Runtime Creation of Continuous Secure Zones in Many-Core Systems for Secure Applications

Luciano L. Caimi*†, Vinicius Fochi†, Eduardo Wachter†, Fernando G. Moraes†
*UFFS – Av. Fernando Machado 108E, 89802-112, Chapecó, Brazil – lcaimi@uffs.edu.br
†FACIN - PUCRS – Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil – fernando.moraes@pucrs.br

***Abstract*—Security is an important design issue in current many-core systems. Several applications run simultaneously, processing sensitive data. The literature describes different attacks on many-core systems, stealing data from processors, by unauthorized access to the memories, and from the communication infrastructure, by sniffing the packets or even modifying them using Hardware Trojans. Thus protect at the same time the computation and communication infrastructures is a paramount requirement to add security to applications processing sensitive data. This paper presents a method to create at runtime secure zones in the many-core, by reserving the computation and communication resources exclusively to the secure application. The method isolates the processing elements of the secure zone, blocking any traffic to cross the region. Traffic that should cross the secure zone is forwarded to the outside of the region using a dynamic rerouting mechanism. The main contribution of this paper is the proposition of an algorithm to define the secure zone. Results evaluate the cost to execute the proposed algorithm with different secure zone shapes, showing that its execution cost is small and that several secure application may execute in parallel, using different regions of the many-core.**

***Index Terms*—Many-core, MPSoC, Network on Chip, Security, Secure Zone.**

I. Introduction

As the adoption and complexity of many-core systems increases, the concern for data protection appears as a major design requirement. Applications that handle sensitive information running in NoC-based many-core systems should protect data and code, adding security mechanisms to prevent attacks. Thus, it is necessary to protect the application's data from unauthorized access.

The literature presents a diversity of mechanisms used in NoC-based many-core systems to protect communication and computation from attacks. Mechanisms used to protect the communication include: (*i*) firewalls [1]; (*ii*) secure zones [2] ; (*iii*) routing schemes [3]; (*iv*) temporal network partitioning [4] ; (*v*) cryptography [5]. To protect computation the main mechanisms used are: (*i*) spatial and/or logical isolation [6]–[8]; (*ii*) secure zones [9].

A secure zone (SZ) is a common mechanism used to protect both communication and computation resources. Secure zones provide an interface that isolates the execution and data exchanged between tasks of the application using some technique, like cryptography, routing scheme, logical isolation.

The *goal* of this paper is to present a runtime algorithm to create secure zones in NoC-based many-core systems to execute applications with security concerns. Despite the available proposals in the literature, the method herein presented stands out in the reduced hardware and software requirements, as discussed in the state-of-the-art section.

The *contribution* of this work is twofold. First, using orthogonal criteria, we present a classification of the available techniques to create secure zones. Afterwards, the paper presents and evaluates an algorithm to create continuous secure zones, at runtime, in NoC-based many-core systems.

II. State-Of-the-Art

Fernandes et al. [2] propose a design-time method that enables the creation of SZ based on the routing algorithm to mitigate DoS and timing side channel attacks. The Authors extend the Segment-based Routing (SBR) to security purposes creating the SBR Security Zone Awareness (SBR-SZA) that enables the creation of SZs. After running, at design time, the SBR-SZA, the Region-based Routing Algorithm (RBR) is used to create routing restrictions avoiding shared paths between different applications and deadlock-free paths.

Real et al. [6] propose a logical and spatial isolation of sensitive applications through the dynamic creation of SZs to mitigate DoS attacks and guarantee data confidentiality and integrity. The method occurs at runtime. The architecture uses the MPSoCSim, a mesh NoC where each router is connected to a cluster with 4 processors (with local memory), 1 shared memory and 1 shared bus. The SZ only isolates the cluster resources. When a given task sends a message to a task in another cluster, this message uses an insecure channel. The performance overhead of the proposed mechanisms increases with the number of required secure zones.

ARM processors provide the ARM TrustZone (ATZ) [7], a hardware support for the creation at runtime of Trusted Execution Environments (TEEs) and therefore the isolation of applications in the same processor. This feature creates 2 virtual processors and 2 Memory Management Units (MMU), allowing to execute a secure and a non-secure application simultaneously. However, at any instant, only a single domain in the system can be secured. TEE allows the secure partition of the shared memory controlling memory accesses to avoid data extraction and change (mitigating confidentiality and integrity attacks). Nevertheless, in multicore and many-core architectures, applications running on different processors share resources such as the communication infrastructure (NoC, buses) and memory. Thereby, with TEE, applications running on different processors are not protected from each other since sharing the communication infrastructure leads to possible leakage of information.

Isakovic et al. [8] obtain computation and communication protection using spatial isolation with encryption mechanisms. The Authors propose an architectural partitioning of the MPSoC resources at design time to prevent availability, confidentiality and integrity attacks. The Authors adopt security components like a *secure μKernel* and a *secure channel* infrastructure that includes cryptography and firewalls. The Authors propose to migrate the security functions from application components to the security components.

Sepúlveda et al. [9] also protect computation and communication resources using spatial isolation with encryption mechanisms. The Authors propose a NoC-based architecture that implements run-time discontinuous SZs using three cryptographic techniques: hierarchical Diffie-Hellman, hierarchical Tree-based Diffie-Hellman, and mapping key predistribution scheme. The method prevents attacks on availability, confidentiality, and integrity of the system. The architecture adopts two NoCs: (a) data NoC, used for the application data; (b) service NoC used to exchange the security control packets (key exchange, firewall rules, etc.). After mapping the application, one of the key agreement protocol is executed between the mapped PEs using the service NoC. The encryption/decryption is obtained XORing the message with the shared key.

978-1-5386-2311-4/18/$31.00 ©2018 IEEE

TABLE I: Secure Zone Methods.

Proposal	Creation	Shape	Commun. Sharing	Comp. Sharing	Method
Fernandes (2016) [2]	Design	Discontinuous	Yes	No	Routing Tables and encryption
Real (2016) [6]	Runtime	Discontinuous	Yes	No	Spatial and temporal isolation
ARM (2008) [7]	Runtime	Continuous (same CPU)	Yes	Yes	Logical and temporal isolation
Isakovic (2013) [8]	Runtime	Rectangular	Yes	No	Spatial isolation and encryption
Sepulveda (2017) [9]	Runtime	Discontinuous	Yes	No	Spatial Isolation, firewalls and encryption
Our Proposal	Runtime	Rectilinear	No	No	Spatial Isolation, wrappers, rerouting

A. SZ Classification

From the above review, Table I classifies the SZ proposals using a set of orthogonal criteria:

- **Creation time:** the definition of the SZ occurs at design time [2] or runtime [6]–[9].
- **Shape:** the SZ may be discontinuous [2], [6], [9], or continuous [7], [8], with a rectangular or rectilinear shape.
- **Communication sharing:** the SZ may allow that flows belonging to sensitive applications share NoC links
- **Computation sharing:** the SZ may allow that tasks belonging to sensitive applications share the same processor.
- **Methods:** the methods used to create the secure zones include cryptography, routing algorithm, logical isolation, spatial isolation, temporal isolation, rerouting.

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 computation time of the heuristics. However, design time methods are not applicable in dynamic workload scenarios. Thus, these methods are limited to scenarios where the workload is known beforehand, without any change during the life cycle of the system.

Discontinuous SZ secure zones [2], [6], [9] require more efforts to prevent attacks, with the use of encryption or routing schemes, while continuous secure zones can imply internal fragmentation of resources, i.e., reservation of resources without effective use. The use of continuous SZ in [7], [8] still exposes the communication to attackers because the implemented mechanism does not isolate the communication resources.

As observed in Table I, all reviewed works present communication sharing, i.e., the traffic of the secure application is subject to timing, side-channel attacks, DOS and Hardware Trojans. Two situations may expose the communication to attacks. The first one is due to the possibility of flows not belonging to the secure application share the NoC links. The second one is due to the communication of the secure flows with IPs outside the SZ or due to the adoption of discontinuous SZ. The standard solution to protect the communication is to adopt encryption mechanisms.

A common feature observed in Table I is the computation protection, i.e., the methods reserve the processors required to execute the secure applications. The exception is the ARM proposal [7], which enables the secure application to share the same CPU with other applications. As discussed, this proposal creates virtual processors to enhance security.

The *original* contributions of our proposal w.r.t the state-of-the-art are: (*i*) prevent communication and computation sharing at runtime, by reserving processing elements (PEs) and communication channels to execute the secure application (App_{sec}); (*ii*) several SZs may co-exist in parallel. The flows are not exposed to attackers due to the adoption of wrappers to isolate the SZ, blocking all incoming and outcoming packets, corresponding to a full spatial isolation (see isolation and rerouting in Figure 1). The method relies on a rerouting mechanism, able to reroute packets at runtime [10]. With the goal to avoid unnecessary PEs reserved inside the SZ, they are removed from the region, resulting in a rectilinear polygon (see $SZ2$ in Figure 2).

III. System Architecture

The baseline architecture is a homogeneous NoC-based many-core, where each PE contains a 32-bit RISC processor, a DMNI module (a network interface with DMA capabilities) and a local dual-port memory accessed by the processor and DMNI module. The software executing at each PE defines its role in the system. The system has manager PEs (MP), and PEs responsible for executing the applications (SP - slave processor).

Two NoCs interconnects the PEs: *data* and *control* NoC. The data NoC transfers *data messages*, exchanged by applications. The control NoC [10] transfers management packets. Both NoCs contain test wrappers, or simply *wrappers*, in the flow control signals. When activated, the wrapper enables to discard all incoming and outcoming packets of a given port. In the current work, the Operating System (OS) of each PE controls the wrappers connected to the data NoC. The control NoC manages their wrappers, for security reasons, i.e., the applications running at the PEs cannot access the wrappers of the control NoC.

The current work uses the control NoC to transfer the control messages to close and open an SZ. The control NoC has two operation modes: *global* and *restrict*. The *global* mode enables the control messages to pass through the wrappers, even if they are enabled. This mode enables the PEs inside the SZ to exchange messages with manager PEs. The *restrict* mode observes the status of the wrappers, discarding all received control message.

The data NoC observes the status of the wrappers. In this case, an activated wrapper discard all received data messages, and the control NoC returns to the source of the message a new message reporting that the message needs retransmission. This mechanism enables the PEs on both sides of SZ to search a new path.

IV. Threat Model

The resource sharing of the many-core components introduces vulnerabilities to App_{sec}s running on it. It is possible to explore these vulnerabilities in attacks, such as: confidentiality and integrity; timing attack; denial of service; spoofing; hijacking [2], [4], [6].

Our proposal adopts two *fundamental assumptions*: *(i)* an SZ is a continuous region; *(ii)* there is no resource sharing inside the SZ, i.e., only one application executes inside the region. The activation of the wrappers at the boundary of the SZ enables the first assumption. The mapping procedure enables the second assumption. Once defined the SZ, tasks executing in the SZ are migrated to outside the SZ before the App_{sec} mapping. The definition of SZ and the migration of tasks are managed by Algorithm 1 (section V).

The adoption of these assumptions avoids the attacks previously presented. As only the App_{sec} executes in the SZ, and the wrappers block the traffic at the boundary of the region, any access or data modification by an external application is blocked, ensuring data confidentiality and integrity. Timing or DoS attacks to the App_{sec} are automatically refused by the wrappers since no external traffic is allowed inside the SZ. Thus, malicious applications cannot extract any information with temporal correlation to a task running inside the SZ, or overload the resources (routers and processors). In the same way, this method prevents spoofing attacks since data cannot cross the SZ boundary. Side channel attacks, as power-monitoring or electromagnetic attacks, are not considered in this proposal. However,

the feasibility of such attacks in a system with dozens of processors is unlikely to occur.

V. Proposed Method to Create and Set Secure Zones

The proposed method includes: *(i)* shape selection; *(ii)* wrapper activation; *(iii)* retransmission of lost packets in and out the SZ boundaries; *(iv)* launch App_{sec}. Figure 1 illustrates the method. In Figure 1(a) the MPSoC contains one application in execution, app_1. Next, the MP maps an App_{sec}, activating the wrappers at the boundary of the SZ. At this moment (Figure1(b)), the app_1 traffic is blocked by the SZ. Figure 1(c) shows the App_{sec} executing in the SZ, and the traffic of app_1 circumventing the region.

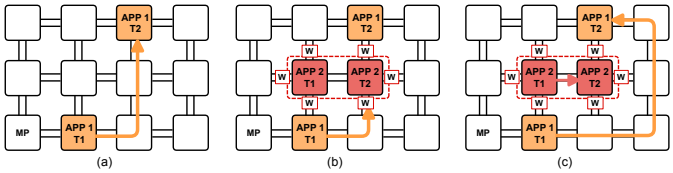

Fig. 1: Secure zone and dynamic reconfiguration of routing paths.

A protocol defines the steps to admit a new application [11]. The MP first executes the algorithm proposed in Section V-A, which determines the location of the SZ, and fires task migrations if necessary to reserve the SZ exclusively to the App_{sec}. After defining the SZ, the MP maps the tasks inside this region, considering as cost function the communication cost between the tasks. After these steps, the MP activates the wrappers of the SZ and transmits a message to start the execution of the App_{sec}.

A. Algorithm to Create Secure Zones at Runtime

Algorithm 1 presents the pseudo-code to create at runtime an SZ. The algorithm inputs are the number of App_{sec}'s tasks (`app.#tasks`), the number of tasks PEs may execute simultaneously (`#tasks.PE`), the cluster side size (`cluster_side`), and the manager position (`MP.pos`).

Definition 1. *Shape*: a rectilinear polygon, enclosing a set of PEs, without any task executing inside it, avoiding computation and communication sharing.

Definition 2. *Fragmentation*: shape area minus the number of PEs to execute App_{sec}. The algorithm removes PEs from rectangular regions to avoid internal fragmentation, leading to rectilinear polygons.

Definition 3. *MAX_MIGRATION*: design time parameter, defining the maximum number of task migrations. Once a shape defined, it may contain tasks belonging to other applications. The algorithms migrate the tasks running in the selected region to guarantee the exclusive use of the processors by the *App_sec*. Due to the cost of the task migration, the algorithm limits the number of task migrations.

The first step of the algorithm, lines 1–5, is the definition of the $shape\_set$. The loop computes the number of PEs to execute App_{sec} according to `app.#tasks` and `#tasks.PE` (line 3). The function *shapes* returns a set of rectangular shapes. For example, consider $app.\#tasks = 7$ and $\#tasks.PE = 2$. The first iteration ($t = 1$) requires 7 PEs, returning shapes $\{$`(3,3)`, `(2,4)`, `(4,2)`$\}$, with a fragmentation equal to 2, 1 and 1 for each shape. The second iteration ($t = 2$) requires 4 PEs, returning shapes $\{$`(2,2)`, `(1,4)`, `(4,1)`$\}$. For these shapes, there is no fragmentation.

If the $shape\_set$ is empty (line 6–7) the algorithm returns $FALSE$, meaning that the cluster does not have a shape to execute App_{sec}. In this case, the MP runs a reclustering function, borrowing resources from neighbor clusters, to guarantee a shape with resources to execute App_{sec}. After this process, Algorithm 1 is re-executed.

If the cluster may receive App_{sec} (*else* block), the algorithm sorts the $shape\_set$ in ascending order, using as criteria the shape size.

Algorithm 1: Search resources to create the SZ

```
   Input:  app.#tasks, #tasks.PE, cluster_side, MP.pos
   Output: sh // selected shape
1  shape_set ← ∅
2  for t from 1 to #tasks.PE do
3  |  PEs_needed ← ceil(app.#tasks / t)
4  |  shape_set ← shape_set ∪ shapes(PEs_needed, cluster_side)
5  end
6  if shape_set = ∅ then
7  |  return FALSE
8  else
9  |  sort(shape_set, largest) // sort shape set according to the area, largest first
10 |  for mig# from 0 to MAX_MIGRATION do
11 |  |  if mig# = 1 then
12 |  |  |  sort(shape_set, smallest) // sort shape set, smallest first
13 |  |  end
14 |  |  forall sh[i] in shape_set do
15 |  |  |  forall PE in (cluster_side × cluster_side) do
   |  |  |  |  // SWS: Sliding Window Search
16 |  |  |  |  usedPEs ← SWS(PE.xy, sh[i].Δxy, MP.pos, cluster_side)
17 |  |  |  |  if usedPEs = mig# then
18 |  |  |  |  |  sh[i].x ← PE.x
19 |  |  |  |  |  sh[i].y ← PE.y
20 |  |  |  |  |  if sh[i].fragmentation > 0 then
21 |  |  |  |  |  |  x ← sh[i].x + sh[i].Δx - 1
22 |  |  |  |  |  |  for y from 1 to sh[i].fragmentation do
23 |  |  |  |  |  |  |  PE.pos(x, sh[i].y).invalid ← 1
24 |  |  |  |  |  |  end
25 |  |  |  |  |  end
26 |  |  |  |  |  if usedPEs ≠ 0 then
27 |  |  |  |  |  |  foreach PE in sh[i] do
28 |  |  |  |  |  |  |  if PE.used ≠ 0 then
29 |  |  |  |  |  |  |  |  migrate_task(PE, sh[i])
30 |  |  |  |  |  |  |  end
31 |  |  |  |  |  |  end
32 |  |  |  |  |  end
33 |  |  |  |  |  return sh[i]
34 |  |  |  |  end
35 |  |  |  end
36 |  |  end
37 |  end
38 |  return FALSE
39 end
```

The rationale is to minimize the CPU sharing (one task per PE), maximizing the App_{sec} performance.

After the sorting step, the algorithm starts the search process. This process requires three loops: (*i*) outer loop (line 10), controls the number of tasks migrations; (*ii*) intermediate loop (line 14) traverses the shape set; (*iii*) internal loop (line 15) traverses the PE set.

The search process adopts a sliding window search (SWS) procedure (line 16), using as inputs the PE coordinates ($PE.xy$), the shape size ($\Delta.xy$), the manager position, and the cluster side size. If the shape fits in the cluster and does not overlap the manager processor or an active SZ, the SWS procedure returns the number of processors executing tasks inside the region, otherwise -1, in the variable $usedPEs$.

The first iteration of the outer loop disables task migrations ($mig\# = 0$). Thus, after executing the SWS procedure, all PEs must be available, i.e., there is no task executing in the PEs inside the region. If one of the shapes fills this condition (line 17), the bottom left coordinate of the selected shape receives the current PE position (lines 18-19). For this first iteration of the outer loop, only defragmentation (Definition 2) occurs, if necessary (lines 20-25). If the shape has internal fragmentation, the rightmost shape column has some of the PEs marked as invalid (line 23), modifying the shape from a rectangular to a rectilinear format (Definition 1). After this process, the selected shape returns to the caller function.

Subsequent iterations of the outer loop enable task migrations, from 1 to *MAX_MIGRATIONs* (Definition 3). With the goal to reduce the number of task migrations, the shape set is reordered, with the smallest shapes evaluated first. The previous process is repeated, executing the SWS procedure. When the number of used PEs inside

the cluster is equal to the allowed number of task migrations, the shape is selected. Besides defragmentation, tasks in used processors are migrated to outside the region (lines 26-32).

VI. Results

The evaluation was conducted using a clock-cycle description models many-core system (RTL SystemC and VHDL). Operating system and applications are described in C language. The experimental setup includes a 6x6 many-core instance (*cluster_side=6*), a 9-task application (App_{sec}), up to 2 tasks may execute simultaneously per PE (*#tasks.PE=2*). With such configuration, the *shape_set* is equal to: $\{(3,3),(2,5)^*,(5,2)^*,(2,3)^*,(3,2)^*,(1,5),(5,1)\}$. Shapes marked with a '*' have internal fragmentation (equal to 1), requiring the removing of PEs from the shape.

Figure 2 presents the 6x6 many-core, with producer-consumer applications mapped when the system starts – pink PEs, with the goal to create obstacles to the SZ creation, and three SZs. The creation of the SZs occurs according to the following sequence:

- $SZ1$: corresponds to the smallest execution time of Algorithm 1 because the execution of the outer loop finds a valid region in the first iteration;
- $SZ2$: corresponds to an intermediate execution time due to the execution of the SWS several times, with the removal of a PE from the region (this is an example of a rectilinear polygon);
- $SZ3$: corresponds to the largest execution time to find an SZ in this scenario.

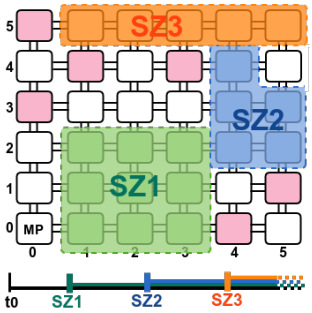

Fig. 2: Secure zone evaluation scenario.

Note that the algorithm creates the three SZs at runtime, with App_{sec}s executing in parallel. Also, the blue SZ (SZ2) requires internal rerouting when the top-left PE send packets to PEs on the right side of the SZ (if XY routing is used the packets would expose the communication). The experimental setup also has scenarios with tasks mapped inside the SZs with the goal to activate the task migration, enabling to evaluate its cost (not presented in Figure 2).

Table II presents the evaluation of the proposed method. The first column shows the Secure Zones created according to Figure 2. Results for execution of *Algorithm 1* (2nd column of the Table) are a function of the number of shapes and the position of selected shape in the cluster. The *Migration* results (3rd column) shows the cost to migrate one task of 4.3 KB at a distance of 4 hops. The *Protocol* delay (4th column) is related to a set of messages send to the PEs of the selected shape to inform the boundaries of SZ. The *Delay to start* App_{sec} (5th column) is the sum of the previous columns that correspond to delay to start the App_{sec} due to the proposed method. This includes the delay to find the shape and an appropriated position, the delay in migrating tasks inside the selected position and, the delay due to the protocol messages exchanged with the PEs of SZ. The internal Source Routing evaluation (6th column) shows the delay to find a new path to a message lost due to the fragmentation.

Results show a low impact on executing Algorithm 1 to create and search a suitable shape position. In the worst-case of the proposed evaluation scenario (SZ3), the delay was 20.5K clock cycles (cc). The cost of the *migration* is a function of the number of migrations and the size of the task's object code. The proposed algorithm tries to minimize the number of migrations first looking for regions that not require migration and after looking by small shapes (and potentially fewer migrations). The cost of the *protocol* varies with the distance of the PEs belonging to the SZ and the MP. The *protocol* has a negligible impact on the delay to start the App_{sec}. The *internal SR* cost impact in the execution time of the App_{sec} due the rerouting mechanism used. The cost varies with the number of the broken paths because when a new path is found, all subsequent messages use it. Overall, results show the effectiveness of the approach and the low impact to start the App_{sec} (94.2K cc, or less than 0.1ms@100MHz).

TABLE II: Secure Zone Performance Evaluation. All results correspond to the number of clock cycles.

Secure Zone	Alg. 1	Migrations	Protocol	Delay to start App_{sec}	Internal SR (App_{sec} delay)
SZ1	3.3K	-	4.8K	8.1K	-
SZ2	12.7K	-	5.1K	17.8K	18.9K
SZ2	12.7K	24.0K	5.1K	41.8K	18.9K
SZ3	20.5K	-	5.3K	25.8K	-
SZ3	20.5K	23.4K	5.3K	94.2K	-

VII. Conclusion

This paper presented a method to create SZs in many-core systems. Relevant characteristics of the method include: (*i*) runtime execution; (*ii*) several SZs may co-exist in parallel; (*iii*) absence of communication and computation sharing; (*iv*) packets are transmitted without cryptography, not penalizing the execution time of the secure application; (*v*) small area overhead (only wrappers added to isolate the PEs); (*vi*) small latency to start the secure applications. In the Authors knowledge, this is the first proposal enabling complete communication isolation, due to the dynamic rerouting of packets. Future works comprise to enhance the secure admission protocol of secure applications and the addition of a lightweight cryptography method to enable the communication of the SZ with external IPs.

VIII. Acknowledgement

Author Fernando Gehm Moraes is supported by FAPERGS (17/2551) and CNPq (302531/2016-5), Brazilian funding agencies.

References

[1] J. Rajesh *et al.*, "Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip," in *NOCs*, 2015, pp. 1–8.
[2] R. Fernandes *et al.*, "A security aware routing approach for NoC-based MPSoCs," in *SBCCI*, 2016, pp. 1–6.
[3] J. Sepúlveda, D. Flórez, and G. Gogniat, "Reconfigurable security architecture for disrupted protection zones in NoC-based MPSoCs," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 10th International Symposium on*, 2015, pp. 1–8.
[4] H. M. G. Wassel *et al.*, "Networks on Chip with Provable Security Properties," *IEEE Micro*, vol. 34, no. 3, pp. 57–68, May 2014.
[5] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-NoCs: Mitigating the threat of a compromised NoC," in *DAC*, 2014, pp. 1–6.
[6] M. M. Real *et al.*, "Dynamic spatially isolated secure zones for NoC-based many-core accelerators," in *ReCoSoC*, 2016, pp. 1–6.
[7] ARM, *ARM Security Technology Building a Secure System using TrustZone Technology*, 2008 (accessed June 19, 2017). [Online]. Available: http://infocenter.arm.com
[8] H. Isakovic and A. Wasicek, "Secure channels in an integrated MPSoC architecture," in *IECON*, 2013, pp. 4488–4493.
[9] 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.
[10] E. Wachter, L. Caimi, V. Fochi, D. Munhoz, and F. Moraes, "BrNoC: A broadcast NoC for control messages in many-core systems," *Microelectr. Journal*, vol. 68, pp. 69 – 77, 2017.
[11] L. Caimi, V. Fochi, E. Wachter, D. Munhoz, and F. Moraes, "Secure Admission and Execution of Applications in Many-core Systems," in *SBCCI*, 2017, pp. 65–71.