Security in Many-Core SoCs Leveraged by Opaque Secure Zones

Luciano Lores Caimi
UFFS
Av. Fernando Machado 108E – Chapecó, Brazil
lcaimi@uffs.edu.br

Fernando Gehm Moraes
PUCRS – School of Technology
Av. Ipiranga 6681 – Porto Alegre, Brazil
fernando.moraes@pucrs.br

***Abstract*—This paper presents an original approach to protect the execution of applications with security constraints in many-core systems. The proposed method includes three defense mechanisms. The first one is the application admission into the many-core using ECDH and MAC techniques. The second is the spatial reservation of computation and communication resources, resulting in an Opaque Secure Zone (OSZ). The key feature enabling the runtime creation of OSZs is a rerouting mechanism responsible for deviating any traffic traversing an OSZ. The last mechanism is the access to peripherals using a secure protocol to open access points in the OSZ border, and lightweight encryption mechanisms.**

***Index Terms*—Many-core, Network-on-Chip, Security, Opaque Secure Zones, Peripheral Access.**

I. Introduction

Many-core systems gradually walk to encompass hundreds of PEs (processing elements) into a single IC. The system and the applications are exposed to malicious attackers at different moments of the applications' life cycle which can compromise their admission and execution due to the resource sharing feature of many-core systems. Examples of attacks include Denial of Service (DoS), timing attacks, spoofing, side-channel attacks, and information leakage [1][2][3].

This paper presents an approach to protect the execution of applications with security constraints, including three defense mechanisms. The first one is the application admission into the many-core system using Elliptic Curve Diffie-Hellman (ECDH) protocol and Message Authentication Code (MAC) techniques. The second one is the spatial reservation of computation and communication resources, resulting in an Opaque Secure Zone (OSZ). The last mechanism is the access to peripherals using a secure protocol to open access points on the border of the OSZ, and lightweight encryption mechanisms.

The ***goal*** of this paper is to detail the creation of the OSZ and the protocol to define access points in the border of the OSZ. Our ***original*** contribution is twofold. The first one is the proposition of a comprehensive set of methods to protect applications during its life cycle. The second one is to introduce the concept of $OSZs$. In the Authors' knowledge, this is the first work adopting rerouting to avoid traffic crossing regions reserved for secure applications. This feature prevents attacks due to resource sharing.

II. Related Work

The execution of a secure application (App_{sec}) comprises at least three assumptions: (*i*) secure admission of the application, to guarantee the object code integrity; (*ii*) application execution in an environment protected from attacks; (*iii*) protection of the communication with peripherals and shared memories.

The application admission corresponds to the object code transfer from an off-chip entity to the many-core system. Two security issues are present, the authenticity of each actor (external entity and many-core), i.e., the guarantee that the other part is whom it says to be, and the integrity of the application must be verified to avoid the tampering of the object code. Solutions to these issues exist for the Internet, computer networks, and software using techniques such as symmetric encryption [4], Diffie-Hellmann (DH) protocol [5] and Elliptic Curve Cryptography (ECC) [6]. However, from Table I it is possible to observe a lack of proposals in the many-core systems literature related to the mutual authentication of external entities and the secure admission of applications.

Techniques to protect applications' execution include firewalls [3], packet certification [7], encryption mechanisms [8][9] and Secure Zones [1][2][10][11]. Firewalls, packet certification, and encryption mechanisms mainly protect the communication resources against attacks such as DoS, information leakage, spoofing, and Hardware Trojan. Secure zones (SZ) protects computation or communications resources.

The protection of the application with peripherals and shared memories avoids unauthorized access to instructions and data, which may also compromise the application execution, due to the information tampering or leakage using techniques such as firewalls [12] and routing schemes [13].

TABLE I
Related works covering the application admission (**A**), its execution (**E**), and communication with I/O devices (**IO**).

Work	M^*	**A / E / IO**
Khernane et al.[4]-2016		(A) Symmetric Encryption
Zhimeng et al.[5]-2016		(A) Diffie-Hellmann
He et al.[6]-2012		(A) Elliptic Curve Diffie-Hellmann
Real et al.[1]-2018	✓	(E) SZ Partition
Sepulveda et al.[2]-2017	✓	(E) SZ, Sym. and Asym. Encryption
Rajesh et al.[3]-2015	✓	(E) Audit and Firewall
Ancajas et al.[7]-2018	✓	(E) Packet Certification
Oliveira et al.[8]-2018	✓	(E) Symmetric Encryption
Silva et al.[9]-2017	✓	(E) Symmetric Encryption
Fernandes et al.[10]-2017	✓	(E) SZ Routing Scheme
Sharma et al.[11]-2018	✓	(E) SZ Symmetric Encryption
Grammatikakis et al.[12]-2015	✓	(IO) Firewall
Reinbrecht et al.[13]-2017	✓	(IO) Routing Scheme
Our Work	✓	(A) Elliptic Curve Diffie-Hellmann (E) Opaque SZ (IO) Lightweight Cryptography

M^* - applied to many-core systems

Most solutions related to the security applied to NoC-based many-core systems consider only one of the three above mentioned assumptions, being limited to the application execution

978-1-7281-3391-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ISVLSI.2019.00091

or the shared memory access mechanism. Table I summarizes related works. Our proposal stands-out from related works because it covers all phases required to execute an application with security constraints.

III. Secure Zones

Resource sharing is a native feature of NoC-based many-core systems. Different applications may execute in the same processor, share the NoC links, as well as shared memories. This feature, resource sharing, is the source of issues related to security. SZ is an approach adopted to reduce resource sharing.

Methods deployed at design time enable the adoption of sophisticated and robust algorithms to provide solutions to the security problem since they do not have limitations related to the heuristics' computation time. However, design time methods do not apply to dynamic workload scenarios. Thus, these methods are limited to scenarios where the workload is known beforehand, without any change during the system lifetime. The literature presents several proposals to create SZs. It is possible to classify such proposals using a set of orthogonal criteria:

- **Creation time:** the definition of the SZ occurs at design time [10] or runtime [1][2].
- **Shape:** the SZ may be discontinuous [2][10][11] or continuous, with a rectangular [14] or rectilinear shape [15].
- **Communication sharing:** the SZ may allow that flows belonging to sensitive applications share NoC links [2][10][11] or the flow inside the SZ is forbidden to other applications.
- **Computation sharing:** the SZ may allow that tasks belonging to sensitive applications share the same processor [2] or applies resource reservation to sensitive applications [1][10][11].
- **Methods:** the methods used by the SZs include cryptography [2][11], routing algorithms [10], spatial and temporal isolation [1], rerouting.

Figure 1 presents examples of SZs. Discontinuous SZs ($SZ2$) require more efforts to prevent attacks (encryption or routing schemes) due to the flows' exposure, while continuous SZs can imply internal fragmentation when using a rectangular shape, due the reservation of resources without effective use ($SZ1$). A rectilinear shape ($SZ4$) prevents internal fragmentation but needs dedicated routing mechanisms to avoid flows crossing the boundary of the region.

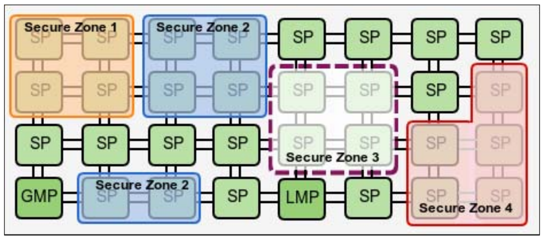

Fig. 1. $SZ1$: continuous and rectangular, $SZ2$: discontinuous, $SZ3$: cont., rect., and opaque, $SZ4$: cont. and rectilinear.

The use of continuous SZ ($SZ1$ and $SZ4$) still exposes the communication to attackers because flows belonging to other applications can transverse the SZ allowing DoS, HT and timing attacks.

IV. Opaque Secure Zones - *OSZs*

According to the previous classification, $OSZs$ are created at runtime, have a rectilinear shape, without computation and communication resource sharing. The PEs of the OSZ are reserved for running a single secure application ($SZ3$, in Figure 1). The only resource sharing exception is communication with I/O devices. The method that enables OSZ is the dynamic rerouting mechanism. The rerouting mechanism ensures that the App_{sec}'s traffic stays inside the OSZ, and deviates all traffic that should cross the OSZ.

A. OSZ Threat Model and Features

OSZ prevents DoS and timing attacks since it is not possible for any external flow to traverse the boundaries of the reserved region. Data confidentiality and integrity is guaranteed because processors of the reserved region do not share the computation with any other application, thus preventing spoofing and hijacking attacks.

Note that due to the resource reservation, there is no application disturbing App_{sec}, improving its performance compared to scenarios where there is NoC congestion or processor sharing.

Hardware Trojans (e.g., inserted into the NoC) are not able to send sniffed data to malicious tasks because no outcoming traffic is allowed and no malicious task can share a PE inside the OSZ.

The proposed method guarantees data application confidentiality and integrity without using encryption inside the OSZ. This feature provides two advantages over solutions that use cryptography, a smaller hardware implementation cost and no delay due to encryption and decryption of messages. Thus, the performance of applications running on an OSZ presents better performance compared to methods requiring flow encryption.

The OSZ also has a smaller area overhead compared to firewall-based solutions since its hardware implementation uses simple gates to mask the control flow signals on the OSZ boundaries (Section V-B), without using tables and rules to check security policies.

B. OSZ Requirements to Meet the Threat Model

Figure 1 presents the reference architecture, an NoC-based many-core system. The software executing at each PE defines its role. The system has a hierarchical architecture organized in clusters, with two types of PEs: (*i*) manager PEs (*MP*), responsible by cluster control and management; (*ii*) PEs executing applications' tasks - slave processors (*SP*). There are two distinct MPs, global manager (GMP) and local managers (LMP). The GMP is an LMP with additional functions, such as communication with external entities to authenticate each other and key management.

Requirements to implement OSZs:

1) Processors' selection to execute App_{sec}. The MPs are in charge to find a continuous region with free SPs to receive App_{sec}. In the absence of a continuous region, the system must support task migration to release SPs in such a way to create the region. Attention must be given to avoid unreachable SPs. For example, in Figure 1 considering all SZs as opaques, the top right SPs are unable to communicate with MPs. Thus, processors' selection should

consider not only the availability of resources but also prevents unreachable SPs.

2) Secure application admission. The Operating System (OS) of each SP receiving an App_{sec} task verifies the task MAC to guarantee the object code authenticity and integrity. The MP enables the App_{sec} execution after receiving the notification that the code of all tasks was correctly verified (Section V-A).
3) Runtime rerouting mechanism. It implies in a dedicated NoC to search paths to circumvent the OSZs, and a data NoC with support to source routing. At the software level, the OS of each SP should be able to resend packets that hit the border of an OSZ and were discarded (Section V-B).
4) Communication of the App_{sec} with peripherals (e.g., shared memory and I/O devices). This requirement may seem contradictory w.r.t the OSZ definition, which states that there is no communication sharing. There is no contradiction because the applications' flow is distinct from the peripherals' flows (Section V-C).

C. *PE Architecture and Peripherals*

The main PE internal modules include (Figure 2): a processor, a DMNI (a network interface with DMA capabilities [16]), a local dual-port memory, a 256-bit pseudo-random number generator (PRNG), wrappers and routers. Two NoCs interconnects the PEs: *data* and *control* NoC. The data NoC transfers *data messages*, exchanged by applications. The control NoC adopts broadcast as the default transmission method, with a small area footprint, corresponding roughly to 20% of the data router [17]. The data NoC adopts duplicated physical channels (ensuring full routing adaptivity), wormhole packet switching, simultaneous support for distributed XY and source routing.

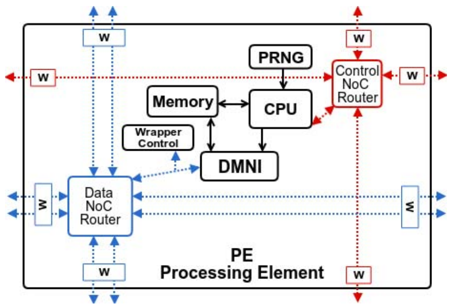

Fig. 2. PE Architecture. Wrappers (W) are added to the control flow signals of the NoC links, enabling to isolate each port individually.

Both NoCs contain test wrappers, or simply *wrappers*, in the control flow signals. The wrappers' function is to block the traffic of a given port to create the OSZs. The incoming and outcoming packets at a given port are discarded when wrappers are enabled.

The control NoC has two operation modes: *global* and *restrict*. The *global* mode enables the control messages to pass through the wrappers, even if they are enabled. This mode enables PEs inside the SZ to exchange messages with MPs. The *restrict* mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded. This mode enables the path discovery mechanism by the control NoC.

Peripherals are connected at unused ports of the mesh NoC (e.g., North ports of top routers). The system requires at least one peripheral: *Application Injector* or simply $AppInj$. It is an external entity attached to the many-core system previously authenticated and enabled to transmit App_{sec}s to execute in the many-core system. The GMP manages the communication between the many-core system and the $AppInj$ by executing the secure application admission protocol.

Architectural assumptions include hardware and software components. At the hardware level, it is necessary to have a control NoC, and wrappers to isolate the PEs. At the software level, only the OSs are modified, responsible for executing the different protocols. Thus, in the Authors' opinion, the proposal may be adapted to other NoC-based many-core systems, not being tightly coupled to the case-study system presented in the results section.

V. Defense Mechanisms

This Section details the defense mechanisms, related to the secure admission and execution of applications. Each phase has internal steps. The application admission phase has five internal steps: (*i*) setup; (*ii*) mutual authentication; *(iii)* OSZ positioning; *(iv)* task's code reception; *(v)* object-code verification. The executions phase three internal steps: *(i)* activation of the wrappers; *(ii)* application execution; *(iii)* OSZ opening. The I/O access occurs during the application execution, and the concern is to enable communication without compromising security.

At the admission phase, the GMP is responsible by the mutual authentication. The LMP defines the OSZ shape and position and sends a broadcast command to the PEs' OS to activate the wrappers. The LMPs also open the OSZ. Thus, App_{sec}s cannot interfere in security management. It is impossible for an App_{sec} to know where other App_{sec}s are or make decisions regarding its mapping.

A. *Secure Application Admission*

The secure admission uses a mutual authentication mechanism based on ECDH to guarantee the authenticity of the $AppInj$ that wants to run an application in the manycore, and a MAC to ensure the integrity of applications during the object code transfer.

Initial steps of the application admission include *setup* and *mutual authentication*. At the *setup* step both $AppInj$/peripherals and the many-core system initially compute a key pair suitable for elliptic curve cryptography. The key pair {Pk, PuK}, corresponds to a private key (Pk) and a public key (PuK), where each key is represented by a random multiplier number and a point over an Elliptic Curve. Next, each one publishes their respective ID and PuK. This phase ends with each part loading the pair {PuK, ID} of each other. The reason to create the keys at the setup phase comes from its high computational cost. At the end of this phase, the GMP contains a table with $AppInj$ and peripherals keys (PuK) and identifiers (ID).

After loading the $AppInj$ key pair, $\{PuK_i,\ ID_i\}$, starts the authentication step:

(a) The $AppInj$ sends a request message encrypted by the many-core system public key (PuK_m). The request message ($AppInj_{req}$) contains the $AppInj$ identifier (ID_i) and a random number ($nonce_i$).

(b) The GMP decrypts $AppInj_{req}$ using it's private key (PK_m), verifying the received ID_i. Next, the GMP sends a reply message (GMP_{rep}) according to the ID_i verification:

(b1) known ID_i: GMP_{rep} encrypted by PuK_i with the tuple $\{ID_m, nonce_i, nonce_m\}$, where ID_m is the many-core ID, and $nonce_m$ is a random number generated by the many-core;

(b2) unknown ID_i: GMP_{rep} encrypted by PuK_m since there is no public key associated to the received ID_i. The reply message is always sent to avoid information leakage;

(c) The $AppInj$ decrypts GMP_{rep}. The $nonce_i$ is evaluated:

(c1) if the $nonce_i$ does not match, it means that the connection with the many-core system is compromised because other actor tried to forge the many-core system ID.

(c2) the $nonce_i$ comparison returns true, corresponding to a correct authentication of the many-core system, once just the lawful pair $\{PuK_m, PK_m\}$ can encrypt/decrypt the initial request message.

(d) The $AppInj$ sends to the many-core system a reply message, $AppInj_{rep}$, with the tuple $\{nonce_i, nonce_m\}$ encrypted with PuK_m.

(e) The many-core decrypts $AppInj_{rep}$ using PK_m. A correct $nonce_m$ authenticates the $AppInj$, once just the lawful pair $\{PuK_i, PK_i\}$ can encrypt/decrypt the message with $nonce_m$. The many-core system verifies $nonce_m$ and ID_i (received on first request message):

(e1) If correct, the many-core system sends an accept message encrypted by PuK_i;

(e2) Otherwise a reject message is sent encrypted by PuK_m.

Once received the accept message, the $AppInj$ generates a *session* key, K_e (or K_p for other peripherals). Then, the $AppInj$ sends the tuple (ID_i, $nonce_m$, K_e) encrypted by PuK_m to the GMP. The GMP keeps the pair $\{ID_i, K_e\}$ to use it in future secure applications deployment by $AppInj$, saving resource consumption and decreasing the latency to start secure applications.

Next steps have three goals: (*i*) define the OSZ location; (*ii*) ask to the $AppInj$ the tasks' codes; (*iii*) transmit to the SPs that will receive the task's codes K_e, and K_ps if the application needs peripheral access.

The GMP receives an application request from the $AppInj$ with the application task graph, using this information to select a given cluster to execute the App_{sec}. Next, the GMP sends to the LMP of the chosen cluster this graph. Using this information, the LMP defines the OSZ shape and location, mapping the application tasks inside it [15]. The resulting task mapping returns to the GMP, which requests the tasks to the $AppInj$ and transmits K_e (and K_ps if necessary) to the SPs of the OSZ using lightweight cryptography.

The $AppInj$ uses K_e to compute a MAC using the SIPHASH algorithm [18]. The $AppInj$ sends the task's object code with the MAC attached at the end of the message. The SP receives the task's object code and, using the same K_e, computes the MAC locally. If the computed MAC matches with the received MAC, the object code integrity is validated.

B. Secure Application Execution

If the previous phase succeeded for all tasks, App_{sec} might execute. The "Close OSZ" step activates the wrappers surrounding the OSZ ("W" in Figure 2), and starts the execution of the App_{sec}.

The wrapper activation occurs using a memory mapped register ($wrapper\_reg$) at each PE. Each bit in the wrapper register enables/disables a given port wrapper of the PE. The wrapper acts over the control flow signals of each NoC port (8 ports for the Data NoC and 4 ports for the Control NoC). Figure 3.a shows the wrapper logic for one port. Thus, the wrappers' area overhead is negligible since its implementation requires a small number of gates, a register, and an FSM.

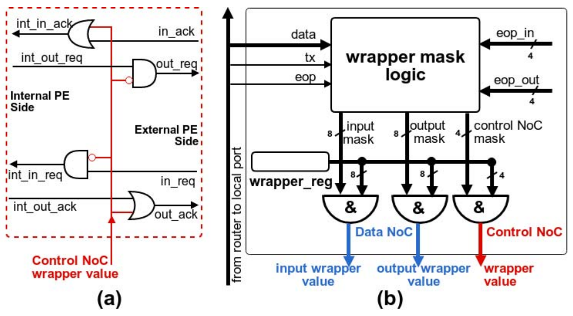

Fig. 3. (a) Wrapper logic for one port; (b) Detailed Wrapper Control masking logic.

When the wrapper value is activated, the internal PE side has the control flow signals masked, disabling all external requests. If a request arrives, it is blocked, and due to the masking, the *ack* signal is high. This simple process results in discarding the packets arriving at the PE. The same process occurs when the PE tries to send a packet, *req* blocked and *ack* enabled. The control NoC uses a single wrapper value, opening or blocking the flows in both directions. The data NoC contains two wrapper values, enabling to selectively block the flow direction (required for I/O communication).

Occurring a packet discarding, the control NoC broadcast a retransmission request to the packet source PE. When the retransmission request arrives at the source PE, the OS uses the control NoC to obtain a new path to the target PE, which avoids the OSZ. The control NoC returns a path, which is used to retransmit the packet, and all subsequent packets through source routing.

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

For security reasons, applications running at the SPs cannot access $wrapper\_reg$, only the OS has access to it. Figure 3.b details the "wrapper control" module presented in Figure 2. The value applied to the wrapper logic explained above is a result of a AND operation between the $wrapper\_reg$ value and a mask value coming from "wrapper mask logic" (WML) module. The default value of mask signals arriving the AND operation is '1', i.e., by default the wrappers values corresponds to $wrapper\_reg$ contents.

C. Access to Peripherals

The communication model assumed in this work is message passing (MPI-like), with the API using $Send()$ and $Receive()$ primitives. At the lower level, the OS communicates with the data NoC with $data\_request$ and $data\_delivery$ packets. A message buffer (in the OS) enables packet retransmission.

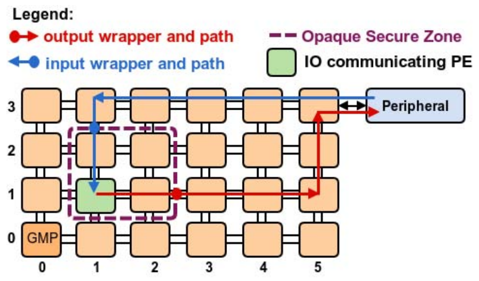

Fig. 4. Example of I/O communication through an OSZ.

The support for I/O communication uses a second API, with $IO\_Send()$ and $IO\_Receive()$ primitives, using a master/slave communication model. The PE is the communication master and the peripherals the communication slaves. At the lower level, the OS communicates with the data NoC with $IO\_request$, $IO\_delivery$, and IO_ack packets. The $IO\_Receive()$ primitive uses the $IO\_request$ at the PE side and the $IO\_delivery$ at the peripheral side. The $IO\_Send()$ primitive uses $IO\_delivery$ at the PE side and the IO_ack at the peripheral side.

The data NoC differentiates the API with a flag in the header field (first flit of the packet). This feature enables to block all data packets arriving at the boundary of the OSZ (in both directions) and to apply selective management of I/O packets.

Figure 4 presents an example of an SP belonging to an OSZ communicating with an external peripheral, using the default XY routing algorithm.

Before transmitting a message to a peripheral, the communicating SP first sends two configuration messages to the boundary of the OSZ, to set the mask registers. The SP knows the boundaries of OSZ, obtained in the 'Close OSZ' step. When the mask configuration message arrives at the target router, the wrapper control module (Figure 2) intercepts the message (i.e., this message is not consumed by the PE) to set the mask value (input mask or output mask value) as shows the Figure 3.b.

The mask configuration messages contain the direction (input or output) and the port side to mask (e.g., north). Note that the WML module (Figure 3) receive the EOP signals (End-Of-Packet) of each data router port. Thus, once an EOP is received in an opened port (input or output), this port is closed. This mechanism ensures that the secure zone receives only one packet for each request.

The procedure of sending and receiving I/O messages opening specific ports, with specific directions is subject to two main issues:

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

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

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

Currently, the support to I/O lightweight cryptography is partially addressed. The peripheral authentication, key management, and distribution are fully implemented. The PRESENT encryption algorithm has a software implementation validated at the many-core system side, enabling to send encrypted messages to the peripheral. For performance reasons a hardware module still needs to be integrated into the DMNI and peripheral.

VI. Evaluation

This Section initially evaluates the two first steps of the Application Admission, corresponding to the system setup and mutual authentication. The evaluation goal is to show the time saving due to the authentication process made before the application admission. To execute this experiment, the $AppInj$ and GMP hardware architectures were described using OVPSim APIs [19], an instruction-set accurate simulator. The cryptographic functions of this work use the TweetNaCl [20], a compact and self-contained public-domain C library. The library uses Curve25519 on the ECDH key exchange.

Table II shows the two first protocol steps and the number of instructions of these actions in the MIPS and ARM processors. The ARM processors use a SIMD ISA. The setup and authentication step actions require for the MIPS/ARM processor, 239.2M/83.9M and 2244.9M/ 841.4M instructions respectively. The total amount of instructions for the two first steps are 2,484.1M and 925.3M instructions, in MIPS and ARM processors, respectively. Considering the processors running at 500 MHz with a CPI equal 1, the time consumption relative to the mutual authentication of the $AppInj$ and the many-core system corresponds to 4.96s for MIPS and 1.85s for ARM.

TABLE II
Setup and authentication step evaluation.

Step	MIPS Instructions ($x10^6$)	ARM Instructions ($x10^6$)
Setup	239.2	83.9
Authent.	2244.9	841.4
TOTAL	**2484.1**	**925.3**

Although this overhead seems high, in the order of seconds, **only one execution** of the mutual authentication process occurs for each $AppInj$ to ensure the authenticity of the parts and does not impact the latency to start the applications.

The second experiment evaluates the wrappers' behavior at the OSZ boundary, as shown in Figure 4, under a DOS attack campaign promoted by a malicious task located at SP (3,3), including several periods with the wrapper opened. The evaluation uses a clock-cycle accurate RTL SystemC description of the many-core system presented in Figure 1 and Section IV-C.

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

The third experiment evaluates the performance impact using the proposed mechanism to communicate with I/O devices. The experiment uses the same platform of the previous experiment, a clock-cycle RTL description of the many-core system. The experiment considers five scenarios (Figure 4):

(1) baseline: an application executing I/O communication without OSZ;
(2) OSZ activated, SPs {(1,1), (2,1), (1,2), (2,2)}, enabling to evaluate the impact of the wrappers' configuration;
(3) OSZ activated with a second OSZ (SPs {(4,1),(4,2)}), blocking the output I/O messages from SP (1,1), requiring the *searchpath* service and the reroute mechanism to retransmit the output message;
(4) OSZ activated with a second OSZ (SPs {(4,2),(4,3)}), blocking the input I/O messages from the peripheral, requiring the *searchpath* service and the reroute mechanism to enable the peripheral to retransmit the message;
(5) OSZ activated with a second OSZ (SPs {(4,1),(4,2),(4,3)}, blocking both output and input I/O messages, requiring two *searchpath* services and reroute mechanisms to correct delivery the I/O message.

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

TABLE III
OVERHEAD TO COMMUNICATE WITH I/O DEVICES.

Scenario	Clock Cycles (CC)	Overhead CC/iteration
(1) I/O	283709	-
(2) I/O + OSZ	317746	680.74
(3) I/O + OSZ + output rerouting	320933	744.48
(4) I/O + OSZ + input rerouting	321861	763.04
(5) I/O + OSZ + rerout. both dir.	326540	856.62

The total execution time increases from 12% up to 15%, an expected result because the secure application is a synthetic application, executing only I/O communication. The relevant result is the one presented in the third column, the communication overhead. The overhead to configure the wrappers and find new paths corresponds to less than 900 clock cycles per I/O access. Once the path configured, it is used for the next packets, without incurring additional overheads. Two main reasons explain this remarkable result: (*i*) a simple wrapper configuration mechanism; (*ii*) the adoption of a dedicated NoC to find the paths when rerouting is required. Our proposal leaves most of the overhead to the hardware, minimizing the amount of required software. These results do not consider the lightweight cryptography (LWC) latency. Additional overhead is expected, but it will not impact the overall results. The reason is that the LWC methods process flows in a pipeline fashion, requiring an initial latency to process the data.

VII. Conclusions and Future Work

This work showed the need to adopt security methods covering the whole application lifetime, using mechanisms to protect the application admission and then reduce the resource sharing to avoid attacks. It also proposed an original procedure to mitigate resource sharing, the OSZs. The rerouting proposal avoids traffic from other applications to cross the regions reserved for secure applications. Such method avoids most of the attacks described in the literature. The hardware overhead due to the adoption of OSZs is smaller than firewall- and encryption-based methods due to the adoption of wrappers and a small dedicated NoC to reroute packets. Finally, the work proposed a robust method to enable OSZs to communicate with I/O devices. Future work includes: (*i*) generate attack scenarios to identify security issues not covered by the proposal; (*ii*) integrate an LWC hardware module to protect the communication with peripherals.

VIII. Acknowledgement

The authors would like to thank Imperas Software and Open Virtual Platforms for their support and access to their models and simulator. Luciano L. Caimi is supported by CAPES (184993/2018-00). Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (302531/2016-5).

References

[1] M. M. Real *et al.*, "Application Deployment Strategies for Spatial Isolation on Many-Core Accelerators," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 55, pp. 1–31, 2018.
[2] J. Sepulveda *et al.*, "Efficient security zones implementation through hierarchical group key management at NoC-based MPSoCs," *Microprocessors and Microsystems*, vol. 50, pp. 164 – 174, 2017.
[3] J. Rajesh *et al.*, "Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip," in *NOCS*, 2015, pp. 8:1–8:8.
[4] N. Khernane *et al.*, "BANZKP: a Secure Authentication Scheme Using Zero Knowledge Proof for WBANs," in *MASS*, 2016, pp. 307–315.
[5] L. Zhimeng and Z. Yanli, "Provable Secure Node Authentication Protocol for Wireless Sensor Networks," in *WISA*, 2016, pp. 221–224.
[6] D. He, J. Chen, and Y. Chen, "A secure mutual authentication scheme for session initiation protocol using elliptic curve cryptography," *Security and Communication Networks*, vol. 5, no. 12, pp. 1423–1429, 2012.
[7] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-NoCs: Mitigating the Threat of a Compromised NoC," in *DAC*, 2014, pp. 1–6.
[8] B. Oliveira, R. Reusch, H. Medina, and F. G. Moraes, "Evaluating the Cost to Cipher the NoC Communication," in *LASCAS*, 2018, pp. 1–4.
[9] M. R. Silva and C. A. Zeferino, "Confidentiality and Authenticity in a Platform Based on Network-on-Chip," in *SBESC*, 2017, pp. 225–230.
[10] R. Fernandes *et al.*, "A security Aware Routing Approach for NoC-based MPSoCs," in *SBCCI*, 2016, pp. 1–6.
[11] G. Sharma and other, "Secure Communication on NoC based MPSoC," in *ATCS*, 2018, pp. 1–6.
[12] M. D. Grammatikakis *et al.*, "High-level security services based on a hardware NoC Firewall module," in *WISES*, 2015, pp. 73–78.
[13] C. Reinbrecht *et al.*, "Timing attack on NoC-based systems: Prime+Probe attack and NoC-based protection," *Microprocessors and Microsystems*, vol. 52, pp. 556–565, 2017.
[14] H. Isakovic and A. Wasicek, "Secure channels in an integrated MPSoC architecture," in *IECON*, 2013, pp. 4488–4493.
[15] L. L. Caimi, V. Fochi, E. Wächter, and F. G. Moraes, "Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications," in *LASCAS*, 2018, pp. 1–4.
[16] M. Ruaro, F. B. Lazzarotto, C. A. M. Marcon, and F. G. Moraes, "DMNI: a Specialized Network Interface for NoC-based MPSoCs," in *ISCAS*, 2016, pp. 1202–1205.
[17] 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.
[18] J.-P. Aumasson and D. J. Bernstein, "A Fast Short-Input PRF," in *INDOCRYPT*, 2012, pp. 489–508.
[19] OVP, "Open virtual platform," Access April 2019 2018. [Online]. Available: http://www.ovpworld.org/technology_ovpsim
[20] D. J. Bernstein *et al.*, "TweetNaCl: A Crypto Library in 100 Tweets," in *LATINCRYPT*, 2015, pp. 64–83.