A novel Network-on-Chip security algorithm for tolerating Byzantine faults

Soultana Ellinidou, Gaurav Sharma, Olivier Markowitch, Jean-Michel Dricot, Guy Gogniat

A novel Network-on-Chip security algorithm for tolerating Byzantine faults

Soultana Ellinidou*, Gaurav Sharma*, Olivier Markowitch*, Guy Gogniat†, and Jean-Michel Dricot*
*Cybersecurity Research Center, Université Libre de Bruxelles, Brussels, Belgium
†Lab-STICC, Université de Bretagne Sud, Lorient, France
Email:{soultana.ellinidou, gsharma, olivier.markowitch, jdricot}@ulb.ac.be, guy.gogniat@univ-ubs.fr

***Abstract*—Since the number of processors and cores on a single chip is increasing, the interconnection among them becomes significant. Network-on-Chip (NoC) has direct access to all resources and information within a System-on-Chip (SoC), rendering it appealing to attackers. Malicious attacks targeting NoC are a major cause of performance depletion and they can cause arbitrary behavior of links or routers, that is, Byzantine faults. Byzantine faults have been thoroughly investigated in the context of Distributed systems however not in Very Large Scale Integration (VLSI) systems. Hence, in this paper we propose a novel fault model followed by the design and implementation of lightweight algorithms, based on Software Defined Network-on-Chip (SDNoC) architecture. The proposed algorithms can be used to build highly available NoCs and can tolerate Byzantine faults. Additionally, a set of different scenarios has been simulated and the results demonstrate that by using the proposed algorithms the packet loss decreases between 65% and 76% under Transpose traffic, 67% and 77% under BitReverse and 55% and 66% under Uniform traffic.**

***Index Terms*—Byzantine Faults, BFT, Network-on-Chip, NoC**

I. Introduction

Byzantine Fault Tolerance (BFT) is the ability of a network to function as desired and correctly reach a sufficient consensus, despite malicious nodes of the system failing or propagating incorrect information to the other nodes. The design of BFT algorithms originates from the introduction of the Byzantine Generals problem by Lamport et al. [1], in which the components of a computer system are abstracted as generals of an army. Loyal generals, which are non-faulty components need to find a way to reach to an agreement (e.g. to attack or retreat), while traitors or faulty components are trying to confound others by sending incorrect messages.

In Distributed systems, the communication process and the behavior of networks in the presence of Byzantine faults have been meticulously studied. Interestingly, Very Large Scale Integration (VLSI) circuits can be viewed as Distributed systems at several levels of abstraction: from gates that communicate via binary signals, to components in a NoC. However, the majority of the existing BFT algorithms cannot be implemented within VLSI systems due to the unavailability of the large amount of resources that is required.

The faults can be classified into transient, intermittent, or permanent faults [2]. Regarding System-on-Chip (SoC), all of the three types of faults can occur in the chip's life cycle. Transient faults appear randomly for one or several cycles. Intermittent faults, which are easily confused with transient faults, occur repeatedly at the same location. They can be tackled by replacing the faulty component hence by removing the fault. Permanent faults can be either logic faults, where transistors or wires are permanently open, or delay faults, where transistors are very slow causing set up and hold timing violation.

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

Different types of faults, coming from the initial three categories, have been introduced, like crash failures, which are permanent faults occurred when a tile halts prematurely or a link disconnects [3]. However, arbitrary failures (also called Byzantine), which are transient faults, have not been explored in the context of Network-on-Chip (NoC). Since the NoC has direct access to all communication resources and information flow within the SoC, attackers have a strong motivation to exploit its possible vulnerabilities. Unfortunately, malicious attacks and malicious hardware modifications of a circuit during the design or fabrication process are common causes of failure and they can cause faulty nodes to exhibit arbitrary behavior, that is, Byzantine faults. Malicious attacks that can cause arbitrary faults within NoC are: Hardware Trojan (HT) [4], DoS [5], HT-Denial of Service (DoS) [6], etc.

Since there is a big gap in the literature of Byzantine faults on NoC, in this paper we address the problem and propose a novel security algorithm for tolerating Byzantine faults. This algorithm is specifically designed for a NoC alternative, called Software Defined Network (SDNoC) [7]. SDNoC provides secure paths in presence of untrusted routers and assures that packets will be successfully delivered to their destination. Our contribution is summarized as:

- a new fault model, in order to introduce the Byzantine faults within NoC,
- a novel algorithm relying on SDNoC for tolerating the Byzantine faults,
- the evaluation and implementation of 3 scenarios within the system by obtaining performance measurements.

The rest of the paper is organized as follows: In Section II we discuss the related work. Thereafter, in Section III the architectural model of SDNoC concept is presented. In Section IV the fault model is introduced, followed by our proposed secure algorithm in Section V. The evaluation results are discussed in Section VI and finally, the conclusion and future work in Section VII.

II. Related work

There is no existing literature on BFT algorithms for NoC, however there is a large number of contributions in Distributed systems following Wireless Sensor Networks (WSN) and Cloud computing. At the end of the 90s, the pioneers Miguel Castro and Barbara Liskov introduced the ”practical Byzantine Fault Tolerance” (pBFT) algorithm [8], which provides practical Byzantine state machine replication by tolerating malicious nodes within a network by assuming that there are independent node failures and manipulated messages are propagated by specific independent nodes. The algorithm is designed to work in asynchronous systems and can process thousands of requests per second with impressive overhead and a slight increase in latency. However it is worth mentioning that the communication between the nodes within the system is heavy and each node not only has to prove that the messages came from a specific peer node but additionally needs to verify that the messages were not modified during the transmission.

Following pBFT, several BFT protocols were introduced to improve its robustness, cost and performance [9, 10, 11], while alternative protocols were introduced that leverage trusted components in order to reduce the number of replicas [12]. Furthermore, WSNs are prone to Byzantine faults because of their limited energy, low calculation capability and dynamic topology. In [13], the authors propose a Byzantine fault-tolerant routing algorithm for large-scale WSN, by ensuring the resistance of timing and energy attacks with help of elliptic curve digital signatures. Afterwards, in [14] a novel distributed fault detection algorithm is presented in order to detect the soft faulty sensor nodes in sparse WSNs, where every sensor node gathers the information only from their neighboring nodes in order to reduce communication overhead.

Cloud-based systems have a more complex architecture in comparison to Distributed systems, they potentially have multiple trust levels and the dynamic change of resources allocated to a service is an easy task in the Cloud. As a result, new BFT algorithms specifically designed for Cloud-based systems have been developed, such as the BFTCloud [15], which is a BFT framework for cloud computing that uses replication techniques to provide the basic fault tolerance and selects voluntary nodes based on Quality of Service (QoS) characteristics and reliability performance. Another interesting contribution by Guisheng Fan et al. [16], proposes a fault detection strategy for cloud module and cloud application, which can make the cloud application to dynamically detect faults at runtime.

III. Architecture Model: SDNoC

Byzantine faults may appear independently in different NoC architectures. In this paper however, we target a NoC alternative called, SDNoC [17, 7, 18]. The SDNoC concept is derived from Software Defined Network (SDN) technology and targets as a main goal the minimization of router's complexity. Precisely, with the help of SDNoC, the routing logic of the routers is exported to a centralized controller which has a general view of the network and can take routing decisions efficiently.

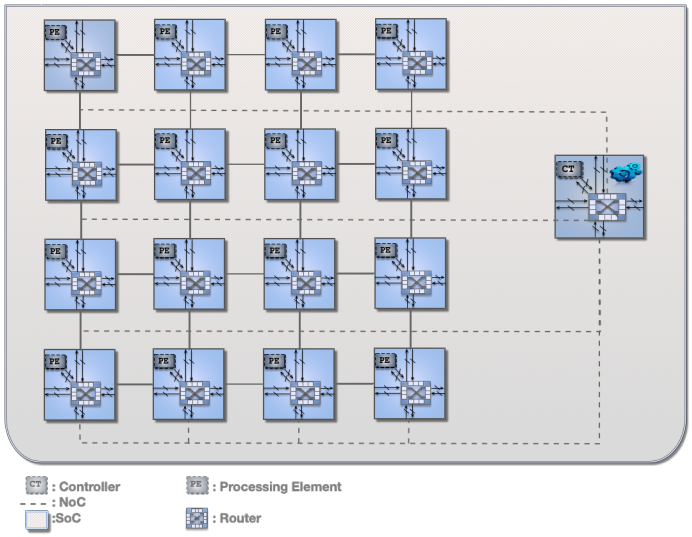


Fig. 1: Software Define Network-on-Chip (SDNoC) architecture

According to the researchers [17, 18], SDNoC could be a potential solution for SoCs due to its advantages: 1) reduction of hardware complexity, 2) high re-usability, and 3) flexible management of communication policies.

Fig. 1 depicts an SDNoC architecture which consists of 16 routers and a centralized controller. The routers are interconnected with the neighboring routers through links and with a Processing Element (PE) through the Network Interface (NI). The centralized controller is interconnected with the routers through direct links and sends configuration packets to the routers in order to manage the routing of the packets in an efficient manner.

IV. Fault Model

Malicious attacks and malicious hardware modifications of a circuit during the design or fabrication often lead to arbitrary failures and can cause faulty nodes to exhibit arbitrary behavior, these are Byzantine failures. Byzantines failures occur when the system is under specific attacks like HT, DoS, HT-DoS, etc.

HT attacks introduce a malicious circuit modification during the design or fabrication process in an untrusted design house or foundry, in which untrusted people, design tools, or components are involved [4]. Such modifications can lead to abnormal functional behavior of a system, degrade performance and provide covert channels or backdoors by which an attacker can leak sensitive information. More precisely, if a router is infected with a HT, it can maliciously change the flit source or destination address or flit type information of a packet. If a Trojan payload modifies the destination address of a packet, that packet could be directed to an unauthorized IP core.

DoS attacks can make the resources of a system unavailable to legitimate nodes. They can also mis-route packets to de-

Algorithm 1 Faulty Node Algorithm

```
Data: n, source, destination
if control_reply[n] ≠ n then
    for i=1:n do
        while control_reply[i] == 0 do
            faulty_node = check_node(i);
              nroute[]=new_route(source, destination, faulty_node);
              control_check(nroute[]);
        end
    end
else
    control_done();
end
```

grade the network performance causing deadlock and virtually link failure [6].

HT can also launch DoS attacks against the NoC [19] of a many-core chip by causing serious damages, including dropping of packets, leaking sensitive information, or modification of functionalities, etc. The consequence of HT-DoS attacks includes bandwidth depletion, incorrect path routing, deadlock and livelock [5].

There is a big number of detection and defense mechanisms specifically designed for each attack separately in literature, however there is no abstract algorithm that can tackle all these attacks at the same time, ensure the right consensus of the network despite the malicious nodes within the system and the correct functionality of the network.

By taking into account the previously mentioned attacks, in this paper we investigate the arbitrary routers by leaving the arbitrary links as a future work. When a NoC is under the above mentioned attacks, the possible arbitrary behavior of a router can include:

- arbitrary deviation from its specification,
- packet redirection,
- packet modification,
- (partial) packet dropping,
- deadlocks or livelocks.

V. Algorithm

Following the architecture and fault model, we design the following algorithm, which consists of 2 cases: a) the Normal Case Operation, where the system has no faults and b) the Byzantine fault Case Operation, where the system has faults.

A. Normal Case Operation

The main network entities are the source router, the destination router, the controller and the routers along the route from source to the destination. The source router is linked with the source PE, which wants to send a packet to a destination PE. The source router will contact the controller, to request a route. Afterwards the controller, with the help of a routing algorithm described on [17], will find a route and it will check all the routers along the route for faulty behavior, after which it will inform the source router for the next hop of the packet. Finally,

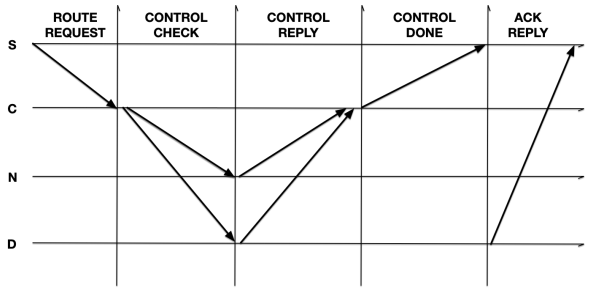


Fig. 2: Messages under Normal Case operation

the source router will wait for a final acknowledgment of the packet by the destination in order to ensure that the packet was successfully delivered. More precisely, each round of the algorithm consists of 6 steps:

- **Step 1:** The source node sends a request to the controller.
- **Step 2:** The controller multicasts the request to the other nodes along the path based on the routing technique that was chosen.
- **Step 3:** The nodes send a reply to the controller.
- **Step 4:** The controller waits for n replies from the nodes. (n is the number of nodes).
- **Step 5:** The controller sends a message to the source node in order to inform him that the nodes along the path are not faulty and to initiate the routing process.
- **Step 6:** The Destination Node sends an acknowledgment to the source node.

In order to implement our algorithm within NoC, a set of 6 network messages were designed (Table I). Network messages are exchanged between the nodes through physical links following the steps of the Normal Case operation algorithm.

Fig. 2 provides an overview of the algorithm, in which the network messages are integrated, in the normal case of no faults. S stands for Source node, C for Controller, N for Nodes along the route and D for Destination node.

B. Byzantine fault Case Operation

In the second scenario, we consider that the system is equipped with Byzantine faults by following the previously described fault model. In this case, the Normal Case Operation

TABLE I: Designed Network messages

Type	Type Value	Description	Contents
ROUTE_REQ	0x01	Sent by source router to controller, which asks a route for a packet.	SRC_ID, DST_ID, Packet_ID, TS
CONTROL_CHECK	0x02	Sent by controller to the nodes along the chosen path.	ACK
CONTROL_REP	0x03	Sent by the nodes on the path to controller.	NODE_ID, TS
CONTROL_DONE	0x04	Sent by controller to the source router.	PACKET_ID, NEXT_HOP, TS
ACK	0x05	Sent by destination router to the source router.	PACKET_ID, TS
ALERT	0x05	Sent by source router to the controller in order to inform him that he didn't receive an ACK from the destination.	DST_ID, Packet_ID, TS

algorithm needs to be enhanced with other 2 algorithms, specifically designed for the controller.

Taking into account the Normal Case Operation algorithm, if a faulty router is present, the first scenario to be considered is that the controller will not receive a reply, CONTROL_REP, from the faulty routers along the route. Thus, we designed Algorithm 1, that is executed by the controller. More precisely, the controller first checks if it received a reply from all the routers along the route. If so, it continues to the next step of the Normal Case algorithm, otherwise it considers the router, from which it did not receive a reply, as faulty one and recomputes a new route for the given source and destination of the packet, excluding this router.

The second scenario to be considered is that a faulty router, along the route, could pretend to be legitimate by replying to the controller. However, it sinks the received packet, such that it can never reach its final destination. As a result the destination will not receive any packet and it will not send an ACK to the source. Thus, we designed the ALERT messages, which will be sent from the source to the controller in order to inform that the packet may not have been received by the destination. When the controller receives an ALERT message, it initiates Algorithm 2.

Based on our architecture, we equipped each router with a counter in each port (north, east, south, west), which is incremented every time that a new packet is imported and decremented every time that a packet is exported. The results are saved in a *TrustTable*, which includes all the values for the different ports. When the controller receives an ALERT message, it requests from all the routers to send their *TrustTable* along with their *RouterID*. The controller calculates and chooses the routes for each individual source-destination pair by storing them in the table *Routes*. The value k indicates the 4 different directions north, east, south, west.

Algorithm 2 is mainly used to identify which are the faulty routers with the help of the table *Suspect*. First the controller checks, whether any input of the *TrustTable* is less than a threshold value (*tv*). This threshold value can be chosen depending on traffic pattern or buffer holding capacity of the system. If so, then it is searching for the previous hop ($neighbor$), in order to identify the possible suspect router. Since the controller calculates the table of *Suspect* of the given *RouterID*, it will also check the tables of *Suspect* of the other *RouterID*'s. If a suspect appears at least in two different *Suspect* tables, because each router could have at least two neighbor routers, this router will be considered as faulty.

VI. Evaluation

Simulations were performed with Garnet2.0 [20], which is a NoC model implementation within the gem5 simulator. We implemented and simulated different scenarios in order to show how a Byzantine fault can affect NoC and also to show the improvements of throughput and packet loss as a result to our proposed algorithms. The first scenario represents the Normal Case Operation which is described in Section V-A. Afterwards various scenarios were implemented, where 1, 3 and 6 Byzantines faults were imported within the NoC and the Byzantine fault Operation algorithms V-B were tested.

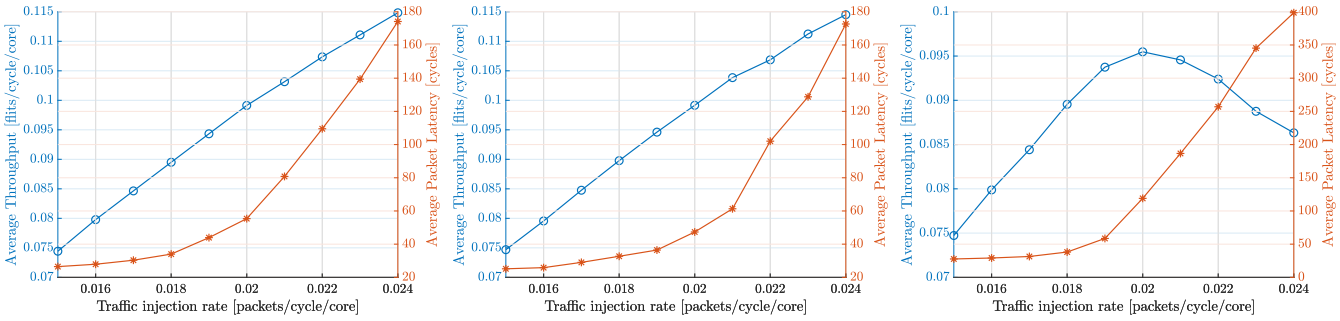
Within the simulations the traffic generated by the processing cores depends on the traffic injection rate (*tir*). The *tir* is the average number of packets injected by the cores into the network per clock cycle ($0 < tir \leq 1$). An 8×8 topology is simulated, by implementing 0, 1, 3 and 6 Byzantine faults within the network. Furthermore, three different traffic scenarios have been evaluated: Transpose, Uniform and Bit-Reverse. It should be noted that for each scenario we perform 40 iterations, of which the average value of throughput and latency are calculated.

The results of the first scenario, which represent the Normal Case Operation of our algorithm, are depicted in Fig. 3a, 3b, 3c. Precisely, in the figures the average throughput and the average packet latency, under different injection rates (0.015-0.024), are presented for Transpose, Uniform and BitReverse traffic respectively. The average throughput and latency tend to be identical for Transpose and Bit-Reverse traffic. The average throughput is in the range of 0.075-0.115 flits/cycle/core and the average latency is between 20-180 cycles. As a result, the controller relies on an accurate view of the network state and it is able to balance the traffic across the network by avoiding the form of congested network areas. However, under Uniform traffic the controller is unable to balance the traffic under high injection because of the source-destination pair is randomly chosen hence, in conjunction with the routing algorithm restrictions applied to the routes, the average latency is in the range of 0-400 cycles and the average throughput in the range of 0.0075-0.0095 flits/cycle/core.

In Fig. 4, three different scenarios are presented in each graph. In the first scenario, a single Byzantine fault is imported, the second scenario considers three Byzantine faults and in the third instance, there are six Byzantine faults. Fig. 4a 4b 4c depict the normalized average throughput under Transpose, Uniform and BitReverse traffic respectively. Fig.

Algorithm 2 Alert Algorithm

```
Data: Routes[ ][ ], TrustTable[ ][ ], RouterID, a=0
for k=1:4 do
    if TrustTable[k][2] < tv then
        for j=1:Routes.rows() do
            for t=1:Routes.column() do
                if Routes[j][t] == RouterID then
                    neighbor == Routes[j-1] [t];
                    if TrustTable[k][1] == direction.neighbor() then
                        a=a+1;
                        Suspect[a]=neighbor;
                    end
                end
            end
        end
    end
end
```



(a) Throughput and Latency under Transpose Traffic. (b) Throughput and Latency under BitReverse Traffic. (c) Throughput and Latency under Uniform Traffic.

Fig. 3: Normal Case Operation Scenario measurements.

TABLE II: Packet loss improvement.

# Byzantine Faults	1		3		6	
# Algorithm	1	2	1	2	1	2
Transpose traffic	24%	15%	56%	47%	76%	65%
BitReverse traffic	24%	14%	55%	46%	77%	67%
Uniform traffic	19%	10%	50%	42%	66%	55%

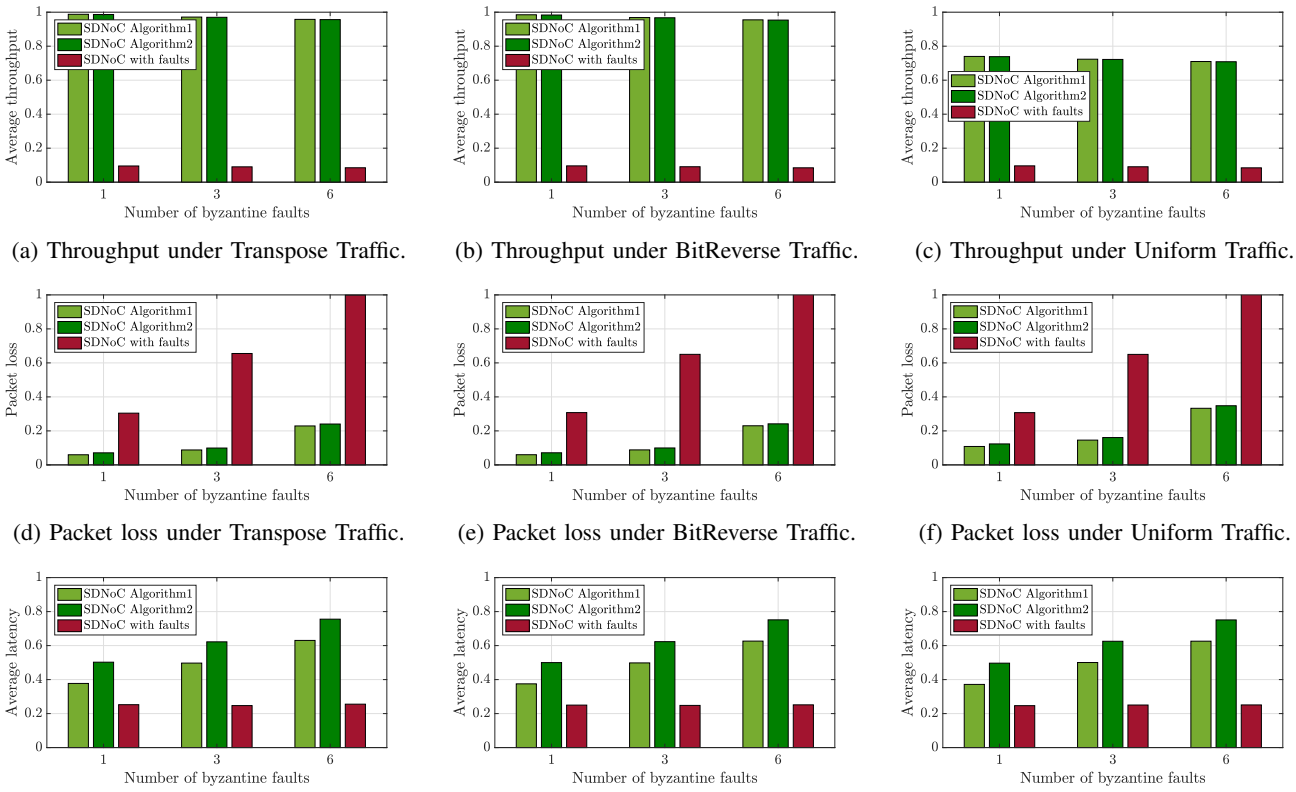
4d 4e 4f show the normalized packet loss rate and Fig. 4g 4h 4i illustrate the normalized packet loss. By taking into account these results, the packet loss improvement is shown in Table II. As far as the throughput is concerned, it improved between 62-64% for Uniform traffic and 87-89% for Transpose and BitReverse traffics. However with the implementation of the algorithms within the system, there is an increase in the functionalities of the network and hence, there is also a latency increase between 10% and 40%.

VII. Conclusion and Future Work

Byzantine faults are a common problem in all systems and can cause network performance decrease, higher packet loss and arbitrary behavior of the nodes. However, it remains an unexplored research problem in the context of VLSI systems and more precisely in the NoC. In this paper we tried to address this by introducing a new fault model in NoC context and by designing and evaluating a novel lightweight algorithm, which includes two cases of operation, and can tolerate Byzantine faults based on SDNoC architecture.

From the results, it is obvious that there is a large throughput decrease and packet loss increase due to the Byzantine faults. Hence, we proposed two different algorithms in order to deal with the reverse arbitrary behavior of the Byzantine fault routers. By applying our algorithms, the NoC continues to function normally by improving the overall packet loss 23%-77% and the average throughput 62%-89%.

Our main goal was to achieve the right consensus of the system and the delivery of the packet from the source to the destination. Furthermore by using the SDNoC architecture, we ensure the authenticity of the network, since there are direct links between the controller and each router. However the confidentiality and integrity of the network are still open research problems and need further exploration.



(a) Throughput under Transpose Traffic. (b) Throughput under BitReverse Traffic. (c) Throughput under Uniform Traffic.

(d) Packet loss under Transpose Traffic. (e) Packet loss under BitReverse Traffic. (f) Packet loss under Uniform Traffic.

(g) Latency under Transpose Traffic. (h) Latency under BitReverse Traffic. (i) Latency under Uniform Traffic.

Fig. 4: Byzantine fault case operation scenarios measurements.

REFERENCES

[1] L. LAMPORT, R. SHOSTAK, and M. PEASE, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[2] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE micro*, vol. 23, no. 4, pp. 14–19, 2003.

[3] T. Dumitras, S. Kerner, and R. Marculescu, "Towards on-chip fault-tolerant communication," in *Proc. of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003.*, pp. 225–232, IEEE, 2003.

[4] S. Bhunia and M. Tehranipoor, *The Hardware Trojan War*. Springer, 2018.

[5] J.-P. Diguet, S. Evain, R. Vaslin, G. Gogniat, and E. Juin, "Noc-centric security of reconfigurable soc," in *First International Symposium on Networks-on-Chip (NOCS'07)*, pp. 223–232, IEEE, 2007.

[6] L. Daoud and N. Rafla, "Routing aware and runtime detection for infected network-on-chip routers," in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 775–778, IEEE, 2018.

[7] K. Berestizshevsky, G. Even, Y. Fais, and J. Ostrometzky, "Sdnoc: Software defined network on a chip," *Microprocessors and Microsystems*, vol. 50, pp. 138–153, 2017.

[8] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[9] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 59–74, 2005.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proc. of the 7th symposium on Operating systems design and implementation*, pp. 177–190, 2006.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, pp. 1–39, 2010.

[12] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 189–204, 2007.

[13] J. Xu, K. Wang, C. Wang, F. Hu, Z. Zhang, S. Xu, and J. Wu, "Byzantine fault-tolerant routing for large-scale wireless sensor networks based on fast ecdsa," *Tsinghua Science and Technology*, vol. 20, no. 6, pp. 627–633, 2015.

[14] M. Panda and P. M. Khilar, "Distributed byzantine fault detection technique in wireless sensor networks based on hypothesis testing," *Computers & Electrical Engineering*, vol. 48, pp. 270–285, 2015.

[15] Y. Zhang, Z. Zheng, and M. R. Lyu, "Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 444–451, IEEE, 2011.

[16] G. Fan, H. Yu, L. Chen, and D. Liu, "Model based byzantine fault detection technique for cloud computing," in *2012 IEEE Asia-Pacific Services Computing Conference*, pp. 249–256, IEEE, 2012.

[17] S. Ellinidou, G. Sharma, S. Kontogiannis, O. Markowitch, J.-M. Dricot, and G. Gogniat, "Microlet: A new sdnoc-based communication protocol for chiplet-based systems," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 61–68, IEEE, 2019.

[18] L. Cong, W. Wen, and W. Zhiying, "A configurable, programmable and software-defined network on chip," in *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, pp. 813–816, IEEE, 2014.

[19] L. Zhang, X. Wang, Y. Jiang, M. Yang, T. Mak, and A. K. Singh, "Effectiveness of ht-assisted sinkhole and blackhole denial of service attacks targeting mesh networks-on-chip," *Journal of Systems Architecture*, vol. 89, pp. 84–94, 2018.

[20] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE international symposium on performance analysis of systems and software*, pp. 33–42, IEEE, 2009.