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

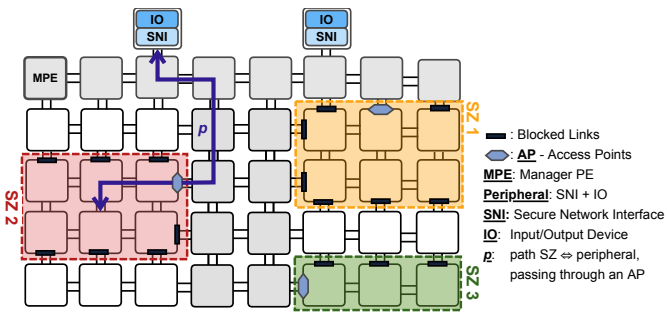


Fig. 1. MCSoC partitioned on secure and gray areas (SA and GA). Three App_{sec} mapped on SZ1 to SZ3. GA is reserved for applications without security constraints.

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES (Finance Code 001), CNPq (grant 309605/2020-2), and FAPERGS grant 21/2551-0002047-4.

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}.

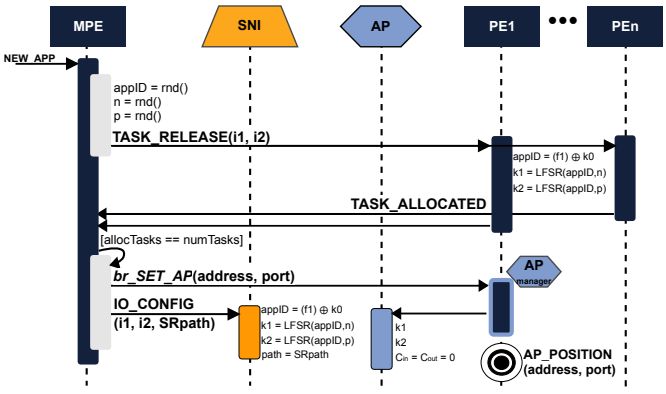


Fig. 2. Sequence diagram of the Application Deploy phase.

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***.

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

PEs restore $\{appID, n, p\}$ upon receiving the initialization flits: $appID = \textbf{i1} \oplus k0_{PE_x}$; $n = MSB(\textbf{i2} \oplus k0_{PE_x})$; $p = LSB(\textbf{i2} \oplus k0_{PE_x})$. 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

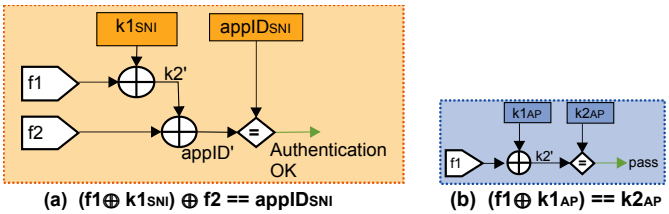
This phase corresponds to authenticated communication between App_{sec} tasks and peripherals. Due to the master-slave communication method, tasks are responsible for starting the communication. Tasks may execute two services: `IO_delivery`, corresponding to sending data to a peripheral; `IO_request`,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

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

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).

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

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



(a) (f1⊕ k1SNI) ⊕ f2 == appIDSNI (b) (f1⊕ k1AP) == k2AP

Fig. 3. Lightweight authentication modules.

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 ($\textbf{f2}_{SNI} \oplus k2_{PE}$) when the packet arrives at the PE. Then it is valid if the retrieved AppID matches the stored AppID.

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 [27].

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 (*in clock cycles*).

Phase	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.

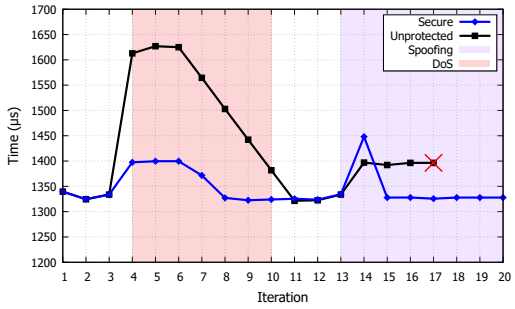


Fig. 4. Application latency for the unprotected and secure systems.

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.

[27] 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. 3-4, pp. 103–122, 2019.