Secure Admission and Execution of Applications in Many-core Systems

Luciano L. Caimi
UFFS - Chapeco, Brazil
PUCRS - Porto Alegre, Brazil
lcaimi@uffs.edu.br

Vinicius Fochi
PUCRS - Porto Alegre, Brazil
vinicius.fochi@acad.pucrs.br

Eduardo Wachter
PUCRS - Porto Alegre, Brazil
eduardo.wachter@acad.pucrs.br

Daniel Munhoz
PUCRS - Porto Alegre, Brazil
daniel.munhoz@acad.pucrs.br

Fernando G. Moraes
PUCRS - Porto Alegre, Brazil
fernando.moraes@pucrs.br

ABSTRACT

Many-core architectures are similar to a computer network, where it is necessary to ensure the security during the execution of sensitive applications. This article discusses two security-related issues: the secure admission of applications and the prevention of resource sharing during their execution. The safe application admission is an open research subject for many-core systems. Although several methods are available for the Internet, computer networks, and software in general, low-cost computational proposals were not yet been proposed for many-core systems. Methods preventing resource sharing adopts firewalls, encryption mechanisms, and resource isolation to deal with the security threats. This paper proposes a protocol, executed at runtime, to tackle these issues. The application admission authenticates trusty entities. An entity authenticated might deploy applications, requiring only a MAC verification to guarantee the application integrity. Secure applications are mapped into continuous secure zones (*SZ*), with the reservation of *all* Processing Elements (PEs) and communication resources. All traffic flows that should cross the *SZ* are rerouted to the outside of the *SZ*. Such isolation approach avoids Deny-of-Service (DoS), timing, and spoofing attacks and guarantees confidentiality and integrity. The cost of the protocol is the latency required to start the secure applications. Results evaluate this latency, showing the effectiveness on adopting the proposed protocol to execute sensitive applications on many-core systems.

CCS CONCEPTS

•**Networks** →**Security protocols;**

KEYWORDS

Many-core systems, Security, Application Admission, Secure Zones

ACM Reference format:
Luciano L. Caimi, Vinicius Fochi, Eduardo Wachter, Daniel Munhoz, and Fernando G. Moraes. 2017. Secure Admission and Execution of Applications in Many-core Systems . In *Proceedings of SBCCI '17, Fortaleza - Ceará , Brazil, August 28-September 01, 2017,* 7 pages.
DOI: 10.1145/3109984.3110015

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBCCI '17, Fortaleza - Ceará , Brazil
© 2017 ACM. 978-1-4503-5106-5/17/08. . . $15.00
DOI: 10.1145/3109984.3110015

1 INTRODUCTION

The many-core architecture assumed in this work are NoC-based MPSoCs, due to their inherent scalability, providing massive parallelism and high performance to the users. These architectures contain processing elements (PEs), interconnected through routers. In such systems, several applications execute simultaneously, sharing computation (processors) and communication (routers) resources.

These MPSoCs are similar to a computer network, where it is necessary to ensure security during the execution of sensitive applications [13]. Two questions require an answer to guarantee the secure execution of applications: (*i*) how to receive from the MPSoC external environment an application and trust on its identity and integrity?; and (*ii*) how to execute an application without sensitive data leakage?

The application admission corresponds to the object code transfer from an off-chip entity to the MPSoC. Concerning security in this process, the MPSoC must trust on the entity transmitting the application and the integrity of the application must be verified to avoid the insertion of malicious code. Solutions to these issues exist for the Internet, computer networks, and software in general [11][7][12]. However, a low-cost protocol for application admission targeting MPSoCs is a gap observed in the literature.

At execution time, a malicious attacker may have access to sensitive computation or communication data. A secure application that processes sensitive data may have its security harmed by a malicious process. Examples of attacks on such systems include DoS, timing, spoofing, side-channel attacks and memory leakage [13][9][14].

The objective of the paper is to propose a protocol executed at runtime to tackle these issues: application admission and secure execution. The protocol authenticates trusty entities, by creating an entity key. The authenticated entities may deploy applications on the MPSoCs, with an attached MAC (Message Authentication Code), which ensures at the same time integrity and authentication. Reserving and isolating computation and communication resources inside a region of the MPSoC guarantees the secure execution. After the application execution, the memory is erased to avoid information leakage.

The original contributions are the lightweight methods adopted in the protocol. The authentication of trusty entities avoids the cost of key exchange protocols for each application admitted in the system since the key exchange occurs once for each entity. For each new application admitted in the system, only a MAC verification is required. The execution of the application inside a reserved region eliminates the need of firewalls and data encryption.

This paper is organized as follows. Section 2 presents proposals related to application admission and works related to the secure execution of applications in MPSoCs. Section 3 details the architecture model assumed in the current work. Section 4 presents the main contribution of this paper, the protocol for secure admission and execution of secure applications. Section 5 evaluates the protocol regarding the latency to start the secure applications. Section 6 concludes the paper and points-out direction for future works.

2 STATE-OF-THE-ART

The use of cryptographical schemes to register and authenticate users and equipment is widely used in many computing areas [11][1][16]. In the same way, the secure application admission (also called deploy in other areas) is proposed in [7][12]. However, to the author's knowledge, there is no proposal for secure admission of applications for MPSoCs.

Sepúlveda et al. [14] and Isakovic et al. [10] protect communication and computation, by adopting firewalls and cryptography. Sepúlveda et al. [14] adopt two NoCs: data NoC, used for the application data; service NoC used to exchange the security control packets. This proposal adopts *SZ*, defined at runtime, encrypting the packets. The *SZ* can be a discontinuous region inside the system. Packets from other applications may traverse the *SZ*, making the approach vulnerable to timing attacks. Firewalls ensure access control, authentication, and confidentiality. Isakovic et al. [10] propose an architectural partitioning of the MPSoC resources at design time. The Authors adopt security components like a *secure µkernel* and a *secure channel* infrastructure. The Authors propose to migrate the security functions from application components to the security components, instead of using *SZ*. To obtain a secure environment for applications, the Authors use physical separation of applications and secure channels to data exchange.

Hu et al. [9] and Fernandes et al. [5] protect communication. The computation is exposed to resource sharing. Both approaches are applied at design time. Hu et al. [9] proposes an approach to select the position of the firewalls: between a PE and a router or between routers. The goal of the Authors is to reduce the communication overhead required for security information in packets headers. Results show a 30% to 63% overhead reduction over a standard solution, i.e., firewalls connected to the NIs. Fernandes et al. [5] propose the creation of *SZ* based on the routing algorithm. The Authors use the Region-based Routing Algorithm (RBR) method to create *SZ* and the Segment-based Routing (SBR) to guarantee deadlock free paths. The Authors claim that the method mitigates DoS and timing attacks. Results show up to 16.56% overhead in the size of the routing tables over the baseline system.

Real et al. [13] and ARM TrustZone [2] protect computation. Both approaches are applied at run time. Real et al. [13] uses a clustered MPSoC. The Authors propose the creation of *SZ*, mapping the application to one or more clusters. If an application task needs to communicate with a task in another cluster, the message is sent through an insecure channel. Thus, computation is protected, but the communication is exposed to attacks. ARM processors provide the ARM TrustZone (ATZ) [2], a hardware support for the creation of Trusted Execution Environments (TEEs) and therefore the isolation of applications in the same processor. In multi-core architectures, applications running on different processors share resources such as the communication infrastructure and memory. Thereby, with TTE, applications running on different processors have its communication exposed to attacks.

Table 1 summarizes the characteristics of state-of-art proposals. The last row of the Table compares our proposal to the state-of-art. The proposal main novelties include: (*i*) secure admission of the applications; (*ii*) runtime protection of the computation and communication; (*iii*) due to the adoptions of continuous *SZ*, firewalls and traffic encryption is not required.

3 ARCHITECTURE OVERVIEW

Figure 1 presents the baseline architecture, a homogeneous NoC-based MPSoC, where each PE contains a 32-bit RISC processor, a DMNI module (a network interface with DMA capabilities), a local dual-port memory accessed by the processor and DMNI module, and a 192-bit pseudo-random number generator (PRNG). The software executing at each PE defines its role in the system. The system has manager PEs, global manager (GMP) and local managers (LMP); and PEs executing applications, slave processors (SP).

Two NoCs interconnects the PEs: *data* and *control* NoC. The data NoC transfers *data messages*, exchanged by applications. The data NoC adopts duplicated physical channels, wormhole packet switching, simultaneous support for distributed XY and source routing. The control NoC transfers the *control messages*. The control NoC has the following features (similar architecture to [15]): adoption of broadcast as the default transmission mode, bufferless router, each message has one flit. The control NoC router has a small area footprint, corresponding roughly to 20% of the data router.

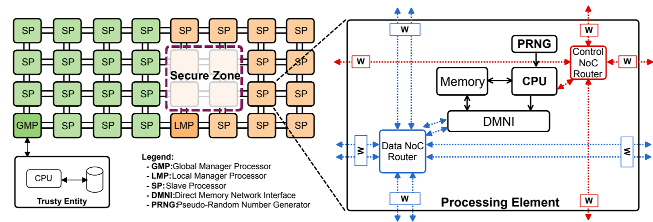

Figure 1: NoC-based MPSoC. Wrappers (W) are added to the control signals of NoCs links, enabling to isolate ports individually.

Definition 1. App_{sec}: application modelled as a task graph, with security constraints.

Definition 2. *SZ*: a Secure Zone is a continuous isolated area in the system, with a rectangular shape, with the SPs and routers of the *SZ* reserved to execute a single App_{sec}. The App_{sec} traffic is enclosed within the *SZ*, and any other traffic crossing the *SZ* is rerouted to the outside of the *SZ*.

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. 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 data NoC observes the status of the wrappers. A data message arriving in an activated wrapper is always discarded, and the control NoC returns to the source of the message a new broadcast reporting that the message needs retransmission.

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*

Table 1: Related works on secure execution of applications in MPSoCs.

Proposal	Protection		Method	Services	Design Time	Run Time
	Comput.	Comm.				
Sepulveda (2015)	Yes	Yes	Discontinuous *SZ*: cryptography and firewalls	Access control; Authentication; Confidentiality	-	Key exchange; cipher and decipher messages
Isakovic (2013)	Yes	Yes	Cryptography and firewalls	Access control; Authentication; Data integrity	Secure Kernel (key management and secure channel management) and secure tools (protocols, algorithms) provided	-
Hu (2015)	No	Yes	Firewall	Access control; Confidentiality	Application mapping; Firewall positioning and configuration	-
Fernandes (2016)	No	Yes	Discont. *SZ*: table-routing algorithm (RBR/SBR)	Timing attacks; DoS;	Task Mapping; Run RBR and SBR algorithms to calculate paths and router tables	-
Real (2016)	Yes	No	*SZ*: spatial and temporal isolation	Data integrity; Confidentiality inside the Cluster	-	Application mapping; *SZ* creation;
ARM (2008)	Yes	No	*SZ*: spatial isolation	Access control; Data integrity;	Application development using ATZ API	Switch to TTE mode and police execution;
Our Proposal	Yes	Yes	Secure Admission and Continuous *SZ*, without cryptography or firewalls	Authentication; Data integrity; Timing attacks; DoS; Spoofing.	-	Entity Authentication, Secure Admission, Key exchange, Shape Definition, *SZ* Creation

mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded.

The GMP manages the communication between the MPSoC and the external world. The MPSoC may execute applications with or without security constraints. In the latter case, *trusty entities* (Definition 3) transmit the secure applications. In this context, the MPSoC may be a node in an IOT network with higher computational power, and an entity is a set of nodes transmitting secure applications to the MPSoC.

Definition 3. *Trusty Entity*: an external entity to the MPSoC enabled to transmit App_{sec}s to be executed by the MPSoC.

4 SECURE ADMISSION AND EXECUTION PROTOCOL

This Section presents the *Secure Admission and Execution of Applications* protocol. The protocol has 7 phases: *(a)* system initialization; *(b)* registration of trusty entities running an Elliptic Curve Diffie-Hellmann (ECDH) key agreement protocol [8]; *(c)* application admission including shape definition and task mapping; *(d)* internal key exchange protocol; *(e)* tasks allocation and MAC verification [3]; *(f)* close the *SZ* and application execution; *(g)* at the end of application execution open the *SZ* and release the memory contents of SPs.

4.1 System Initialization

At system startup, the GMP computes and fills some structures to reduce the computation time of the phases executed at runtime. First, the GMP reads a random number from the PRNG module, sending it through a `polynomial_value` message to all SPs, using the control NoC. The PEs have a set of polynomials defined at design time, to be used by an LSFR. After receiving the `polynomial_value` message from the GMP, the SPs and the GMP select the same polynomial according to the generated random number. Encryptions or decryptions executed between PEs use the LFSRs configured using this process. Next, the GMP computes the set(P, p), used at the *entity registration* phase.

Definition 4. set(P, p): set of tuples with points over an elliptic curve, P, and 192-bit prime numbers, p. set(P, p) = $\{(P_0, p_0), (P_1, p_1), \ldots\}$.

4.2 Entity Registration and Key Agreement

As shown in Figure 1, external entities may deploy secure applications to execute in the MPSoC. The goal of this phase is to define the *entity key* (Definition 5). During the task allocation phase (Section 4.5), K_e will be used by the entity to compute the MAC of each task object code, and by the SPs to check the received MAC.

Definition 5. K_e: key generated by the ECDH key agreement protocol between an entity and the MPSoC.

The option for the use of ECDH is due to its better performance, considering the execution time, when compared to other systems with equivalent security levels. This protocol adopts 192-bit Elliptic Curves, which provides equivalent security levels to 2048-bit classical Diffie-Hellman [6].

The external entities may also be connected to the Internet and run secure protocols such https or TLS/SSL to guarantee the security of the received application. Thus, with the proposed protocol, the security between entities and the MPSoC is guaranteed, but the security between the external world and the entities must be ensured by the entities using appropriate protocols.

The entity registration phase runs the ECDH key agreement protocol using the 25519 curve[4]. Figure 2 shows the registration steps.

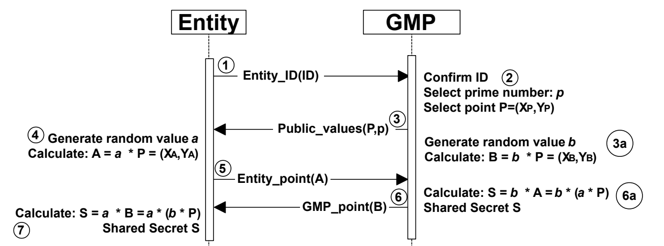

Figure 2: Entity registration with Diffie-Helmann Elliptic Curve.

The first action executed by an entity is to send its `entity_ID` to the MPSoC (1 in Figure 2). The MPSoC accepting the request (2) sends an element of the set(P, p) (Definition 4) to the entity (3).

Using the public values (P, p), the entity and the GMP compute new points, A and B, as follows:

Entity: $A = a \ * \ P \ (mod \ p)$

GMP: $B = b \ * \ P \ (mod \ p)$

where: a and b are random numbers, independently generated by the entity and the GMP at this step of the protocol (3a and 4).

The new points, A an B, are public exchanged (5 and 6). Now, each actor is able to compute a common secret point:

Entity: $S = a \ * \ B \ (mod \ p) \ = a \ * \ (b * P) \ (mod \ p)$

GMP: $S = b \ * \ A \ (mod \ p) \ = b \ * \ (a * P) \ (mod \ p)$

The computed S points are equal because in the EC domain commutative and associative properties are true. Thus, at the end of this phase (6a and 7), the entity and the GMP share a secret point S=(X_S,Y_S). The S point X-coordinate is the K_e. The GMP keeps the pair `(entity ID,` K_e`)` to use it in all future secure applications deployed by the entity, saving resource consumption and decreasing the latency to start the secure applications.

4.3 Application Admission

The application admission phase determines the location of the SZ and maps the App_{sec}'s tasks (Definition 1). Figure 3 presents the application admission steps. An entity requests to the GMP the execution of a new App_{sec}, transmitting the `entity_ID` and the App_{sec} graph (number of tasks, inter-task dependencies) - 1 in the Figure. The GMP selects the cluster with enough resources to execute App_{sec} (2), sending to the LMP of the selected cluster the App_{sec} graph and a tag identifying that the application is secure (3).

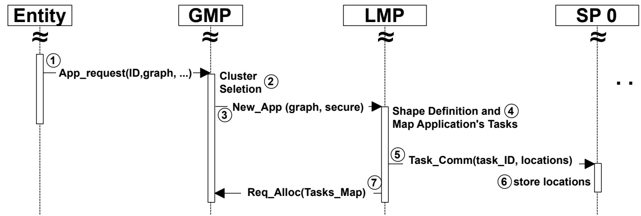

Figure 3: Application admission phase protocol.

Algorithm 1 presents the pseudo-code to create at runtime a SZ with a rectangular shape (Definition 2). Line 1 returns the *shape_set*, i.e., the rectangular regions that may execute App_{sec}. The function responsible for computing the *shape_set* considers the number of App_{sec}'s task, the cluster size, and the manager position. This function favors square shapes to reduce the number of hops between the communicating tasks.

If the *shape_set* is empty (line 2 to 3) the algorithm returns *FALSE*, meaning that the cluster does not have a rectangular shape to execute App_{sec}. In such case, the LMP runs a reclustering function, borrowing resources from neighbor clusters, to guarantee a shape with resources to execute App_{sec}. After the reclustering process, Algorithm 1 is re-executed.

If the cluster may receive App_{sec}, the algorithm creates the set *AR*, where each element of *AR* contains: (*i*) the number of SPs not executing tasks inside the shape (*nbFreeSPs*); (*ii*) the bottom left address of the SP inside the shape; (*iii*) the shape size. The loop starting at line 7 evaluates each shape of the *shape_set*. For each SP of the cluster, a sliding window search (SWS) returns the number of SPs not executing any task inside the shape (line 10), adding the tuple to the *AR* set (line 11). If the number of free SPs is equal to

Algorithm 1 Search resources to create the SZ

```
1:  shape_set = find_shapes(#tasks, cluster size, manager position)
2:  if shape_set = ∅ then
3:      return FALSE
4:  else
5:      // set Available Resources: tuples {nbFreeSPs, SP, shape}
6:      AR ← ∅
7:      for all shape in shape_set do
8:          for all SP of the cluster do
9:              // SWS = Sliding_Window_Search
10:             nbFreeSPs = SWS((SP.x,SP.y),(shape.Δx,shape.Δy))
11:             AR ← AR ∪ {nbFreeSPs, SP, shape}
12:             if nbFreeSPs = |shape| then
13:                 return SP, shape
14:             end if
15:         end for
16:     end for
17:     sort(AR, nbFreeSPs)
18:     for each SP in AR[0].shape do
19:         if task running in SP then
20:             migrate_tasks(SP)
21:         end if
22:     end for
23:     return AR[0].SP, AR[0].Shape
24: end if
```

the number of elements of the shape (|*shape*|) the function returns the SP address and the shape size (lines 12-14).

If all SPs of the cluster were visited with all possible shapes and there is no free region to execute App_{sec}, it is necessary to migrate tasks to obtain a SZ. Line 17 sorts the *AR* set according to the number of free SPs. After sorting, the first element of *AR* is the one with the largest number of free SPs. Lines 18 to 22 visit the SPs inside the shape, and if an SP has running tasks, the tasks are migrated to outside the shape boundary (line 20). Line 23 returns the coordinates of the bottom left SP of the SZ and the shape size.

After defining the SZ the LMP maps the tasks inside this region, considering as cost function the communication cost between the tasks (4). Note that mapping means definition of tuples {*task_ID, SP location*}, not the object code transfer.

Definition 6. Application's task map: set of tuples {*task_ID, SP location*} with the address of each App_{sec} task .

Steps 5 and 6 on Figure 3 corresponds to the transmission of the addresses of the communicating pairs to the SPs that will receive the tasks. At the end of this phase, the LMP request to the GMP the object code of the tasks (7).

4.4 Entity Key Exchange

This phase of the protocol transmits K_e to the SPs of the SZ, encrypted with a new key, K_m (internal MPSoC key). Each task receives a MAC to guarantee the integrity of the object code. The MAC is created using the SIPHASH algorithm [3] and K_e. Therefore, the SPs need K_e to compute the MAC attached to each task, and for security reasons K_e must remain in the GMP.

Figure 4 presents the entity key exchange steps. The LMP transmits the application's task map (Definition 6) at the last step of the previous phase to the GMP (`Req_Alloc` message, 1 in Figure 4). The GMP reads a random number from the PRNG (2), transmitting it to each SP inside the SZ using the data NoC. This step avoids

broadcast transmission to prevent the reception of the random number by other SPs (3). The GMP and the SPs create K_m also using the SIPHASH algorithm and the transmitted random number (4).

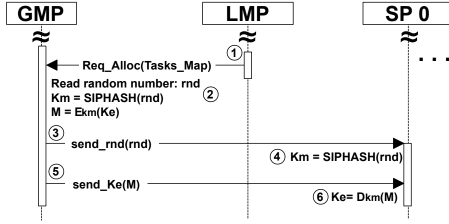

Figure 4: Entity key exchange.

The GMP uses K_m to encrypt K_e, using an LFSR. The resources initialization phase defines the polynomial of the LSFR (Section 4.1). After encrypting K_e, the GMP sends the encrypted value to each SP inside the *SZ* using the data NoC (5). Then, each SP decrypt the received message using K_m (6), obtaining K_e, to be used in the next protocol phase.

4.5 Tasks Allocation and MAC Verification

The goal of this phase of the protocol is to allocate the App_{sec}'s tasks in the SPs of the *SZ*, guaranteeing their integrity.

This phase starts with the GMP requesting to the external entity the object code of the App_{sec}'s tasks (1 in Figure 5). The entity generates a MAC for each task, using the SIPHASH algorithm and K_e. Then, the entity sends the task ID, the object code of the task and the MAC to the GMP (2). The GMP creates a `task_allocation` message to an SP according to the mapping received at the end of the Application Admission phase, sending the task identification (appID and taskID), the object code of task and the MAC (3). The SP stores the object code in the memory and computes the MAC also using the SIPHASH algorithm and K_e (obtained in phase 4.4) (4). The SP compares the received MAC with the computed MAC and sends the `task_allocated` message with the MAC comparison result to the LPM (5).

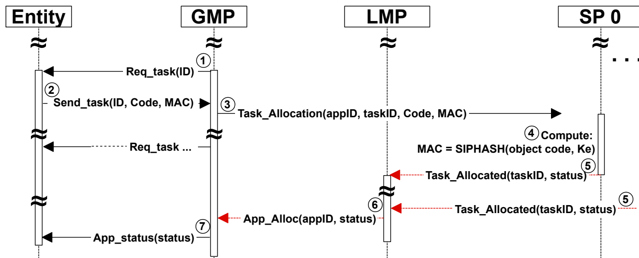

Figure 5: Tasks allocation and MAC verification phase. Red arrows: broadcast messages transmitted through the control NoC.

After mapping all tasks, the LMP notifies the GMP the status of the allocation phase (6). If all App_{sec}'s tasks were correctly received, the App_{sec} might start its execution. Otherwise, the process is interrupted, and the GMP notifies the external entity that the *App_sec* was corrupted during its allocation (7).

4.6 Closing the *SZ* and App_{sec} Execution

If the previous phase succeeded, the App_{sec} might execute. The goal of this phase is close the wrappers surrounding the *SZ* ("W" in Figure 1), and start the execution of the App_{sec}. Figure 6 shows the *SZ* closing and application execution steps.

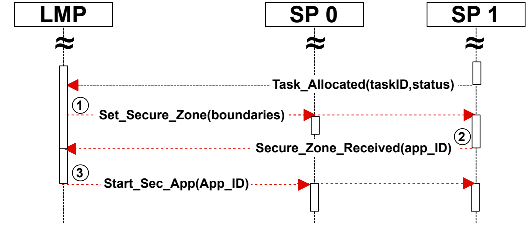

Figure 6: Close *SZ* phase.

This phase starts with the LMP `Set_Secure_Zone` broadcast message (1 in Figure 6) through the control NoC with the upper right and lower left corners of the *SZ*. All PEs receive this message, each one verifying if it is on the *SZ* boundary. If the SP is on the *SZ* boundary, the OS writes in a data structure the wrappers to be closed. The SP located at the upper right corner of the *SZ* transmits to the LMP a `Secure_Zone_Received` message (2). Once received this message, the LMP can start the execution of App_{sec}, by broadcasting a `Start_Sec_App` message (3), using the control NoC in *global* mode. This message enables the SPs to activate the wrappers to block incoming/outcoming traffic and releases the tasks belonging to App_{sec} to execute.

Now, the wrappers discard all messages that should traverse the *SZ*. The control NoC transmits to the source of the discarded messages (*restrict* mode) a retransmission request. The non-secure applications use the control NoC to find an alternative path to circumvent the *SZ* and retransmit the non-delivered messages.

4.7 Opening the *SZ* and Memory Clear

This phase of the protocol opens the *SZ* and cleans the memory contents of the SPs inside it to prevent any information leakage to be used by an attacker. Figure 7 shows the open *SZ* phase.

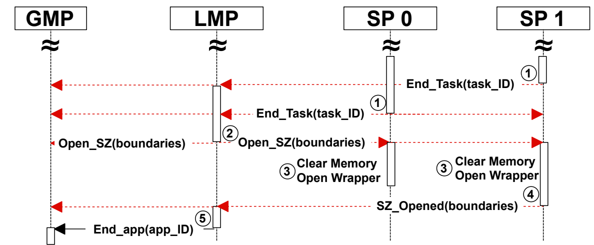

Figure 7: Open SZ phase of protocol.

When an App_{sec} task finishes its execution (1 in Figure 7), the SP sends an `End_Task` message with its task ID, using the control NoC in *global* mode. When all App_{sec} tasks finish their execution, the LMP transmits an `Open_SZ` message (2). All SPs inside the *SZ* clear their memory to prevent information leakage, erase K_m and K_e, and then open the wrappers (3), releasing the allocated resources. Then, the PE located at the upper right corner of the *SZ* transmits to the LMP an `SZ_Opened` message (4). Finally, the LMP clears the internal structures to release the cluster resources

previously allocated to App_{sec} and sends an End_app message to the GMP (5).

5 RESULTS AND DISCUSSIONS

The evaluation was conducted in a clock-cycle RTL SystemC description of the MPSoC presented in Section 3. This Section evaluates the protocol phases that impact in the latency to start the App_{sec}, i. e. the phases 4.3 to 4.6. The first two phases correspond to the system initialization and entity authentication, respectively. Both phases do not impact in the latency to start the applications. The last phase is responsible for releasing the application resources, occurring after the application execution.

The evaluation uses an 8x4 MPSoC, with two 4x4 clusters, as shown in Figure 1. The first result concerns the number of clock cycles required by the Application Admission phase (Section 4.3). Three scenarios were created, varying the *SZ* size: 2x2, 2x3 and 3x2. For each scenario, the already mapped tasks in the cluster induce different positions for the *SZ*.

Figures 8.a, 8.b and 8.c present the scenario for a 2x2 *SZ*. Each figure contains a set o previous mapped tasks, ({A,B}, {C,D}, {E,F}), making Algorithm 1 to select the *SZ* at the bottom-left, middle and top-right positions of the cluster, respectively.

Figure 8.d presents the scenario for a 2x3 *SZ*. The same method was applied to make Algorithm 1 select different positions inside the cluster. With tasks {A,B} previously mapped, the *SZ* location is located in the bottom-left position of the cluster (starting at address 5,0). With tasks {C,D} and {E,F} the initial *SZ* address is (4,1) and (6,1), respectively. Figure 8.e presents the scenario for a 3x2 *SZ*. With tasks {A,B}, {C,D,E,F}, {G,H} previously mapped, the *SZ* location is located at addresses (5,0), (5,1) and (5,2) respectively.

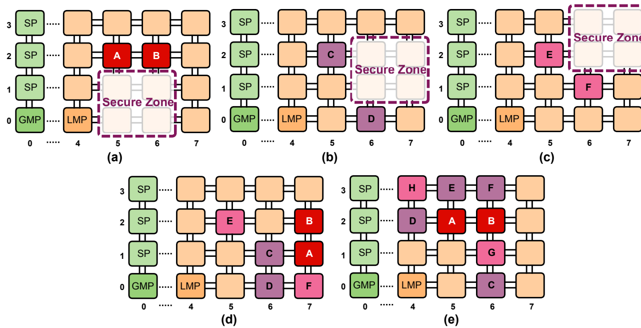

Figure 8: (a) 2x2 *SZ* at the bottom-left position of the cluster; (b) 2x2 *SZ* positioned at the middle of the cluster; (c) 2x2 *SZ* at the top-right position of the cluster; (d) previous mapped tasks for a 2x3 *SZ*; (e) previous mapped tasks for a 3x2 *SZ*.

Table 2 presents in the 1st and 2nd columns the number of App_{sec}'tasks and the *SZ* size, respectively. The 3rd column lists the previously mapped tasks in the cluster. The 4th column presents the bottom-left address of the *SZ*, computed according to Algorithm 1.

The 4th column of the Table corresponds to the number of clock cycles required to execute to Algorithm 1. The 5th column presents the number of clock cycles to run the task mapping. Both columns correspond to the step 4 of Figure 3. The last column of the Table corresponds to the Application Admission phase execution time.

This first set of results shows that the Application Admission phase is not time-consuming. Despite the exhaustive search made by Algorithm 1, the search space is small (few shapes to evaluate). The mapping algorithm is also fast because the search space is delimited by the *SZ*, with enough resources to map the tasks. Finally, the total time to execute this phase, including the cluster selection and the exchanged messages (steps 3, 5, 7) reached in the worst-case 15,384 clock cycles, or 153.3μs@100MHz. It is worth to mention that this performance may be penalized if tasks migrations (typically 40μs@100MHz for a 32 KB task) and reclustering are required (not evaluated in the current proof-of-concept implementation).

Table 2: Admission Phase evaluation scenarios and results (cc: clock cycles).

App_{sec} #Tasks	SZ size	Previous tasks	Shape from PE	Shape (cc)	Map (cc)	Total (cc)
3	2x2	(A,B)	5x0	946	5852	12154
		(C,D)	6x1	1191	6412	12954
		(E,F)	6x2	1370	6860	13584
5	2x3	(A,B)	5x0	1471	6736	13584
		(C,D)	4x1	1610	7216	14184
		(E,F)	6x1	1768	7536	14634
5	3x2	(A,B)	5x0	1983	6522	13854
		(C,D,E,F)	5x1	2042	7372	14896
		G,H)	5x2	2207	7802	15384

The second experiment evaluates the Entity Key exchange phase (Section 4.4). This phase of the protocol consumes, in average, 21,744 clock cycles. The dominant cost is the LSFR rotation time required to obtain the value used to encrypt (step 2 in Figure 4) and decrypt (6) K_e. The LSFR consumes 8,200 clock cycles at each side (GMP and SPs). The SIPHASH algorithm consumes 2,270 clock cycles.

The third experiment evaluates the Tasks Allocation and MAC Verification phase - TAMV (Section 4.5). Table 3 presents in the 1st column the task object code size (in KBytes), in the 2nd column the number of clock cycles required to compute and verify the MAC, and in the last column the total time of this protocol phase.

Table 3: Task Allocation and MAC verification evaluation results.

Object Code size (KB)	MAC step (cc)	Total (cc)
2	67,226	73,105
3	100,122	106,547
4	133,018	139,966
5	165,914	173,362
10	330,394	340,434

The Table shows that the number of clock cycles to compute and verify the MAC is a function of the task object code size, being equal to 32,896 cc/KB. The equation below presents the number of clock cycles to compute and verify the MAC.

$$TAMV_{phase} \approx 2{,}646 + (\textit{Object Code Size in KB} * 32{,}896)$$

where: 2,646 cc is the average value between steps 3 and 5 of Figure 5.

Compared to the other protocol phases, this step dominates the latency to start secure applications. The reason to explain this larger delay is due to the MAC computation, which operates on 64-bit

blocks, and the result of each block is used in the next block, thus being a sequential operation.

The last experiment evaluates the steps presented in Section 4.6, corresponding to the impact to close SZ in 4 scenarios, changing the SZ size from 2x2 up to 3x3 SPs. Table 4 presents the evaluation, considering the SZ size. The number of clock cycles to close an SZ starts when the LMP sends the `Set_Secure_Zone` message until the upper right PE in SZ effectively activates their wrappers after the `Start_Sec_App` message. The *Close Secure Zone* results present a small increase in the number of the clock cycles, proportional to the distance, in hops, to the upper right PE of the SZ.

Table 4: SZ close phase evaluation.

SZ Shape	2X2	2X3	3X2	3X3
Total (cc)	2,302	2,312	2,316	2,322

Figure 9 presents a timeline for the four phases impacting the latency to start an App_{sec} (the Figure does not respect a scale related to each phase). The key exchange and MAC verification phases are executed in parallel, for each SP with App_{sec} tasks. The delay to start each of these phases corresponds to send the encrypted K_e and the tasks object code, respectively.

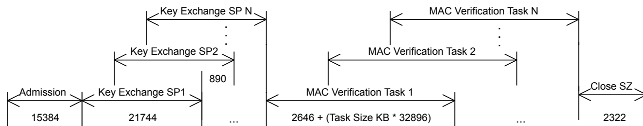

Figure 9: Ttimeline for the four phases impacting the latency to start an App_{sec}.

6 CONCLUSION

This work presented a protocol executed at runtime for secure application admission and execution. The proposal includes: the authentication of entities that deploy the application; the secure admission using a MAC ensuring integrity and authentication; the secure execution through isolation of computational and communication resources inside a Secure Zone, and; the release of previously allocated resources avoiding information leakage. Results show a low impact on the latency to start a secure application. The dominant operation that increases the latency concerns to MAC calculation and verification steps. Adding the delay of each phase, the secure application is delayed for less than 4 ms (@100MHz) for 10KB tasks.

Future works include: (*i*) improve the Task Allocation and MAC verification phase making parallel the reception of the task object code with the MAC calculation; (*ii*) evaluate the cost reduction obtained with the initialization phase; and (*iii*) evaluate the cost of task migration and recluster in the Shape Definition and Mapping phase.

7 ACKNOWLEDGEMENT

The Author Fernando Gehm Moraes is supported by CNPq funding agency.

REFERENCES

[1] A. M. Allam, I. I. Ibrahim, I. A. Ali, and A. E. H. Elsawy. 2003. Efficient zero-knowledge identification scheme with secret key exchange. In *MWSCAS*, Vol. 1. 516–519. https://doi.org/10.1109/MWSCAS.2003.1562331
[2] ARM. 2008. ARM Security Technology Building a Secure System using TrustZone Technology. In *ARM*. http://infocenter.arm.com
[3] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. *SipHash: A Fast Short-Input PRF*. Springer Berlin Heidelberg, Berlin, Heidelberg, 489–508. https://doi.org/10.1007/978-3-642-34931-7_28
[4] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *In Public Key Cryptography (PKC), Springer-Verlag LNCS 3958.*
[5] R. Fernandes et al. 2016. A security aware routing approach for NoC-based MPSoCs. In *SBCCI*. 1–6. https://doi.org/10.1109/SBCCI.2016.7724054
[6] V. Gupta, S. Gupta, S. Chang, and D.s Stebila. 2002. Performance Analysis of Elliptic Curve Cryptography for SSL. In *WiSE*. 87–94. https://doi.org/10.1145/570681.570691
[7] O. Hanka and H. Wippel. 2011. Secure deployment of application-tailored protocols in future networks. In *International Conference on the Network of the Future*. 10–14. https://doi.org/10.1109/NOF.2011.6126668
[8] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. 2003. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York,, Secaucus, NJ, USA.
[9] Y. Hu et al. 2015. Automatic ILP-based Firewall Insertion for Secure Application-Specific Networks-on-Chip. In *INA-OCMC*. 4. https://doi.org/10.1109/INA-OCMC.2015.9
[10] H. Isakovic and A. Wasicek. 2013. Secure channels in an integrated MPSoC architecture. In *IECON*. 4488–4493. https://doi.org/10.1109/IECON.2013.6699858
[11] M.; Chaudet C. Khernane, N.; Potop-Butucaru. 2016. BANZKP: a Secure Authentication Scheme Using Zero Knowledge Proof for WBANs. *CoRR* abs/1602.00895 (2016).
[12] C. Kuntze N.; Rudolph. 2013. Secure deployment of SmartGrid equipment. In *IEEE Power Energy Society General Meeting*. 1–5. https://doi.org/10.1109/PESMG.2013.6672120
[13] M. M. Real et al. 2016. Dynamic spatially isolated secure zones for NoC-based many-core accelerators. In *ReCoSoC*. 1–6. https://doi.org/10.1109/ReCoSoC.2016.7533900
[14] J. Sepúlveda et al. 2015. Reconfigurable security architecture for disrupted protection zones in NoC-based MPSoCs. In *ReCoSoC*. 1–8. https://doi.org/10.1109/ReCoSoC.2015.7238098
[15] E. Wachter et al. 2013. Topology-Agnostic fault-tolerant NoC routing method. In *DATE*. 1595–1600. https://doi.org/10.7873/DATE.2013.324
[16] L. Zhimeng and Z. Yanli. 2016. Provable Secure Node Authentication Protocol for Wireless Sensor Networks. In *WISA*. 221–224. https://doi.org/10.1109/WISA.2016.51