Lightweight Machine-Learning-driven Security Threat Detection in NoC-based Manycores

Angelo Elias Dal Zotto and Fernando Gehm Moraes
School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
angelo.dalzotto@edu.pucrs.br, fernando.moraes@pucrs.br

2025 38th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI) | 979-8-3315-5555-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/SBCCI66862.2025.11218641

***Abstract*—This work presents a non-intrusive, lightweight machine-learning-driven approach for runtime detection of Hardware Trojan (HT) in NoC-based manycore systems. The proposed solution uses XGBoost to detect subtle traffic anomalies caused by such HTs. Unlike previous works, which often rely on offline inference or expensive hardware modifications, this approach is implemented as a software-based runtime detection model and assessed on a Register Transfer Level (RTL)-modeled platform. The results show an average detection rate of 92% in real applications with low overheads, making it a feasible solution for runtime security in manycore systems.**

***Index Terms*—NoC-based manycores, Security, Machine Learning, XGBoost, Hardware Trojan.**

I. Introduction and Related Work

As manycore systems grow in complexity, they are increasingly vulnerable to security threats, including Hardware Trojans (HTs) and malicious applications. Addressing these challenges requires runtime threat detection without the overhead of computationally intensive or costly hardware solutions [1], [2]. Previous works on threat detection in NoC-based manycore systems employed classical algorithms based on traffic latency monitoring [3]. These algorithms could not account for variations in traffic patterns, such as those caused by different application mappings or concurrent traffic. Recent proposals have turned to Machine Learning (ML) techniques to address these limitations.

Yao et al. [4] propose a Random Forest (RF)-based method to detect and localize DoS attacks in NoCs, using packet paths and flit latency as features. Training data is generated from synthetic traffic in OPNET [5], implying offline inference.

DetectANN [6] employs a 3-layer fully connected ANN with 30 hidden neurons to detect injected faults, using metrics like buffer and link utilization, ECC error rate, and router temperature. Evaluated on gem5 [7] with synthetic and PARSEC traffic [8], it incurs 0.9% area overhead.

Sniffer [9] uses a Perceptron for detecting and localizing flooding DoS attacks from Malicious IPs (MIPs), leveraging buffer flit time, end-to-end latency, and buffer utilization. Trained with Rodinia [10], it embeds detection IP in each router, causing 3.3% area and 3.92% power overheads. However, it assumes uncompromised routers.

Security Engine [11] uses XGBoost to detect DoS attacks, collecting counters, router IDs, packet paths, and flit latency per traversed flit. These features are probed in routers at every flit traversed and sent to the centralized Security Engine. Training uses the Fast Fourier Transform (FFT) application for traffic. Simulated on gem5 and Garnet NoC model [12], no overhead data is provided, implying offline use.

AGAPE [13] applies a Generative Adversarial Network (GAN) to detect faults, misrouting, and packet drops from HTs, converting attribute time series into matrices and applying convolutional layers. After the GAN training, only the Discriminator is used. Along with DetectANN's features, it adds retransmission rate and average end-to-end latency. Despite claiming a 7.2% area overhead on PARSEC and gem5, the absence of manycore details raises questions about the accuracy of this result.

Hu et al. [14] introduce an approach using SVM for detection and KNN for classifying HTs, based on average packet latency and link utilization. Tested with synthetic traffic on gem5 and Garnet, inference is performed offline.

ML-based threat detection in NoC manycores often uses system-level simulators. Half employ offline inference [4], [11], [14] and omit overhead data. The rest use hardware-accelerated inference [6], [9], [13], reporting low overhead. However, these metrics, derived from gem5 modeling without RTL synthesis, lack confidence [15].

This work proposes an ML-based approach to **detect** traffic affected by HTs in NoC-based manycores using a **lightweight software runtime** and assessed in an **RTL-modeled platform**. This work targets threats, such as HTs, that can cause **subtle**, hard-to-detect traffic anomalies.

II. Pre-Runtime Training

This section presents our first *contribution*: an ML training flow to obtain a Regression Model for each application.

Manycores often use Communication Task Graphs (CTG) to model applications, where vertices correspond to tasks, and edges carry messages from a sender to a receiver task.

Figure 1 shows the latency of the messages of a given edge over time. The curve in green shows the edge with its vertices mapped at a distance of 1 hop (*h=1*). The curve in orange corresponds to the edge with four hops between communicating tasks (*h=4*). Note that the application edge has a *temporal signature* that does not change with different mappings, even if its average latency increases [16].

In Figure 1, the red curve shows the edge behavior with *h=1* but with an HT between its link (*h=1 w/ trojan*). Note that there are 2 points where the HT interfered with the message: (*i*) a disturbance at ≈4.2ms that cannot be detected by former solutions that relied solely on latency monitoring, as it would also classify the entire *h=4* line as affected by HT; and (*ii*) a spike at ≈10.3ms that is easy to detect.

The **threat model** used in this work involves HTs inserted at NoC links, which delay message arrivals. Every time a message passes the link, the HT has a 25% chance of delaying

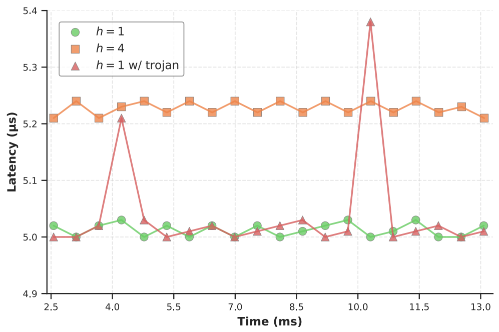

Fig. 1. Latency of messages over time for an edge.

it by 240 to 352 clock cycles. This combination of minor delays and randomized intervals increases the likelihood of interference while keeping the threat hard to detect, in contrast to more easily detectable attacks like flooding DoS [17], [18].

Our proposed approach employs XGBoost [19] to model the *temporal traffic signature* of applications. XGBoost was selected primarily due to its efficient inference process. In contrast to algorithms used in related work, such as ANN, GAN, and SVM, XGBoost demands significantly fewer arithmetic operations. This efficiency allows runtime threat detection without the need for dedicated hardware.

XGBoost, or eXtreme Gradient Boosting, is a gradient-boosting ensemble. In *gradient boosting*, base learners are built iteratively based on errors of the previous learners [20]; therefore, already correct predictions are not modified. XGBoost commonly uses Decision Trees (DT) as its base learners.

Figure 2 illustrates the XGBoost training process. The training begins by selecting a *bias*, the initial prediction $\hat{y}^0$ for all dataset targets y_i. The *residuals* r^i are used to build the XGBoost estimators instead of the targets y. A residual is the difference between a predicted value $\hat{y}_i$ and the corresponding actual value y_i.

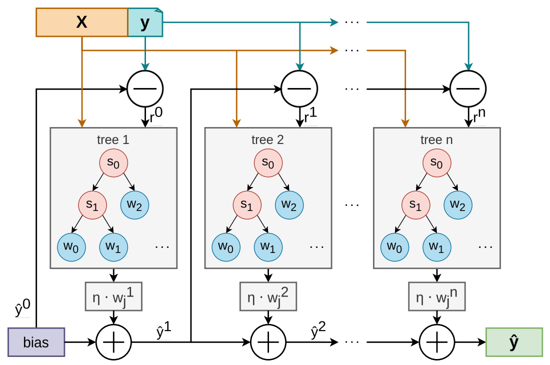

Fig. 2. XGBoost training flow. X: features. y: targets. $\hat{y}$: predicted values.

In each decision tree estimator (t^i), a node (j) can be a split (s_j) or a leaf containing a weight (w_j). In XGBoost, w_j of t^i (w_j^i) is computed to minimize a regularized loss function.

The weights of a tree w_j^i are multiplied by a learning rate η. The final tree prediction $\hat{y}^i$ is summed with the previous predictions, and residuals r^i are computed to build the next tree. This step is repeated to build n trees, controlled by a hyperparameter. The final prediction $\hat{y}$ is the sum of all individual tree predictions, weighted by η, with the initial bias.

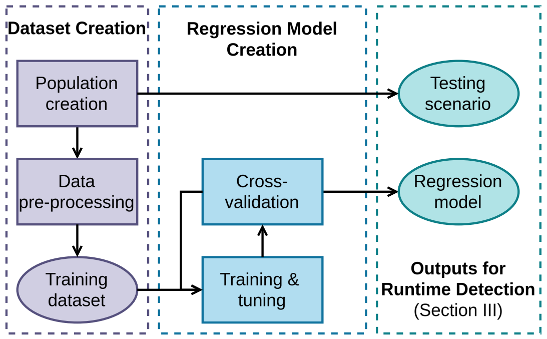

Fig. 3. Proposed machine learning flow (executed for each application).

Our proposed approach trains an XGBoost model at pre-runtime. The pre-runtime refers to the period before the system enters the runtime, during which applications are executed. Figure 3 shows the pre-runtime training flow, divided into two phases: (*i*) Dataset Creation; and (*ii*) Regression Model Creation. The third phase of the proposed approach, Runtime Detection (Section III), uses the outputs of the pre-runtime.

The *Population creation* defines the minimum system size to map an application, considering that a Processing Element (PE) can execute a single task and an overhead of three management tasks. All possible mapping permutations within the free space are generated, considering the management tasks statically mapped. These mapping permutations are randomly split into two, creating a training scenario with 75% of the mappings, and the *Testing scenario* with 25%. The training scenario is simulated in RTL without HTs inserted at its links.

In *Data pre-processing*, data is extracted from simulation logs and filtered to include only the target application, resulting in a *Training dataset* with four features: (*i*) the **timestamp** of a message, counted since the application starting time (Δt); (*ii*) an integer number of **hops** of the edge; (*iii*) the **edge ID**; and (*iv*) an integer **message size**, in flits. The target variable is the message end-to-end **latency**.

The Regression Model Creation begins by *Training* a model using the XGBRegressor Python API [21], fed by the Training dataset. The model is initially trained with default hyperparameters. As our objective is to provide lightweight runtime inference, we iteratively explore *tuning* the model size by decreasing the number of trees and the maximum tree depth, allowing the Mean Percentage Squared Error (*MAPE*) of the *cross-validation* to increase up to 10% from the default hyperparameters MAPE. This metric quantifies the mean percentage by which the prediction of the resulting *Regression model* deviates from the actual latency.

This method generates **one model per application**, identified by a *model ID*. We applied the proposed flow to create regression models for two applications: (*i*) **A/V**: Audio/Video decoding benchmark with two pipelines; and (*ii*) **MPEG**: a pipelined MPEG decoding application.

Table I presents results for the regression models. The 1st column lists the trained applications. The 2nd and 3rd columns list the number of tasks and edges for each application. The 4th and 5th columns show the tuned hyperparameters obtained from Regression Model Creation (Figure 3). The 6th column shows the average MAPE obtained from the training cross-validation. The model size (#estimators and Max. tree depth) needed to learn the application behavior is small compared to XGBoost defaults, which are 100 and 6. Furthermore, the average MAPE shows that the predicted latencies are, on average, less than 2.5% away from the correct ones.

TABLE I
PRE-RUNTIME TRAINING RESULTS.

App.	#tasks	#edges	#estimators	Max. tree depth	Avg. MAPE
A/V	7	7	**15**	**6**	0.023
MPEG	5	4	**17**	**4**	0.008

III. RUNTIME DETECTION

This section presents our second *original* contribution. The regression model built at pre-runtime must be integrated into the manycore to detect traffic affected by HTs. Integrating the model into the manycore is a twofold effort, as the model needs to be ported to the manycore system and the input features monitored. Figure 4 shows how the proposed flow is integrated into the manycore and tested.

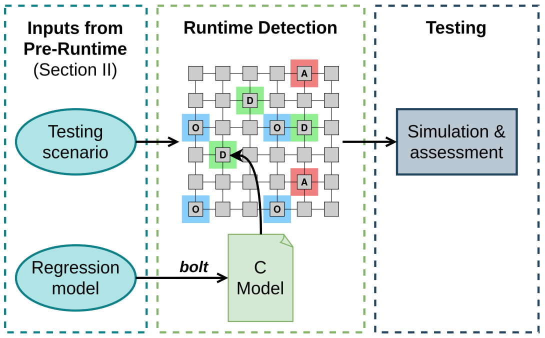

Fig. 4. Runtime detection and assessment. The model is inserted into a decision task, in green. Blue PEs are observation tasks, and red PEs are actuation tasks.

In Figure 4, the *Regression model* was trained using a Python API, leaving two options for deployment on the target platform: *a)* use the same Python API on the manycore, requiring all XGBoost dependencies and a Python interpreter; or *b)* convert the model into a standalone function in a compiled language. We selected the second option, converting the model to C, due to its lower dependency and memory requirements, making it suitable for lightweight runtime inference.

We created a tool called Boost Learning Transpiler (*bolt*) to convert the *Regression model* to a *C Model* (Figure 4). Unlike similar tools like *m2cgen* [22], *bolt* converts internal XGBoost data types to the corresponding C data type. XGBoost *indicators* are converted to *bool*, and both floating-point precision and integer size are kept from the original model.

Figure 5 lists a sample inference code converted to C, related to the training results depicted in Figure 2. Each tree t^i traverses its splits s_j, `ti_sj`, to assign the weight of one of its leaves w_j, `ti_wj`, to `wi`. In C, the splits (lines 2–3,11–12) are operations that compare a feature with a value, both learned during the training. Finally, the only arithmetic operation is the final accumulation of the weights with the bias (line 19). The learning rate (η), `eta`, is multiplied directly at the leaves at build-time.

```
1  float w1;
2  if (t1_s0) {
3      if (t1_s1)
4          w1 = eta * t1_w0;
5      else
6          w1 = eta * t1_w1;
7  } else {
8      w1 = eta * t1_w2;
9  }
```

```
10 float w2;
11 if (t2_s0) {
12     if (t2_s1)
13         w2 = eta * t2_w0;
14     else
15         w2 = eta * t2_w1;
16 } else {
17     w2 = eta * t2_w2;
18 }
```

```
19 prediction = bias + w1 + w2 + ... + wn;
```

Fig. 5. Sample XGBoost inference in C.

In Figure 5, the model mainly involves traversing the XGBoost trees using comparison operations (lines 2–9,11-18). Still, the final accumulation of the weights (line 19) is the most computation-intensive step, due to floating-point arithmetic. On a system without hardware floating-point, we identified that ≈75% of the inference time is spent in this accumulation.

Bolt can quantize leaf weights to integers to reduce accumulation overhead. It multiplies each leaf by a power-of-two constant, then shifts the accumulated result right by the constant's binary logarithm. This allows integer additions followed by a final shift, reducing inference time on systems without floating-point or multiplication support.

XGBoost represents its categorical data as multiple *indicators*, meaning that each category of a variable turns into an individual *bool* variable. *Bolt* decreases the number of input variables by collapsing the categorical indicators into a unique integer variable, modifying the splits to compare whether the resulting variable equals the category indicator (e.g., if it is equal to 1 for category number 1). Therefore, the number of input variables is decreased without modifying the inference result. The comparison overhead is unchanged, as comparing booleans or integers often uses the same instruction.

Once the *Regression model* is converted to the *C Model*, the Runtime Detection (Figure 4) starts by inserting the model into the manycore. This work adopts the Memphis-V manycore, publicly available at https://github.com/gaph-pucrs/Memphis-V. Memphis-V is modeled at RTL (SystemVerilog), with PEs consisting of: (*i*) an XY-routing wormhole data NoC [23]; (*ii*) a broadcast NoC for control messages [24]; (*iii*) a RISC-V RV32IMAC processor [25]; and (*iv*) a Network Interface with Direct Memory Access (DMA) [26].

The software stack comprises a small operating system kernel and management tasks. The management strategy is the ODA [27], with Observation (T_O), Decision (T_D), and Actuation (T_A) tasks executing in user space. A Low-Level Monitor (LLM) gathers monitored data from kernel space and sends it to a T_O [28]. A T_O broadcasts its monitoring goals at boot time, such as power, **security**, or QoS. Based on the monitored metric, the LLM constructs the monitoring message and sends it to the nearest compatible T_O.

For Runtime Detection (Figure 4), we monitor features at a **message** grain to reduce the NoC traffic. Whenever the OS receives a message from the NoC targeting a user task, it generates a monitoring message with the timestamp at which the monitored message was sent, its size in flits, the number of hops of its path, its edge ID, and its end-to-end message latency. The monitoring message is sent to the nearest T_O capable of monitoring *security* goals.

The T_D with the corresponding *model ID* receives the monitoring data and executes the *C Model* (Figure 4), resulting in a *predicted* message end-to-end latency. If the monitored latency exceeds the predicted latency by a configurable threshold (e.g., 5%), the monitored message is classified as affected by a security threat.

This approach enables runtime inference by parallelizing the computation and communication loads. The ODA strategy pipelines management. Multiple T_O can be strategically spread through the manycore to parallelize the monitoring communication and its computation. Each T_D implements a model for a single application, allowing the division of the model memory overhead and the inference load. Actuation (T_A tasks) is out of the scope of the present work, as our goal is threat detection.

The final step of Figure 4 consists of Testing the Runtime Detection. The *Testing scenario* from Section II, consisting of different mappings, has HTs inserted at every output port link of PEs with a *sender* task. The scenario is tested with RTL *Simulation*, with the system containing the proposed detection strategy. Detection results are extracted from simulation logs for *assessment*.

IV. Results

Table II presents the Runtime Detection performance metrics, considering the target platform executing at a clock frequency of 100 MHz. The 1st column lists the assessed applications. The 2nd column defines whether the model uses leaf quantization. The 3rd column presents the *inference* latency, i.e., the time needed to infer the model and then classify the message as whether or not affected by the security threat. The 4th column lists the *detection* latency, i.e., the time taken from the moment the HT delays the packet until the T_D detects it. The 5th column presents the increase in application execution time caused by the proposed strategy. Finally, the last column shows the memory footprint of the model.

TABLE II
Runtime detection performance metrics. LQ: Leaf Quantization.

App.	LQ	Avg. inf. latency	Avg. det. latency	Avg. exec. time inc.	Mem. footprint
A/V	✗	$34.6\mu s$	$225.8\mu s$	3.0%	$9.80KB$
	✓	$4.9\mu s$	$159.7\mu s$	3.0%	$10.78KB$
MPEG	✗	$39.0\mu s$	$183.3\mu s$	3.3%	$2.82KB$
	✓	$4.4\mu s$	$119.0\mu s$	3.3%	$2.75KB$

In Table II, the execution time increase occurs due to the overhead of monitoring messages, which increases NoC and CPU loads, but it is negligible. The low inference latency reveals that our approach is lightweight, and the low detection latency enables real-time threat detection. The memory footprint is a result of how big the model is (Table I), with more, denser trees (i.e., with more depth), resulting in a larger footprint. Nevertheless, the memory footprint is adequate for resource-constrained manycores in both applications.

The Leaf Quantization (LQ) in Table II shows a decrease of at least 85% in inference time, which translates to smaller detection latencies. The decrease for MPEG is bigger despite A/V having a denser model due to the additional trees of MPEG, which translates to floating-point additions. The memory footprint showed negligible difference for MPEG and an increase for A/V; however, in a system without hardware floating-point support, the required *soft float* functions, provided by GCC, resulted in an overhead of $3.76KB$. Despite lacking hardware floating-point, Memphis-V does not need these libraries when using quantized leaves, as the model's input features are integers.

Table III shows the predictive performance of the Runtime Detection. The 1st column lists the assessed applications. The 2nd column defines whether the model uses leaf quantization. The 3rd column presents the *recall*. Recall is the model's ability to detect instances of traffic affected by an HT. However, a high recall could mean the model flags all messages as affected by the HT. Therefore, the 4th column presents the precision, which considers how often the HT is *correctly* detected. A well-tuned model should present a good balance between recall and precision, and this trade-off is given by the *F1-score* in the 5th column, which is the harmonic mean of the two other metrics.

TABLE III
Runtime detection predictive performance. LQ: Leaf Quantization.

App.	LQ	Recall	Precision	F1-Score
A/V	✗	91.9%	78.5%	84.6%
	✓	91.8%	78.4%	84.6%
MPEG	✗	91.8%	100 %	95.7%
	✓	91.8%	100 %	95.7%

Given the high recall in Table III, our approach shows high detection rates. The precision varies from high in MPEG to acceptable for A/V. A/V's lower precision results from its more frequent message exchange, which triggers more monitoring messages and worse traffic crossings that can occur on some mappings. These two problems can lead to increased NoC congestion, which increases the message latency, hence wrongly detecting it as affected by a threat. Leaf quantization shows negligible predictive performance loss.

V. Conclusions and Future Work

We presented a flow for training an XGBoost model in pre-runtime that learns an application temporal signature in a NoC-based manycore. This learned model is inserted in the manycore and, with an ODA approach, enables runtime detection of traffic affected by an HT. It shows a high detection rate across realistic benchmarks. The XGBoost model presented low overhead in terms of performance and memory. Training data and runtime performance assessment are obtained by RTL simulations with clock cycle accuracy. Our noninvasive runtime detection does not require modifications to the existing manycore architecture; it only requires monitoring capabilities and software support.

Future work include: (*i*) simplify the monitoring process to reduce the NoC load, hence decreasing execution time overhead and detection latency; (*ii*) engineer new features to account for potential traffic crossings in an application mapping.

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), 305621/2024-6; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), 23/2551-0002200-1.

References

[1] V. Gohil, H. Guo, S. Patnaik, and J. Rajendran, "ATTRITION: Attacking Static Hardware Trojan Detection Techniques Using Reinforcement Learning," in *CCS*, 2022, pp. 1275–1289, https://doi.org/10.1145/3548606.3560690.

[2] H. Weerasena and P. Mishra, "Breaking On-Chip Communication Anonymity Using Flow Correlation Attacks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 20, no. 4, pp. 12:1–12:27, 2024, https://doi.org/10.1145/3677034.

[3] J. Rajesh, D. M. Ancajas, K. Chakraborty, and S. Roy, "Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip," in *NOCS*, 2015, pp. 1–8, https://doi.org/10.1145/2786572.2786580.

[4] 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, https://doi.org/10.1109/ITAIC49862.2020.9338861.

[5] "OPNET Network Simulator." [Online]. Available: https://opnetprojects.com/opnet-network-simulator

[6] 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, https://doi.org/10.1109/MM.2020.3003576.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1—-7, 2011, https://doi.org/10.1145/2024716.2024718.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008, pp. 72–81, https://doi.org/10.1145/1454115.1454128.

[9] 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, https://doi.org/10.1109/JETCAS.2021.3083289.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54, https://doi.org/10.1109/IISWC.2009.5306797.

[11] 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, https://doi.org/10.1145/3479876.3481589.

[12] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009, pp. 33–42, https://doi.org/10.1109/ISPASS.2009.4919636.

[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, https://doi.org/10.23919/DATE54114.2022.9774693.

[14] 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, https://doi.org/10.1109/ISCAS46773.2023.10182218.

[15] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *ReCoSoC*, 2012, pp. 1–6, https://doi.org/10.1109/ReCoSoC.2012.6322869.

[16] A. E. Dalzotto and F. G. Moraes, "A Machine Learning Approach for Traffic Anomaly Detection in NoC-based Manycores," in *SBCCI*, 2024, pp. 1–5, https://doi.org/10.1109/SBCCI62366.2024.10703991.

[17] C. G. Chaves, S. P. Azad, T. Hollstein, and J. Sepúlveda, "DoS attack detection and path collision localization in NoC-based MpsoC architectures," *Journal of Low Power Electronics and Applications*, vol. 9, no. 1, p. 7, 2019, https://doi.org/10.3390/jlpea9010007.

[18] S. Charles and P. Mishra, "A Survey of Network-on-Chip Security Attacks and Countermeasures," *ACM Computing Surveys*, vol. 54, no. 5, pp. 101:1–101:36, 2022, https://doi.org/10.1145/3450964.

[19] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *SIGKDD*, 2016, pp. 785–795, https://doi.org/10.1145/2939672.2939785.

[20] C. Wade, *Hands-On Gradient Boosting with XGBoost and scikit-learn: Perform accessible machine learning and extreme gradient boosting with Python*, 1st ed. Packt, 2020, 310 pages.

[21] "XGBoost: eXtreme Gradient Boosting." [Online]. Available: https://github.com/dmlc/xgboost

[22] "m2cgen: Model 2 Code Generator." [Online]. Available: https://github.com/BayesWitnesses/m2cgen

[23] F. G. Moraes, N. L. V. Calazans, A. V. Mello, L. H. Möller, and L. C. Ost, "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip," *Integration, the VLSI journal*, vol. 38, no. 1, pp. 69–93, 2004, https://doi.org/10.1016/j.vlsi.2004.03.003.

[24] E. Wachter, L. L. Caimi, V. Fochi, D. Munhoz, and F. G. Moraes, "BrNoC: A broadcast NoC for control messages in many-core systems," *Microelectronics Journal*, vol. 68, pp. 69–77, 2017, https://doi.org/10.1016/j.mejo.2017.08.010.

[25] W. A. Nunes, A. E. Dal Zotto, and F. G. Moraes, "RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems," in *LASCAS*, 2024, pp. 1–5, https://doi.org/10.1109/LASCAS60203.2024.10506171.

[26] 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.

[27] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "A generalized software framework for accurate and efficient management of performance goals," in *EMSOFT*, 2013, pp. 1–10, https://doi.org/10.1109/EMSOFT.2013.6658597.

[28] A. E. Dalzotto, C. d. S. Borges, M. Ruaro, and F. G. Moraes, "Non-intrusive Monitoring Framework for NoC-based Many-Cores," in *SBESC*, 2022, pp. 1–7, https://doi.org/10.1109/SBESC56799.2022.9965177.