A Machine Learning Approach for Traffic Anomaly Detection in NoC-based Manycores

Angelo Elias Dal Zotto and Fernando Gehm Moraes
PUCRS - School of Technology, Av. Ipiranga 6681, Porto Alegre, Brazil
angelo.dalzotto@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—NoC-based manycores are increasingly vulnerable to traffic anomalies, which can compromise the performance and security of these systems. Early proposals to detect traffic anomalies primarily focus on static latency analysis. Machine learning (ML) is a potential solution to this problem, as it can support variations in mapping and workloads. However, state-of-the-art ML detectors use costly techniques, usually based on artificial neural networks, adopt synthetic NoC traffic, and use high-level simulations, which do not accurately reflect actual systems. This paper introduces an ML-based flow for detecting traffic anomalies using XGBoost regression, trained on a dataset that captures various traffic characteristics, such as latency and hop count, and different mappings, using a manycore modeled at the RTL level. Our flow presents high accuracy in identifying deviations from expected traffic behavior. This work enhances the security of manycores and opens new opportunities for real-time monitoring and traffic anomaly detection.**

***Index Terms*—Traffic Anomaly, XGBoost, NoC-based manycores, Security.**

I. Introduction and Related Work

Manycore systems, such as ET-SoC-1 [1], can house over a thousand cores on a single chip. These systems are intrinsic to modern interconnected environments, particularly in IoT networks. However, this interconnectivity renders them susceptible to various forms of cyberattacks, such as malware, phishing, ransomware and DDoS, as well as hardware-specific threats like side-channel attacks [2], hardware bugs [3], and hardware Trojans (HTs) [4]. The use of third-party intellectual property cores as a solution to cope with design complexity and time-to-market pressures further worsens these vulnerabilities. Thus, security is a critical concern in developing and deploying manycore systems.

The existing literature offers various strategies for threat *detection* [5], *localization* of these threats [6], and the corresponding *countermeasures* [7] in NoC-based manycores. This work focuses on threat detection.

In early research on threat localization, Rajesh et al. [8] introduced a runtime latency auditor for NoCs named RLAN. This auditor generates Proximal Analogous Packets (PAPs), slightly modified versions of existing packets. It then compares their latencies to identify any malicious activities. However, this work does not consider factors that could impact the system, such as: (*i*) an HT could affect both the original packet and the PAP; (*ii*) variations in application mapping could influence latencies; and (*iii*) the use of synthetic traffic does not accurately represent actual application behavior, where latencies can fluctuate based on the workload.

Research related to threat detection in NoCs started to use machine learning (ML) due to the limitations of methods that rely exclusively on packet latency monitoring. Kulkarni et al. [9] presents one of the first ML proposals for anomaly detection in manycore systems. In their research, they assess K-Nearest-Neighbors (K-NN), Linear Regression (LR), Decision Trees (DT), and unsupervised learning algorithms. They choose to implement Support Vector Machines (SVM) due to their advantageous balance between classification performance and hardware complexity. The SVM algorithm is integrated into a 16-core NoC implemented on an FPGA. Kulkarni et al. employ synthetic traffic generated by an external data injector to the FPGA, which does not accurately represent real-world workloads. Another limitation lies in the NoC topology, with four routers connected to four cores and a central router connected to these four routers. This architecture compromises the scalability of the proposed solution.

Sudusinghe et al. propose methods for detecting flooding DoS [10] and eavesdropping attacks [11] using ensemble learners, which are combinations of multiple ML models to enhance performance. The Authors claim their techniques can identify attacks in unpredictable NoC traffic patterns caused by varying application mappings. The routers in the system have probes that send relevant features to centralized units that detect the attacks. Their experimental setup employs the Gem5 simulator in a 4x4 system configuration and achieves nearly 90% accuracy in detecting attacks.

Some authors also propose using Neural Networks (NN). TSA-NoC [12] and AGAPE [13] detect HTs without manually configured thresholds. While TSA-NoC is a 3-layer NN to detect HTs, AGAPE uses a Generative Adversarial Network capable of detecting, localizing, and neutralizing HTs. These approaches have as a drawback the NN area overhead, which every NoC router incorporates. The authors claim low overhead due to the adoption of complex processors with hierarchical caches and out-of-order execution, features that do not apply to embedded systems. In addition to this drawback, NN insertion in these works is invasive in the router design, requiring total control over an IP that an HT might infect. These works also simulate the system using Gem5.

The motivations for the present work are the limitations found in early work on threat localization and in works using ML. ML is a potential solution for traffic profiling, which can consider different variations in mapping and workloads that classical algorithms cannot tackle. However, state-of-the-art ML techniques for anomaly detection in NoCs are limited in *applicability* and *confidence*. The first limitation, *applicability*, is due to costly techniques, usually based on NN, replicated at every router in the NoC [8], [12]–[14]. The second limitation, *confidence*, is due to the adoption of synthetic NoC traffic that does not represent real application scenarios [9], [15]–[18] and

the use of high-level simulators [10]–[15], [17], [18], such as Gem5, which do not reflect the actual accuracy of the systems.

Our *goal* is to detect anomalies in the NoC traffic by using an ML flow that profiles real applications in an actual manycore, modeled at the RTL level with clock cycle accuracy. Our *original contribution* is twofold: (*i*) create and use application profiles instead of synthetic NoC traffic; (*ii*) detect subtle, uncorrelated traffic collisions that occur due to disturbing malicious traffic in applications.

II. Application Profile

We assume that applications are modeled by Communication Task Graphs (CTG). A CTG is a graph where vertices correspond to tasks, and edges carry messages from a producer to a consumer task. Figure 1 shows a CTG for the MPEG application. Task t_0 initiates the application by producing frames to be decoded by t_1, t_2, and t_3, then outputted by t_4. This CTG has four edges: $e_{0,1}$, $e_{1,2}$, $e_{2,3}$, and $e_{3,4}$.

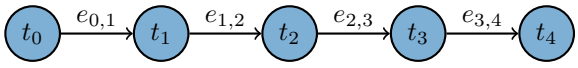


Fig. 1: CTG of the MPEG application.

Figure 2 curve C (2C), shows the communication behavior of the MPEG application, where each graph represents the latency for a given edge. The x-axis represents the time, and the y-axis the latency. In Figure 2C, the application is mapped according to Figure 3a, with all edges traversing one hop.

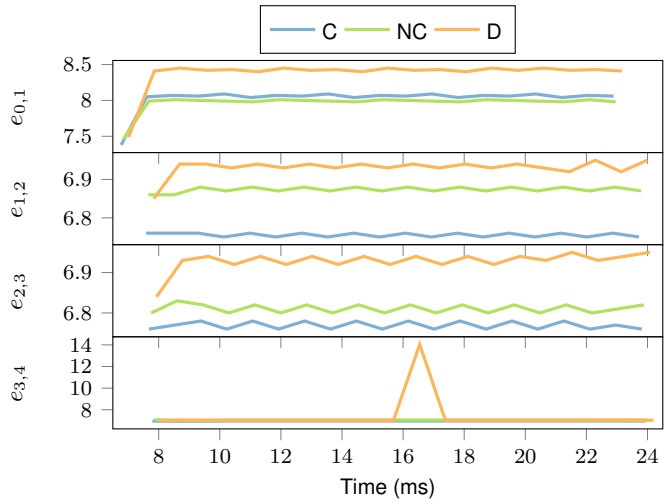
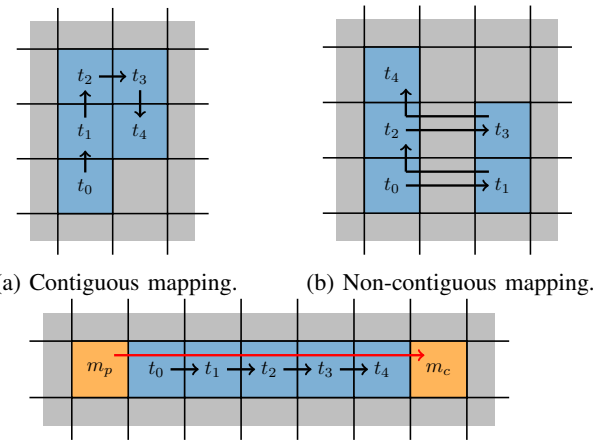


Fig. 2: Latency of MPEG messages for each edge ($e_{0,1}$, $e_{1,2}$, $e_{2,4}$, $e_{3,4}$). The y-axis represents the latency in μs.

Figure 2NC corresponds to the mapping presented in Figure 3b, with an average of 2.5 hops between communicating tasks. The number of hops between communicating tasks changes the communication latency, not its behavior. *This behavior cannot be captured by models that only monitor latency, which was adopted in early work*. Summarizing:

- An application represented by a CTG has a communication behavior for each of its edges;
- This behavior does not change by the number of hops between edges, i.e, it is not a function of the mapping.



(a) Contiguous mapping. (b) Non-contiguous mapping.

(c) Mapping with malicious interference.

Fig. 3: Mappings for the MPEG application.

What happens when malicious traffic is inserted into the NoC? Malicious traffic can be caused by an HT, a compromised peripheral, or malicious or faulty software that generates traffic to the links used by an application.

Figure 3c presents the MPEG application with a contiguous mapping and a malicious producer task (m_p) sending messages to a malicious consumer (m_c), competing for the same links used by the MPEG application. The malicious application depicted in Figure 3c generates packets with 96 flits. The application continuously produces disturbing traffic, with the time interval between messages randomized between 100 and 500 μs. This combination of small messages and randomized intervals increases the likelihood of traffic collisions, leading to hard-to-detect, temporary Denial-of-Service (DoS) attacks.

Figure 2D represent the edges' latency when subjected to disturbing traffic. While there are minor changes in edges $e_{0,1}$, $e_{1,2}$, and $e_{2,3}$, these cannot be classified as malicious behavior. The variations could be attributed to different mappings or loads, as the overall communication behavior remains similar despite the added latency offset. However, edge $e_{3,4}$ shows a spike in latency between 16 and 18 ms, indicative of a collision that constitutes a temporary DoS attack.

The challenge is to develop an application profile capable of monitoring the latency of each edge in the NoC to detect potential malicious interferences, whether from software or hardware. ML is suitable for this purpose because it can generate a regression model for each application, accurately capturing variations in communication behavior. Among available regression models, including LR, DT, and K-NN, XGBoost [19] stands out as a recent and widely used option and is used in this work.

III. Machine Learning Flow

The proposed ML flow, presented in Figure 4, builds a model to compare the predicted application behavior with the actual behavior, enabling malicious traffic detection. Section III-A details the Dataset Creation phase, represented by Figure 4DC, explaining how traffic characteristics from Section II are extracted and prepared for model training. Section III-B focuses on the Regression Model Creation phase, denoted by Figure 4RMC, detailing the training and validation

of the XGBoost model. Section III-C discusses the Classification Inference phase, indicated by Figure 4ID, describing how the trained model is used to infer the presence or absence of traffic anomalies in the NoC.

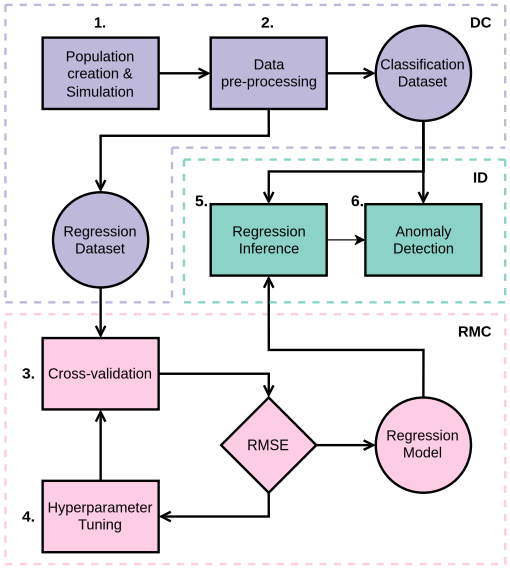


Fig. 4: Machine learning model flow.

A. Dataset Creation Phase

The initial step of the flow is to *create a population*, Figure 4(1). This step starts by defining the manycore size to simulate various application mappings. We assume that each Processing Element (PE) executes a single task. The minimum manycore size is chosen based on three criteria: (*i*) the number of tasks in the application being profiled, (*ii*) the mapping task (map), placed at the top-left corner of the manycore, and (*iii*) two additional PEs reserved for a disturbing traffic generator (m_p and m_c), located at the bottom-left and top-right corners. Figure 5 illustrates this setup for the MPEG application. The manycore size is 3x3, with six PEs for the five MPEG tasks (in blue), and the map, m_p and m_c tasks.

In Figure 5, the blue area is reserved for the MPEG application, and all possible mapping permutations within this space are generated, resulting in 720 different mapping scenarios. The aim is to account for variations in hop counts between edges, flow directions, and potential flow crossings. One limitation of this approach is that it does not consider fragmented mapping; however, disturbances arising from fragmentation are beyond the scope of this work as the mapping heuristic can be tuned to avoid fragmentation, thus keeping the mapping similar to the training scenarios. These scenarios are mirrored to create another 720 scenarios with the same mapping but without the disturbing traffic generator.

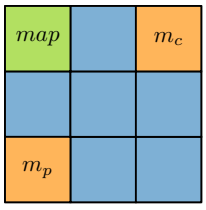


Fig. 5: Manycore mapping setup for MPEG profiling.

The population creation step ends with the RTL simulation. Probes are inserted into the local output port of the routers to log: (*i*) the system time when the PE has ended receiving an entire message; (*ii*) the ID of the message sender; (*iii*) the ID of the message receiver; (*iv*) the latency of the received message, measured from the moment it begins transmission at the sender's NI until the recorded timestamp.

The data pre-processing step, Figure 4(2), extracts data from the simulation logs. Messages unrelated to the profiled application are removed based on sender and receiver IDs. These IDs are then converted into categorical values, enabling the identification of the CTG edge associated with each message. All message timestamps are transformed from system-time to application-time, meaning they are made relative to the timestamp of the first message sent by the application, which is set to zero. The number of hops between each application edge is obtained from the mapping and added to the data.

Two datasets are created from this step. The first one, called the *Regression Dataset*, is a table with five columns, {**timestamp**, **hops**, **sender ID**, **receiver ID**, **latency**}, containing the relevant characteristics of all messages (in this example, the dataset has 57,600 lines). The Regression Dataset uses a random subset containing 75% of the simulated scenarios without the disturbing traffic generator (in this example, 540 scenarios).

The second dataset is called the *Classification Dataset*, which is originated from 25% of the scenarios with the disturbing traffic generator (in this example, 180) that do not have its equivalent mapping present in the Regression Dataset. Additionally to the five columns in the Regression Dataset, the Classification Dataset includes a *tag*, indicating whether an instance is affected by the malicious traffic. This tag originates from comparing the latency column of the Classification Dataset with the latency of equivalent scenarios simulated without the disturbing traffic generator, which were not added to the Regression Dataset. If the latency of an instance of the Classification Dataset deviates beyond a configurable threshold (e.g., 5%) from what is expected, that instant is identified as being affected by disturbing traffic.

B. Regression Model Creation phase

This phase (Figure 4RMC) builds and validates a regression model capable of reproducing the application communication behavior. The step in Figure 4(3) makes the cross-validation using the Regression dataset as input for the XGBRegressor Python API from the Distributed Machine Learning Community (*dmlc*) [20]. The model is initially trained with default hyperparameters, such as limiting the number of estimators (decision trees) to 100 and the maximum estimator tree depth to 6. The model is validated using cross-validation.

The cross-validation step results in an average Root Mean Squared Error (RMSE). This metric represents how far each latency generated by the regression model is from the actual latency produced by the application. The hyperparameter tuning step pictured in Figure 4(4) iteratively simplifies the XGBoost model by decreasing two hyperparameters – number of estimators and maximum tree depth – until the RMSE begins to increase. For the MPEG application, the default hyperparameters result in an average RMSE of 3.24 μs, meaning that the model deviates $\approx 3\mu s$ from the actual latency of any transmitted message. When tuning the hyperparameters

to use 16 estimators and limiting the maximum tree depth to 6, the model achieves 3.71 μs of RMSE, which, despite being almost a 15% increase, is negligible. Finally, after the hyperparameter tuning, the flow produces the XGBoost regression model.

C. Inference and Detection Phase

The first step of this phase is the *Regression Inference*, Figure 4(5). In this step, the classification dataset columns {**timestamp**, **hops**, **sender ID**, **receiver ID**} are used to infer a *latency* using the regression model.

The last step of the flow is the Anomaly Detection, presented in Figure 4(6). This step uses the latencies inferred from the last step. It compares them against the actual latency in the Classification Dataset, using the same threshold (in this example, 5%) used to tag the Classification Dataset. The result is the binary classification of every message as either positive for being affected by disturbing traffic or negative (not affected). The classification is evaluated using the tag present in the Classification Dataset.

IV. Results

Our target platform is MA-Memphis [21], modeled at the RTL level, with a data NoC, a broadcast NoC for control messages, a RISC-V RV32IM, and a Network Interface (NI) with Direct Memory Access (DMA) [22]. This platform is derived from the Memphis manycore [23]. The software stack is composed of a small operating system and a set of libraries.

Our threat model consists of any traffic source that can modify the behavior of the NoC, **even if this change is subtle**. The origin of this threat can be: (*i*) an HT in the data NoC or any module that can insert traffic into the NoC; (*ii*) a malicious or malfunctioning peripheral; (*iii*) a malicious or malfunctioning application. Attacks that drop packets are not in the scope of this work, given the fact that this type of attack or failure is catastrophic for the system.

We evaluated the proposed flow with four applications: **MPEG**: a pipelined MPEG decoding application with 6 tasks; **DTW**: a pattern-recognition application following a fork-join model with 6 tasks; **Dijkstra**: shortest path algorithm, with master-slave model and 7 tasks; **A/V**: audio and video decoding benchmark with two pipelines, with 7 tasks.

Table I presents results for the applications. The first three rows of the Table present the **Regression Model** parameters (Section III-B). The first two rows show the tuned parameters corresponding to the step outlined in Figure 4(5). The third row shows the average RMSE obtained from the cross-validation of the regression model, as detailed in Figure 4(3,4).

TABLE I: Results compilation.

Application	MPEG	DTW	Dijkstra	A/V
$n\_estimators$	16	20	28	30
max_depth	6	6	6	6
Avg. RMSE (μs)	3.71	0.78	5.20	23.17
Recall	1.00	1.00	1.00	1.00
Precision	1.00	1.00	0.95	0.67
F1-score	1.00	1.00	0.98	0.80

The last three rows focus on the **Classification Inference** results (III-C). The metric *recall* measures the model's ability to detect instances of anomalous traffic, with a recall of 1 indicating that all threats have been detected. However, recall alone is insufficient for evaluation, as a high recall could mean the model is flagging all traffic as anomalous. To balance this, *precision* is also presented, which considers false positives. A well-tuned model should present a good balance between recall and precision, and this trade-off is given by the *F1-score*, which is the harmonic mean of the two metrics.

Figure 6 pictures the confusion matrices for the four tested applications. Together with Table I, we can observe that both MPEG and DTW applications are easily predicted and classified by our flow, even with models with few estimators. Dijkstra and A/V applications are increasingly complex, requiring slightly larger models. However, the flow can still detect anomalies at a perfect rate for both applications. Despite that, the A/V application presented worse RMSE. When the model fails to approximate the application-produced latency correctly, it deviates further and induces the classification to flag an anomaly, resulting in a higher number of false positives and, therefore, decreasing the precision. Nevertheless, even with lower precision, A/V presents a high true negative rate (99%) due to the unbalanced nature of the classification dataset, meaning that 99% of the non-disturbed instances are correctly classified. This unbalance is not present in the regression dataset; therefore, it does not affect the model creation. Note that the threats were correctly detected (*true positives*) using the Classification dataset, i.e., with mapping instances not used in training.

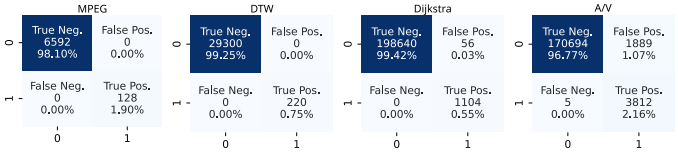


Fig. 6: Confusion matrices for tested applications.

V. Conclusions and Future Work

This work presented a flow for detecting anomalous traffic in NoC-based manycore systems using ML. The training data is obtained from real applications and simulated at the RTL level with clock cycle accuracy. A tree-based ensemble, specifically XGBoost, was chosen for its low computational overhead, dismissing the need for complex hardware, such as neural networks, for inference. Our noninvasive approach requires no modifications to the existing NoC architecture; it only requires a monitoring structure connected to the router's local ports. The results demonstrate a high rate of anomalous traffic detection across different benchmarks, attesting to the effectiveness of the proposed method.

Directions for future work include: (*i*) simplify the monitoring process by reducing the granularity of the application profile, eliminating the need to probe every message sent by applications; (*ii*) expand the range of monitored data and adjust training hyperparameters, enabling the inclusion of real-time constraints and improving precision when necessary; (*iii*) deploy the ML to execute at runtime.

ACKNOWLEDGMENTS

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Finance Code 001; Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), 309605/2020-2 and 407829/2022-9; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), 21/2551-0002047-4 and 23/2551-0002200-1.

REFERENCES

[1] D. Ditzel *et al.*, "Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip," in *HCS*, 2021, pp. 1–23.

[2] D. Moghimi, "Downfall: Exploiting Speculative Data Gathering," 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi

[3] T. Ormandy, "Zenbleed," 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[4] M. Xue, C. Gu, W. Liu, S. Yu, and M. O'Neill, "Ten years of hardware Trojans: a survey from the attacker's perspective," *IET Computers & Digital Techniques*, vol. 14, no. 6, pp. 231–246, 2020.

[5] S. Charles and P. Mishra, "A Survey of Network-on-Chip Security Attacks and Countermeasures," *ACM Computing Surveys*, vol. 54, no. 7, pp. 1–36, 2021.

[6] S. Charles, Y. Lyu, and P. Mishra, "Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4510–4523, 2020.

[7] R. F. Faccenda, L. L. Caimi, and F. G. Moraes, "Detection and Countermeasures of Security Attacks and Faults on NoC-Based Many-Cores," *IEEE Access*, vol. 9, pp. 153 142–153 152, 2021.

[8] 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.

[9] A. Kulkarni, Y. Pino, M. French, and T. Mohsenin, "Real-Time Anomaly Detection Framework for Many-Core Router through Machine-Learning Techniques," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 1, pp. 1–22, 2016.

[10] C. Sudusinghe, S. Charles, and P. Mishra, "Denial-of-Service Attack Detection using Machine Learning in Network-on-Chip Architectures," in *NOCS*, 2021, pp. 35–40.

[11] C. Sudusinghe, S. Charles, S. Ahangama, and P. Mishra, "Eavesdropping Attack Detection Using Machine Learning in Network-on-Chip Architectures," *IEEE Design & Test*, vol. 39, no. 6, pp. 28–38, 2022.

[12] K. Wang, H. Zheng, and A. Louri, "TSA-NoC: Learning-Based Threat Detection and Mitigation for Secure Network-on-Chip Architecture," *IEEE Micro*, vol. 40, no. 5, pp. 56–63, 2020.

[13] K. Wang, H. Zheng, Y. Li, J. Li, and A. Louri, "AGAPE: Anomaly Detection with Generative Adversarial Network for Improved Performance, Energy, and Security in Manycore Systems," in *DATE*, 2022, pp. 849–854.

[14] M. Sinha, S. Gupta, S. S. Rout, and S. Deb, "Sniffer: A Machine Learning Approach for DoS Attack Localization in NoC-Based SoCs," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 2, pp. 278–291, 2021.

[15] K. Madden, J. Harkin, L. McDaid, and C. Nugent, "Adding Security to Networks-on-Chip using Neural Networks," in *SSCI*, 2018, pp. 1299–1306.

[16] A. Vashist, A. Keats, S. M. Pudukotai Dinakarrao, and A. Ganguly, "Securing a Wireless Network-on-Chip Against Jamming-Based Denial-of-Service and Eavesdropping Attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2781–2791, 2019.

[17] J. Yao, Y. Zhang, Z. Mao, S. Li, M. Ge, and X. Chen, "On-line Detection and Localization of DoS Attacks in NoC," in *ITAIC*, 2020, pp. 173–178.

[18] S. Hu, H. Wang, and B. Halak, "Cascaded Machine Learning Model Based DoS Attacks Detection and Classification in NoC," in *ISCAS*, 2023, pp. 1–5.

[19] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," *CoRR*, vol. abs/1603.02754, pp. 1–13, 2016.

[20] Distributed (Deep) Machine Learning Community, "XGBoost: eXtreme Gradient Boosting," 2014. [Online]. Available: https://github.com/dmlc/xgboost

[21] A. E. Dalzotto, M. Ruaro, L. V. Erthal, and F. G. Moraes, "Management Application - a New Approach to Control Many-Core Systems," in *SBCCI*, 2021, pp. 1–6.

[22] M. Ruaro, F. Lazzarotto, C. Marcon, and F. G. Moraes, "DMNI: A specialized network interface for NoC-based MPSoCs," in *ISCAS*, 2016, pp. 1202–1205, https://doi.org/10.1109/ISCAS.2016.7527462.

[23] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, "Memphis: a framework for heterogeneous many-core SoCs generation and validation," *Design Automation for Embedded Systems*, vol. 23, no. 4, pp. 103–122, 2019, https://doi.org/10.1007/s10617-019-09223-4.