Secure Network Interface for Protecting IO Communication in Many-cores

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

***Abstract*—Many-core systems have gained popularity due to their high performance and parallelism but pose significant security challenges. One crucial concern that remains insufficiently addressed in current research is protecting Input/Output (IO) communication within many-core systems. To tackle this issue, this paper introduces a design for a Secure Network Interface with Peripherals (SNIP), which incorporates security mechanisms to safeguard communication between internal components in many-core systems. The proposed SNIP makes an original contribution by developing an NI incorporating security features within its design, filling a research gap in the communication of many-core systems with peripherals. The paper evaluates the SNIP, focusing on the trade-offs between security and overheads. Results show that the SNIP protects IO communication in many-core systems from potential attacks while adding minimal overhead. The SNIP proposal design offers a promising solution to address the security challenges in many-core systems, particularly in safeguarding IO communication.**

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

I. Introduction

Many-core systems deliver high performance through parallelism, addressing the current demand for embedded devices with power consumption and communication constraints. A many-core system comprises Processing Elements (PEs) interconnected by Networks-on-Chip (NoCs). An NoC, consisting of routers and links, is responsible for forwarding data and control messages between PEs. Network Interfaces (NIs) connect PEs to the routers within the NoC.

An NI in an NoC-based system is the communication link between on-chip components, such as processors, memory modules, and dedicated hardware modules with the NoC [1]. Its primary function is to manage data exchange by converting incoming and outgoing traffic into a format compatible with the NoC infrastructure. To accomplish this, the NI offers flow control, buffering, routing, and protocol conversion, enabling seamless communication within NoC-based systems and contributing to their overall efficiency and scalability.

Besides acting as the communication bridge between on-chip components, the NI enables communication with input and output modules, such as accelerators or shared memories, herein named *IO devices*. These IO devices often use 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 with AXI protocol [2]). This may require protocol converters, specialized buffer management, and tailored QoS policies to maintain a high-performance data exchange with IO devices. Furthermore, NIs also offer configurable interfaces that facilitate the integration of new hardware modules, fostering adaptability and extensibility in NoC-based systems.

Incorporating security as a fundamental requirement in the design of NIs is essential to protect against threats and ensure the integrity of communication between on-chip components and IO devices [3, 4]. Literature related to many-core that present methods to secure communication with IO devices is scarce, with most focusing on shared memory protection [5]. On the other hand, several works present many-cores with IO devices but without security concerns [6, 7, 8]. Therefore, there is a gap to fulfill: *how to protect the communication of applications with IO devices*?

The *goal* of this work is to design an NI for communication with IO devices, integrating security mechanisms to safeguard communication with internal components in many-core systems. This NI is named “Secure Network Interface with Peripherals” (*SNIP*). The development of an NI incorporating security features within its design is our *original* contribution.

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

II. Related Work

Aghaei et al. [1] refer to the NI as Network Adapter (NA), stating that it directly impacts the system power, latency, throughput, and area. The Authors review different NAs and evaluate the parameters which affect the NA architectures. The Authors consider the security-aware design of communication architectures such as an NoC an increasing necessity, as their complexity may lead to new weaknesses. At the same time, the NoC itself may contribute to the system’s security 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.

Kapoor et al. [9] divide the PEs of a many-core system into two categories: secure cores, which store and process secret information; and 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. [10] 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 (+one clock cycle), with an area overhead of 4.1%.

Ahmed et al. [11] 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. Even though the proposal offers a routing solution to the RAHT attack, it does not prevent unauthorized communication between HT and IO devices.

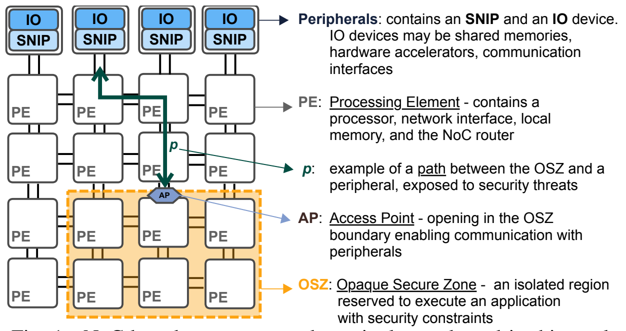
Restuccia et al. [12] propose the AKER framework, which aims to prevent unauthorized access to shared resources in a many-core. Hardware wrappers enforce the access control policy (which defines the allowed communication pairs) between the cores and the interconnect; illegal transactions are filtered 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.

The reviewed works highlight the threats that IO devices can pose to system security. The solutions protect mainly the NoC [10] or add mechanisms that severely impact performance and the system area [9]. There is a gap related to mechanisms that effectively protect the many-core against malicious IO devices. This work aims to fill these gaps by proposing the SNIP, which incorporates a lightweight authentication mechanism and protects the communication between PEs and IO devices against attacks described in the threat model section.

III. Baseline Platform and Threat Model

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 four elements:

- Opaque Secure Zone (OSZ [13]) – 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 [14]) – opening in an OSZ border that controls the entry and exit of packets.


IO
SNIP
PE
p
AP
Peripherals: contains an SNIP and an IO device. IO devices may be shared memories, hardware accelerators, communication interfaces
PE: Processing Element - contains a processor, network interface, local memory, and the NoC router
p: example of a path between the OSZ and a peripheral, exposed to security threats
AP: Access Point - opening in the OSZ boundary enabling communication with peripherals
OSZ: Opaque Secure Zone - an isolated region reserved to execute an application with security constraints
Fig. 1. NoC-based many-core and terminology adopted in this work.

- 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.

A. 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 ($MalApp$) or by a malicious IO device ($MalPerph$).

As a 3PIP (third-party intellectual property module), the IO device may be a source of vulnerabilities. Embedding an IO device into the system enables the following attacks:

- **DoS:** $MalPerph$ 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. Also, a peripheral can be the target of a DoS attack performed by another component.
- **Misrouting:** $MalPerph$ sends its messages to the wrong target, leaking sensitive information or disrupting the execution of applications.
- **Spoofing:** a $MalApp$ accesses an IO device without authorization, stealing or corrupting sensitive data.

A packet traversing the non-isolated region of the network might have its information compromised by an HT infecting a router. The latency monitoring of path p enables the detection of threats in this communication path. Protection of the path p is out of the scope of this work.

IV. SNIP Design Requirements

Considering the baseline platform and the threat model, this section defines the requirements for designing a secure network interface.

Although the SNIP is a security module, it still keeps the functions of a standard NI. The first three requirements capture this aspect:

R1 Enable the communication between IO devices and PEs by defining a communication API, ensuring seamless integration within the network.

R2 Abstract the NoC protocol for the IO devices, thereby increasing the number of compatible devices. IO devices only see the external protocol, as the AXI stream. This abstraction also offers a security advantage by hiding the system's inner workings.

R3 Minimize the silicon area footprint. Since this module is instantiated multiple times within the system, keeping the area overhead as low as possible is essential to avoid excessive resource consumption.

Malicious IO devices can execute attacks in two primary ways: sending unrequested packets and directing packets to incorrect recipients. The following two requirements aim to mitigate these harmful behaviors.

R4 Enforce a master-slave communication model in which only an application can start the exchange of messages. This prevents the IO device from injecting unwanted packets into the network (avoid **DoS** attacks).

R5 The SNIP only sends messages to authorized applications through a fixed source-routing path set by trusted entities. Thus, the IO device cannot communicate with unauthenticated applications (avoid **misrouting** attacks).

The next requirements include a SNIP authentication protocol to protect against **Spoofing** attacks.

R6 Register the applications authorized to interact with an IO device.

R7 Grant access to an IO device only to authorized applications. A malicious application cannot directly access the sensitive information the IO device provides.

R8 Be able to execute an authentication protocol, assert the authenticity of packets, and perform key derivation procedures.

R9 Be able to execute an authentication protocol, assert packet authenticity, and perform key derivation.

Final requirements relate to security mechanisms such as packet discarding and address obfuscation.

R9 Quickly discard packets that fail the authentication, avoiding **DoS** attacks targeting the peripheral.

R10 Build packets using source routing, which obfuscates the addresses of the communicating devices, making the attack more difficult to occur, given that if a packet is intercepted, the attacker does not have access to the source and target addresses of the intercepted packet (avoid **misrouting** attacks).

V. SNIP Architecture

The SNIP has six main modules, as illustrated in Figure 2. Two modules, Packet Handler and Packet Builder, enable simultaneous communication to and from the NoC. The Packet Handler stores the sensitive data for communication with the application in the Application Table (ApT), while the Packet Builder retrieves it when necessary. FIFO buffers hold data sent to or received from the IO device until consumption. The Key Generator produces and updates authentication keys. Next subsections detail these components, except for the buffers.

This work focuses on the security aspects of IO communication and not on device connectivity. Thus, the SNIP-peripheral interface implements a simple data and address oriented protocol. To support more elaborate protocols (e.g., AXI stream), a Network Adapter module would have to be inserted between the SNIP and the peripheral. Although we do not focus on IO device protocols, attack campaigns consider attempts to insert malicious data into the input buffer.

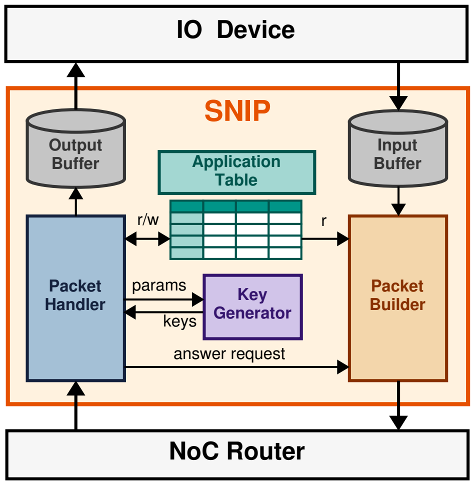


Fig. 2. SNIP architecture and interfaces.

The Packet Handler meets design requirements **R4** (master-slave communication) and **R9** (discard malicious packets). Table ApT meets design requirements **R5** (store the path to applications using SR), **R6** (application registering), **R7** (access to authorized applications), and **R8** (authentication protocol and key derivation). The Packet Builder meets design requirement **R10** (packet transmission using SR to obfuscate the source and target address).

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.

TABLE I
Services supported by the SNIP.

Service code	Packet Source	Function
IO_INIT	Manager PE	Packet received at system startup with the initialization key – $k0$
IO_CONFIG	Manager PE	Configure a line of the ApT with $\{appID, path, k1, k2, status\}$
IO_RENEW	Manager PE	Renew the $appID$ keys $\{k1, k2\}$ receiving parameters $\{n, p\}$
IO_CLEAR	Manager PE	Clear and deallocate the ApT row indexed by $appID$
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

The SNIP waits for an IO_INIT packet from the Manager PE (MPE) at the system startup. The MPE is a dedicated PE responsible for management functions, including security protocols. This IO_INIT packet contains the key used by the MPE to encode sensitive information – $k0$. The SNIP only starts operating after receiving an IO_INIT; any other IO_INIT packet received afterward is discarded.

When the MPE sends a packet, the sensitive header information is encoded by $k0$. Packets sent by applications contain two flits, $\{f1, f2\}$, which encode $\{k1, k2, appID\}$, used for the authentication process. The tuple $\{k1, k2\}$ corresponds to the authentication keys, and $appID$ is a unique random application identifier.

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.

The handling process is divided into distinct phases, controlled by the finite state machine (FSM) depicted in Figure 3. Aiming for ease of comprehension, each phase is implemented as a set of states in the actual FSM.

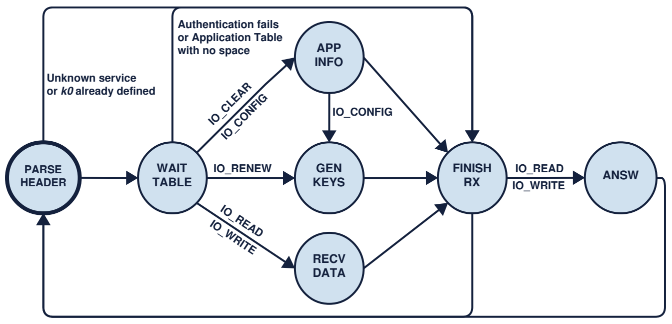


Fig. 3. Abstract FSM controlling the Packet Handler.

Each packet contains two header flits (target and payload size), followed by a set of flits corresponding to the message header and, optionally, the payload flits. The ***PARSE_HEADER*** phase reads the message header, storing relevant flits (e.g., $servicecode$, $appID$, $k0$, $f1$, $f2$, n, p, $path$) into registers. All services of Table I, except IO_INIT, make the FSM switch to the ***WAIT_TABLE*** phase. All other unknown services switch to the ***FINISH_RX*** phase, where the packet is discarded.

The function of the ***WAIT_TABLE*** phase is twofold: find a free row in `ApT` (service IO_CONFIG), or find a row that matches with the $appID$ included into the packet using the authentication detailed in Section V-B. The incoming packet is discarded if the authentication fails or the `ApT` has no space.

The ***APP_INFO*** treats the IO_CONFIG and IO_CLEAR services, writing in the row found during the ***WAIT_TABLE*** phase. The IO_CONFIG service writes $\{appID, path\}$ into the row. The IO_CLEAR service releases this row.

The ***GEN_KEYS*** waits for the generation of the $\{k1, k2\}$ keys by the Key Generator module (Section V-C), subsequently writing them in the `ApT` row.

The ***RECV_DATA*** phase treats IO_WRITE and IO_READ. The packet's payload is transmitted to the Output Buffer for both services. For an IO_WRITE service, the packet's payload has the data to the IO device; and for an IO_READ operation, the packet's payload has, e.g., the address and the number of flits to receive from the external device.

The ***FINISH_RX*** phase discards the remaining flits of the packet. This phase also discards packets with unknown services or unauthenticated applications.

The ***ANSW*** phase concludes the IO_WRITE and IO_READ transactions. For an IO_WRITE transaction, the SNIP sends an IO_ACK to the application. For an IO_READ the Packet Handler requests the Packet Builder to assemble and send a message back to the application containing the data stored in the Input Buffer, which was written by the IO device upon receiving the payload 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.

Figure 4 presents the abstract FSM to access the `ApT`. The ***WAIT*** state switches to another phase when the Packet Handler has finished storing the message header.

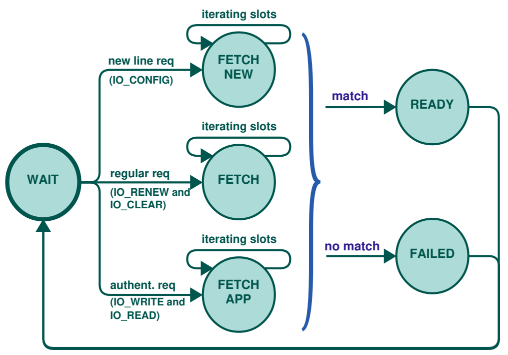


Fig. 4. FSM managing the primary interface access to the `ApT`.

The IO_CONFIG starts the ***FETCH_NEW*** phase. 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.

IO_RENEW and IO_CLEAR start the ***FETCH*** phase. This phase retrieves $appID$ using Equation (1). The FSM search for a row with an $appID$ that matches the retrieved one, returning it to the Packet Handler.

$$appID = appID_{PE} \oplus k0 \tag{1}$$

where: $appID_{PE}$ is the received $appID$, encoded for safe transport in the NoC, and $k0$ the initialization key.

An application transmitting an IO_WRITE or IO_READ packet includes in the message header flits the $f1$ and $f2$, as presented in Equation 2.

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

IO_WRITE or IO_READ initiate the ***FETCH_APP*** phase, corresponding to the packet authentication process. The FSM searches for a row in the ApT that satisfies Equation (3). This equation uses two fields stored in the ApT, $\{k1_{SNIP}, appID_{SNIP}\}$, and flits $\{f1, f2\}$. The row satisfying this equation returns to the Packet Handler.

$$(\boldsymbol{f1} \oplus k1_{SNIP}) \oplus \boldsymbol{f2} == appID_{SNIP} \qquad (3)$$

Note that a forged IO_CONFIG packet will result in the values decoded by $k0$ being different from those in the forged packet. Thus, any other forged packet must forge $\{f1, f2\}$, with the correct $\{k1_{SNIP}, appID_{SNIP}\}$, initially encoded by $k0$. Therefore, the *pending* state is essential in the security process, as it prevents attempts to register new applications in the ApT.

The ***READY*** and ***FAILED*** states notify the Packet Handler the row to be written or signalize an invalid row, respectively.

The Packet Builder uses the read-only interface by providing the *appID*. The ApT uses a combinational circuit to find the matching row, outputting its fields for the Packet Builder.

C. Key Generator

The Key Generator creates and updates the keys used in the Authentication Protocol. This module produces $\{k1, k2\}$ keys, using a Linear-Feedback Shift Register (LFSR) as a pseudo-random key generator. Although an LFSR is not considered the most robust method to generate pseudo-random numbers, it provides a distributed and area-efficient way to generate the authentication keys. For the IO_CONFIG service, the LFSR uses $appID$ as a seed, and $k1$ is obtained after n rounds in the LFSR and $k2$ after p more rounds. For the IO_RENEW service, the same procedure is executed to generate new keys for the application, but this time using $k2$ as seed. The $\{k1, k2\}$ keys are stored in the ApT row indexed by $appID$. Note that n and p are randomly generated for each IO_CONFIG and IO_RENEW services.

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.

VI. 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 system has a 4x4 size, with an application running on an OSZ and a malicious application generating attacks.

A. Security Analysis

This section illustrates the SNIP under DoS and spoofing attacks. Figure 5 illustrates the IO transaction latency between the DTW (Dynamic Time Warping) application and the SNIP (y-axis, freq. 100 MHz) for each transaction (x-axis). Three scenarios are evaluated: (a) *baseline*, no malicious traffic (avg. latency: 8.80 μs); (b) *non-sec*, security mechanisms disabled (avg. latency: 19.18 μs); (c) *secure*, security mechanisms enabled (avg. latency: 8.81 μs), overlapped with the baseline curve. The malicious application, in scenarios (b) and (c), generates 512-flit packets (256 32-bit words) at each 8μs (DoS attack), randomly forging authentication flits (spoofing attack).

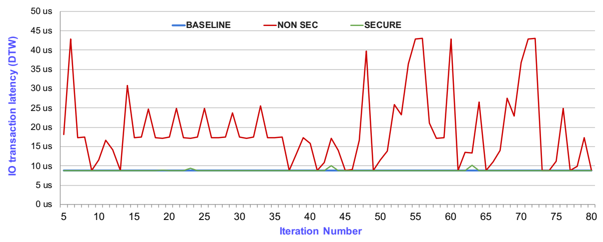


Fig. 5. IO transaction latency (time between IO_WRITE–IO_ACK or IO_READ–IO_DELIVERY) the for the DTW application under DoS and spoofing attack. The graph removed the warm-up period (5 first iterations).

The SNIP handles all packets in the *non-sec* scenario, increasing the packet latency. Besides increasing the latency, characterizing a successful attack, this malicious traffic can access and transmit sensitive data. The *secure* scenario had a negligible impact on the IO transaction latency, mainly occurring due to the key renewal (spikes in the *secure* curve), given that the packets are discarded at the moment the authentication fails. The periodic key renewal is an important security mechanism in the proposal, making the spoofing attack more difficult. If authentication is forged, the packet must still have a valid *AppID*, and the service must be what SNIP expects.

The DTW application is periodic. For the three simulated scenarios, the average iteration time was 359 μs (*baseline*), 364μs (*non-sec*, +1.4%), and 366μs (*secure*, +1.94%). Thus the execution time overhead was $\approx$ 2%. This attack did not impact the iteration latencies as it is a computation-intensive application, i.e., a low communication rate with the IO device.

We repeated the attack campaign for a communication-intensive application, *synthetic*. The computation-communication rate of the DTW was 359:8.8 (μs), while for the synthetic, it is 81:8.75 (μs). The behavior of the IO transaction latency remained similar to the previous experiment, as shown in Figure 6(a). The malicious traffic generates DoS in the *non-sec* scenario, and in the *secure* scenario, the latency overlaps the baseline latency. The transaction latency increases in the 2nd and 8th iterations in the *secure* scenario due to the collision between the synthetic and malicious flows.

Figure 6(b) shows the iteration latency for the synthetic application. The average iteration latency between the 5th and 26th iterations (excluding warm-up and end of the application) is 81μs, 101μs (+23.6%), and 86μs (+6.6%) for the *baseline*, *non-sec* and *secure* scenarios. This result shows the effect of the attack in the *non-sec* scenario, given that the malicious application is using the SNIP, and data is probably leaking.

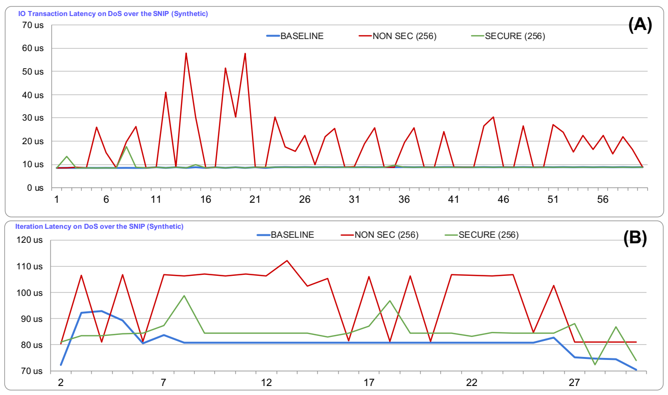


Fig. 6. I/O and iteration latencies for the synthetic application under attack.

In addition, attack campaigns were carried out with the IO device injecting data into the Input Buffer without being requested. In this case, all IO data was discarded.

Such results, considering applications with distinct profiles, demonstrate the SNIP approach's effectiveness in protecting the IO transactions. The execution time has neglectable overhead for real applications, as DTW (2%), or up to 6.6% in a "communication intensive" synthetic benchmark.

B. Area Evaluation

Each processing element (PE) has two routers (data and control routers), a processor, local memory, and a network interface. We synthesized the SNIP (`ApT` with 4 rows, and input/output buffers for 16 *depth* 16-bit flits *size*), the Data Router (two disjoint channels, with 16-bit flits *size* and 8-flit *depth* input buffers), and the Control Router (a router that transmits 1-flit packets in broadcast, with a CAM size with 8 rows) for a 28 nm FDSOI process from ST Microelectronics. Table II presents the synthesis results.

TABLE II
SYNTHESIS RESULTS FOR THE SNIP, DATA ROUTER, AND CONTROL ROUTER - 28NM FDSOI - CADENGE GENUS 21.12-S068_1.

Synthesis results	SNIP	Data Router	Control Router
Cell Area (μm^2)	8,665	15,019	4,657
Net Area (μm^2)	2,470	5,000	2,090
Total Area (μm^2)	11,135	20,019	6,747
Cell Instance Count	3,666	7,776	3,203
Timing slack@500 MHz (ps)	373	136	29
Total estimated power (mW)	3.956	7.463	2.697

Routers represent no more than 20% of the PE area (in [15], the NoC represents 7.7% of the PE area). Table II demonstrates that the SNIP has a low area consumption, representing 55.6% 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.

VII. 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) master-slave communication API, (ii) application authentication, (iii) lightweight packet authentication, (iv) frequent key renewal, and (v) obfuscation via source-routing. The security evaluation involved applications with different profiles, demonstrating the SNIP's effectiveness in thwarting attacks such as Denial of Service (DoS) and spoofing while having a minimal IO iteration latency overhead (2 to 6.6%) and area impact (equivalent to 55% of a data router).

Potential 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 oversees attack monitoring and countermeasures implementation 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.

References

[1] B. Aghaei *et al.*, "Network adapter architectures in network on chip: comprehensive literature review," *Cluster Computing*, vol. 23, no. 1, pp. 321–346, 2020.
[2] ARM, "AMBA® AXITM and ACETM Protocol Specification," February 2013.
[3] S. Charles *et al.*, "Securing Network-on-Chip Using Incremental Cryptography," in *ISVLSI*, 2020, pp. 168–175.
[4] ——, "A Survey of Network-on-Chip Security Attacks and Countermeasures," *ACM Comput. Surv*, vol. 54, no. 5, pp. 101:1–101:36, 2022.
[5] C. Reinbrecht *et al.*, "Gossip NoC - Avoiding Timing Side-Channel Attacks through Traffic Management," in *ISVLSI*, 2016, pp. 601–606.
[6] C. Lee *et al.*, "Towards enhanced I/O performance of a highly integrated many-core processor by empirical analysis," *Cluster Computing*, pp. 1–13, 2021.
[7] Z. Jiang *et al.*, "I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance," in *DAC*, 2021, pp. 1159–1164.
[8] S. Zhao *et al.*, "Timing-Accurate General-Purpose I/O for Multi- and Many-Core Systems: Scheduling and Hardware Support," in *DAC*, 2020, pp. 1–6.
[9] H. K. Kapoor *et al.*, "A Security Framework for NoC Using Authenticated Encryption and Session Keys," *Circuits Syst Signal Process*, vol. 32, no. 6, pp. 2605–2622, 2013.
[10] S. Baron *et al.*, "Security mechanisms to improve the availability of a Network-on-Chip," in *ICECS*, 2013, pp. 609–612.
[11] M. M. Ahmed *et al.*, "What Can a Remote Access Hardware Trojan do to a Network-on-Chip," in *ISCAS*, 2021, pp. 1–5.
[12] F. Restuccia *et al.*, "Aker: A Design and Verification Framework for Safe and Secure SoC Access Control," in *ICCAD*, 2021, pp. 1–9.
[13] L. L. Caimi *et al.*, "Security in Many-Core SoCs Leveraged by Opaque Secure Zones," in *ISVLSI*, 2019, pp. 471–476.
[14] R. F. Faccenda *et al.*, "SeMAP – A Method to Secure the Communication in NoC-based Many Cores," *IEEE Design & Test*, vol. preprint, pp. 1–7, 2023.
[15] A. Rovinski *et al.*, "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, 20119.