A mixed criticality approach for the security of critical flows in a Network-on-Chip

Ermis Papastefanakis[†‡], Xiaoting Li[*], Laurent George[‡]
[*]ECE Paris, 75015 Paris, France
[†]Thales Communications and Security, 92622 Gennevilliers, France
[‡]Université Paris-Est, LIGM / ESIEE, Champs sur Marne, France
Email: ermis.papastefanakis@thalesgroup.com, xiaoting.li@ece.fr, laurent.george@univ-mlv.fr

***Abstract*—In this paper, we explore a new approach for the security of flow transmission in a Network-on-Chip. This approach is based on a dual mixed criticality approach where the criticality of a flow (HI or LO) is associated to timeliness and security constraints on flow transmission. Flow of HI criticality should be protected from Denial of Service (DoS) and side channel attacks while LO flows are assumed to be non secured. We propose a solution to detect abnormal flow profiles in a Network-on-Chip, that can result from a security attack and we provide a mixed criticality approach to mitigate the impact of an attack. This is done by switching the nodes of a Network-on-Chip subject to a security attack from LO to HI criticality and taking protection measures in HI mode to contain the security attack. Our solution relies on a mechanism, configured and managed by a hypervisor applied in all IPs connected to the Network-on-Chip.**

***Keywords*—Network-on-chip, security, safety, mixed criticality.**

I. Introduction

Embedded architectures gain momentum as they advance to nanoscale technologies that will allow them to include hundreds of cores (manycores) while maintaining a competitive performance to power ratio. Apart from the standard General Purpose Processors (GPPs) and Graphics Processing Unit (GPU), new architectures demonstrate a high degree of heterogeneity by integrating Intellectual Property (IP) elements such as Digital Signal Processors (DSPs), dedicated accelerator cores and Field Programmable Gate Arrays (FPGAs) fabric. The high number of cores and their diversity provide an increase in processing power and allows handling larger, more complex computation loads and greater information volume. As a result these architectures unlock new sets of possibilities, being present in domains such as Internet of Things (IoT) infrastructures (gateways), Cyber-Physical Systems (CPSs) , where they play important roles in systems that interact heavily with the physical world. Security as well as safety in such domains is taken more and more under consideration and will inevitably become mandatory through standardization and evaluation or certification schemes.

When considering the evaluation of a system, a certain level of security is targeted. Protection profiles is a term introduced by Common Criteria [1] for a document that specifies security requirements (security functionalities, assets to be protected, threats, attacking agents, etc.). They are specific for a category of systems and their definition depends on the system's function, application domain, criticality etc. As a result security schemes can be restrictive or permissive according to the system's protection profiles with the objective to reflect the level of evaluation required. Security features can be implemented in various levels starting from hardware and reaching up to the higher levels of software. They are designed so that a security feature in one level exposes a configuration interface to the higher level. That way, a solution can be adapted to best serve the system's security requirements. Consequently when conceiving security implementations it is beneficial to include assumptions that have been identified in protection profiles. This way an implementation can possess qualities making it a suitable candidate for evaluation.

Manycore embedded systems increasingly rely on the Network-on-Chip (NoC) paradigm that allows them to overcome the scalability limitations posed by buses at the interconnect level [2]. The idea behind them is to implement routing principles taken from the computer networks inside the System-on-Chip (SoC)'s ecosystem. This involves adapting them to fit the constraints that SoC design imposes (chip surface, power consumption, clock distribution). Their advantage lies in the fact that as the number of IP elements increases, a NoC becomes more efficient in aspects such as throughput, Quality of Service (QoS), IP compatibility, ...

The NoC can be a powerful link in the chain of elements and partially shape the SoC's potential performance. It is a key component in the SoC's communications and has a great impact on guarantying the timing determinism and throughput. NoC have a modular and distributed structure and has capacities in reconfiguration and redundancy. This makes NoC a good candidate to implement security mechanisms to grant that its performance cannot be compromised due to security attacks. In manycore platforms an important component is the resource manager which can be responsible for task placement, resource monitoring, redundancy, reconfiguration, statistics, etc. This role is often assigned to a hypervisor which is a low-level software block that is necessary in manycores and is in charge of managing the different operating system instances executing on the platform. We consider that enhancing security in these two elements (NoC, hypervisor) can have an impact on the real-time properties of a system. Security is mostly treated as a standalone subject, we consider that there is an interest to put its impact in perspective with other aspects such as timeliness of the communications in the NoC.

In existing research, work has been done to address different aspects of security in NoC-based architectures. In [3] the authors propose a mechanism to protect against packet injection and information extraction through filters between secure and non-secure areas in the NoC applied at routing level. In [4] a framework is presented implementing the prin-

ciples of key exchange and encrypted communications between IP elements. In [5] and [6], solutions are proposed to prevent illegal memory accesses and packet injection by implementing firewall-like wrappers to the IP elements. Finally in [7] a mechanism against DoS and side channel attacks is presented based on the principle that giving an attacker priority will make it impossible to exploit resource sharing to extract information from communication channels. At the same time setting a static threshold on channel utilization will prevent the attacker from exploiting the priority given in order to perform DoS attacks. From a QoS perspective Kalray's MPPA architecture [8] provides a mechanism to locally manage traffic quotas that can potentially be used for applying security mechanisms similar to the one proposed in this paper. Also in [9] and [10] the authors propose protocols that guarantee timing in mixed-criticality context. The difference of our approach is that the attack is mitigated at the node subject to a security attack occurs, without broadcasting a criticality switch to other nodes, changing HI/LO modes locally at the node(s) instead of globally in the whole SoC.

From the different types of threats our scope is DoS and side channel attacks. In a DoS scenario the goal is to occupy the maximum amount of shared resources, having for objective to render the system (all or parts of it) inoperable. In a similar scenario where a side channel attack is performed, the goal is to extract information indirectly by probing the communication channels. In this paper we consider two criticality levels LO and HI for the flows sent in the NoC. The criticality is associated to security and timeliness constraints that should be granted for highly critical messages. We adopt a mixed-criticality approach, in which we separate the NoC flows in two criticality levels, high (HI) and low (LO), which represent the levels of importance in terms of security, safety and timeliness. We focus on the impact that a security attack can have on critical flows (worst-case traversal time (WCTT)) and as a result to their execution time. The challenges we identify are, firstly to improve the NoC's ability to deal with abnormal behavior coming from a node (security aspect) and secondly to guaranty performance and timing requirements for HI criticality flows.

Our contribution: Our implementation is at NoC and hypervisor levels and will allow NoC traffic to be managed in an efficient and intelligent way. We propose a mixed-criticality approach such that the system is able to shift between criticality levels to protect itself from attacks. We propose a mechanism that will allow a hypervisor to configure the hardware with the criteria on which the system detects anomalies in the traffic of each IP as well as with the limit to impose locally to the attacking node. In this state, our proposed mechanism provides the possibility to apply quotas locally to the misbehaving nodes in the NoC limiting rogue traffic. In addition, the hypervisor is given the possibility to monitor the mechanism, reconfigure it on runtime and take further action. The goal is to make sure that critical tasks will always have access to resources they require and will maintain their performance. As a result their timeliness constraints will be preserved even in the case of security attacks.

In section II, we introduce the problem considered in this paper by characterizing the use case, the attack scenarios and the mixed criticality model in the context of security and timeliness problems. Section III presents the platform and network model we consider for our NoC system. Then, in section IV we describe the solution we consider to limit the effect of an attack on the timeliness constraints of a highly critical task communicating to the NoCs. As a proof of concept, we show the behaviour of our solution with illustrative examples. Finally we conclude this paper.

II. The considered problem

A. Use Case

We consider an architecture based on the NoC platform described in Figure 1. The hardware platform consists of many processing elements such GPPs, GPUs, DSPs, etc. or Input/Output (I/O) peripheral controllers. The hypervisor is responsible for virtualizing, separating and managing hardware resources by means of Partitions. Partitions are executing individual tasks or running a full Operating System (OS) on a subset of the available hardware resources provided by the hypervisor. In such an architecture multiple partitions might co-exist and share resources, each performing its own tasks potentially ignoring the existence of the other. Partitions will try to occupy resources while the hypervisor is in charge of monitoring and enforcing the operation of the entire platform. Depending on the application domain these partitions might be statically defined at design time or created dynamically and operated by third parties. For example in cloud infrastructures the owner of a container might not be aware of other owners he might be sharing the hardware platform with. Similarly in telecommunication infrastructures, communication gateways can provide the possibility to different service providers to co-exist in the same hardware, each delivering services for his subscribers. Finally in IoT gateways, partitions can be predefined and it is not normally expected to have third parties residing in the platforms. However the risk here is that such platforms are usually placed in locations that provide physical access to third parties and as a result an attacker can potentially tamper with the platform and install malicious software (e.g. a backdoor or rootkit).

In all three cases, it is essential that the virtualized environment guarantees isolation in terms of data confidentiality and resource availability to the partitions that require such features. Ensuring such policies is part of the role of the hypervisor, who will try to maintain performance while supervising the activities of each partition. However since this supervision is implemented in software, there is performance penalty resulting from the time it takes the hypervisor to approve each partition's access requests and enforce the necessary rules. To tackle this issue, hardware assisted virtualization involves providing mechanisms to take place at a hardware level. In such a case the hypervisor configures the policies and delegates the supervision to the hardware, reclaiming an important part of performance that would otherwise be lost.

We aim at proposing a proof of concept for our solution. Therefore we consider the case where a bare metal partition is composed of a simple task in charge of communicating through the NoC. We also use the term Partition to refer when needed to the single task (executing on one or more cores) in charge of sending a flow. We consider a critical partition (A) and a non-critical partition (B) launched by a hypervisor. When partition (B) performs a DoS attack, it will continuously try to retrieve

memory locations occupying the main memory. By doing that it is not allowing partition (A) to utilize the bandwidth that would be available under normal conditions. As a result the latency of flows coming from partition (A) should increases, impacting the task's execution time. In a CPS with real-time constraints this could have undesired outcomes ranging from environmental damage to equipment damage or even worse. In a similar fashion, at a side channel attack, partition (B) will try to estimate the algorithm or even extract data such as cryptographic keys, by profiling the channel usage of tasks using the same NoC paths. An example of such an attack is described in [11]. Such attacks can lead to data corruption, information theft or system compromise.

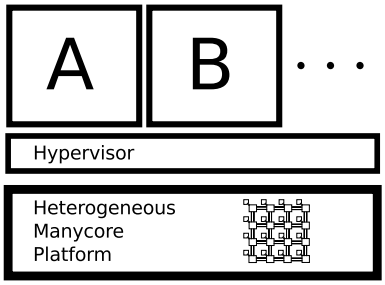

Fig. 1. Basic representation of a virtualization platform

B. Attack scenarios

We proceed by explaining how a DoS attack can take place on a NoC-based platform. We consider as point of entry, a vulnerable partition (B) that lacks security features due to the fact that has a non-critical role. The attacker will execute partitions that access memory locations in a way that in relation to the chip's architecture will limit or block the execution of critical partition (A). From a hardware perspective this can be made possible through the cache memory hierarchy and the interconnect among other things. When a partition is running, the execution of an instruction involves memory transactions that generate traffic in the NoC. Accessing a memory location triggers the cache coherency mechanism that will verify if the requested data can be found in a close proximity to the requesting IP. An attacker will try to access memory in such a way that this mechanism will not be able to cope with the requests ending up soliciting the information at the source, the external memory, a peripheral etc. This will result in the creation of high load for caches as well as high network traffic. Large caches with a high degree of associativity can absorb part of the load but they will not stop the attack. At the same time NoC architectures can offer resource partitioning using distributed caches and dedicated networks for traffic within a partition. As a result a part of the generated traffic will occupy dedicated channels and will not traverse shared NoC channels again alleviating part of the attack but will still not being able to prevent the external memory from being accessed. Eventually the data source will need to respond to requests coming from critical partition (A) and to high frequency requests coming from the attacker partition (B) using shared channels. At this point the critical partition will delay its execution waiting for data that would otherwise take less time to be delivered. Consequently it will execute in a much larger timeslot than the one it was provisioned at design time. Considering systems that have real-time constraints this can potentially prevent the system from performing its functions correctly.

To resolve this issue we need to guarantee that critical partition (A) will continue to run correctly even in a high load scenario. In order to be able to shift the node subject to an attack between LO and HI criticality, to guaranty the resources for critical partition, we need to provide a mechanism that will allow the hypervisor to detect the abnormal behavior and to limit a partition's resource usage to a degree that will limit or block the attack.

C. Mixed-Criticality

We consider a mixed-criticality system that can switch nodes between two criticality levels (LO and HI), depending the level of security required by the the system. In LO mode, all partitions are guaranteed to continue functioning maintaining their performance. This corresponds to the case where not security threat is detected. In the case of a security attack, the criticality of the node subject to an attack switch to HI. In that case, no guarantee is given for lower criticality partitions which, depending on the chosen policy, can continue with less resources or be completely stopped. This signifies that the resource allocation can shift towards high criticality partitions through for example arbitration scheme change, bandwidth allocation throttling or other mechanisms.

We assume that partitions having security, timeliness and safety constraints are critical (denoted partition A in our experiments) and other partitions are non-critical (denoted partition B in our experiments). In addition we consider that non-critical partitions are more vulnerable to an attacker. When the node is functioning in LO mode, its partitions have the necessary resources and can execute as planned. When a node executing a non-critical partition is compromised, an attacker can potentially occupy additional system resources. This can be part of an effort to perform a side channel attack or to saturate the NoC in an effort to perform a DoS attack. In such a case the system finds itself in a limited performance state where critical partition (A) might not be able to occupy the necessary resources resulting in a degradation in performance and consequently in the response time of the system itself. To avoid such an outcome the system must shift to HI criticality mode and be able to ensure that critical partitions continue using the same resources unaffected. This can happen at the cost of delaying or even stopping non-critical partitions.

III. Platform and network model

Before proposing our mixed criticaly mechanism, we define the reference platform we use in order to implement it and validate its functionality.

A. Platform

We use a 4x4 2D mesh NoC topology such as the one presented in Figure 2. The IP elements that are connected to each node can be processing cores, accelerator cores or peripheral controllers. A Network Interface (NI) serializes/deserializes their requests and transfers them to and from the routers. XY dimension routing is used to determine the paths that packets will take to traverse the NoC. To manage the flows, wormhole switching is used, to use small-sized buffers in the routers. On/Off flow control is used to help neighbor router to determine whether they can send/receive more traffic.

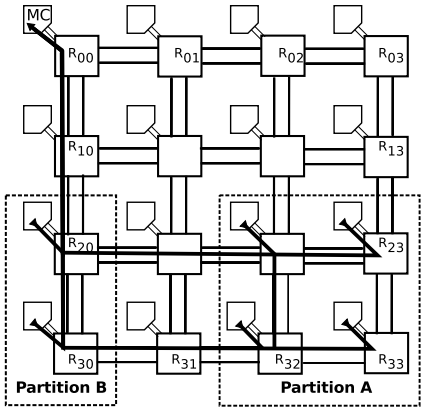

Fig. 2. 2D mesh NoC architecture

Each router R_{xy} consists of five links, four located at the edges **N**orth, **E**ast, **W**est, **S**outh (NEWS), used to connect with neighbor routers and the fifth is used to connect with the **L**ocal (L) IP$_{xy}$. An illustration of a router R_{xy} is given in Figure 3. For example, R_{xy_W} signifies the West link of router R_{xy}. Here x and y are the coordinates of the router inside the 2D mesh and they range from 0 to 3 for a 4x4 NoC. In order to traverse a router, there are two steps that

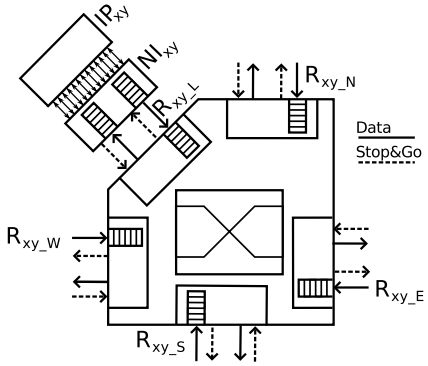

Fig. 3. Architecture of a NoC router R_{xy}

a **fl**ow control dig**it** (flit) has to pass, each taking one clock cycle. In the first one, buffering and routing take place while the second deals with arbitration and output. From a time standpoint, during the first cycle a header flit enters a router and is stored in a small size buffer that can hold up to four flits. At the same cycle it passes through a routing mechanism to determine which output link it wants to reserve. During the second cycle the arbiter (one in each output) will decide which of the potentially competing inputs will take over the output link. At the same time the output register (no output buffers) holds the flit that will traverse the link. These two steps are pipelined and initially two steps are needed to forward the header flit but each of the payload flits will only require one cycle to follow through the path. In this work, we consider round robin arbitration in which we guarantee that all inputs will be allocated to an output channel through a token that is assigned successively to each input in a circular manner.

There are different routers in each node each leading to a different network. These networks are responsible for handling different kinds of traffic which is a common practice in existing SoCs. Cache coherency within a partition is handled by one network and main memory accesses by another. As a result information exchanged between the nodes of one partition will use separate resources that the ones to access the chip's memory. This allows take distribute the load to different channels and makes it more difficult to saturate them.

The **NI** is in charge of serializing and deserializing packets. Packets are then split into smaller size flits in order to travel in the NoC. When an IP element makes a request for a memory location the NI will encapsulate that into a packet, split it into flits and send them one by one to the router. When they reach their destination the local NI will reassemble the packet, deserialize it and forward it to the IP element that will handle the request. The same applies for the response.

This platform is implemented in Verilog and is able to synthesize on a FPGA (Xilinx Virtex-7). Measurements can be taken through a cycle accurate simulator or through data logs of the FPGA output stream.

B. Network model

In this section we present how the network flows are generated when a partition is executed. As explained in Section II-B when a partition is executed it will inevitably generate memory location accesses which will result in NoC traffic. Whether it is accessing data from the chip's external memory or from a peripheral controller's registers, a partition will generate requests. The responses are locally stored in caches close to the core running the partition in order to accelerate further access. When more than one partitions run in parallel we need to consider that multiple cores might be accessing the same memory locations. There is a necessity to preserve data coherency during concurrent accesses which can be done both in software and hardware. The latter is mostly preferred due to its high yield in performance however it is more difficult to characterize in respect to timeliness.

When a core tries to access data that has been previously modified by another core, the cache coherency mechanism will ensure that all the copies of this data are up to date. The more available cores, the more complex the memory hierarchy. When requesting a memory location, a core will receive an entire line that will be locally stored in its cache. In a similar case when running out of space to place new lines, a cache will start flushing lines writing them back to higher levels in the memory hierarchy eventually reaching the chip's main memory.

In most platforms, as well as in ours, a cache line is 64 bytes long. In the example below we indicate a simple algorithm that would generate cache misses constantly forcing the memory hierarchy to look for the cache lines in high levels. This method is known as cache thrashing and results in great performance loss since the partitions will have to fetch their data directly from memory which has a slower response time and will become congested. We consider a last level cache of 3MB (3145728 Bytes) with a degree of associativity of 12. In order to fill the cache we use a table of minimum size twice the size of the cache equal to 6MB (6291456 Bytes). We declare the array in such a way that a row has the same length as a cache line (64 Bytes) and align it at the beginning of a line in memory. We then perform enough iterations to overcome the degree of associativity.

Listing 1. Simple cache thrashing implementation

```
1        char array[49152 * 2][64]
             __attribute__ ((aligned (64)))
             ;
2        void cache_fill(void) {
3        for (int i=0;i<49152 * 2;i++) {
4             array[i][0] = 'a';
5             }
6        }
```

As seen in line 4 of Listing 1 we iterate twice the size of the cache, although the same result could have been achieved with less iterations and a smaller table, for the sake of simplicity we take this approach. Assuming that the array is not already present in the cache we normally expect cache misses for the first 49152 lines. For the rest of the lines in the **for** loop the cache will have to fetch the requested data replacing the first part of the table. As a result as many times as we execute this loop, the core will constantly generate writeback requests to the main memory each containing a complete memory line (9 flit long). We can imagine that number being much larger if we consider busrt mode accesses, when multiple cache lines are transferred in sequence.

C. Use Case Flows

In order to represent the use case in Section II-A, we consider two partitions hosted on the same platform. An attacker i.e. a non-critical partition (B), and a critical partition (A) are both under the supervision of the hypervisor. To implement a scenario where the two partitions coexist in space and time we assume that both partitions execute concurrently and are placed in the cores as depicted in Figure 2.

The memory accesses of each partition generate requests and the memory responds with flows that are defined as τ_A and τ_B. Flow τ_B follows path:

$$\mathcal{P}_B = \{\text{IP}_{33}, R_{33\_L}, R_{32\_E}, R_{31\_E}, R_{30\_E}, R_{20\_S}, R_{10\_S}, R_{00\_L}, \text{IP}_{00}\}$$

while flow τ_A follows path:

$$\mathcal{P}_A = \{\text{IP}_{30}, R_{30\_L}, R_{20\_S}, R_{10\_S}, R_{00\_L}, \text{IP}_{00}\}$$

The rest of the paths follow the same logic.

We can observe that for their memory accesses (based on XY-dimension routing) the two partitions share both the NoC channels $\mathcal{P}_{shared} = \{R_{00}, R_{10}, R_{20}, R_{30}\}$ as well as the access to the memory controller placed at IP_{00}.

We assume that flow τ_B constantly access memory and asks to modify locations as described in listing 1.

IV. Proposed solution

The solution we propose in order to limit the effects of a DoS or a side channel attack is an implementation of a monitoring and a control mechanism that will keep records on the rate of incoming and outgoing traffic and will change the NoC access quota at a node level.

The NoC access monitoring mechanism will measure the bandwidth usage of a node for a timeslot. Should the node exceed a certain threshold, its behavior will be considered abnormal for that timeslot. If this behavior persists for a specific number of timeslots a traffic shaping mechanism will be triggered. This mechanism can limit the traffic that is available for the node for the next timeslots. The number of timeslots can be used to keep a balance between aggressively ensuring execution times or tolerating legitimate traffic (e.g. burst transfers). In the first case we might obtain false positives, which can be considered acceptable when applied to non-critical tasks since we can afford to slow them down at the benefit of critical ones.

The hypervisor is responsible for the configuration of both mechanisms during boot or dynamically during the chip's operation. In this use case the system starts and the hypervisor sets a LO criticality mode and both partitions are given normal access to the NoC. Partition (A) however is not allowed to exceed a certain traffic quota for a long period of time. When it launches a DoS attack that quota is used continuously until the NoC system's criticality changes to HI, the traffic shaper limits the NoC access quota and the injection rate is constrained. During this operation partition (B) maintains access to the resources required to continue its execution and keep reaching its deadlines.

A hypervisor can benefit greatly from such a mechanism as it does not generate overhead apart from the configuration phase, after that it can function autonomously. In addition a hypervisor can use the information provided by the mechanism to draw conclusions on the system's behavior more globally.

In order to validate the mechanism we used Vivado's cycle accurate simulator with our NoC model. Partitions (A) and (B) are placed at two source nodes and are launched almost simultaneously (partition (B) preceding partition (A) by 1000 cycles). They have the same duration when launched separately without any competition from other partitions. Sharing the channels in the NoC path as well as the output to the sink node that absorbs all the traffic. When the mechanism is disabled we observe that both partitions have the same duration, as expected since the round-robin arbiters are handling requests without any distinction between the two flows generated by the partitions. We also notice that this duration is increased in comparison to the one of each partition when executed in isolation.

When the mechanism is enabled and both partitions are launched, we notice that the duration of partition (B) launched by the attacker is increased in favor critical partition (A) that is decreased. The limit in bandwidth we set in the attacker partition will affect the degree at which we observe this behavior.

We ran both partitions multiple times, each time increasing the bandwidth allocation for the attacker (partition B). The obtained data-set is depicted in the graphs below (Figures 4, 5, 6). What follows are our observations on the mechanism's behavior. As we can see in **figure 4** the more bandwidth allowed to the attacker B the higher duration of the critical partition (A), until the point after 80% where they share resources equally. It is visible that as the bandwidth becomes more limited the critical partition obtains an execution time closer to the one it would have if it were executed in isolation (dotted line). Concerning DoS attacks here we have the possibility to apply a threshold that would allow the critical partition to obtain a duration that would allow it to not lose its execution deadlines. In **figure 5** we can observe the same thing; the gap between the two partitions becomes greater as the bandwidth availability for the attacker decreases. In regard

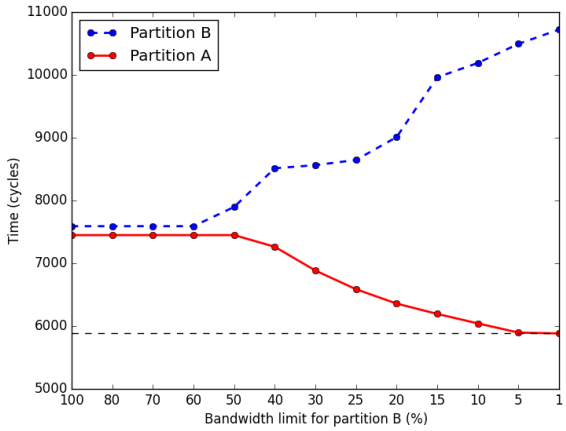

Fig. 4. Partition (A) and (B) duration graph

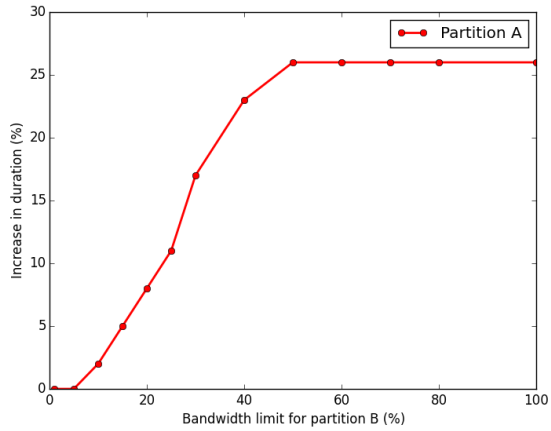

Fig. 6. The percentage of increase in duration for partition (A)

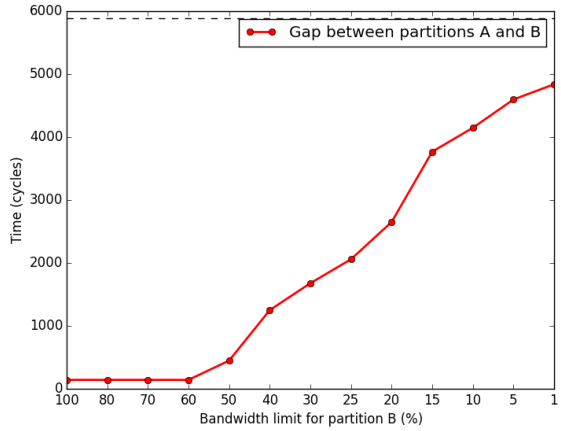

Fig. 5. The gap between the execution times of the two partition (A) and (B)

to side channel attacks, this mechanism offers the possibility to apply it in such a manner that the gap remains wide enough that would make it inopportune for the attacker to obtain information from the critical partition. The quota for the attacker needs low enough that the gap between the two partitions will minimize arbitration for NoC access. As a result the attacker's capacity to extract information from the critical partition will be limited. In **figure 6** we can see how much the execution time of the critical partition (A) increases as we allow more bandwidth for the attacker B. In this particular case we observe that if the mechanism is not activated the critical partition would be impacted by a 25% increase in execution time. This graph allows to determine (depending on partition (A)'s deadline) until which point the critical partition can maintain its timeliness. By using that information we can decide how to configure the threshold in order to guarantee an upper bound for the critical partition.

V. Conclusion

In this paper, we consider the problem of security in NoC in the case of DoS and side channel attacks. We propose a mixed-criticality approach for securing HI critical flows. HI flows have security and timeliness constraints whereas LO flows are supposed unsecured. We propose a solution preserving the timeliness constraints and the security of HI flows in a NoC in the case of an attack. The solution relies on a mechanisms implemented in hardware that is managed by the hypervisor and makes enforcing the security measures we propose more efficient.

References

[1] "Common criteria website," https://www.commoncriteriaportal.org, 2016, [Online; accessed 25-May-2016].

[2] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 684–689.

[3] S. Evain and J.-P. Diguet, "From noc security analysis to design solutions," in *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*. IEEE, 2005, pp. 166–171.

[4] C. H. Gebotys and R. J. Gebotys, "A framework for security on noc technologies," in *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*. IEEE, 2003, pp. 113–117.

[5] M. D. Grammatikakis, K. Papadimitriou, P. Petrakis, A. Papagrigoriou, G. Kornaros, I. Christoforakis, and M. Coppola, "Security effectiveness and a hardware firewall for mpsocs," in *High Performance Computing and Communications, 6th Intl Symp on Cyberspace Safety and Security*. IEEE, 2014, pp. 1032–1039.

[6] S. Baron, M. Silva Wangham, and C. Albenes Zeferino, "Security mechanisms to improve the availability of a network-on-chip," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference on*. IEEE, 2013, pp. 609–612.

[7] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*. IEEE, 2012, pp. 142–151.

[8] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa®-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654–1663, 2013.

[9] L. S. Indrusiak, J. Harbin, and A. Burns, "Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip," in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE, 2015, pp. 47–56.

[10] A. Burns, J. Harbin, and L. S. Indrusiak, "A wormhole noc protocol for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*. IEEE, 2014, pp. 184–195.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.