the keys, enhancing key strength.

IV. THREAT MODEL AND COUNTERMEASURES

The section initially presents the Threat Model (Section IV.A.), describing the potential attacks from malicious applications and peripherals. It highlights the risks of unauthorized access, data corruption, and system overloads, establishing the need for effective countermeasures. The Countermeasures subsection (Section IV.B.) then presents the strategies the SNIP employs to mitigate these risks.

A. Threat Model

This section discusses the harmful behaviors in IO communication that this work aims to protect the system against. We assume that the hardware implementing the NoC-based many-core system is secure. Threats may arise from application tasks (software) or IO devices connected to the SNIP. Thus, attacks can be initiated by a malicious application (*MalApp*) or a malicious IO device (*MalPeriph*).

If a *MalApp* is loaded into the system and has access to create custom packets, it can perform the following attacks:

- **DoS (flooding).** Occurs when the *MalApp* sends a high volume of invalid packets to the IO device, generating excessive traffic and harming access to it.
- **DoS (configuration).** Occurs when the *MalApp* forges configuration packets (IO_CONFIG), filling the SNIP with invalid application requests, rendering it incapable of accepting new legitimate applications.
- **Spoofing.** The *MalApp* accesses an IO device without authorization, stealing or corrupting sensitive data.

A *MalPeriph* may perform the following attacks:

- **DoS.** A *MalPeriph* may attempt to send more data than requested, generating oversized packets that can overload the communication infrastructure and system components.
- **Information leakage.** A *MalPeriph* sends messages to a malicious target, leaking sensitive information.
- **Spoofing.** A *MalPeriph* attempts to access a sensitive application by sending forged packets.

B. Countermeasures

To protect the IO communication against the threats previously described, the SNIP may execute three main countermeasures: (i) authentication; (ii) packet discard; (iii) warning generation.

The SNIP employs the **IO Authentication** protocol, based on [19], to enforce authentication and authorization principles. The protocol begins with the MPE sending a command to the SNIP (IO_CONFIG), specifying the applications authorized to interact with the IO device. Additionally, the SNIP must verify packet authenticity, send packets with the correct authentication fields, and perform key derivation. An important feature of this protocol is that the SNIP only communicates with authorized applications using a fixed source-routing path set by the MPE. As a result, the IO device is prevented from sending messages to unauthorized applications or using forged paths.

The **Packet Discard** mechanism is a countermeasure that focuses on quickly rejecting and eliminating packets that fail authentication, removing them from the NoC, and reinforcing the principle of availability.

Both aforementioned countermeasures are applied immediately upon detecting malicious actions to neutralize threats without delay. However, due to this automatic response, the MPE remains unaware of suspicious behavior on the SNIPs. Therefore, the third key countermeasure of the SNIP is **Warning Generation**, which notifies the MPE whenever a security anomaly is detected at the SNIP.

The SNIP issues four types of warnings: (i) *Failed authentication*, triggered when an incoming packet fails authentication; (ii) *Write on a full table*, indicating that the SNIP received an IO_CONFIG request, but the table has no available space; (iii) *Row overwrite*, reporting that a slot in the SNIP table for an authenticated application was replaced; and (iv) *Abnormal peripheral*, signaling that the peripheral is not adhering to the correct communication protocol.

V. SNIP ARCHITECTURE

The SNIP has seven main modules, as illustrated in Figure 2: (i-ii) Packet Handler and Packet Builder, enable simultaneous communication to and from the NoC.; (iii) Application Table (ApT), stores sensitive data to enable communication with the applications; (iv-v) FIFO buffers hold data sent to or received from the IO device until consumption; (vi) Key Generator produces and updates authentication keys; (vii) Warning Manager detects suspicious behavior and sends packets to the MPE. The next subsections detail these components, except for the buffers.

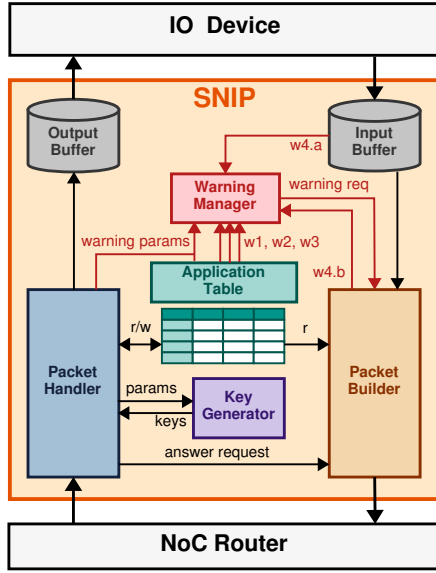


Fig. 2 : SNIP architecture and interfaces (adapted from [10]).

This work addresses the security of communication between PEs and peripherals. Figure 2 would have an additional module between the SNIP and the IO device, responsible for adapting the chosen protocol (e.g., AXI stream) to the protocol used by the SNIP, which in the current implementation is data and address-oriented. To support more complex protocols, such as AXI stream, a Network Adapter module would need to be introduced between the SNIP and the peripheral. While we do not explore IO device protocols in detail, attack scenarios consider attempts to insert malicious data into the input buffer.

A. Packet Handler

The SNIP acts as a slave to the system since it waits for incoming packets to define its action. The Packet Handler is responsible for receiving packets from the NoC and carrying out the appropriate response. It executes all the decision-making, acting as a manager to the other components.

Upon receiving a packet, the Packet Handler analyzes its service code, which refers to the function of the packet. Table I displays the services the SNIP supports. The SNIP discards any received packet whose *appID* that is not in *ApT*.

The packet handling process is divided into phases, controlled by an FSM, depicted in Figure 3. Every packet begins with two header flits (target and payload size) followed by the message header and, if present, payload flits.

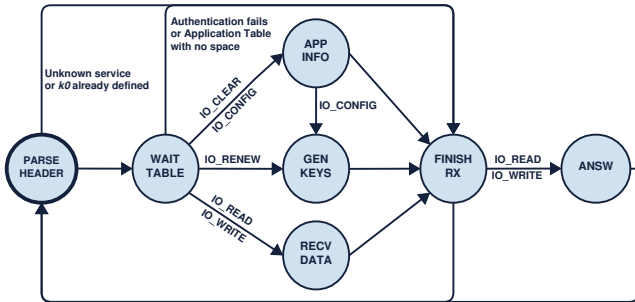


Fig. 3 : Abstract FSM controlling the Packet Handler [10].

Table I. Services supported by the SNIP [10].

Service code	Packet Source	Function
IO_INIT	Manager PE	Packet received at system startup with the initialization key – k_0
IO_CONFIG	Manager PE	Configure a line of the <i>ApT</i> with $\{appID, path, k_1, k_2, status\}$
IO_RENEW	Manager PE	Renew the <i>appID</i> keys $\{k_1, k_2\}$ receiving parameters $\{n, p\}$
UNBLOCK_WARNINGS	Manager PE	Enable the SNIP to send warning packets
IO_CLEAR	Manager PE	Clear and deallocate the <i>ApT</i> row indexed by <i>appID</i>
IO_WRITE	Application	Write data into an IO device Application waits an IO_ACK from SNIP
IO_READ	Application	Request data from an IO device Application waits an IO_DELIVER packet

Each phase in Figure 3 execute the following actions:

- **PARSE_HEADER**: reads and stores relevant flits, such as *servicecode*, *appID*, and other parameters, into registers. All services from Table I, except IO_INIT, trigger the transition to the **WAIT_TABLE** phase. Unknown services lead to the **FINISH_RX** phase, where the packet is discarded.
- **WAIT_TABLE**: the FSM searches for a free row in *ApT* for IO_CONFIG or checks for a matching *appID* using authentication. Packets are discarded if authentication fails or no space is available in *ApT*.
- **APP_INFO**: handles IO_CONFIG and IO_CLEAR services. For IO_CONFIG, it writes $\{appID, path\}$ into the row, while IO_CLEAR releases the row.
- **GEN_KEYS**: waits for the Key Generator to produce $\{k_1, k_2\}$, which are written into the *ApT* row.
- **RECV_DATA**: processes IO_WRITE and IO_READ services by transmitting the packet's payload to the Output Buffer. For IO_WRITE, the payload contains data for the IO device; for IO_READ, the payload contains the address and the number of flits to be received.
- **FINISH_RX**: remaining flits are discarded, including packets with unknown services or unauthenticated applications.
- **ANSW**: completes IO_WRITE and IO_READ transactions. For IO_WRITE, an IO_ACK is sent to the application. For IO_READ, the Packet Handler requests the Packet Builder to send a message back to the application with data from the Input Buffer, which was populated during the **RECV_DATA** phase.

B. Application Table – *ApT*

The SNIP uses the *ApT* to allow authorized applications to access the IO device connected to the SNIP. Each line of the *ApT* has the following fields: ***appID***: application identifier; ***path***: path between the SNIP and the application AP; ***k1*** and ***k2***: authentication keys, used to certify the authenticity of packets; ***status***: it may assume *free*, *pending*, and *used* values.

Note that the *ApT* authenticates applications and not tasks. This “application granularity” reduces the *ApT* size and thus silicon area compared to a table with “task granularity”.

The ApT has two interfaces, enabling the SNIP to send and receive packets simultaneously. The primary interface (read-write) is connected to the Packet Handler, and the secondary interface (read-only) is connected to the Packet Builder.

As in the Packet Handler module, an FSM controls the access to the ApT. The main phases of the FSM include:

- **FETCH_NEW**: started by an *IO_CONFIG* packet. This phase searches for a row with *free* status; if no row is *free*, the FSM searches for a row with a *pending* status. If there is a match (*free* or *pending*), the FSM returns the row to the Packet Handler. When a row is configured, its status is set to *pending*, and changes to *used* only after the first transaction with the application. This procedure avoids attacks that try to fill the ApT by flooding the SNIP with *IO_CONFIG* packets.
- **FETCH**: started by an *IO_RENEW* or *IO_CLEAR* packet. This phase retrieves *appID* using the authentication method [19]. The FSM searches for a row with an *appID* that matches the retrieved one, returning it to the Packet Handler.
- **FETCH_AP**: corresponds to the packet authentication process. The FSM searches for a row in the ApT that successfully authenticates the packet.
- **READY** and **FAILED**: notify the Packet Handler the row to be written or signalize an invalid row, respectively.

C. Key Generator

The Key Generator is responsible for creating and updating the keys used in the Authentication Protocol. It generates two keys, $\{k1, k2\}$, using a Linear-Feedback Shift Register (LFSR), which acts as a pseudo-random key generator. The key size is a design time parameter, in our case we use 16-bit keys. While LFSRs are not the most robust method for generating pseudo-random numbers, they offer a distributed and area-efficient solution for generating authentication keys. For the *IO_CONFIG* service, the LFSR uses *appID* as the seed, producing *k1* after *n* rounds and *k2* after an additional *p* rounds. For the *IO_RENEW* service, new keys are generated using *k2* as the seed, following the same procedure. The generated $\{k1, k2\}$ keys are stored in the ApT row indexed by *appID*, with *n* and *p* being randomly generated for each *IO_CONFIG* and *IO_RENEW* service.

D. Packet Builder

The Packet Builder assembles and sends packets to the applications. These packets can be either *IO_DELIVERY* messages with the data requested from the IO device, or *IO_ACK* to acknowledge data from the application. Once the Packet Handler receives a valid packet from an application, it uses the *Answer Request* (Figure 2) interface to notify the Packet Builder to send an answer. The parameters specifying the packet to be sent are *appID*, *messageType*, and *requestSize*.

Upon receiving a request, the Packet Builder registers the parameters, raises a busy signal, and generates the packet. The information required to build the packet header, such as the authentication keys and the source-routing path, is retrieved from the ApT through the secondary interface. If the

outgoing packet is an *IO_DELIVERY*, data sent by the IO device is retrieved from the Input Buffer and sent in the packet payload.

Since only one request can be handled at a time, if another request needs to be issued while the Packet Builder is busy, the Packet Handler stays blocked until the completion of the current request.

E. Warning Manager

The Warning Manager module is responsible for detecting suspicious behavior and generating warning packets to send to the MPE.

As depicted on Figure 2, SNIP components alert the Warning Manager through specific warning signals (*w1*, *w2*, *w3*, *w4.a*, *w4.b*) when irregularities occur. The Warning Manager collects relevant data via the *Warning Parameters* interface and issues a *Warning Request* to the Packet Builder, which assembles and sends the warning packet into the NoC.

The four types of warnings the SNIP can detect are:

- W1 Failed authentication.** Issued when an incoming packet fails authentication, indicating that a malicious agent is attempting unauthorized access using incorrect keys.
- W2 Write on a full table.** Triggered when an *IO_CONFIG* request is made, but the table is full. This suggests a malicious attempt to configure the SNIP, as the MPE should not exceed the table's capacity.
- W3 Overwritten row.** Occurs when a pending table row is overwritten by a new *IO_CONFIG* request. This may result in a legitimate application losing access to the peripheral, indicating a potential malicious configuration attempt.
- W4 Abnormal peripheral.** Detected when a peripheral does not follow the correct communication protocol. This includes cases where (i) the peripheral attempts to write data while access control is disabled, or (ii) it writes more flits than requested by an *IO_READ* operation.

The ApT is responsible for detecting the **W1**, **W2**, and **W3** warnings. The Input Buffer identifies the first trigger for the **W4** warning, related to an access control breach, while the Packet Builder detects the second trigger, which involves the reception of excessive flits.

Figure 4 illustrates the structure of the Warning Manager, which consists of four interfaces:

1. **Anomaly Detection**: includes the warning signals (*w1*, *w2*, *w3*, and *w4*) that other components use to signal a security anomaly.
2. **Warning Parameters**: collects diagnostic information from other components, which is later included in the warning packet.
3. **Warning Request**: communicates with the Packet Builder to request the sending of a warning packet; it comprises the *warning_req* signal, a *warning_ack*, and the necessary parameters for packet assembly.

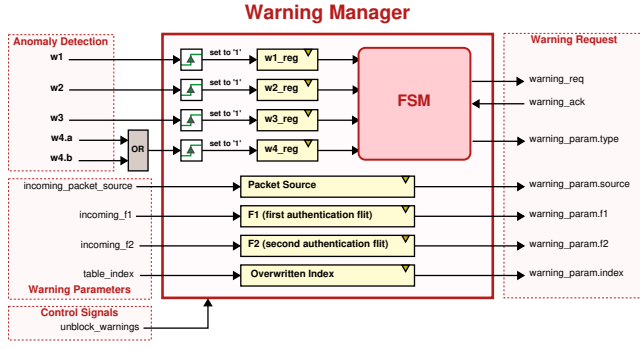


Fig. 4 Warning manager internal blocks and external interfaces.

4. **Control Signals:** contains the *unblock_warnings* signal, which is triggered by the Packet Handler to clear any blocked warnings.

Anomaly Detection signals are used to identify security threats, remaining active as long as the threat persists. However, to prevent multiple warnings from being sent for the same anomaly, warnings are triggered only when a rising edge is detected on the signal. Each Anomaly Detection signal is, therefore, filtered by an edge detector.

For each signal, there is a corresponding register (*w1_reg*, *w2_reg*, *w3_reg*, *w4_reg*). When a rising edge is detected, the appropriate register is activated, and the Warning Parameters (*Packet Source*, $\{F1, F2\}$, *Overwritten Index* from Figure 4) are stored in registers. The Warning Manager FSM then requests the Packet Builder to send a warning packet.

The FSM transitions to one of four states, each handling a different warning type based on the active register. In the corresponding state, the Warning Manager raises the *warning_req* signal and sends the relevant parameters. Once the Packet Builder acknowledges with the *warning_ack* signal, the FSM returns to idle and resets the warning register.

After a warning is processed, the register is deactivated, and the Warning Manager waits for the next anomaly. Since there is only one set of parameter registers shared by all warning types, only one warning can be sent at a time, minimizing area overhead.

To prevent excessive warning packets from flooding the NoC, each warning type has a limit on how many times it can be sent, which is defined at design time. This is particularly important for *w4*, as a malicious peripheral could flood the network. Once a warning type reaches its limit, it is blocked, and further detections are ignored until the MPE sends a *UNBLOCK_WARNINGS* packet to the SNIP.

F. Warning structure

When the SNIP detects a suspicious behavior, the Packet Builder sends a warning message to the MPE. These warning packets follow the format shown in Figure 5:

The packet header (yellow) contains routing information for reaching the MPE, which can use either XY routing or a source-routing path. These fields are configured by the MPE during the SNIP initialization via the *IO_INIT* packet, making it possible to reconfigure the path to the MPE whenever the SNIP is reset.

The *Packet Size* and *Service Code* fields (in blue) are used by the PE to correctly receive and handle the packet.

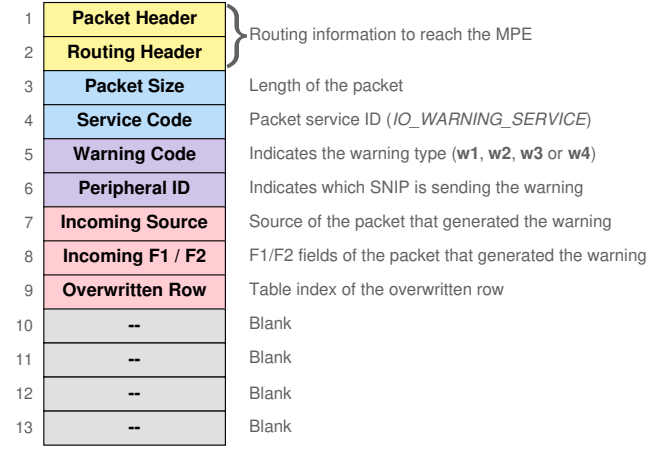


Fig. 5 : Structure of the warning packet sent by the SNIP.

The purple fields are used to identify the warning: **Warning Code** defines the type of the warning, while **Peripheral ID** specifies which SNIP sent it.

The red fields provide information to help the MPE assess the attack and apply appropriate countermeasures. These include the **Source** of the packet that triggered the warning and its authentication flits $\{F1, F2\}$. If a row in the ApT was overwritten, the index of the **Overwritten Row** is included.

VI. ATTACK DETECTION

Based on the warnings received from the SNIP, the MPE can identify ongoing attacks targeting the SNIP and their respective warning profiles:

Unrequested peripheral data: if a peripheral sends unrequested messages or violates the communication protocol, the SNIP issues a *w4* warning. This may indicate malicious intent (e.g., data leakage) or a peripheral not following correctly the communication protocol. The MPE can flag the peripheral as suspicious and monitor for further evidence before deploying countermeasures.

Peripheral DoS: the peripheral tries to flood the NoC by writing excessive data to the SNIP. In this case, the SNIP sends several *w4* messages to the MPE, which will result in blocking the *w4* warning. The MPE can request to unblock the warnings through a *UNBLOCK_WARNINGS* packet. If this continues to happen, then it is clear the peripheral is attempting to flood the NoC (DoS attack).

Read/write spoofing: if a malicious application attempts to access a peripheral by forging the $\{F1, F2\}$ authentication flits, the SNIP will fail to authenticate the packet and trigger a *w1* warning. The malicious application will receive no response from the SNIP.

Configuration spoofing: a malicious entity, other than the MPE, sends a fake *IO_CONFIG* packet to the SNIP, attempting to configure a row in the ApT table to gain unauthorized access. This attack is not immediately detected, leaving the row in a *pending* state. Since the attacker lacks knowledge of the keys generated by the Key Generation process, any attempt to access the SNIP will trigger a *w1* warning. If the MPE configures additional applications in the ApT table, the malicious row will eventually be overwritten, resulting in a *w3* warning.

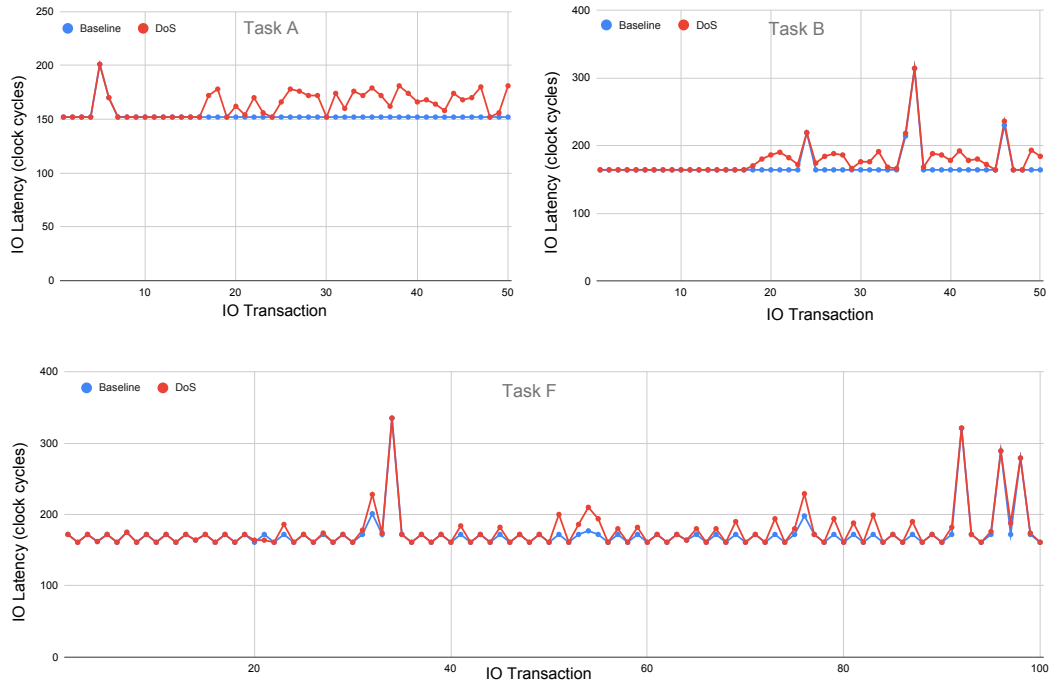


Fig. 6 : IO latency values for three tasks from *synthetic* app: Task A, Task B, and Task F.

Configuration flooding: since there is no mechanism to verify the authenticity of *IO_CONFIG* packets, an attacker could attempt a DoS attack by filling the *ApT* rows with invalid applications. While the *pending* status mechanism prevents the table from being fully filled, a legitimate application might lose access to the SNIP. This attack results in a stream of *w3* warnings to the MPE as malicious entries overwrite valid ones, along with a *w1* warning for each failed read/write operation attempted by the overwritten legitimate application.

By monitoring these warnings, the MPE can detect the occurrence of different attacks in the SNIPs, thus being able to deploy effective countermeasures and resume proper system behavior.

VII. RESULTS

This section presents the SNIP behavior against attacks and its resulting area cost. The system is modeled at the RTL level (VHDL and SystemC), allowing clock cycle accuracy. The simulated scenarios have applications running on an OSZ and malicious applications and malicious peripherals generating attacks.

A. Execution Time

The first evaluation assesses the impact of a DoS flooding attack on the total execution time of six applications, commonly used in the many-core research community [20, 21]:

- *Dijkstra* - 6 tasks (3x2 OSZ), 1 IO: computes the shortest path between two points in a graph represented by an adjacency matrix;
- *DTW* (Dynamic Time Warping) – 6 tasks (3x2 OSZ), 1 IO: given a set of reference templates, the application determines which one best matches an unknown pattern;
- *Fixed* - 14 tasks (4x4 OSZ), 3 IOs: image authentication based on spectral analysis;

- *MPEG* - 5 tasks (3x2 OSZ), 1 IO: decodes patterns for video information;
- *Synthetic* - 6 tasks (3x2 OSZ), 2 IOs: an application graph with fork and join tasks.
- *VOPD* (Video Object Plane Decoder) – 12 tasks (4x3 OSZ), 3 IOs: simulates the interaction between hardware modules of a Video Object Plane (VOP) encoder.

The OSZ shape is defined at runtime, being the smallest rectangle that encloses the application tasks, considering one task for PE and respecting the system dimensions. The SNIP configuration in every scenario is the same: 16-bit key size and 4-slot *ApT*.

These applications are evaluated under two scenarios: **baseline** (no attack) and **DoS** (injection rate: 50% of one NoC link bandwidth). Table II presents the execution time in milliseconds.

Table II.: Execution time of applications while the IO device is under a DoS attack (ms@100MHz).

Application	System Size	Baseline	DoS 50% bandwidth	Overhead
Dijkstra	4x4	7.5432	7.5436	0.01%
DTW	4x4	8.3467	8.3530	0.08%
Fixed	5x5	5.5585	5.5590	0.01%
MPEG	4x4	14.4426	14.4430	0.003%
Synthetic	4x4	7.0030	7.2066	2.91%
VOPD	5x5	4.0659	4.0676	0.04%

The overhead values presented in Table II are below 0.1%, except for the *synthetic* application. For DoS attacks, the SNIP's fast packet discard mechanism is crucial in reducing contention on the NoC, helping mitigate significant impacts on IO communication latency. Additionally, the round-robin arbitration algorithm, used by the NoC router, ensures that malicious traffic does not dominate a given link.

The *synthetic* application showed higher overhead due to its IO-intensive profile, where any disruption in IO latency directly affects execution time. Figure 6 highlights three tasks from the *synthetic* application (A, B, and F) that exchange packets with the IO device during the DoS attack. Task A and Task B perform 50 IO transactions each, while Task F performs 100.

In all cases, the latency values remain identical for both the Baseline and DoS scenarios until a specific iteration (16 for Task A and Task B, and 20 for Task F). After this point, the attack begins, and IO communication is impacted by the malicious traffic competing for the same link, leading to increased latency in the DoS scenario.

B. Security Analysis

This section presents two attack scenarios and demonstrates how the Secure Manager in the MPE can use the information provided by the Warning Manager to implement countermeasures and protect the system.

Scenario 1: IO device under DoS from *MalApp*

Figure 7(a) depicts the first scenario, where a secure application (*AppSec*) communicates with two IO devices (*IO1* and *IO2*). Additionally, a malicious application (*MalApp*) is running on PE X3Y2. At a given moment, Figure 7(b), *MalApp* begins injecting malicious packets into the NoC at a specified *Injection Rate* (I_{rate}). Although these packets contain valid fields, the authentication keys are invalid, prompting the SNIP to send *w1* warnings to the MPE, reporting the invalid access attempts (indicated by the orange arrow in Figure 7(b)). Upon receiving these warnings, the MPE checks the source of the malicious packet (from field 7 of the warning packet) and applies a countermeasure. In this case, the MPE terminates the application running on PE X3Y2 (Figure 7(c)).

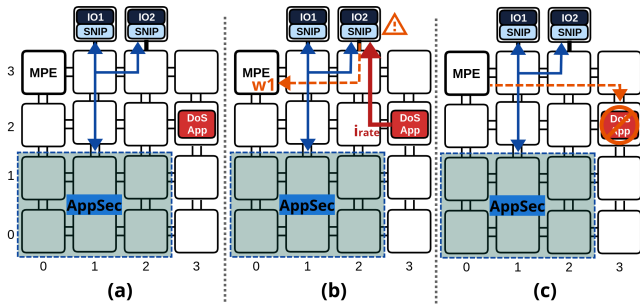


Fig. 7 : Scenario 1: *MalApp* causing flooding DoS to *IO2*.

Since the MPE receives failed authentication warnings immediately when the DoS campaign begins, only a few malicious packets reach the SNIP before *MalApp* is terminated. As a result, the execution time of the secure application remains unaffected.

The focus of this scenario is not the countermeasure itself but rather the importance of the warning generation mechanism and how it enables the MPE to make system-level decisions. This scenario serves as a starting point, and the threats can be expanded, along with the complexity of the countermeasures, as needed.

Scenario 2: *MalPeriph* attempting to inject data

Figure 8(a) shows the same *AppSec* running in the system and communicating with two IO devices, *IO1* and *IO2*. In this scenario, however, *IO1* is a malicious peripheral (*MalPeriph*). Additionally, the scenario includes a third peripheral, *IO3*, which is not utilized at the start of the execution.

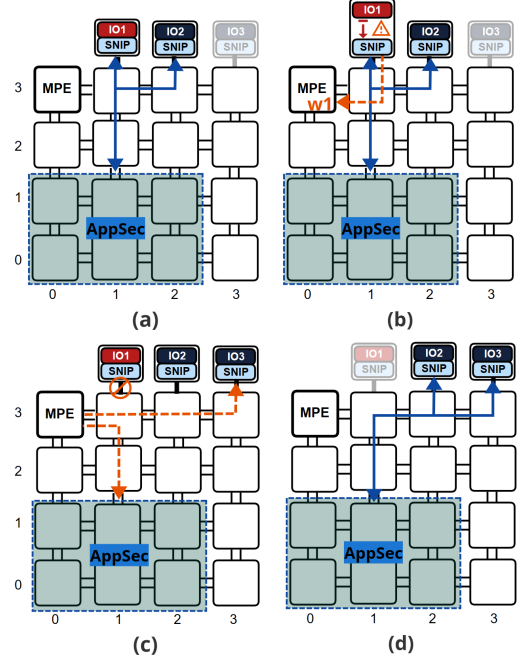


Fig. 8: Attack scenario 2: *MalPeriph* attempting to inject unrequested data into the NoC.

After the 20th iteration, the *MalPeriph* begins injecting extra data into the system. As depicted in Figure 8(b), the SNIP detects this behavior and sends a *w1* warning to the MPE. In response, the MPE initiates a process to switch the IO device. Following the orange arrows in Figure 8(c), the MPE configures *IO3* to be used by the application and informs *AppSec* that future IO transactions with device *IO1* should be rerouted to *IO3*. Finally, Figure 8(d) illustrates the IO communication after implementing the countermeasure.

The execution time for each iteration is shown in Figure 9, comparing the *synthetic* application's normal execution (Baseline) with the scenario involving the attack and countermeasure (Attack). Up to iteration 20, the iteration times show minimal variation, following the same pattern in both cases. However, from iteration 21 to 26, the Attack scenario takes longer to complete due to the overhead of switching to the new IO device. The average increase in execution time for iterations 21 to 26 is 7.21%. This overhead includes the time required to calculate new paths to *IO3* and the additional key renewal necessary to update the keys after the attack.

C. Area Evaluation

Each processing element (PE) has two routers (data and control routers), a processor, local memory, and a network interface.

The *data NoC* is a wormhole packet-switched network-on-chip (NoC) that does not use virtual channels with two key

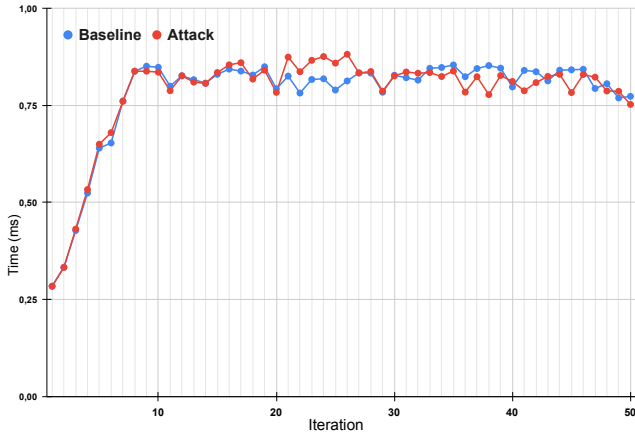


Fig. 9: Synthetic application iteration execution time for Baseline and Attack scenarios (ms@100MHz).

features: (i) two physical channels acting as separate NoCs, allowing fully adaptive routing; (ii) area overhead minimization by using a 16-bit flit size. This NoC supports both the default XY routing algorithm and source routing (SR), where the packet header specifies required turns to avoid faulty paths or bypass an OSZ.

The *control NoC* [22], named *BrNoC*, is a lightweight NoC in which each packet consists of a single flit. In the default broadcast mode, packets are sent to all PEs, enabling path establishment from a source PE to a target PE even in the presence of faults or HTs in the data NoC. Unicast transmission is also supported, allowing path creation between source and target PEs using a backtracking approach. For security reasons, access to the control NoC is restricted to the operating system (OS), thereby preventing malicious applications from using it.

We synthesized the SNIP (ApT with 4 rows, and input/output buffers for 16 depth 16-bit flits size), the Data Router (16-bit flits size and 8-flit depth input buffers), and the Control Router (CAM size with 8 rows) for a 28 nm FD-SOI process from ST Microelectronics. Table III presents the synthesis results.

Table III.: Synthesis results for the SNIP, Data Router, and Control Router - 28nm FDSOI - CADENCE GENUS 21.12-s068.1.

Synthesis results	SNIP	Data Router	Control Router
Cell Area (μm^2)	12,449	15,019	4,657
Net Area (μm^2)	3,775	5,000	2,090
Total Area (μm^2)	16,224	20,019	6,747
Cell Instance Count	5,852	7,776	3,203
Timing slack@500 MHz (ps)	263	136	29
Total estimated power (mW)	5.844	7.463	2.697

Routers represent no more than 20% of the PE area (in [23], the NoC represents 7.7% of the PE area). Table III demonstrates that the SNIP has a low area consumption, representing 82.3% of the data router area. The number of rows of the ApT directly impacts the area overhead. According to the expected workload, it may be necessary to increase the ApT to make each peripheral available to more applications simultaneously. Also, a small additional area must be considered in a complete implementation, corresponding to converting the protocol used in the input and output buffers to the selected protocol.

Three of the related works presented in this paper showed the area overhead of their implementation of security: in the NI (Kapoor et al. [11]), in the router (Baron et al. [12]) and entire system size (Sankar et al. [16]). All of these approaches present solutions with scalability associated to the number of PEs in the platform. In our solution, the overhead scales only with the number of IO devices in the system. Therefore, once defined the number of SNIPs, the number of PEs does not increase this proposal area overhead.

VIII. CONCLUSION

The proposed SNIP design presents a promising approach to addressing security challenges in many-core systems, specifically protecting IO communication. The SNIP integrates security mechanisms that safeguard communication between internal components in many-core systems and bridges a research gap regarding the communication between many-core systems and peripherals. These security mechanisms encompass: (i) host-device communication API, (ii) application authentication, (iii) lightweight packet authentication, (iv) frequent key renewal. The security evaluation involved applications with different profiles, demonstrating the SNIP effectiveness in thwarting attacks such as Denial of Service (DoS) and unauthorized access from *MalPeriph*, applying local (packet discarding, authentication) and system-level countermeasures (change peripheral and terminate application) based on reported suspicious behavior. The costs are minimal IO iteration latency overhead (7.21%) and area impact (equivalent to 82.3% of a data router).

During the evaluation, none of the applications presented an increase on the packet latency during an IO transaction caused by path congestion on the AP. Since the utilized benchmark only has applications that exchange packets with up to 3 IO devices, one future research is the inclusion of more IO intensive applications as benchmark to investigate the possibility of path congestion due to the AP.

Other future research directions include: (i) developing a range of distinct modules to establish the interface between the SNIP and IO devices; (ii) creating a high-level protocol that analyzes the warnings information to detect more complex attacks, such as Distributed DoS (DDoS), and develop improved countermeasures for prevention or mitigation; (iii) dynamically modify paths to the Access Point (AP) through randomization; (iv) evaluate the robustness of the proposed method when multiple applications simultaneously access the same SNIP.

ACKNOWLEDGEMENTS

This work was financed in part by CAPES (Finance Code 001), CNPq (grants 309605/2020-2 and 407829/2022-9), and FAPERGS (grants 21/2551-0002047-4 and 23/2551-0002200-1).

REFERENCES

- [1] B. Aghaei, M. Reshadi, M. Masdari, S. Sajadi, M. Hosseinzadeh, and A. Darwesh, "Network adapter architectures in network on chip: comprehensive literature review," *Cluster Computing*, pp. 321–346, 2020, <https://doi.org/10.1007/s10586-019-02924-2>.

- [2] ARM, “AMBA® AXITM and ACETM Protocol Specification,” February 2013, <https://developer.arm.com/documentation/ih0022/e/>.
- [3] S. Charles and P. Mishra, “Securing Network-on-Chip Using Incremental Cryptography,” in *ISVLSI*, 2020, pp. 168–175, <https://doi.org/10.1109/ISVLSI49217.2020.00039>.
- [4] —, “A Survey of Network-on-Chip Security Attacks and Countermeasures,” *ACM Comput. Surv.*, vol. 54, no. 5, pp. 101:1–101:36, 2022, <https://doi.org/10.1145/3450964>.
- [5] J. Sepúlveda, A. Zankl, D. Flórez, and G. Sigl, “Towards protected mp soc communication for information protection against a malicious noc,” in *ICCS*, 2017, pp. 1103–1112, <https://doi.org/10.1016/j.procs.2017.05.139>.
- [6] C. Reinbrecht, A. Susin, L. Bossuet, and J. Sepúlveda, “Gossip NoC - Avoiding Timing Side-Channel Attacks through Traffic Management,” in *ISVLSI*, 2016, pp. 601–606, <https://doi.org/10.1109/ISVLSI.2016.25>.
- [7] 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, <https://doi.org/10.1007/s10586-021-03288-2>.
- [8] 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, <https://doi.org/10.1109/DAC18074.2021.9586156>.
- [9] 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*, 2020, pp. 1–6, <https://doi.org/10.1109/DAC18072.2020.9218686>.
- [10] G. Comarú, R. F. Faccenda, L. L. Caimi, and F. G. Moraes, “Secure Network Interface for Protecting IO Communication in Many-cores,” in *SBCCI*. IEEE, 2023, pp. 1–6, <https://doi.org/10.1109/SBCCI60457.2023.10261655>.
- [11] H. K. Kapoor, G. B. Rao, S. Arshi, and G. Trivedi, “A Security Framework for NoC Using Authenticated Encryption and Session Keys,” *Circuits Syst Signal Process*, vol. 32, no. 6, pp. 2605–2622, 2013, <https://doi.org/10.1007/s00034-013-9568-5>.
- [12] S. Baron, M. S. Wangham, and C. A. Zeferino, “Security mechanisms to improve the availability of a Network-on-Chip,” in *ICECS*, 2013, pp. 609–612, <https://doi.org/10.1109/ICECS.2013.6815488>.
- [13] M. M. Ahmed, A. Dhaville, N. Mansoor, and S. M. P. Dinakarrao, “What Can a Remote Access Hardware Trojan do to a Network-on-Chip,” in *ISCAS*, 2021, pp. 1–5, <https://doi.org/10.1109/ISCAS51556.2021.9401297>.
- [14] A. Dhaville, M. M. Ahmed, N. Mansoor, K. Basu, A. Ganguly, and S. M. P. Dinakarrao, “Defense against on-chip trojans enabling traffic analysis attacks based on machine learning and data augmentation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. 12, pp. 4681–4694, 2023, <https://doi.org/10.1109/TCAD.2023.3278618>.
- [15] F. Restuccia, A. Meza, and R. Kastner, “Aker: A Design and Verification Framework for Safe and Secure SoC Access Control,” in *ICCAD*, 2021, pp. 1–9, <https://doi.org/10.1109/ICCAD51958.2021.9643538>.
- [16] S. Sankar, R. Gupta, J. Jose, and S. Nandi, “Sec-noc: A lightweight secure communication system for on-chip interconnects,” *IEEE Embedded Systems Letters*, vol. 16, no. 2, pp. 214–217, 2024, <https://doi.org/10.1109/LES.2023.3333561>.
- [17] L. L. Caimi and F. Moraes, “Security in Many-Core SoCs Leveraged by Opaque Secure Zones,” in *ISVLSI*, 2019, pp. 471–476, <https://doi.org/10.1109/ISVLSI.2019.00091>.
- [18] R. F. Faccenda, G. Comarú, L. L. Caimi, and F. G. Moraes, “SeMAP – A Method to Secure the Communication in NoC-based Many Cores,” *IEEE Design & Test*, vol. preprint, pp. 1–7, 2023, <https://doi.org/10.1109/MDAT.2023.3277813>.
- [19] R. F. Faccenda, G. Comarú, L. L. Caimi, and F. G. Moraes, “Lightweight Authentication for Secure IO Communication in NoC-based Many-cores,” in *ISCAS*, 2023, pp. 1–5, <https://doi.org/10.1109/ISCAS46773.2023.10181962>.
- [20] W. N. Costa, L. P. Lima, and O. A. de Lima Junior, “Extracting Packet Dependence from NoC Simulation Traces Using Association Rule Mining,” *Analog Integrated Circuits and Signal Processing*, vol. 106, no. 1, pp. 235–247, 2021, <https://doi.org/10.1109/sbcci.2018.8533244>.
- [21] S. Kashi, A. Patooghy, D. Rahmati, and M. Fazeli, “An Energy Efficient Synthesis Flow for Application Specific SoC Design,” *Integration, the VLSI Journal*, vol. 81, pp. 331–341, 2021, <https://doi.org/10.1016/j.vlsi.2021.08.005>.
- [22] E. Wachter, L. L. Caimi, V. Fochi, D. Munhoz, and F. G. Moraes, “BrNoC: A broadcast NoC for control messages in many-core systems,” *Microelectronics Journal*, vol. 68, pp. 69 – 77, 2017, <https://doi.org/10.1016/j.mejo.2017.08.010>.
- [23] A. Rovinski and others Mishra, “Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL,” *IEEE Solid-State Circuits Letters*, vol. 2, no. 12, pp. 289–292, 2011, <https://doi.org/10.1109/LSSC.2019.2953847>.