# Secure Admission of Applications in Many-cores

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

***Abstract*—The adoption of many-cores systems makes the concern for data protection a critical design requirement. A secure application that processes sensitive data may have its security harmed by a malicious process. The literature contains several proposals to protect many-cores against attacks, focusing for example in the protection of the application execution or the access to shared memories. However, a complete solution, covering the entire application lifetime is a gap to be fulfilled. Designers must follow these steps to protect the execution of an application: provide a secure admission process; give to the application a protected environment to execute; protect the access to memory and peripherals. The article main goal is to cover the secure admission by using mutual authentication, and Message Authentication Code (MAC) to guarantee the integrity of applications during the object code transfer. This work also presents the protocol to cover the other steps, proposing a complete framework to execute applications with security constraints in many-cores. The evaluation shows that the mutual authentication in a phase previous to the object code transmission, minimize the latency to start a secure application and there is a latency dependency relative to the MAC verification with the object code size. After the MAC verification, the application starts its execution in a secure environment.**

***Index Terms*—Many-core, Network on Chip, Mutual Authentication, Security, Secure Zones.**

## I. Introduction and Related Works

Many-core systems are similar to a computer network, where it is necessary to ensure security during the execution of applications processing sensitive data. At runtime, 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 [1]–[3].

The execution of a secure application ($App_{sec}$) comprises at least three assumptions. The first one is the secure admission of the application, to guarantee the object code integrity. The second assumption regards the application execution in an environment protected from attacks. The third assumption is related to the 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. Concerning security in this process, each actor (external entity and many-core) must 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 in general [4]–[6], but not yet applied to many-cores.

Techniques to execute applications in a protected environment include firewalls, encryption mechanisms, and resource isolation through secure zones – $SZ$s. In the scope of this work, $SZ$ is defined as a continuous isolated area in the system, with a rectilinear shape, with the processing elements and routers of the $SZ$ reserved to execute a single $App_{sec}$.

The protection of the communication with peripherals and shared memories avoids unauthorized access to instructions and data, which may compromise the application execution, due to the information tampering or leakage [7], [8].

Most solutions related to security applied to NoC-based many-cores consider only one of the three above mentioned assumptions, being limited to the application execution or the shared memory access mechanism. Table I summarizes related works, presenting in the second column if the work is applied the many-cores and in the remaining columns the protected phases. Note that our proposal stands-out from related works because it covers all steps required to execute an application with security constraints.

**TABLE I:** Related works.

| Work | Applied for many-cores | Admission | Execution | IO Access |
|---|---|---|---|---|
| Khernane et al. [4] | | ✔ | | |
| Zhimeng et al. [5] | | ✔ | | |
| He et al. [6] | | ✔ | | |
| Real et al. [1] | ✔ | | ✔ | |
| Sepulveda et al. [2] | ✔ | | ✔ | |
| Rajesh et al. [3] | ✔ | | ✔ | |
| Reinbrecht et al. [7] | ✔ | | | ✔ |
| Grammatikakis et al. [8] | ✔ | | | ✔ |
| Our Work | ✔ | ✔ | ✔ | ✔ |

The secure framework adopts the following methods. The *secure admission* uses Diffie-Hellman with Elliptic Curves to mutual authentication of communicating parts, and the computation of a Message Authentication Code (MAC) appended at each application task to provide the integrity of the object code during the transfer from an external entity to the many-core. The *secure execution* takes heuristics executed at runtime to create $SZ$s, promoting the exclusive use of computation and communication resources inside the region, and when necessary, reroute the traffic that should traverse an $SZ$ [9]. The protection of the communication with *peripherals* and shared memories uses a master-slave mechanism with private key cryptography [10].

Our *goal* is to propose and evaluate a lightweight mechanism to guarantee the secure admission of applications into many-cores from an external entity. The proposed protocol promote the mutual authentication between external entities and the many-core. The authenticated entities might deploy applications on the many-core, with an attached MAC, which ensures at the same time integrity and authentication of object code.

The reader might ask: "why the concern about the admission phase, if methods are available for distributed systems and the Internet?". The answer is the need to evaluate and choose methods that can be applied intra-chip, at a low cost. Which authentication method is safe, has a small area and memory requirements, and has reduced latency to start the applications? How to check the integrity of applications in a many-core before starting the application execution? The answer to these questions is the *contribution* of this work, showing that lightweight methods can be adopted in many-cores to authenticate secure applications.

## II. Architecture Overview

Figure 1 presents the baseline architecture, a homogeneous NoC-based many-core. Each Processing Element (PE) contains a processor, a DMNI module (a network interface with DMA capabilities), a local dual-port memory, and a 256-bit pseudo-random number generator (PRNG). The software executing at each PE defines its role in the system. The system has a hierarchical architecture organized in clusters, with two types of PEs: (*i*) managers PEs responsible by cluster control and management (application mapping and allocation, cluster resources control, secure zone definition); (*ii*) PEs executing applications' tasks, the slave processors (SP). There are two distinct manager PEs, global manager (GMP) and local managers (LMP). The functions of GMP and LMP are identical except that the GMP communicate with external entities to authenticate each other and perform the internal key management.

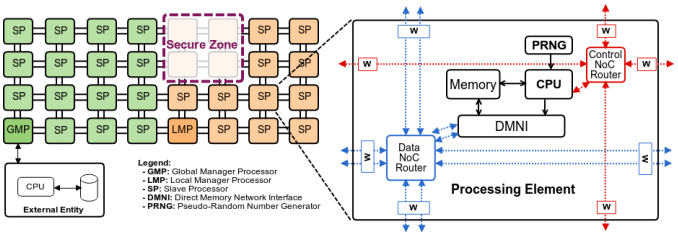


**Fig. 1:** NoC-based many-core. Wrappers (W) are added to the control signals of NoCs links, enabling to isolate ports individually.

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 mode, and a small area footprint, corresponding roughly to 20% of the data router [11].

Both NoCs contain test wrappers, or simply *wrappers*, in the flow control signals. The function of the *wrappers* is to block a given PE port to create the $SZ$s. 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 GMP manages the communication between the many-core and the external world. The many-core may execute applications with or without security constraints. In the latter case, an *authenticated external entity* ($Entity$) transmits the secure applications. The $Entity$ is an external entity to the many-core previously authenticated and enabled to transmit $App_{sec}$s to be executed by the many-core.

## III. Mutual Authentication for Many-cores

### A. System setup phase

At the setup phase, both $Entity$ ($E_i$) and the many-core ($M$) initially compute a key pair suitable for elliptic curve cryptography, step 1 in Figure 2. The key pair {Pk, PuK} consisting in a private key (Pk) and a public key (PuK), represented by a random multiplier number and a point over an Elliptic Curve, respectively. Next, each one publishes their respective ID and PuK (step 2). This phase ends with each part loading the pair {PuK, ID} of each other (step 3), using a secure channel.

A secure channel to publish and load the public keys may avoid man in the middle and hijacking attacks. This process can be done using mechanisms like https, TLS/SSL or VPNs. However, the protocol deployed in the next section ensures that even if the public keys use unsecured channels, the mutual authentication phase guarantees that only the legitimate actors involved in the process will correctly exchange data.

### B. Mutual Authentication phase

As shown in Figure 1, external entities may deploy secure applications to execute in the many-core. This phase provides mutual authentication between an external entity and the many-core.

The mutual authentication uses the ECDH protocol. 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 256-bit Elliptic Curves, which provides equivalent security levels higher than 2048-bit classical Diffie-Hellman [12].

If an entity wants to send an $App_{sec}$ to run in the many-core, it must first execute the mutual authentication phase, before sending the application code.

Initially, the entity sends a request message encrypted by the many-core public key ($PuK_m$) loaded at the setup phase, step 1 in Figure 3. The request message contains the entity identifier and a random number, i.e., the pair $\{ID_i, nonce_i\}$. Note that any entity, even in an attempt to insert a malicious application, may send this message because the many-core has a published public key.

The GMP decrypts the request message using its private key ($PK_m$), verifying the received $ID_i$. If $ID_i$ is valid, the many-core sends a reply message encrypted by the Entity public key ($PuK_i$) – step 2. This message contains 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. Otherwise, if the entity $ID_i$ is not found, the reply message is encrypted using the many-core public key ($PuK_m$) since there is no public key associated with the received $ID_i$. The reply message is always sent to avoid information leakage.

At the $Entity$ side, if the received $nonce_i$ does not match, this means that the connection with the many-core is compromised because other actor tried to forge the many-core ID. A correct received $nonce_i$ corresponds to a correct authentication of the many-core, once just the lawful pair $\{PuK_m, PK_m\}$ can encrypt/decrypt the initial request message.

After verifying the correct reception of $nonce_i$, the Entity seeks the $PuK_m$ according to the received $ID_m$. The Entity encrypts the received $nonce_m$ using $PuK_m$, and send this data to the many-core – step 3.

At the many-core side, the received $nonce_m$ is decrypted using $PK_m$. A correct $nonce_m$ authenticates the Entity, once just the lawful pair $\{PuK_i, PK_i\}$ can encrypt/decrypt the message with the $nonce_m$. This way, if the $nonce_m$ and the $ID_i$ (received on first request message) are correct, the many-core sends an accept message to the Entity. Otherwise, a reject message is sent – step 4.

Once finished the mutual authentication steps the entity generates a *session* key $K_e$, used during the task allocation phase. The entity uses this key to compute a Message Authentication

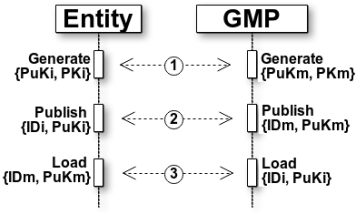


**Fig. 2:** Sequence diagram of Setup phase of protocol.

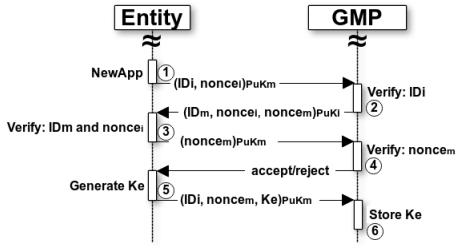


**Fig. 3:** Sequence diagram of Mutual Authentication phase of protocol.

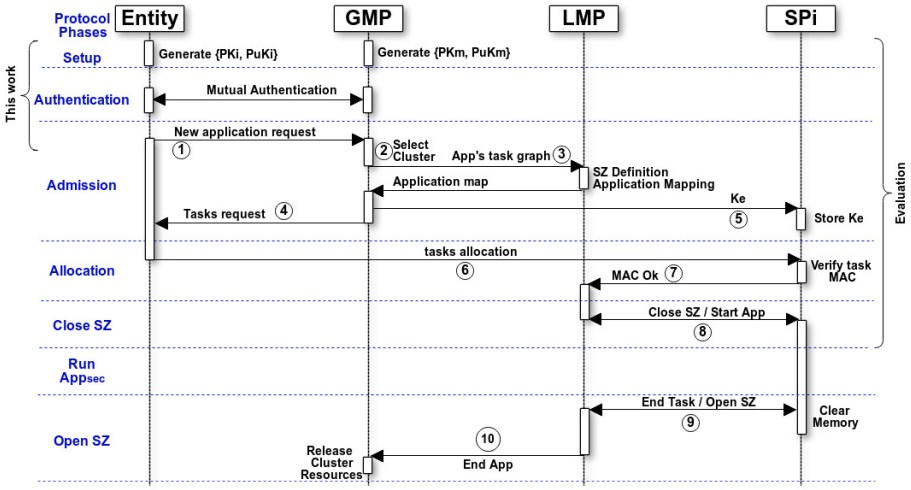


**Fig. 4:** General view of the protocol to execute an $App_{sec}$.

Code (MAC) appended at the end of each task object code. The many-core uses this key to verify the integrity of each task deployed into the system.

The entity sends the tuple ($ID_i$, $nonce_m$, $K_e$) encrypted by the many-core public key ($PuK_m$), where $ID_i$ and $nonce_m$ are used to authenticate the message – step 5.

The GMP keeps the pair $\{ID_i, K_e\}$ to use it in future secure applications deployed by the entity, saving resource consumption and decreasing the latency to start the secure application – step 6.

### C. General view of the protocol

Figure 4 presents a general view of all phases of the protocol. The names of the phases are presented on the left side of the Figure, in blue color. The phases presented in previous sections, the focus of this paper, are used to ensure the mutual authentication of the actors involved in the communication avoiding execution of applications from malicious sources. The result of the previous sections is the session key, $K_e$.

The Admission phase has three goals: (*i*) define the secure zone ($SZ$) location; (*ii*) ask to the Entity the tasks' codes; (*iii*) transmit to the SPs that will receive the task's codes the $K_e$ to verify the MAC (1 to 5 in Figure 4).

The Entity uses $K_e$ to compute MAC using the SIPHASH algorithm [13], during the allocation phase of the protocol. After receiving the destination of $App_{sec}$'s tasks, the Entity uses $K_e$ to generate a MAC to each task's object code. Thus, the allocation sends the task's object code with the respective MAC appended – 6. The SP receives the task's object code and, using the same $K_e$, compute the MAC locally. If the calculated MAC match with the received MAC the object code integrity is verified.

If the previous phase succeeded for all tasks, $App_{sec}$ might execute – 7. The Close $SZ$ phase proceeds by activating the wrappers surrounding the $SZ$ ("W" in Figure 1) and start the execution of the $App_{sec}$ – 8. The wrappers discard all messages that should traverse the $SZ$. The control NoC transmits to the source of the discarded messages 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 [9] using a rerouting mechanism.

The Open $SZ$ phase cleans the memory contents of the SPs inside it to prevent any information leakage to be used by an attacker – 9. Also, the SP's Operation System erase the value of $K_e$ key and release the wrapper opening the $SZ$. Finally, the LMP clears the internal structures to release the cluster resources previously allocated to $App_{sec}$ and sends a message to the GMP that release its internal structures about the cluster resources – 10.

## IV. Results and Discussions

This Section evaluates the first two phases of the protocol initially, corresponding to the system setup (Section III-A) and mutual authentication (Section III-B). Both phases do not impact directly in the latency to start a secure application. A third evaluation measures the time spent in the admission, allocation, and cloze SZ (Section III-C) phases, that impact directly in the latency to start a secure application.

The first experiment measures the cost of the Setup and Mutual Authentications phases. The hardware architecture of the External Entity and GMP was described using OVPSim APIs [14], an accurate instruction simulator. The cryptographic functions of this work use the TweetNaCl [15], a compact and self-contained public-domain C library. The library uses Curve25519 on the EDCH key exchange, the base of our proposed mutual authentication.

**TABLE II:** Setup and authentication phases evaluation.

| Phase | Step | Actions | MIPS # Instructions (x$10^6$) | ARM # Instructions (x$10^6$) |
|---|---|---|---|---|
| Setup | 1 | generate Key pair | 239.2 | 83.9 |
| Authent. | 1 | generate $nounce_i$<br>encrypt request | 224.4 | 84.1 |
| | 2 | decrypt request<br>generate $nounce_m$<br>encrypt reply | 449.0 | 168.3 |
| | 3 | decrypt reply<br>encrypt $nounce_m$ | 449.0 | 168.3 |
| | 4 | decrypt $nounce_m$<br>encrypt A/R | 449.0 | 168.3 |
| | 5 | decrypt A/R<br>generate Ke<br>encrypt Ke | 449.0 | 168.3 |
| | 6 | decrypt Ke | 224.5 | 84.1 |
| **TOTAL** | | **Setup + Authentication** | **2245.0** | **845.3** |

Table II shows the two first phases of the protocol ($1^{st}$ col.), the steps of the protocol ($2^{nd}$ col.), the executed actions ($3^{rd}$ col.), and the number of instructions of these actions in the MIPS-like OR1k ($4^{th}$ col.) and ARM A9 (with SIMD) processors ($5^{th}$ col.). The three main actions, generate a key pair, encrypt data and decrypt data, requires a similar number of instructions: 239.2M, 224.4M and 224.5M – MIPS processor, and 83.9M, 84.1M and 84.1M – ARM processor. The total amount of instructions of the two first phases of the protocol are 2,245.0M and 845.3M instructions, in processors MIPS and ARM, respectively. Considering the processors running at 500 MHz with a CPI=1, the time consumption relative to the mutual authentication of the External Entity and the many-core is 4.49s for MIPS and 1.68s for the ARM.

Although this overhead is relatively significant, in the order of seconds, only one execution of the mutual authentication process occurs for each entity to ensure the authenticity of the parts and does not impact the latency to start the applications. The cost to compute the session key, $K_e$, corresponds to step 5 in Table II, being this cost smaller than 1 second. Thus, it is possible to consider to change this key periodically to increase the security of the system. Remember that the transmission of this key to the PEs occurs by using a lightweight cryptography method, and its repeated use by different applications could be an opportunity for attacks.

The second experiment evaluates the protocol phases that impact in the latency to start $App_{sec}$, i. e. the phases *Admission* to *Close SZ*. The evaluation uses a clock-cycle RTL SystemC description of the many-core presented in Section II.

The evaluation uses an 8x4 many-core, with two 4x4 clusters, and positioning a 2x2 $SZ$ as shown in Figure 1. The $App_{sec}$ have fours tasks with sizes from 1.3KB to 9.6KB.

Table III presents the protocol phase in the $1^{st}$ col., the actions of each phase in the $2^{nd}$ col., and the number of clock cycles to execute each action. The Admission and Close SZ phases present a low impact in the latency. The dominant operation that increases the latency to start the Applications concerns the MAC computation and verification. The number of clock cycles is a function of the task object code size because the MAC computation is a sequential operation, where each block computation (256 bits of the object code) is used for the next block.

**TABLE III:** Admis., Alloc. and Close SZ phases evaluation.

| Phase | Actions | Clock Cycles (x$10^3$) |
|---|---|---|
| Admission | SZ Definition & App mapping | 6.56 |
| | $K_e$ Distribution | 21.70 |
| Allocation | Task1: 9.6 KB | 318.44 |
| | Task2: 5.4 KB | 186.84 |
| | Task3: 4.4 KB | 147.36 |
| | Task4: 1.3 KB | 45.37 |
| Close SZ | Wrappers activation | 2.30 |
| **TOTAL** | **Adm. + Alloc. + Close SZ** | 728.57 |

Results show a low impact on the latency to start an $App_{sec}$. Adding the delay of each phase, the secure application is delayed for 728.57K clock cycles, less than 1.5 ms (@500MHz) considering the sum of three phases, with the sequential MAC verification of four tasks in allocation phase.

This result takes account the sum of MAC computation and verification at distinct SPs executed sequentially, although there is a parallelism degree in this stage to be evaluated in future works.

## V. Conclusion

The *original* contribution of the paper is the authentication method of entities that deploy applications on the many-cores. To the Authors' knowledge, there is no similar proposal for many-cores. The paper details a setup and a mutual authentication phases, and evaluates its costs for two processors architectures, being in the order of few seconds. This cost is acceptable because only trusty applications run in the many-core, reducing the probability of attacks from malicious applications.

Concerning the latency to start a secure application, the operation that increases the latency is the MAC computation and verification steps. The application is delayed by a few milliseconds to start. Once executing, the application has computation and communication resources reserved in a Secure Zone. Thus, there is no execution time overhead because it is not necessary to encrypt the application messages.

Future works include: (*i*) evaluate the impact in size ordering of task's allocation to take advantage of parallelism in MAC computation and verification on distinct SPs; and (*ii*) implements and evaluate a new peripheral access mechanism that combines the routing algorithm, the tasks mapping, and a lightweight cryptographic scheme.

## VI. Acknowledgement

Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (302531/2016-5), Brazilian funding agencies. Luciano L. Caimi and Vinicius Fochi are supported by CAPES (184993/2018-00, 184851/2018-00).

## References

[1] M. M. Real, P. Wehner, V. Migliore, V. Lapotre, D. Göhringert, and G. Gogniat, "Dynamic spatially isolated secure zones for NoC-based many-core accelerators," in *ReCoSoC*, 2016, pp. 1–6.

[2] J. Sepulveda, D. Flórez, R. Fernandes, C. Marcon, G. Gogniat, and G. Sigl, "Reconfigurable security architecture for disrupted protection zones in NoC-based MPSoCs," in *ReCoSoC*, 2015, pp. 1–8.

[3] J. Rajesh, D. M. Ancajas, K. Chakraborty, and S. Roy, "Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip," in *NOCS*, 2015, pp. 1–8.

[4] N. Khernane, M. Potop-Butucaru, and C. Chaudet, "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] C. Reinbrecht, A. Susin, L. Bossuet, and J. Sepúlveda, "Gossip NoC - Avoiding Timing Side-Channel Attacks through Traffic Management," in *ISVLSI*, 2016, pp. 601–606.

[8] M. D. Grammatikakis, P. Petrakis, A. Papagrigoriou, G. Kornaros, and M. Coppola, "High-level security services based on a hardware NoC Firewall module," in *WISES*, 2015, pp. 73–78.

[9] L. Caimi, V. Fochi, E. Wachter, D. Munhoz, and F. G. Moraes, "Activation of Secure Zones in Many-core Systems with Dynamic Rerouting," in *ISCAS*, 2017, pp. 144–147.

[10] B. Oliveira, R. Reusch, H. Medina, and F. G. Moraes, "Evaluating the Cost to Cipher the NoC Communication," in *LASCAS*, 2018, pp. 1–4.

[11] E. Wachter, 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.

[12] V. Gupta, S. Gupta, S. Chang, and D. Stebila, "Performance Analysis of Elliptic Curve Cryptography for SSL," in *WiSe*, 2002, pp. 87–94.

[13] J.-P. Aumasson and D. J. Bernstein, "A Fast Short-Input PRF," in *INDOCRYPT*, 2012, pp. 489–508.

[14] OVP, "Open virtual platform," 2018. [Online]. Available: http://www.ovpworld.org/technology_ovpsim

[15] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A Crypto Library in 100 Tweets," in *LATINCRYPT*, 2015, pp. 64–8.