PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

ANGELO ELIAS DAL ZOTTO

A MACHINE LEARNING APPROACH FOR REAL-TIME TRAFFIC ANOMALY DETECTION IN NOC-BASED MCSOCS

Porto Alegre
2026

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

A MACHINE LEARNING APPROACH FOR REAL-TIME TRAFFIC ANOMALY DETECTION IN NOC-BASED MCSOCS

ANGELO ELIAS DAL ZOTTO

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2026

FICHA CATALOGRÁFICA

ANGELO ELIAS DAL ZOTTO

A MACHINE LEARNING APPROACH FOR REAL-TIME TRAFFIC ANOMALY DETECTION IN NOC-BASED MCSOCS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March, 2026.

COMMITTEE MEMBERS:

Profa. Dra. Luiza de Macedo Mourelle (PEL/UERJ)

Prof. Dr. Ney Laert Vilar Calazans (PPGEEL/UFSC)

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

UMA ABORDAGEM DE APRENDIZADO DE MÁQUINA PARA DETECÇÃO DE ANOMALIAS DE TRÁFEGO EM TEMPO REAL EM MCSOCS BASEADOS EM NOC

RESUMO

Sistemas em chip *Many-Core* (MCSoCs – *Many-Core Systems-on-Chip*) podem ter milhares de processadores interconectados por redes intra-chip, oferecendo benefícios em consumo de potência e em área de chip, permitindo escalabilidade. MCSoCs estão presentes em dispositivos críticos, incluindo sistemas automotivos, médicos e industriais. Consequentemente, uma falha de segurança em um MCSoC pode causar danos físicos ou falhas em infraestruturas críticas. Tais falhas podem ter origem em hardware, decorrentes da integração de propriedades intelectuais de terceiros ao chip, ou em ataques à cadeia logística, que podem levar à inclusão de cavalos de Tróia de hardware (HT – *Hardware Trojan*), ou em software, decorrentes de aplicações maliciosas ou com mau funcionamento. Um MCSoC comprometido pode violar a confidencialidade, a integridade e a disponibilidade da informação. Esta Tese propõe uma estrutura de detecção de ameaças de segurança leve, de tempo real e não invasiva baseada em Aprendizado de Máquina (AM). Essa estrutura é desenvolvida e avaliada com a plataforma MCSoC sintetizável Memphis-V, o que permite mitigar as limitações identificadas no estado da arte quanto à aplicabilidade e à confiança nos métodos de detecção de ameaças de segurança em MCSoCs. Tais limitações decorrem do uso de técnicas custosas que inviabilizam a aplicação de AM em MCSoCs e de simuladores de alto nível e de tráfego sintético que podem não representar com precisão as reais características dos MCSoCs. A estrutura de detecção de ameaças proposta é implementada por meio de um fluxo de AM que define a criação do dataset e dos modelos de AM, a conversão desses modelos em código C e a integração desse código na plataforma MCSoC por meio de um mecanismo de monitoramento que permite detectar ameaças em tempo de execução. Os resultados experimentais demonstram escalabilidade e alto desempenho de detecção contra HTs furtivos, alcançando uma pontuação F1 superior a 80%. A estrutura proposta reduz a latência de inferência por até 89,4% através de quantização, permitindo detecção em tempo real com latências abaixo de 61 μs e sobrecarga negligenciável no desempenho do sistema. Esta Tese demonstra que é viável detectar ameaças de segurança em tempo real sem comprometer o desempenho ou a escalabilidade de um MCSoC.

Palavras-Chave: Sistemas em Chip Many-Core (MCSoC); Redes Intra-Chip (NoC); Aprendizado de Máquina (AM); Detecção de Ameaças de Segurança; Cavalo de Troia de Hardware; XGBoost.

A MACHINE LEARNING APPROACH FOR REAL-TIME TRAFFIC ANOMALY DETECTION IN NOC-BASED MCSOCS

ABSTRACT

Many-Core Systems-on-Chip (MCSoCs) can have thousands of processors interconnected by networks-on-chip, offering benefits in power consumption and chip area that enable scalability. MCSoCs are present in critical devices, including automotive, medical, and industrial systems. Consequently, a security failure in an MCSoC can cause physical damage or failures in critical infrastructures. Such failures can originate in hardware, resulting from the integration of third-party intellectual property into the chip or from supply chain attacks, which can lead to the inclusion of Hardware Trojans (HTs), or in software, resulting from malicious or malfunctioning applications. A compromised MCSoC can violate the confidentiality, integrity, and availability of information. This Thesis proposes a lightweight, real-time, and non-invasive security threat detection framework based on Machine Learning (ML). This framework is developed and evaluated with the Memphis-V synthesizable MCSoC platform, which allows mitigating the limitations identified in the state-of-the-art regarding applicability and confidence in security threat detection methods in MCSoCs. Such limitations arise from the use of costly techniques that make the application of ML in MCSoCs infeasible, and from high-level simulators and synthetic traffic that may not accurately represent the real characteristics of MCSoCs. The proposed threat detection framework is implemented through an ML flow that defines the creation of the dataset and ML models, the conversion of these models into C code, and the integration of this code into the MCSoC platform through a monitoring mechanism that allows detecting threats at runtime. Experimental results demonstrate scalability and high detection performance against stealthy HTs, reaching an F1-score above 80%. The proposed framework reduces inference latency by up to 89.4% through quantization, allowing real-time detection with latencies below 61 μs and negligible overhead on system performance. This Thesis demonstrates that it is feasible to detect security threats in real-time without compromising the performance or scalability of an MCSoC.

Keywords: Many-Core Systems-on-Chip (MCSoC); Network-on-Chip (NoC); Machine Learning (ML); Security Threat Detection; Hardware Trojan; XGBoost.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

3PIP – Third-Party Intellectual Property

A/V – Audio/Video

AE – Actuation Enforcer

AES – Advanced Encryption Standard

AI – Artificial Intelligence

ALU – Arithmetic Logic Unit

ANN – Artificial Neural Network

API – Application Programming Interface

BOLT – Boost Learning Transpiler

CSR – Control and Status Register

CTG – Communication Task Graph

CTU – Central TampML Unit

D-MEM – Data Memory

DFC – Distributed Flit Collector

DIV – Division

DMA – Direct Memory Access

DMNI – Direct Memory Network Interface

DNN – Deep Neural Network

DOS – Denial of Service

DT – Decision Tree

DTW – Dynamic Time Warping

ECC – Error-Correction Code

FF – Flip-Flop

FFT – Fast Fourier Transform

FIFO – First-In-First-Out

FN – False Negative

FP – False Positive

FPGA – Field-Programmable Gate Array

FPR – False Positive Rate

FPU – Floating-Point Unit

GAN – Generative Adversarial Network

GNN – Graph Neural Network

GPR – General Purpose Register

HAL – Hardware Abstraction Layer

HT – Hardware Trojan

I-MEM – Instruction Memory

I/O – Input/Output

IBUF – Instruction Buffer

ID – Instruction Decode

ILP – Instruction-Level Parallelism

IP – Intellectual Property

IPC – Inter-Process Communication

IOT – Internet of Things

ISA – Instruction Set Architecture

ISE – Instruction Set Extension

JSON – JavaScript Object Notation

KNN – K-Nearest Neighbors

LIF – Leaky Integrate and Fire

LLM – Low-Level Monitor

LQ – Leaf Quantization

LR – Linear Regression

LRU – Least Recently Used

LS – Load/Store

LST – Least Slack Time

LSTM – Long Short-Term Memory

LUT – Look-Up Table

M-MODE – Machine mode

MA – Management Application

MCSOC – Many-Core System-on-Chip

MEMPHIS-V – Many-core Modeling Platform for Phivers

MIP – Malicious Intellectual Property

ML – Machine Learning

MLP – Multi-Layer Perceptron

MPEG – Moving Picture Experts Group

MMR – Memory-Mapped Register

MMU – Memory Management Unit

MPI – Message Passing Interface

MPN – Multiple Physical Networks

MSE – Mean Squared Error

MTIMER – Machine Timer

MUL – Multiplication

NBC – Naïve Bayes Classifier

NI – Network Interface

NOC – Network-on-Chip

ODA – Observe-Decide-Act

OS – Operating System

PAP – Proximal Analogous Packet

PE – Processing Element

PHIVERS – Processor Hive for RS5

PLIC – Peripheral Local Interrupt Controller

PMA – Physical Memory Address

PMP – Physical Memory Protection

QOS – Quality of Service

RBF – Radial Basis Function

RELU – Rectified Linear Output Unit

RF – Random Forest

RLAN – Runtime Latency Auditor for NoCs

RMSE – Root Mean Squared Error

RNN – Recurrent Neural Network

RT – Real-Time

RTOS – Real-Time Operating System

RTL – Register-Transfer Level

S-MODE – Supervisor mode

SCA – Side-Channel Attack

SNN – Spiking Neural Network

SOC – System-on-Chip

SVM – Support Vector Machine

TLB – Translation Look-aside Buffer

TN – True Negative

TP – True Positive

U-MODE – User mode

VC – Virtual Channel

VMA – Virtual Memory Address

WI – Wireless Interface

WINOC – Wireless Network-on-Chip

XOSVM – Custom extension for offset and size virtual memory

XU – Execution Unit

CONTENTS

1. INTRODUCTION

A Many-Core System-on-Chip (MCSoC) is a System-on-Chip (SoC) with simple processors replicated multiple times, rather than a few complex processors, which offers benefits in power consumption and chip surface area [Asanovic et al., 2009]. They address the Power Wall, Memory Wall, and Instruction-Level Parallelism (ILP) Wall [Manferdelli et al., 2008]. ET-SoC-1 [Ditzel et al., 2021] is an example of an MCSoC that can house over a thousand cores, or Processing Elements (PE), on a single chip. A broad spectrum of electronic devices uses these systems, from computers and tablets to routers, webcams, household appliances, and self-driving cars. These systems are intrinsic to modern interconnected environments, particularly in Internet of Things (IoT) networks.

MCSoCs are often interconnected by a Network-on-Chip (NoC). NoC is an on-chip communication infrastructure that interconnects PEs, decoupling computation from communication. The NoC structure contains routers and links. Routers implement network control logic that defines the path for each packet exchanged between a source and target PE, routing it from an input port to an output port. Links connect routers and connect each router to its local PE. NoC architectures offer a significant advantage over traditional bus systems due to their superior scalability. As the number of cores in a system increases, NoCs provide more efficient, manageable communication, thereby minimizing the bottlenecks common in bus architectures, especially in MCSoCs.

Historically, security in computing systems has been a software-centric theme, with estimates from 2005 indicating that 80% of embedded systems security flaws originated in software [Coburn et al., 2005]. However, MCSoC interconnectivity renders it susceptible to various forms of cyberattacks, such as malware, ransomware, and Denial of Service (DoS), as well as hardware-specific threats like Side-Channel Attacks (SCA) [Moghimi, 2023], hardware bugs [Ormandy, 2023], and Hardware Trojans (HTs) [Xue et al., 2020]. The use of Third-Party Intellectual Property (3PIP) cores to address design complexity and time-to-market pressures further exacerbates these vulnerabilities. Consequently, security is critical in the design and deployment of MCSoCs.

The literature offers strategies for security threat detection and its localization [Charles and Mishra, 2021], as well as corresponding countermeasures [Faccenda et al., 2021], in MCSoCs. Threat detection in NoC-based MCSoCs, the focus of this Thesis, is challenging due to its complex, dynamic nature, making it difficult for traditional rule-based approaches to effectively identify anomalies or potential threats [Charles et al., 2020].

Machine Learning (ML) emerges as a suitable solution for threat detection in MCSoCs due to its ability to discern subtle, nonlinear patterns. ML is a field of Artificial Intelligence (AI) that enables machines to learn from data and improve their performance on tasks without being explicitly programmed, allowing them to adapt to NoC behavior and potentially identify previously unseen threats.

Classical ML algorithms have been extensively explored in the literature to support threat detection in MCSoCs [Kulkarni et al., 2016]. More recently, two ML techniques for threat detection in MCSoCs have emerged: ensemble learners [Yao et al., 2020] and neural networks [Wang et al., 2020]. Both solutions offer better detection performance than classical single-learner models. Still, the overhead of learning models in MCSoCs is often neglected in the literature [Hu et al., 2023; Hathal et al., 2024; Wang and Halak, 2025].

1.1 Motivation

Demonstrating the feasibility of ML models for threat detection in MCSoCs is a key aspect of this Thesis. The motivation is the recognition of limitations found in early work on NoC threat detection [Rajesh et al., 2015] and in work using ML for this purpose [Sudusinghe et al., 2022; Wang et al., 2024].

ML is a potential solution for threat detection in MCSoCs through traffic profiling, as it can account for variations in mapping and workloads that classical algorithms cannot handle. However, the state-of-the-art ML techniques for threat detection in NoCs and NoC-based MCSoCs have limitations in applicability and confidence.

The first limitation is applicability, arising from the assumption that ML techniques are applied either with centralized engines or with costly techniques, such as Deep Neural Networks (DNNs), replicated on every router of the NoC. Centralized solutions limit the system's scalability, while replicated and costly techniques hinder practical implementation. The second limitation is confidence, which is influenced by the adoption of synthetic NoC traffic that does not accurately reflect real application scenarios, as well as by the use of high-level simulations that may not accurately reflect the actual behavior of MCSoCs.

This research advances the state-of-the-art by moving the assessment of NoC security from abstract, simulation-based environments to a high-fidelity Register-Transfer Level (RTL) model. This Thesis demonstrates the feasibility of a non-invasive, ML-based detection approach under realistic hardware constraints and application workloads, providing an analysis of the trade-offs between security monitoring and system performance.

1.2 Thesis Statement

This Thesis demonstrates that it is feasible to detect **subtle security threats** that cause traffic anomalies through a **lightweight**, **real-time**, and **non-invasive** Machine Learning approach, by profiling the behavior of real applications on a **synthesizable platform** modeled in RTL to achieve **trustworthy results** and mitigate the limitations in applicability and confidence found in the state-of-the-art.

1.3 Objectives

The primary objective of this Thesis is to demonstrate the feasibility of machine learning models for detecting security threats in MCSoCs, considering variations in task mapping and workload.

The *specific* goals of the Thesis include:

G1. Evaluate the feasibility of machine learning models to detect threats in MCSoCs;

G2. Develop such models to accurately detect threats, considering mapping variations correctly;

G3. Propose a monitoring technique considering the model constraints and the performance trade-offs in an RTL-modeled MCSoC;

G4. Insert the proposed model into the RTL-modeled MCSoC;

G5. Assess the model in runtime.

1.4 Scope of the Thesis

The primary focus of this work is detecting traffic anomalies that indicate security threats in MCSoCs. The scope of this Thesis encompasses a system for monitoring application behavior and detecting traffic anomalies. The considered threats can originate from either hardware (e.g., Hardware Trojans) or software (e.g., malicious applications), as long as their malicious activity manifests as a disturbance in the NoC traffic, thereby affecting the performance of a legitimate application.

It is **not** the goal of the Thesis to specify the attack type, as DoS (e.g., temporary traffic blocking), eavesdropping (e.g., an HT sending sensitive data to a malicious application), or misrouting (e.g., unexpected traffic in a given link). The goal is to detect traffic anomalies.

The development and implementation of countermeasures to mitigate the detected attacks are **not** objectives of this Thesis. The work focuses exclusively on the accurate and efficient identification of the threat, leaving the proposal of countermeasures as future work.

1.5 Original Contributions

The original contributions of this Thesis are:

1. **An RTL-modeled MCSoC evaluation platform:** The development of the Memphis-V platform, a complete MCSoC modeled in synthesizable SystemVerilog. The platform includes RISC-V processors, NoCs, and a complete software stack with an Operating System (OS), enabling the execution of realistic benchmarks. This contribution directly addresses the limitations in confidence of the state-of-the-art by providing a high-fidelity environment for evaluating security threats.

2. **A lightweight, real-time, and non-invasive security threat detection approach:** An ML-based framework to detect security threats at runtime, i.e., the framework is executing on the MCSoC alongside applications. The approach is non-invasive, requiring no modifications to the NoC routers. Furthermore, it operates with low computational and monitoring overhead and a low memory footprint (lightweight). This enables real-time security threat detection, i.e., with detection latency below the typical embedded RTOS (Real-Time Operating System) timer period, e.g., between 1–10 ms [Barry and The FreeRTOS Team, 2024]. It is evaluated on the Memphis-V platform using real-world application benchmarks, thereby mitigating the limitations in applicability of existing solutions.

1.6 Document Organization

The remainder of this document is organized as follows. Chapter 2 briefly describes the ML and MCSoC background required for this Thesis. Chapter 3 presents the state-of-art in NoC and MCSoC threat detection. Chapter 4 describes the first contribution of this Thesis, the Memphis-V MCSoC. The next Chapters detail the second contribution of this Thesis. Chapter 5 presents an overview of the proposed framework that uses ML to detect threats on the MCSoC. Chapter 6 describes the process of generating data to train and test the ML model. Chapter 7 details the model training and its integration into the target MCSoC. Chapter 8 presents the experimental results. Finally, Chapter 9 concludes this work and points out directions for future work.

2. BACKGROUND KNOWLEDGE

This Chapter presents the background knowledge required for this Thesis, briefly explaining the ML foundations needed to meet the objectives detailed in Chapter 1, as well as the architectural assumptions of MCSoCs.

An ML algorithm can be *supervised* or *unsupervised*. In supervised learning, an algorithm builds a model to predict an output from one or more inputs. Unsupervised learning has no supervising output. Thus, it learns the relationships and structures from the data [James et al., 2023].

Supervised learning can be divided by the characteristics of the target variable it is trying to predict. Problems with quantitative responses are referred to as *regression* problems, while problems with qualitative responses are *classification* problems [James et al., 2023]. Most ML algorithms for regression in this Chapter have variants for classification and vice versa.

The input for an ML algorithm is a *dataset*, i.e., a collection of data objects [Tan et al., 2019]. A dataset is represented as a matrix, where each row corresponds to an observation of an event, also called *instance*. An instance can contain multiple *attributes*, also called features, that capture the characteristics of an object. The attributes are represented by the dataset columns, with the last column containing the *target*, i.e., the value the ML algorithm aims to learn. A row containing all attributes without the target is called a *feature vector*.

Sections 2.1-2.6 explore specific supervised learning paradigms, as they are used in the literature to predict whether an MCSoC is affected by a security threat:

- Section 2.1: instance-based learning;
- Section 2.2: probabilistic learning;
- Section 2.3: search-based learning;
- Section 2.4: optimization-based learning;
- Section 2.5: ensemble learners;
- Section 2.6: XGBoost, the algorithm chosen for this Thesis.

Section 2.7 briefly explains the metrics commonly used to evaluate an ML model predictive performance. Section 2.8 summarizes the MCSoC requirements for deploying the presented ML algorithms. Section 2.9 presents the MCSoC architectural assumptions and threat model. Finally, Section 2.10 summarizes the Chapter with final remarks regarding the application of ML for security threat detection in MCSoCs.

2.1 Instance-based learning

Instance-based learning involves delaying the modeling of the training data until it is needed to classify the test instances, thus making it a type of *lazy learner* [Tan et al., 2019]. Algorithms in this category make decisions based on the similarity between new data points and instances seen during training. A typical example of instance-based learning is the **K-Nearest Neighbors (KNN)** algorithm [Cover and Hart, 1967], where predictions are made based on the similarity of a given instance to its nearest neighbors in the training set.

KNN classification is straightforward. Figure 2.1 pictures an instance ($\times$) of a 2-dimensional KNN model with $k = 3$, denoted by 3 neighbor instances (+ and $-$) inside the dashed circle. In KNN, an instance is represented as a data point in a multidimensional space, where the number of attributes dictates the number of dimensions. The similarity of each test instance, e.g., Euclidean distance, is computed for all its neighbors. The classification for the test instance is based on the class of its k-nearest neighbors, where k is specified by the designer.

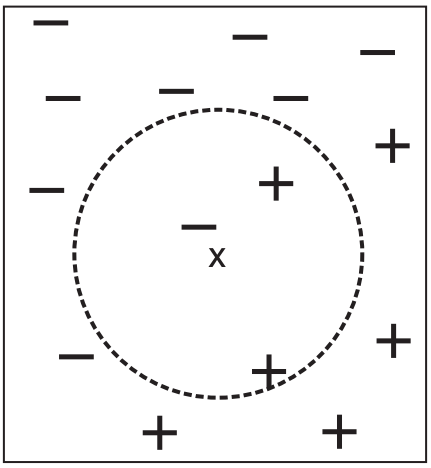


Figure 2.1: Example of 2-dimensional KNN with $k = 3$. Source: Tan et al. [2019].

In Figure 2.1, the 3-nearest neighbors of instance $\times$ are circled by a dashed line. Among these three neighbors, two are from the + class, and one is from the $-$ class. Therefore, with a majority vote, $\times$ is classified as belonging to the + class.

The classification performance of KNN is strongly influenced by the value of k, the distance metric used, and the number of attributes [Faceli et al., 2011]. Nevertheless, lazy learners maintain the training data in memory to compute the similarities for all training instances when classifying a test instance. Therefore, it is inefficient for embedded systems due to their usual small memory capacity.

2.2 Probabilistic Learning

Probabilistic methods focus on modeling uncertainty and probability distributions within the data, providing a degree of confidence for predictions. Probabilistic methods use Bayes' theorem, which provides the probability of an event based on prior knowledge of conditions that may be related to it [Faceli et al., 2011].

Bayesian probability provides the statistical foundation for computing probabilities when evidence from multiple sources must be combined with prior beliefs. Bayes' theorem is presented in Equation (2.1), stating the relationship between the conditional probabilities $P(Y|X)$, the posterior probability, and $P(X|Y)$, the likelihood. $P(X)$, the evidence, and $P(Y)$, the prior probability, are the probabilities solely in terms of X and Y, respectively.

$$P(Y|X) = \frac{P(X|Y) \times P(Y)}{P(X)} \tag{2.1}$$

A well-known probabilistic learning algorithm is the **Naïve Bayes Classifier (NBC)**. NBC uses a naïve approach to the multidimensional problem, assuming that the attributes are conditionally independent, to reduce the computational cost of conditional probabilities [Tan et al., 2019]. Equation (2.2) presents the conditional probability used by NBC as the product of the conditional probability for every attribute x_i in d dimensions.

$$P(x|y) = \prod_{i=1}^{d} P(x_i|y) \tag{2.2}$$

Equation (2.2) makes possible to estimate only $P(x_i|y)$ conditional probabilities separately. NBC computes $P(y|x)$, the probability of observing a class y in an instance of attributes x. Thus, the probability of a test instance x belonging to a class y is given by Equation (2.3). This equation is applied to every class for a given test instance. The class that produces the highest probability is the one classified by NBC.

$$P(y|x) \propto P(y) \times \prod_{i=1}^{d} P(x_i|y) \tag{2.3}$$

Probabilistic learning methods, such as NBC, are widely used for forecasting an outcome given input data and diagnosis, aiming to identify the most likely cause of the observed effects [Faceli et al., 2011]. Due to its probabilistic characteristics, NBC is robust to irrelevant attributes and isolated noise points, but correlated attributes can harm the classifier's performance [Tan et al., 2019].

2.3 Search-based learning

One way to build an ML model is to view it as a search problem in a solution space, which is often solved using **Decision Trees (DT)** [Faceli et al., 2011]. DT is a divide-and-conquer strategy. Therefore, the search problem is divided into subproblems, and a tree combines all subsolutions. This tree is an acyclic directed graph, where each node represents a *split* that leads to another node, which may be another split or a *leaf*.

Figure 2.2 shows an example of a decision tree to classify two different classes. The starting node of the tree is called *root* node, and each leaf provides the classification result. Root and internal nodes, i.e., every node that *isn't* a leaf node, test an attribute of a dataset and take the edge corresponding to the test result. Note that in Figure 2.2, there is only one internal node (ATT X). The depth of the tree presented in the Figure is 2.

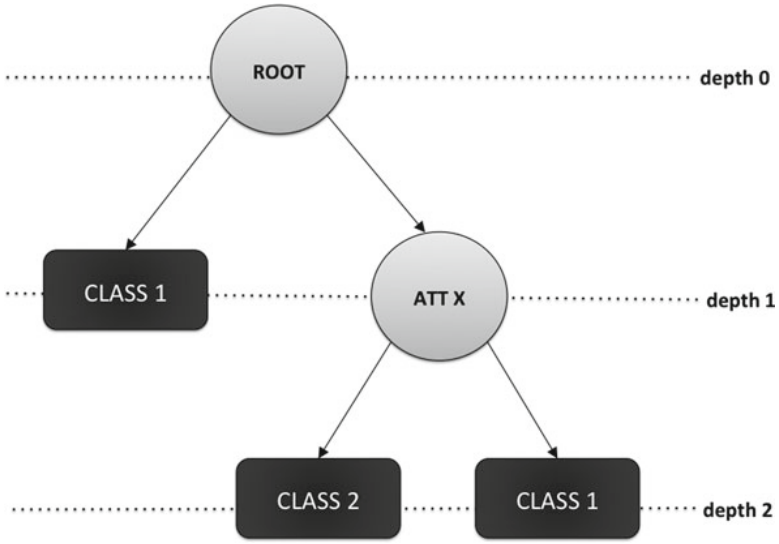


Figure 2.2: Example of a two-level deep decision tree with two nodes and three leaves. Source: Barros et al. [2015].

A dataset can build multiple different decision trees. The challenge is to build a well-performing decision tree, which involves making design decisions about how the tree is induced, the split criterion, and the stopping criteria for tree splits. A common approach uses a greedy top-down induction [Barros et al., 2015].

Hunt's algorithm for DT induction is the basis for top-down greedy approaches. It consists of two steps applied recursively: if the stop criterion is met, the tree node is a leaf and labeled with the target class. Otherwise, an attribute is selected for a split, splitting the tree into two nodes.

Standard stopping criteria are: class homogeneity, when all node instances belong to the same class; tree depth, when the tree reaches a maximum stipulated depth; and failing to reach a threshold for the splitting criterion [Barros et al., 2015].

The splitting criterion determines which attributes to use for splitting a node into two child nodes. The Gini index, Equation (2.4), is a common criterion for splitting that measures the node impurity by computing the squared probabilities of a class i appearing at the node for all K classes.

$$G = 1 - \sum_{i=1}^{K} p_i^2 \tag{2.4}$$

An attribute chosen for the split must have all its resulting nodes computed for the Gini index. The weighted Gini index, Equation (2.5), measures the impurity of the generated split. It is defined as the weighted average of the Gini indices at each node j resulting from the split, where the weight is proportional to the ratio of the number of elements n_j in the child node to the original number of elements n in the parent node before the split. The attribute that produces the lowest impurity is selected for the split.

$$G_S = \sum_{j=1}^{m} \frac{n_j}{n} \times G_j \tag{2.5}$$

Decision trees are prone to overfitting, which occurs when a model fits the training data too well, often leading to poor generalization and poor performance on unseen data. Therefore, decision trees are limited because they cannot grow too much without overfitting. A way to avoid overfitting is to reduce the tree size by using *pruning* techniques [Barros et al., 2015]. Another way to avoid overfitting in decision trees and maintain their performance is to aggregate many smaller trees, creating an ensemble [Wade, 2020].

2.4 Optimization-based learning

Optimization-based learning refers to a class of ML methods that optimize specific parameters or functions to improve model performance [Faceli et al., 2011]. Optimization typically involves defining a loss or objective function that quantifies the model's performance on a given task.

Linear Regression (LR) is an optimization-based linear model. LR builds an approximate function $\hat{y} = \hat{\beta}_0 + \sum_{i=1}^{d} \hat{\beta}_i x$, where each attribute x_i is multiplied by a learned parameter β_i. The objective is to minimize the error generated from the difference between the training data points and the approximate function [James et al., 2023]. The error is usually measured using the Mean Squared Error (MSE) [Abu-Mostafa et al., 2012]. **Logistic regression** is similar to LR, but it bounds its output to a probability between 0 and 1. Therefore, it is useful for classification [Abu-Mostafa et al., 2012].

Support Vector Machine (SVM) can learn nonlinear decision boundaries in an attribute space [Tan et al., 2019]. Figure 2.3 illustrates how SVM works by finding the optimal hyperplane, the black line, that best separates different classes while maximizing the margins. A margin is the distance between the hyperplane and the nearest data points of each class, called support vectors, with arrows drawn in the Figure. Although building a hyperplane, SVM can handle non-linearly separable data by mapping the input space into a higher-dimensional feature space using *kernels*. Usually, the used kernel is the *Gaussian*, also known as the Radial Basis Function (RBF).

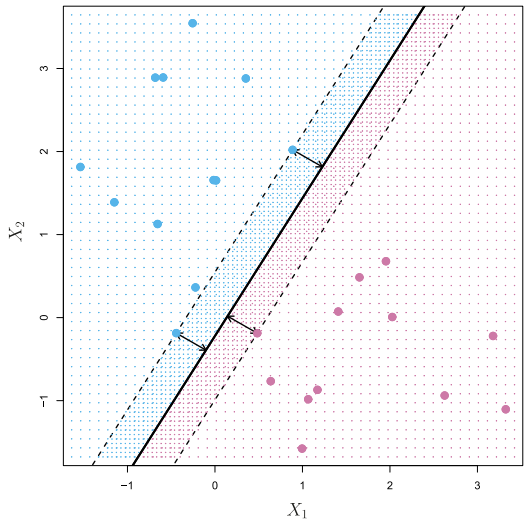


Figure 2.3: Separating hyperplane for X_1 and X_2 attributes with highest margins from the nearest data points. Source: James et al. [2023].

A **Perceptron** is a basic Artificial Neural Network (ANN). Figure 2.4 pictures the Perceptron architecture, composed of three inputs x_1 to x_3, a bias b, and an output y. The weighted links represent the synaptic connection between the inputs and the neuron.

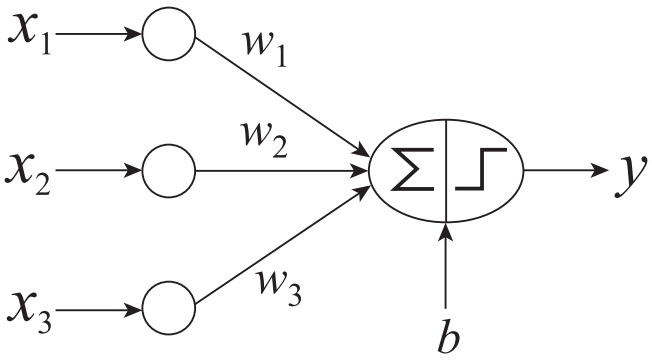


Figure 2.4: Perceptron with 3 inputs x and a bias b. Source: Tan et al. [2019].

The output of the Perceptron is given by the weighted sum of the inputs with a *bias* factor. This result is used by an *activation function*, which is usually the signal in a Perceptron, yielding either +1 or −1. The perceptron learns the weights using an iterative algorithm. However, the algorithm will only converge if the training data are linearly separable [Tan et al., 2019].

Generalizing a Perceptron allows it to classify nonlinear problems [Faceli et al., 2011]. This generalization is done with a **Multi-Layer Perceptron (MLP)**. An MLP is an ANN with multiple Perceptron *nodes* arranged in multiple *layers*. Figure 2.5 pictures a 3-layer MLP with five inputs. The first layer is the input layer, where each input is linked to a single node. The middle layers, called *hidden layers*, receive signals from the input nodes or preceding hidden layers. Finally, the output layer has one node for each output class.

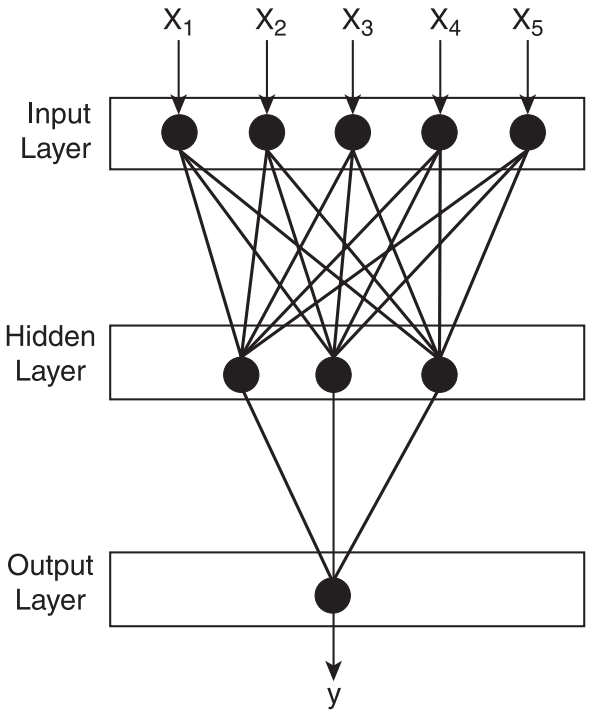


Figure 2.5: A fully-connected ANN with a single hidden layer with five input nodes and one output node. Source: Tan et al. [2019].

In Figure 2.5, all nodes are connected to every other node between any two layers, i.e., all layers are *fully-connected*. Each node multiplies each input by a learned weight and sums the computed weight with a learned bias. An activation function provides a node's output, which is typically the sigmoid function or the Rectified Linear Output Unit (ReLU) activation function [Tan et al., 2019].

Spiking Neural Network (SNN) is a specialized class of ANN. SNN differentiates from traditional ANN by using temporal pulses called spikes [Jeyasothy et al., 2022]. This enables SNNs to learn tasks that involve temporal dynamics, such as processing time-series information. With spikes, information is often encoded as analog values into spike patterns. The activation function in an SNN determines when a neuron should produce a spike based on its input. A typical activation function is the Leaky Integrate and Fire (LIF) [Jeyasothy et al., 2022]. LIF accumulates incoming spikes until a *membrane potential* threshold is reached. At this point, it emits an output spike and resets the membrane potential. In the absence of stimulus (i.e., input spikes), the potential decays gradually back to its resting state.

Graph Neural Network (GNN) is a class of DNN. DNNs are ANNs that contain more than one hidden layer. GNN is engineered to perform inference on data represented as graphs. The fundamental mechanism in a GNN is to update node representations by aggregating their neighbors' representations and their own representations from the previous layers [Wu et al., 2022]. A limitation of deep GNNs is over-smoothing, in which all nodes have similar representations, resulting in inferior performance.

Generative Adversarial Network (GAN) is a DNN architecture primarily for *generating* synthetic data that resembles real data. A GAN consists of two neural networks, the generator and the discriminator, which are trained simultaneously through a game-like process, improving both networks in the process [Foster, 2023]. The generator aims to create realistic data samples from random noise, mapping the noise to data that resembles the training set. Conversely, the discriminator is a neural network classifier trained to distinguish between actual training data and fake data generated by the generator.

Recurrent Neural Network (RNN) is a class of DNNs designed for processing sequential data. An RNN has recurrent structures that allow the network to maintain an internal state, or memory, that persists across time steps and influences the current output [Graves, 2012]. This enables RNNs to model temporal dynamics and relationships in data such as time series. The influence of an input decays or grows exponentially as it passes through recurrent connections, making RNNs unsuitable for learning long-term dependencies.

Long Short-Term Memory (LSTM) is a type of RNN with a redesigned memory mechanism to address the limitations of RNNs with long-term dependencies. LSTM recurrent neurons, or cells, are configured by three *gates*: an input gate, a forget gate, and an output gate. These gates allow the network to selectively add, remove, or retrieve information from the cell state, thereby regulating which information is preserved and which is discarded. This mechanism allows the network to maintain relevant information over long time periods, enabling LSTMs to effectively learn long-range dependencies in sequential data [Graves, 2012].

Optimization-based learning is considered a *black-box* approach, as the models built with these techniques are encoded in hard-to-interpret equations, whereas techniques such as DT are easily interpretable [Faceli et al., 2011]. Nevertheless, methods such as ANNs can combine features exponentially in hidden layers to construct more complex features, differentiating them from methods such as DT that can learn partitions in the attribute space but cannot combine them in an exponential manner [Tan et al., 2019].

2.5 Ensemble learning

An ensemble algorithm transforms models, called *weak learners*, that perform poorly, often just slightly better than random guessing, into models called *strong learners*, with good performance, by joining two or more models [Wade, 2020]. There are two main types of ensembles: bagging and boosting.

A bagging ensemble does bootstrap aggregating. *Bootstrap* consists of sampling with replacement, i.e., it samples from a dataset repeatedly, thus repeating some instances in the training set to maintain the same size as the original data. This helps with random fluctuations in training data but can degrade the classifier performance because it reduces the effective training set size [Tan et al., 2019]. Finally, each classifier is *aggregated* using the majority vote for classifiers or the average for regressors.

Random Forest (RF) is a bagging ensemble that uses DT as its base learners. In RF, each tree is built from a different data sample. Unlike standard bagging with decision trees, RF also randomizes the attributes used for splitting at each internal node of the tree. A drawback of bagging with decision trees is that if all base learners have similar errors, the ensemble will inherit those errors [Wade, 2020].

Boosting is otherwise an iterative process that changes the training samples based on difficult-to-classify sets [Tan et al., 2019]. While bagging aggregates the results of base learners, boosting builds each learner on top of the previous one, increasing the weight of instances that are being misclassified. Finally, the base learners are aggregated using weighted majority voting, where the weights depend on each learner's performance.

Gradient Boosting is another boosting technique. In Gradient Boosting, instead of weighting the errors of the previous learner to build the new one, as in standard boosting, it fits the new learner entirely based on those errors [Wade, 2020]. Therefore, each subsequent learner does not change the already correct predictions.

2.6 XGBoost

XGBoost [Chen and Guestrin, 2016] stands for eXtreme Gradient Boosting and is a Gradient Boosting ensemble commonly using DTs as its base learners. The algorithm constructs regression trees sequentially. XGBoost uses unique regression trees that iteratively improve residuals, rather than the target variables. A residual is the difference between the predicted and the actual value. Figure 2.6 illustrates the XGBoost sequential training process. XGBoost begins with a *bias* that defaults to the average of the targets y. The bias acts as the initial prediction $\hat{y}^0$ for all instances.

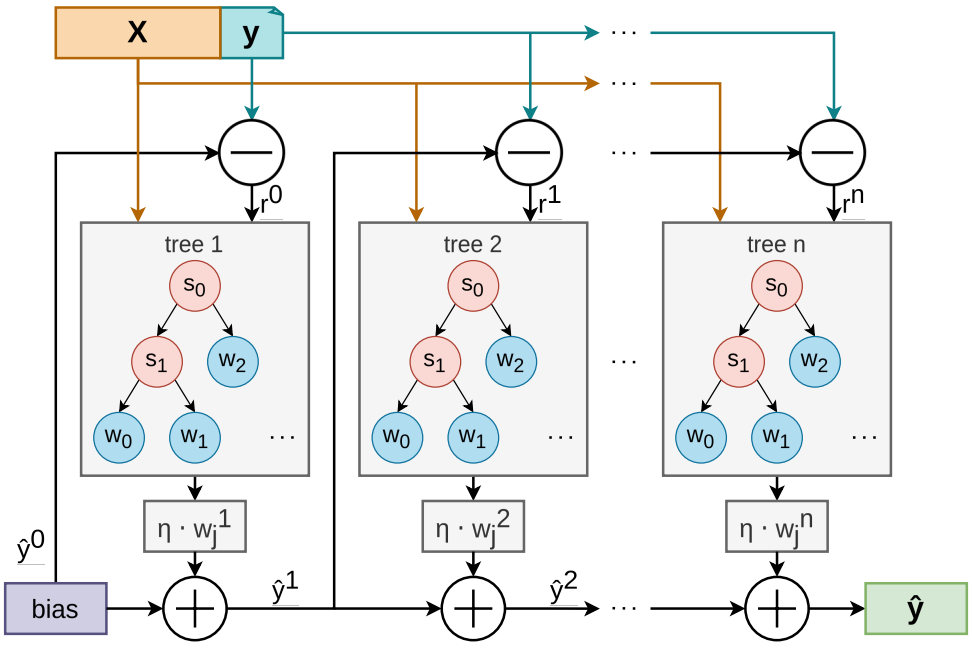


Figure 2.6: XGBoost training flow. X: features. y: targets. $\hat{y}$: predicted values. Source: the Author.

An XGBoost decision tree estimator (tree t) is built using the residuals r^{t-1}. In a DT, a node can be a split (s_j) or a leaf containing a weight (w_j). The weight of a tree (w_j^t) is computed as the value that minimizes a loss function. Equation (2.6) defines a commonly used loss function, the squared residuals for all N instances. A regularization term $\frac{1}{2}\lambda w_j^{t2}$ is added to penalize tree complexity, thus reducing variance and preventing overfitting.

$$L(y_i, \hat{y}_i) = \sum_{i=1}^{N} \frac{1}{2}(y_i - \hat{y}_i)^2 \tag{2.6}$$

The value of w_j^t that minimizes the regularized loss function is given in Equation (2.7). It is obtained as the sum of the residuals of all instances (I) in a leaf divided by the number of such residuals plus the regularization factor λ.

$$w_j^t = \frac{\sum_{i=1}^{I}(y_i - \hat{y}_i)}{I + \lambda} \tag{2.7}$$

The splits for XGBoost regression trees are performed using a similarity score. In Equation (2.8), the similarity, S, at the leaf of a tree (S_j^t) is given by the sum of the residuals at the leaf, squared, divided by the number of such residuals plus the regularization factor.

$$S_j^t = \frac{[\sum_{i=1}^{I}(y_i - \hat{y}_i)]^2}{I + \lambda} \tag{2.8}$$

The split decision is made by maximizing the gain, γ, defined by Equation (2.9) as the difference from the similarities in both left and right leaves of a split to the similarity score at the root node originating the split.

$$\gamma = S_{left} + S_{right} - S_{root} \tag{2.9}$$

XGBoost employs a greedy approach to select a split condition based on the mean of two split features. As the number of attributes and instances increases, computing the means, similarities, and split gains can become infeasible. Therefore, XGBoost speeds up split computation by using an approximate greedy algorithm that partitions the data into quantiles and computes only the quantile splits.

After building a tree, it can be post-pruned by removing the leaves from bottom to top that do not yield a sufficient gain (γ). Furthermore, the regularization factor itself already penalizes the three growth. When building XGBoost trees, limiting the maximum tree depth is also possible and is set to 6 by default.

In Figure 2.6, w_j^t is multiplied by a learning rate (η). The final tree prediction, $\hat{y}^t$, is a result of the sum of the previous predictions, and residuals r^t are computed to build the next tree. This step is repeated n times, controlled by a hyperparameter, which defaults to 100. The final prediction, $\hat{y}$, is the sum of all individual tree predictions with the initial bias.

One of the primary advantages of XGBoost lies in its ability to handle complex datasets with high dimensionality and a mix of feature types. It's also less prone to overfitting than traditional DT due to built-in regularization and pruning [Wade, 2020]. However, XGBoost's main drawbacks include the potential complexity of parameter tuning and longer training time compared to simpler Gradient Boosting models.

2.7 Model performance evaluation

None of the ML methods presented in this Chapter dominates all others across all possible datasets. One specific method may work better on a particular dataset, but others may be more effective on a similar yet distinct dataset. An ML algorithm should be evaluated to determine how well its predictions align with the observed data, assessing its performance [James et al., 2023].

The most widely used performance metric in regression applications is the MSE [Faceli et al., 2011]. Equation (2.10) defines the MSE, which measures the squared distances from the predicted values p_i from their actual values y_i in a test dataset and then averages them by the number N of test instances. MSE is non-negative, and a small value represents a good approximation of the actual values.

$$MSE = \frac{\sum_{i=1}^{N}(y_i - p_i)^2}{N} \tag{2.10}$$

In classification, a commonly used metric is *accuracy*, which represents the rate at which the model correctly labels data. For binary classification, Equation (2.11) represents the accuracy. The accuracy is given by the sum of True Positives (TP) and True Negatives (TN) divided by the total number of test instances (N). TPs are values correctly labeled as the positive class, while TNs are values correctly labeled as the negative class in binary classification. Likewise, False Negatives (FNs) and False Positives (FPs) are values that are incorrectly labeled as negative and positive, respectively.

$$Accuracy = \frac{TP + TN}{N} \tag{2.11}$$

Accuracy alone cannot represent the model performance when the classes are unbalanced, i.e., when TP or TN are too close to N. Labeling all targets as the dominant class will result in an accuracy close to 1, but the least frequent class will be left undetected.

In cases where the classes are unbalanced or when detecting the positive class is essential, such as in the medical field [Faceli et al., 2011] or in anomaly detection, the *recall*, also known as *sensitivity*, is often used. Recall corresponds to the correctly labeled positive class rate and is defined by Equation (2.12), where the TP is divided by TP plus FN.

$$Recall = \frac{TP}{TP + FN} \tag{2.12}$$

Recall alone is also insufficient for accurate measurement. When the model favors predicting positive labels, FN will decrease, and recall will approach 1. Therefore, it is necessary to use another metric, such as precision, that represents how *correct* the positive class is labeled. Equation (2.13) defines the precision as the rate between TP and the sum of TP and FP.

$$Precision = \frac{TP}{TP + FP} \tag{2.13}$$

Precision and recall are typically not presented in isolation, and the trade-off between the two helps assess a well-performing model [Faceli et al., 2011]. Therefore, the F_1-score, Equation (2.14), represents this trade-off by computing the harmonic mean between precision and recall.

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{2.14}$$

2.8 Machine Learning Requirements for MCSoCs

While the previous sections detailed various Machine Learning paradigms in a general context, applying these techniques within an MCSoC presents its own challenges. The massive parallelism of MCSoCs makes them a target for ML-based applications, such as CNN computation [Mourelle et al., 2024; Nunes et al., 2025]. However, their inherent resource constraints impose significant limitations on the complexity of a deployed model. MCSoCs exhibit complex, non-linear behaviors, such as NoC traffic contention and thermal hotspots [Weber et al., 2024], that are difficult to model with traditional analytical or rule-based methods. ML excels at learning these subtle patterns, as it can adapt to changing workloads and application mappings, unlike static policies. This enables context-aware decisions on security, resource allocation, and power management [Bellman et al., 2020].

The distributed nature of an MCSoC architecture is adequate for decentralized ML. Lightweight models can be deployed across the chip, enabling scalable, localized decision-making without a central bottleneck. Despite its potential, deploying ML models at runtime in MCSoCs requires an evaluation beyond predictive performance, as MCSoCs often operate with limited memory and strict power budgets. Thus, the feasibility of ML depends on balancing predictive and computational performance [Batzolis et al., 2023]. ML inference requires processing cycles that could otherwise be used by user applications, potentially degrading overall system performance. The additional computations for inference contribute to the system's power budget. Furthermore, ML models also require on-chip memory to store parameters (e.g., neural network weights), which is a critical constraint in systems with limited local memory [Warden and Situnayake, 2019].

Table 2.1 presents the machine learning paradigms presented in previous sections, comparing their predictive performance in relation to inference computation overhead and memory footprint. This Table provides a qualitative comparison, as precise overheads depend on specific implementation, hardware, and dataset characteristics. As summarized, there is a trade-off between a model's predictive performance and its implementation overhead. A low memory footprint is a requirement for deployment in MCSoCs, often making models with high memory demands, such as KNNs and DNNs, infeasible.

Computational overhead also presents a challenge due to the frequent lack of dedicated Floating-Point Units (FPUs) in MCSoCs. Consequently, models that rely on floating-point arithmetic must either be executed via software emulation, which severely degrades real-time inference capabilities, or undergo quantization. While converting a model to operate with integer or fixed-point arithmetic can accelerate inference, this process can potentially degrade the model's predictive performance. This trade-off makes models that rely less on arithmetic operations, such as tree-based models, suited for this domain.

Table 2.1: Qualitative comparison of ML model inference characteristics. Source: the Author.

ML Method	Predictive Performance	Inference Computation Overhead	Memory Footprint
KNN	**Moderate**. Performance degrades with redundant, irrelevant, or high-dimensional features.	**Very High**. Requires computing the distance to training instances and sorting to find the nearest ones.	**Very High**. Requires storing the entire training dataset.
Naïve Bayes	**Moderate**. Limited by the strong feature independence assumption.	**Very Low**. Computes the probability for each output class, with a number of multiplications proportional to the number of features.	**Very Low**. Stores conditional probability tables.
Linear/Logistic Regression	**Low to Moderate**. Limited to learning only linear decision boundaries.	**Very Low**. A single dot product operation.	**Very Low**. Stores the regression coefficients.
Decision Tree (Single)	**Moderate**. Highly prone to overfitting if not properly pruned.	**Very Low**. A simple traversal of the tree structure (comparison operations).	**Low to Moderate**. Stores the tree's nodes and split conditions.
SVM	**High**. Effective with non-linear kernels such as RBF.	**Moderate to High**. Kernel computation scales with the number of support vectors.	**Moderate**. Stores the support vectors and some coefficients.
Shallow ANNs	**High**. Universal approximators capable of learning nonlinear functions.	**Moderate**. Consists of matrix multiplications for a few layers.	**Moderate**. Stores weight parameters for all layers.
DNNs	**Very High**. State-of-the-art for highly complex, unstructured, non-linear problems (e.g., spatial or sequential data).	**Very High**. Numerous, large-scale matrix multiplications across many layers.	**High to Very High**. Requires storing a large number of weight parameters.
Tree Ensembles (e.g., XGBoost)	**High to Very High**. Suitable for structured data, where each data point is described by a set of features.	**Moderate**. Averages or sums the traversals for many shallow trees.	**Moderate**. Stores the ensemble's collection of all trees.

Ultimately, the successful application of ML in MCSoCs systems lies in the trade-off between a model's predictive performance and its implementation cost. Complex models may achieve high accuracy but can be infeasible due to their overhead. Therefore, this Thesis focuses on developing and validating a **lightweight**, **real-time**, and **non-invasive** ML framework that provides effective security monitoring while respecting the resource constraints of the underlying hardware.

2.9 MCSoC Architecture and Threat Model

The architectural model adopted in this Thesis is an MCSoC for embedded systems, imposing constraints on power, area, and cost. An MCSoC contains simple processors replicated multiple times, which can result in hundreds or even thousands of cores on a single chip [Asanovic et al., 2009]. Interconnecting such a large number of cores is a significant challenge for performance and energy consumption. Hence, shared buses are replaced by a **Network-on-Chip (NoC)** for inter-core communication [Benini and De Micheli, 2002].

Figure 2.7 presents a NoC interconnecting 16 cores. A NoC uses routers, or routing nodes, to route the data between its links. NoCs often use packet-switching routers, as in the context of this Thesis, in which data is structured into packets that are routed by each router individually. Packets are then split into *flits* to traverse the routers [Bjerregaard and Mahadevan, 2006].

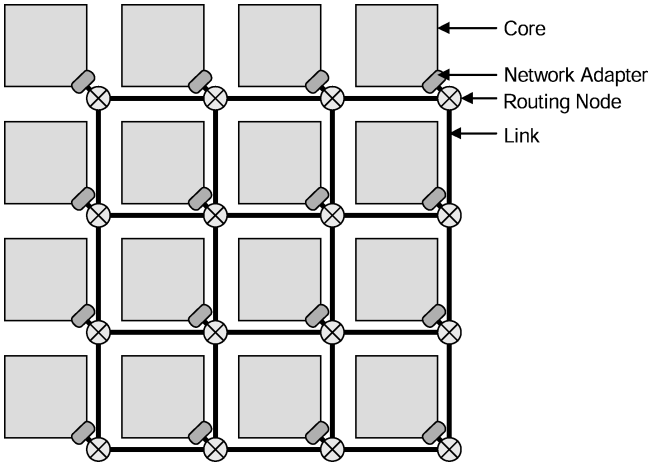


Figure 2.7: A 4x4 2D mesh NoC. Source: Bjerregaard and Mahadevan [2006].

In Figure 2.7, the routers are arranged in a 2D mesh topology, a topology often used because its regular 2-dimensional structure is well-suited for chip layout. In a NoC, the packet path is determined by a routing algorithm. For example, *XY* is a routing algorithm in which the packet moves along the columns (X-axis in a mesh) and then along the rows (Y-axis in a mesh) to reach its destination. XY-routing is deterministic, as its path is determined solely by its source and destination, and also minimal, as it chooses the shortest path [Bjerregaard and Mahadevan, 2006].

Two of the most common flow control strategies in NoC are *wormhole* and *store-and-forward*. In store-and-forward, a router receives a packet in its entirety, buffering all flits before forwarding it to the next router. In wormhole flow control, the first flit is routed as soon as it arrives at the router, reducing latency. However, it holds all links along the path until the last flit is routed, then releases them.

Virtual Channels (VC) and Multiple Physical Networks (MPN) are two methods for improving performance in NoCs. A router with VC has multiple buffers per input port, each behaving like a channel, allowing multiplexing on a link among these VCs. MPNs instead partition channel widths across multiple independent, parallel networks. VC yields better performance under high distributed loads, while MPNs perform better in areas with traffic hotspots, offering better area efficiency [Yoon et al., 2013].

The Network Interface (NI, or Network Adapter in Figure 2.7) connects the core to the router. The processing core, the NI, and local private memories form the MCSoC's **Processing Element (PE)**. The absence of shared memory dictates that all inter-core communication and synchronization must be performed explicitly via message passing.

The message-passing is managed at the hardware level by the NI, which handles packet transmission and reception, often with Direct Memory Access (DMA) capabilities to offload the processor. At the software level, a lightweight operating system running on each core typically provides a message-passing Application Programming Interface (API), similar to the Message Passing Interface (MPI), along with the necessary drivers to abstract the low-level network details from the user application.

Complementing this hardware model, applications are represented as Communication Task Graphs (CTGs). CTG is a model for representing functional parallelism, where an application is composed of independent parts and thus divided into tasks [Rauber and Rünger, 2013]. In this model, an application is decomposed into a set of concurrent tasks, represented by the graph's nodes (or vertices), while the communication and data dependencies between them are represented by directed edges. The CTG model naturally maps onto the MCSoC architecture, where each task node can be assigned to a processor core and the communication edges correspond directly to the message-passing operations executed over the NoC.

A parallel application is often structured in a pattern that is effective for many different applications [Rauber and Rünger, 2013]. Figure 2.8 illustrates two common parallel patterns of CTGs. Pipeline (a) offers a continuous data flow, overlapping communication and computation. Master-slave (b) divides parallel computation to *slaves* while the master performs serial computation or coordination for the application.

(a) Pipeline. (b) Master-slave.

Figure 2.8: Examples of common parallel patterns modeled as Communication Task Graphs (CTGs). Source: the Author.

The architectural features described, with a large number of cores, a distributed memory model, and a reliance on a shared NoC for all communication, create a scalable platform. However, these same features also introduce security vulnerabilities that are not present in monolithic, shared-bus, or single-core systems.

MCSoCs are powering critical devices, including IoT hubs, automotive systems (e.g., self-driving cars), medical equipment, and industrial control systems [Cerrolaza et al., 2020]. Consequently, a security failure is not just a data breach. It can also lead to physical harm or the failure of critical infrastructure. The design of an MCSoC often relies on integrating 3PIP cores to address time-to-market constraints. This creates a vulnerability: a malicious or compromised Intellectual Property (IP) core can be integrated into the chip, creating an HT [Li et al., 2016]. Furthermore, a supply chain attack could target the foundry and modify the IPs to include an HT, for example, in the NoC links [Das et al., 2024].

A compromised core through software exploits can also become a source of attacks. Once compromised, the core becomes a source of attacks against the system, using the NoC as its primary vector. The NoC itself is a shared resource. This means a malicious or misbehaving task running on one core can disrupt or spy on the communication of other, critical tasks, creating a system-wide vulnerability from a single point of failure [Caimi et al., 2021]. A compromised MCSoC can violate one of the three pillars of information security [NIST, 2017]: confidentiality, integrity, and availability. A confidentiality violation can occur due to the outcome of an *eavesdropping* attack, in which sensitive data can be obtained from the NoC.

A violation of integrity often occurs due to **misrouting** or **data-tampering** attacks. In a data tampering attack, the data is modified, corrupting its integrity. Misrouting deliberately redirects existing, legitimate packets to an incorrect destination. The target of a misrouting can be any target other than the packet's destination, the source of the packet, or coordinated across various affected points of the NoC to a single target. Misrouting is considered an integrity attack because it involves tampering with a packet's header to deliberately redirect it to an incorrect destination. This action also results in a loss of availability, as it causes data loss and potential system failure.

A violation of availability is the primary outcome of a DoS attack. A DoS attack can prevent or slow legitimate traffic from passing through a particular link or router. Simple **flooding DoS** attacks generate packets to overwhelm the NoC, while **packet-drop** attacks drop packets along the transmission path and can also cause system failure. More subtle forms of DoS attacks in MCSoCs can occur due to injected faults, route looping, temporary halts, or malicious software execution.

Injected faults into NoCs with Error-Correction Code (ECC) trigger retransmissions, compromising performance. **Temporarily halting** packet transmission can slow the NoC without causing a system failure, allowing it to resume normal flow afterwards.

Route looping is a coordinated attack in which two malicious entities keep a packet transmission in a live-lock, sending it back and forth between themselves. **Malicious software execution** generates traffic that can slow down the NoC while appearing to the system as legitimate. This malicious software can be unintended, compromised, or simply misbehaving due to programming bugs.

This Thesis aims to detect subtle traffic anomalies caused by the aforementioned attacks, which may originate from HTs or malicious software. The **threat model** considered in this work encompasses all attacks that result in modifications to traffic behavior within the NoC without critical integrity violations of packets, while maintaining the intended routing and ensuring that important monitoring and control flits remain unmodified.

2.10 Final Remarks

While a wide range of security threats exists, this Thesis focuses on those that manifest as anomalies in the NoC traffic patterns. The objective is to detect the malicious behavior at runtime by observing its impact on the communication of legitimate applications. This goal narrows the problem to runtime, performance-based security threat detection, for which a carefully chosen ML approach is required.

In this context, although ANNs offer high predictive performance, they are often computationally expensive to deploy on resource-constrained hardware. In contrast, tree-based ensembles, particularly XGBoost, provide an effective balance between predictive performance and computational efficiency. This balance, combined with XGBoost's inherent advantage against overfitting, makes it well-suited for profiling application behavior based on monitored communication features.

3. STATE-OF-THE-ART

This Chapter reviews and discusses related work on threat-detection solutions for NoCs and NoC-based MCSoCs. Each of the following sections is divided by the type of algorithm used by the related work to detect such threats:

1. Section 3.1 lists solutions using ad-hoc algorithms, i.e., that do not apply machine learning;
2. Section 3.2 describes solutions that do not use neural networks and are composed by a single learner;
3. Section 3.3 presents proposals that use ensemble learners, i.e., combinations of multiple ML models to enhance performance, but that are not composed by neural networks;
4. Section 3.4 presents solutions using neural networks.

Finally, Section 3.5 discusses the related work presented in this Chapter.

3.1 Ad-hoc solutions

In the early stages of research on threat detection in NoCs, Rajesh et al. [2015] introduced the Runtime Latency Auditor for NoCs (RLAN) as a tool to detect subtle, condition-based DoS attacks. Figure 3.1 illustrates the architecture of RLAN. The SoC Firmware (in orange) is responsible for controlling the RLAN IP in this system. It is designed to insert a timestamp into the packet before forwarding it to the NI. Upon receiving a packet, the firmware extracts the previously inserted timestamp to calculate the packet's latency.

RLAN includes a hardware IP that generates Proximal Analogous Packets (PAPs), which are modified versions of existing packets that mirror their size and route. The SoC Firmware receives the PAPs, and RLAN assesses whether the PAP latency deviates from the original packet's latency. Repeated deviations indicate abnormal network activity. The Authors use an 8x8 NoC simulated with BookSim [Jiang et al., 2013], employing traffic profiles from the PARSEC benchmark suite [Bienia et al., 2008] provided by Netrace [Hestness et al., 2010]. They reported an average recall of 98.9% and a false positive rate ranging from 0% to 17.88% across eight scenarios involving single and multiple applications running concurrently. The insertion of RLAN increased router area and power by 12.73% and 9.84%, respectively, while PAP traffic increased NoC latency by 5.47%. However, the study did not account for an HT that might affect both the original packet and the PAP.

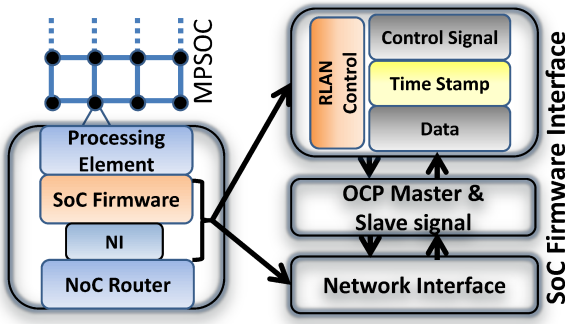


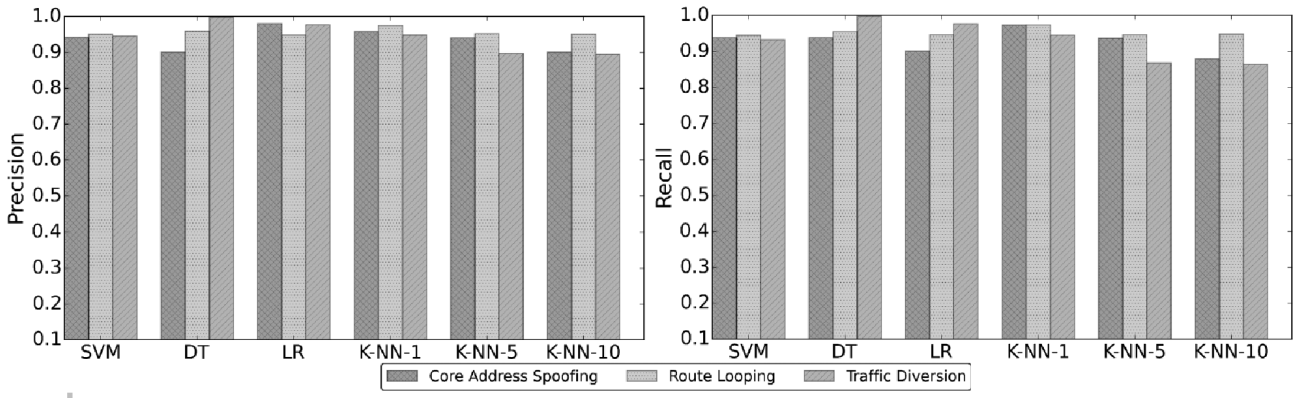
Figure 3.1: RLAN architecture. SoC Firmware controls RLAN packet generation. Source: Rajesh et al. [2015].

Charles et al. [2020] present a method to detect and localize flooding DoS generated by malicious 3PIP using expected packet arrival rate and packet arrival latency. Their approach monitors the arrival rate at every router a packet traverses and the latency at the destination IP. If a packet violates a threshold on those metrics, an attack is flagged. This solution is limited by variations in applications, application mapping, and input values, requiring those parameters to be well-defined or predictable. The work is assessed using the gem5 [Binkert et al., 2011] simulator along with Garnet [Agarwal et al., 2009] NoC. A 4x4 NoC is simulated with SPLASH-2 [Woo et al., 1995] benchmark traces, and an 8x8 NoC is simulated with synthetic traffic. Router area and power overhead are stated as 6% and 4%, respectively, but these metrics are obtained using a different design than the one evaluated for detection performance. IP memory overhead is negligible, but the computation required to analyze packet latency is not disclosed, suggesting that is focus solely on NoC characteristics.

3.2 Single learner solutions

Research on threat detection in NoC systems has increasingly turned to ML due to the limitations of ad-hoc solutions. Kulkarni et al. [2016] present one of the earliest ML-based proposals for detecting threats in MCSoCs, specifically targeting misrouting attacks. Their research evaluates various ML algorithms, including KNN, LR, DT, and unsupervised learning methods. Following a correlation analysis of different features extracted from NoC traffic, they select the following attributes for algorithm evaluation:

- Source core of the message;
- Destination core of the message;
- Packet transfer path;
- Distance, as defined by the packet hop count at each router.



The three attacks are actually misrouting attacks, differing in the target of the tampered packet. In "Core Address Spoofing", the target is coordinated to a single PE; in "Route Looping", the target is the source of the packet; in "Traffic Diversion", the target is random.

Figure 3.2: Precision and recall for the three types of attacks detected by the six models proposed by Kulkarni et al. [2016]. Source: Kulkarni et al. [2016].

Figure 3.2 compares the average precision and recall for the three types of attacks using supervised learning algorithms, as assessed by Kulkarni et al. [2016]. Results from unsupervised learning are excluded from the graph due to their insufficient performance. Notably, SVM and KNN (with $k = 1$) achieve better overall performance than LR and DT. However, the Authors chose to implement SVM, considering its advantageous balance between classification performance and hardware complexity.

Figure 3.3 illustrates the architecture into which the SVM algorithm is integrated. This architecture comprises a 16-core hierarchical NoC with a single Attack Detection Module that uses the proposed SVM. The system is implemented on a Field-Programmable Gate Array (FPGA). The solution employs synthetic traffic generated by a Traffic Generator Module. Another limitation is the NoC topology, which consists of four routers, each connected to four cores, and a central router connected to them. This design may limit the scalability of the proposed solution. Nonetheless, the reported overhead for integrating the SVM algorithm into the system is minimal, accounting for only 2% of the system's area and 1% of its power.

Similarly, Vashist et al. [2019], besides evaluating KNN, SVM, and DT, also include MLP. Their research focuses on eavesdropping and jamming DoS attacks in Wireless NoCs (WiNoCs), characterized by sustained burst errors detected by an ECC module. In their approach, the machine learning classifier uses the number of burst errors within a block, flits transmitted, and flits received as features. Given the specific characteristics of their WiNoC platform, they propose a clustered detection approach that incorporates a *security unit* within each Wireless Interface (WI). The effectiveness of their system is tested using system-level simulation with back-annotated router RTL performance figures on an 8x8 many-core setup with 4 WIs, employing synthetic traffic.

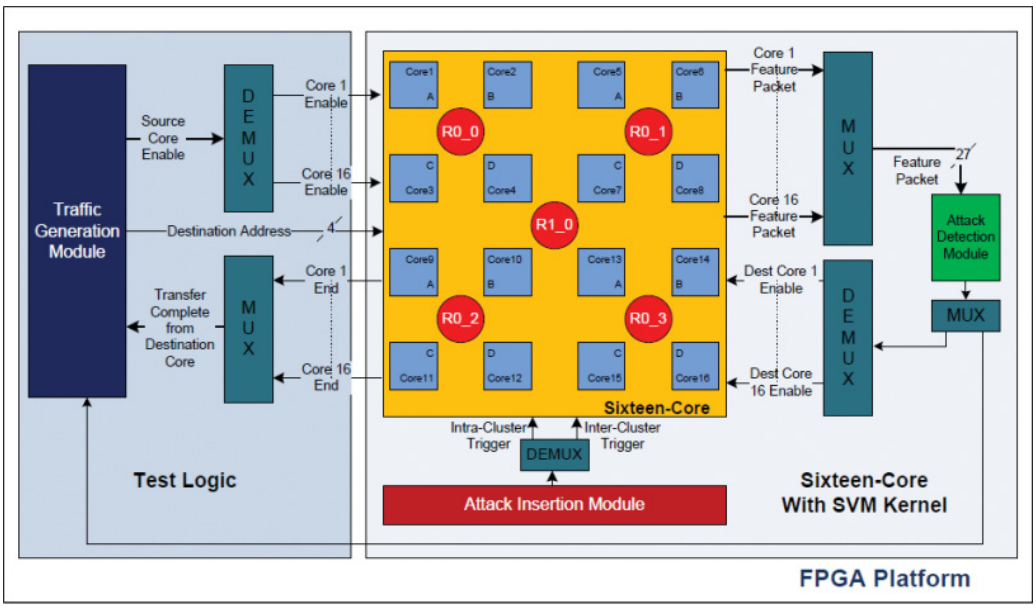


Figure 3.3: FPGA platform consisting of a hierarchical NoC in tree topology with an external SVM attack detection module. Source: Kulkarni et al. [2016].

In Table 3.1, recall and precision scores are listed for each ML approach used by Vashist et al. [2019]. Among the approaches, MLP with a single hidden layer of 20 nodes and DT showed poor detection performance, as evidenced by their low recall, but the Authors do not justify this result. Both SVM and KNN (with $k = 1$) showed good classification performance, but the Authors indicate that SVM could not detect sporadic variations, so KNN was used in their final implementation. Their approach resulted in 2.6% and 4.2% additional area and power, respectively, in relation to the WI.

Table 3.1: Classifier performance for WiNoC eavesdropping and DoS detection obtained by Vashist et al. [2019]. Source: the Author.

Classifier	MLP	SVM	KNN	DT
Precision	1.00	0.98	0.99	1.00
Recall	0.48	0.98	0.99	0.52

Hu et al. [2023] proposed a cascaded methodology first to detect and then classify condition-based misrouting HTs. Following a correlation analysis, the Authors selected average packet latency and average link utilization as the primary attributes for their research. They assessed various algorithms for both detection and classification stages, including SVM, Logistic Regression, KNN, DT, and RF. Figure 3.4 depicts their proposed workflow, which consists of using SVM for attack detection, followed by KNN for classification, with the latter fed by the SVM output. They tested their approach using gem5 and Garnet, simulating an 8x8 NoC with synthetic traffic that exhibited distinct communication patterns. The results, obtained offline, showed an average precision of approximately 0.87 and an average recall of roughly 0.86. However, the Authors did not disclose any overhead metrics associated with their approach.

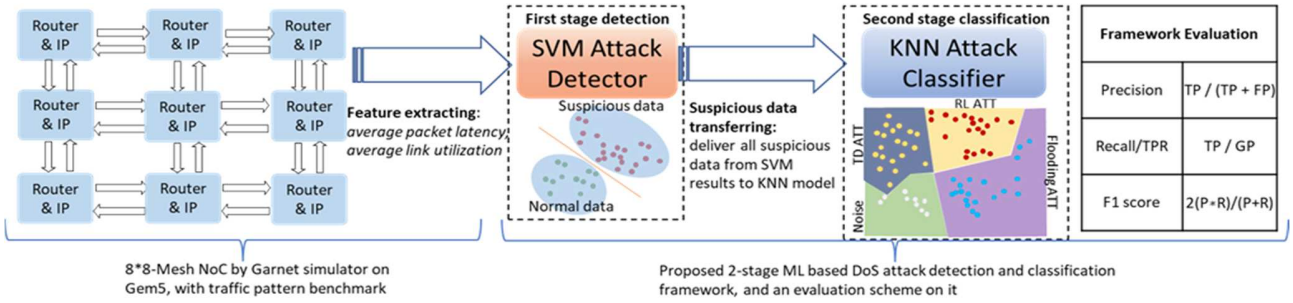


Figure 3.4: Proposed attack detection and classification flow by Hu et al. [2023]. Source: Hu et al. [2023].

3.3 Ensemble learner solutions

Yao et al. [2020] propose a method based on RF that simultaneously detects flooding DoS attacks in NoCs and localizes their source. To train their model, they consider the following features:

- Source core of the message;
- Destination core of the message;
- Packet transfer path;
- End-to-end packet delay.

They achieve an average precision of 1.0 and a recall of 0.957 in detecting traffic *suffering* from a DoS attack and a precision of 0.993 and a recall of 0.950 in identifying the traffic *causing* the DoS attack. However, the data is extracted using synthetic traffic in a 4x4 mesh simulated with OPNET [2005] computer network simulator, which may introduce inaccuracies. Furthermore, the Authors do not provide details on the power, area, or implementation method of the detection system for the target platform.

Sudusinghe et al. detect flooding DoS [Sudusinghe et al., 2021] and eavesdropping attacks [Sudusinghe et al., 2022] using XGBoost. They claim their techniques can identify attacks in unpredictable NoC traffic patterns caused by varying application mappings. Their experiments use gem5 and Garnet in a 4x4 system configuration with shared memory and local caches at the PEs. The routers in their system are equipped with probes that collect relevant attributes for every flit traversing a router and send this data to a centralized unit for attack detection.

Sudusinghe et al. [2021] propose a Security Engine for detecting DoS attacks, incorporating six distinct counters and indexes, along with the following attributes: (*i*) port used by the flit to exit the router; (*ii*) virtual channel; (*iii*) number of hops; and (*iv*) time taken by the flit to traverse the router.

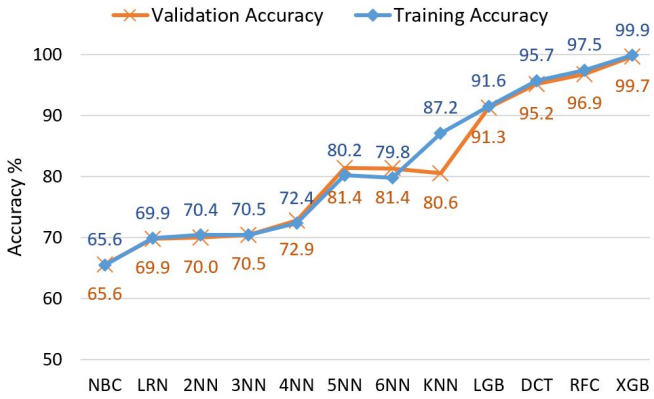


Figure 3.5: Accuracy obtained by Sudusinghe et al. [2021] when detecting DoS attacks using different ML models. Source: Sudusinghe et al. [2021].

Sudusinghe et al. [2021] evaluate a range of ML models, including NBC, KNN, DT, RF, LightGBM, XGBoost, and ANNs (2–6 layers), for their effectiveness in detecting DoS attacks. The accuracy of each model is detailed in Figure 3.5, with XGBoost achieving the highest one. Despite not assessing other metrics, the Authors select XGBoost for their final implementation. Their model computes the probability of a DoS attack occurring as the average of instances classified as indicative of an attack over a time window. In three simulated scenarios, they report average precision and recall of 0.95 and 0.96, respectively. However, the Authors evaluate their model using only one benchmark application, the Fast Fourier Transform (FFT), which could bias the results due to the varying computational and communication characteristics of different applications. While the classifier's performance appears promising, the Authors do not provide details on the Security Engine IP's overhead. Moreover, they do not address potential scalability issues associated with a centralized approach, as this dedicated IP is unique for the entire MCSoC.

Sudusinghe et al. [2022] propose a Decision Unit to detect eavesdropping attacks. The Authors provide limited details on the set of attributes used to train their model, but state that each router in each distinct test scenario has a different subset of 10 attributes, potentially biasing the results. They propose an algorithm for selecting which models to execute at each time in the Decision Unit, based on a trigger mechanism that selects only PEs with a packet transfer ratio above a predefined threshold. The ML algorithm is executed for the selected router and its neighbors. The Authors evaluate LR, DT, RF, XGBoost, and MLP and, based on model accuracy, select XGBoost for the final model. The average precision ranges from 0.788 to 0.992, while recall ranges from 0.770 to 0.985 across scenarios with up to 50% of the synthetic traffic snooped. The router monitoring probes increase their area by 19.2%. Moreover, the Authors use a dedicated service NoC to transfer the attributes to the centralized Decision Unit, with 6% area overhead and 7% power overhead compared to a single NoC. It is possible that scalability is not an issue in their approach, since the model is executed on a filtered subset of PEs, but this cannot be confirmed without knowing the overheads of the centralized Decision Unit IP.

Ding et al. [2024] present a Bayesian change point detection on NoC traffic that feeds an RF model. The method first identifies abrupt changes in traffic features, then outputs a change-point probability. This probability, along with a topological structure feature, is then fed into the RF model to detect HT-induced attacks. The RF model detects flooding DoS, routing loop, misrouting, and packet-drop attacks, with recall ranging from 97.1% to 99.6% and a False Positive Rate (FPR) ranging from 0.03% to 0.7%. The method is assessed offline using gem5 and Garnet with synthetic traffic, so no overhead metrics are given. Furthermore, the Authors do not provide specific details on the initial traffic features fed to the Bayesian algorithm, nor do they define the topological structure feature for the RF model.

Hathal et al. [2024] aim to detect flooding DoS in a 4x4 MCSoC executing an FFT benchmark. Their threat model assumes an infected IP that increases network traffic by 50%. The Authors chose XGBoost among 12 tested models due to its superior predictive performance in their tests, achieving 99.9% precision and 99.7% recall. The Authors assess their proposal using gem5 and Garnet, but do not disclose the processor architecture. The approach uses a centralized detector that implements the XGBoost model with distributed monitoring, but neither is detailed. Furthermore, no overhead metrics are given.

3.4 Neural Network Solutions

Madden et al. [2018] employ SNN to identify flooding DoS attacks, considering spatial and temporal variations. The only input attribute is the request-to-send signal from the routers' output ports. Their approach is centralized and comprises three fully connected layers. Figure 3.6 illustrates their proposed architecture, in which the SNN features one input neuron for each output port in the NoC and a total of 1,000 neurons. The evaluation is conducted in a system with 16 PEs, using synthetic traffic modeled from a "multimedia-system" [Liu et al., 2015], simulated with Noxim [Catania et al., 2016]. As the model evaluation is performed offline, no overhead metrics are provided. The Authors report an average accuracy of 86%, but this result has low confidence because not all assessed scenarios have balanced class distributions.

Sniffer [Sinha et al., 2021] is a detection and localization Perceptron for flooding-DoS caused by Malicious IP (MIP). They evaluate their proposal on an 8x8 NoC with x86 and "GPU cores" using the Rodinia benchmark suite [Che et al., 2009]. The attributes used as inputs to the Perceptron are:

- Time taken by a flit in the input buffer;
- Time between the reception of two flits in the input ports;
- Buffer utilization in each virtual channel.

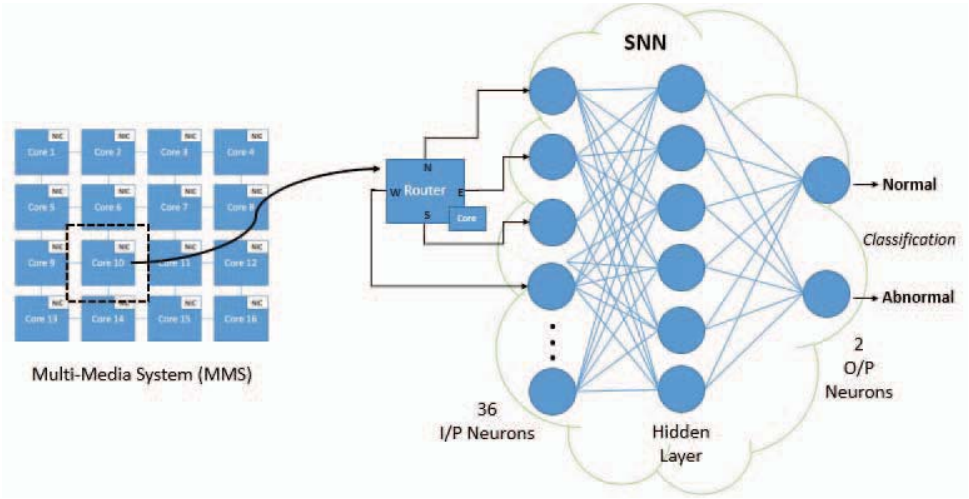


Figure 3.6: Test setup proposed by Madden et al. [2018], where the output ports of every router in the MCSoC are connected to the SNN. Source: Madden et al. [2018]

Sniffer presented an average precision of 0.984 and recall of 0.983 for DoS detection. Figure 3.7 illustrates the Sniffer architecture, with the Perceptron shown in (a). The router (b) is modified in their proposal to insert their Perceptron detection IP. Moreover, the work is evaluated using gem5 and Noxim, but the overheads are obtained from a different design synthesis, which can lead to inaccurate results, such as their claimed 3.3% area and 3.92% additional area and power, respectively, in relation to the router.

DetectANN [Wang et al., 2020] and AGAPE [Wang et al., 2022] are two neural network-based approaches for detecting HTs. DetectANN is a fully connected 3-layer MLP for detecting injected faults, with a hidden layer of 30 neurons. The Authors use the following attributes:

- Number of occupied VCs in each router port;
- Link utilization in each router input port;
- Router temperature;
- Transient error rate detected by an ECC Decoder.

DetectANN is evaluated using gem5 to simulate an 8x8 MCSoC with out-of-order processors with a 2-level cache using synthetic traffic mixed with PARSEC benchmark traffic. Still, only a detection accuracy of about 95% is reported, which can be a misleading metric when presented without further details if the test dataset is unbalanced. The Authors claim an overhead of 0.9% in the router area and a power dissipation of 0.086 mW. Nevertheless, the methodology for obtaining the overheads involves synthesizing a design different from the simulated one, which can lead to inaccuracies in functionality and in the resulting metrics.

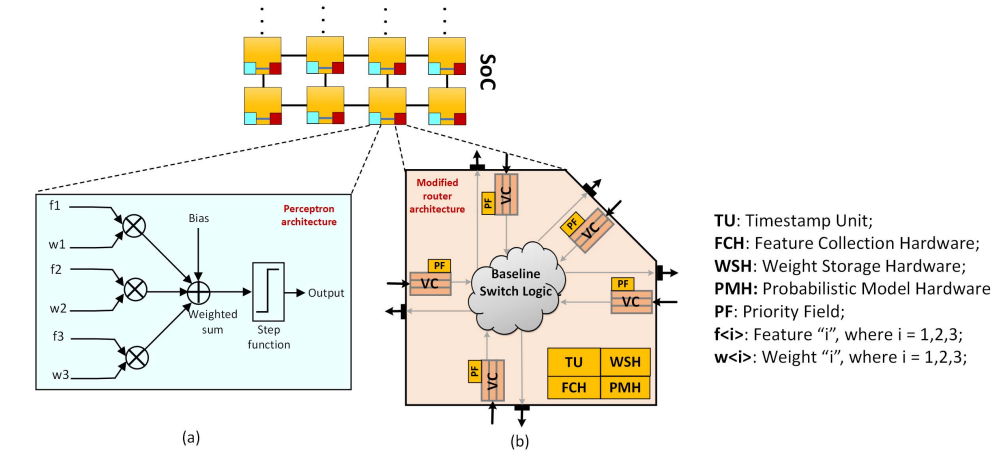


Figure 3.7: Sniffer architecture, with the perceptron (a) and the modified router (b). Source: Sinha et al. [2021].

AGAPE employs a GAN to detect injected faults, misrouting, and packet drops caused by HTs. This detection process involves reconstructing the attribute time series into matrices and applying computationally demanding convolution layers, as depicted in Figure 3.8. After the offline training phase, the Generator is discarded, and only the Discriminator with 5 convolution layers is used for classification. Compared to DetectANN, AGAPE does not use the transient error rate but adds the retransmission rate per port, the packet injection rate per port, and an additional average end-to-end packet latency as attributes. Although it is assessed using the PARSEC benchmark suite and gem5, the lack of detailed information about the MCSoC lowers confidence in the claimed area overhead of 7.2% relative to the router and in the claimed power of 0.34 mW.

Table 3.2 presents the precision and recall scores for the three types of attacks that AGAPE can detect. The results show high classification performance for fault-injection attacks but a notably lower precision for misrouting and packet-drop detection. Packet drops can lead to catastrophic failures in NoCs, as their intra-chip nature typically does not account for packet loss, making their detection impractical.

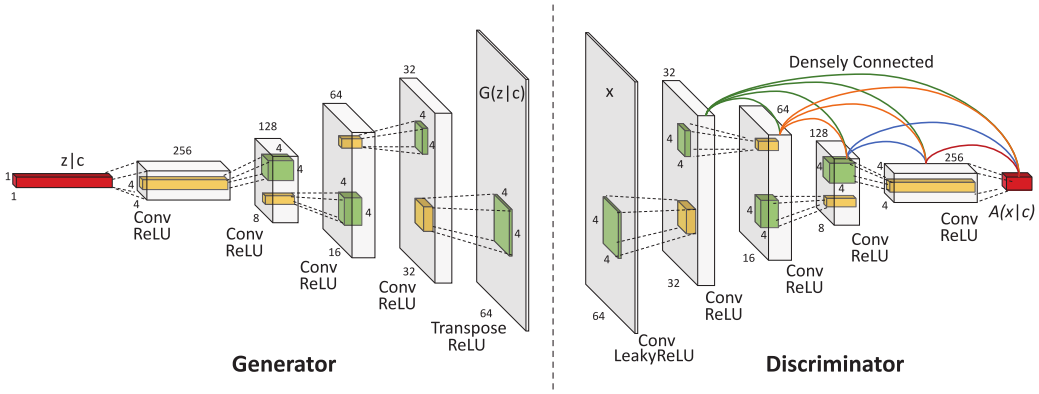


Figure 3.8: GAN architecture used by AGAPE. Adapted from: Wang et al. [2022].

Table 3.2: Classifier performance obtained by AGAPE [Wang et al., 2022]. Source: the Author.

HT	Fault-injection	Misrouting	Packet-drop
Precision	0.873	0.770	0.802
Recall	0.962	0.928	0.979

Figure 3.9 shows the router proposed by Wang et al. [Wang et al., 2020, 2022] in both DetectANN and AGAPE. The neural networks (HT Detection) are inserted into every router of the MCSoC, an invasive design that also results in significant area overhead.

NoCSNet [Abdollahi et al., 2024] is a hybrid opto-electrical NoC attack dataset and an attack detection approach. The dataset aims to capture thermal attacks on the optical routers, which could lead to DoS or data leaks. The Authors use router ID, flit type (head, tail, body), hop count, flit sequence number, buffer occupation, input and output port identifiers, core temperature, and simulation clock cycle as inputs for training the detection models. By evaluating several models' predictive performance, the Authors choose an 8-layer RNN for its ability to capture temporal relationships. The RNN presented 92.9% precision and 92.5% recall. The dataset is obtained using Noxim and HotSpot [Huang et al., 2006] thermal model in a 6x6 NoC assessed offline.

Galimberti et al. [2024] present a 3-layer ANN with 50 neurons in the hidden layer capable of classifying an application under execution in a 4x4 MCSoC running Linux. The approach uses traffic monitors inserted in every PE, which use the NoC to send, in each timeslice, the number of packets that passed through the router to a centralized detector that implements the ANN. The ANN has 50 neurons in the hidden layer and outputs 13 possible applications: 12 from the PARSEC benchmark and 1 deemed *unintended*. The simulation is carried out with gem5, using an MCSoC that contains x86 out-of-order processors with a 2-layer cache and DDR4 memory. The router model considers that routing occurs within 1 clock cycle. On average, the detector achieves a precision of 89.4% and a recall of 89%, with an estimated area of $34,449\,\mu m^2$ and a power of 6.3 mW. The traffic monitor incurs an overhead of $37,200\,\mu m^2$ with $3\,\mu$W of power. However, the Authors do not provide details about the synthesis to support the area and power claims.

GNS [Wang et al., 2024] and TampML [Wang and Halak, 2025] are two approaches for detecting, localizing, and applying countermeasures to attacks in MCSoCs. GNS combines GNN with LSTM to address flooding DoS attacks. It includes traffic monitors on every router that average the occupancy of input buffers over 100 clock cycles and send the results to a global detector. Figure 3.10 pictures the global detector diagram. The data gathered from all traffic monitors is represented as an adjacency matrix to feed the GNN input, capturing spatial communication relationships. The GNN output serves as the LSTM input, along with a timestamp representing the monitored frame. The LSTM outputs the classification result (attack/non-attack).

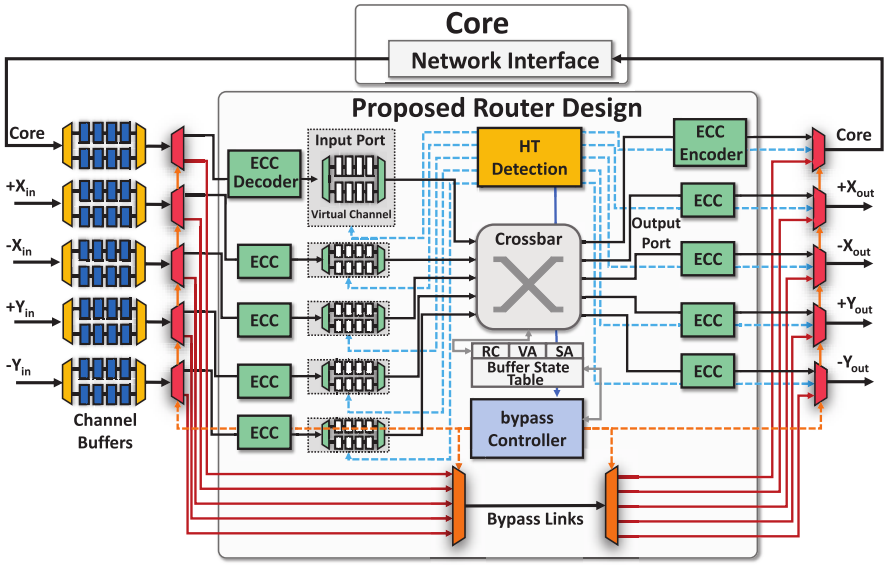


Figure 3.9: Router architecture of DetectANN and AGAPE. Source: Wang et al. [2022].

GNS is evaluated using gem5 and Garnet in a 4x4 x86 configuration to run PARSEC benchmarks, achieving 97% detection precision and 99% recall. The traffic monitor reported an overhead of 1.1% in Look-Up Tables (LUTs) and 2.7% in Flip-Flops (FFs) compared to the router in an FPGA implementation using ProNoC [Monemi et al., 2024]. The global detector (GNN+LSTM) resulted in a 17.3% LUT and 6.7% FF increase w.r.t. a different MCSoC built with 16 mor1kx [OpenRISC, 2012] cores. These overhead metrics have low confidence due to the lack of data regarding the FPGA implementation, and the use of different core architectures for the overhead versus the benchmark assessments.

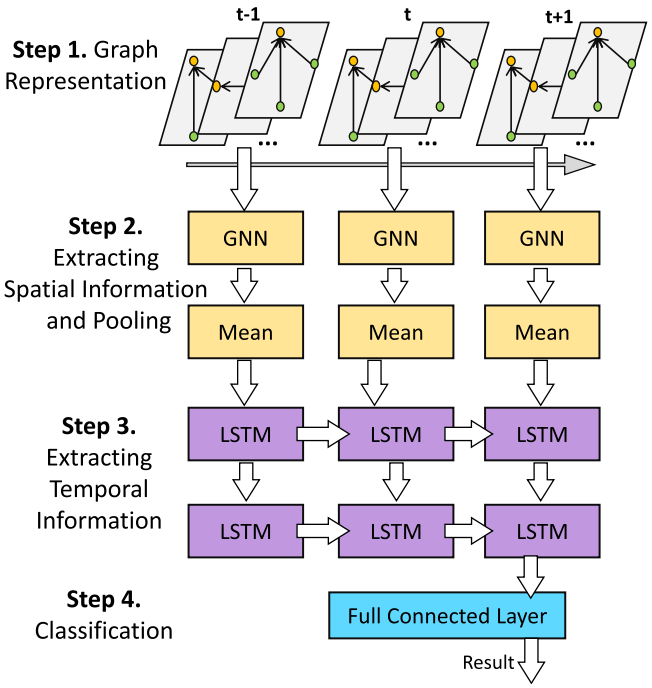


Figure 3.10: GNS global detector. Adapted from: [Wang et al., 2024].

TampML is a framework for addressing data tampering HTs. Its detection approach is based on a 3-layer ANN that classifies flits as tampered or not tampered using packet ID, flit size, message size, and flit width as input parameters. The number of neurons in the hidden layer is not disclosed. The Authors also propose a clustered monitoring architecture with Distributed Flit Collectors (DFC) that sample flits every 5 clock cycles and send the input features to the Central TampML Unit (CTU), which implements the ANN accelerator. The interaction between DFC and CTU uses a separate communication mechanism. The work is assessed using gem5 and Garnet to simulate an 8x8 mesh of Alpha processors, with 4 DFCs monitoring 16 PE clusters. The Authors use synthetic traffic and three PARSEC benchmarks. Among the PARSEC benchmarks, TampML achieved a recall of 78% and a precision of 100%. Overhead is stated as minimal, but no supporting numbers are provided.

3.5 Related work analysis

Table 3.3 presents the related work summary. The columns are organized as follows:

- The first column lists the works presented in this Chapter using ML. Ad-hoc solutions (works in Section 3.1 are omitted;
- The second column lists the detection method and its main characteristics;
- The third column lists the types of threats detected by each work;
- Columns 4–5 present the precision and recall metrics, respectively, for the corresponding method, unless otherwise indicated;
- The sixth column lists the topology of each solution, whether the detection method is centralized or distributed. Works not assessed at runtime are classified as *offline*;
- The seventh column lists the overhead as reported by the Authors. For some works, the confidence in these reported data is low. Such instances are marked as *claimed* overhead values. The justification for this low confidence in certain reported overheads is justified later in this Chapter;
- The eighth column lists the traffic profile used to generate data;
- Finally, the ninth column presents the modeling strategy, whether a simulation tool is used, whether the system is modeled in RTL, or whether a custom implementation is used on an FPGA. Furthermore, it details the MCSoC size, the processor architecture, or whether only the NoC was modeled.

Table 3.3: Related work summary. **ND**: No Data. N: number of PEs. O: number of output ports. Source: the Author.

Author	Method	Attacks Detected	Precision	Recall	Topology	Overhead	Traffic Profile	Modeling
Kulkarni et al. [2016]	SVM (4 attributes)	Misrouting	0.94 (average)	0.94 (average)	Centralized	2%/3% total area/power	Synthetic	FPGA (16 cores, 5 routers)
Madden et al. [2018]	SNN (O attributes, 3 fully-connected layers, 1000 neurons)	Flooding DoS	ND	ND	Offline	ND	Synthetic	Noxim (4x4 NoC-only)
Vashist et al. [2019]	KNN (3 attributes, k = 1)	Eavesdropping; Jamming DoS	0.99	0.99	Clustered	2.6%/4.2% area/power w.r.t. WI (claimed)	Synthetic	System-level simulator (8x8 NoC-only, 4 WIs)
Wang et al. [2020]	MLP (12 attributes, 30 hidden layer neurons)	Injected faults	ND	ND	On every router	0.9% area w.r.t. router & 0.086 mW power (claimed)	Synthetic & PARSEC	gem5 (8x8, 2-level cache, out-of-order)
Yao et al. [2020]	RF (4 attributes)	DoS-affected traffic DoS-causing traffic	1.0 0.993	0.957 0.950	Offline	ND	Synthetic	OPNET (4x4 NoC-only)
Sinha et al. [2021]	Perceptron (3 attributes)	Flooding DoS	0.984	0.983	On every router	3.3%/3.92% area/power w.r.t. router (claimed)	Rodinia	gem5+Noxim (8x8, 8 x86 out-of-order processors)
Sudusinghe et al. [2021]	XGBoost (10 attributes)	Flooding DoS	0.95	0.96	Centralized	ND	FFT application	gem5+Garnet (4x4 cached with shared memory)
Sudusinghe et al. [2022]	XGBoost (10 attributes)	Eavesdropping	0.924 (average)	0.915 (average)	Centralized (condition-activated)	19.2% router area & 6%/7% area/power w.r.t. NoC (claimed); Decision Unit: ND	Synthetic	gem5+Garnet (4x4 cached with shared memory)
Wang et al. [2022]	GAN discriminator (22 attributes, 5 convolution layers)	Injected faults Misrouting Packet-drop	0.873 0.770 0.802	0.962 0.928 0.979	On every router	7.2% area w.r.t. router & 0.34mW power (claimed)	PARSEC	gem5 (ND)
Hu et al. [2023]	SVM (2 attributes)	Misrouting	0.87	0.86	Offline	ND	Synthetic	gem5+Garnet (8x8 NoC-only)
Ding et al. [2024]	Ad-hoc + RF (no attribute details)	Flooding DoS Routing loop Packet-drop Misrouting	FPR: 0.7% FPR: 0.03% FPR: 0.2% FPR: 0.4%	0.97 0.98 0.99 0.97	Offline	ND	Synthetic	gem5+Garnet (4x4 NoC-only)
Hathal et al. [2024]	XGBoost (29 attributes)	Flooding DoS	0.999	0.997	Centralized	ND	FFT application	gem5+Garnet (4x4)
Abdollahi et al. [2024]	RNN (12 attributes)	Thermal attack	0.929	0.925	Offline	ND	Synthetic	Noxim + HotSpot (6x6 NoC-only)
Galimberti et al. [2024]	ANN (N attributes, 3-layer, 50 hidden layer neurons)	Unintended software execution	0.894	0.890	Centralized	71649 μm^2 & 6.3 mW (claimed)	PARSEC	gem5 (4x4 x86 out-of-order)
Wang et al. [2024]	GNN+LSTM ($N^2 + 1$ attributes)	Flooding DoS	0.97	0.99	Centralized	1.1%/2.7% LUT/FF w.r.t. router & 17.3%/6.7% LUT/FF w.r.t. MCSoC	PARSEC	gem5+Garnet (4x4 x86) or FPGA (4x4 ProNoC + mor1kx)
Wang and Halak [2025]	ANN (4 attributes, 3 layers)	Data tampering	1.00	0.78	Clustered monitoring; Centralized detector	ND	Synthetic & PARSEC	gem5+Garnet (8x8 Alpha ISA)
This Thesis	XGBoost (4 attributes, per-application-profiling)	Threat-affected traffic	0.841 (average)	0.919 (average)	Distributed	2.5% app. exec. time; 18.4 kB memory (average)	Realistic workloads with complete software stack	RTL simulation (up to 6x6 RISC-V ISA)

Table 3.3 contains 16 works that use ML to detect threats in MCSoCs. The last line of the Table presents the work proposed in this Thesis. ML is a potential solution for security threat detection in MCSoCs. However, state-of-the-art techniques are limited in **applicability** and **confidence**. The first issue, **applicability**, arises from the use of costly techniques. The following applicability limitations were identified in the state-of-the-art:

***Limitation* 1. Scalability:** 7 works [Kulkarni et al., 2016; Sudusinghe et al., 2021, 2022; Hathal et al., 2024; Galimberti et al., 2024; Wang et al., 2024; Wang and Halak, 2025] use hardware-based centralized machine learning solutions. This poses a scalability issue due to: (*i*) the ML model requires more inputs as MCSoC increases in size; and (*ii*) receiving monitoring packets from the entire system can become infeasible as the system grows. The same holds when the ML model directly probes the system: as the system increases in size, the probes are farther from the classifier. Only Vashist et al. [2019] propose a distributed approach that may not compromise scalability.

***Limitation* 2. Model overhead:** a possible distributed organization is placing a threat detector at every router. Three works [Sinha et al., 2021; Wang et al., 2020, 2022] insert their proposed model in every router. This poses a significant overhead in terms of area and power, mainly if the proposed model is complex and has many attributes, such as the neural networks proposed in some of these works [Wang et al., 2020, 2022]. A significant overhead can also occur in a centralized approach, such as the one proposed by Sudusinghe et al. [2022], which uses a different model for each router, or the one proposed by Wang et al. [2024], which involves DNN models. Such models will incur a significant overhead, even if they are concentrated in a single detector IP.

***Limitation* 3. Monitoring overhead:** two works [Sudusinghe et al., 2021, 2022] obtain their data on every hop traversed by a flit from all routers in the NoC to infer using their model. The traffic overhead becomes infeasible when every flit traversing the NoC generates one or more monitoring flits to send to the ML classifier containing the feature vector. Sudusinghe et al. [2022] even uses a dedicated service NoC to carry the monitored data. Galimberti et al. [2024] and Wang et al. [2024] use the same monitored NoC for monitoring messages, but generate less traffic by monitoring end-to-end messages or by averaging metrics. The monitoring overhead is addressed by three works [Vashist et al., 2019; Hathal et al., 2024; Wang and Halak, 2025] that use distributed monitoring approaches on averaged metrics. Nevertheless, Hathal et al. [2024] do not detail their distributed monitoring approach.

***Limitation* 4. Offline inference:** five works [Madden et al., 2018; Yao et al., 2020; Hu et al., 2023; Ding et al., 2024; Abdollahi et al., 2024] do not present scalability, model overhead, or monitoring overhead limitations. This occurs because they obtain their results offline, i.e., they are not applied in runtime to the target platform. Because the model is not properly implemented at runtime, it is impossible to obtain data on scalability or overhead.

***Limitation* 5. Router modification:** three works [Vashist et al., 2019; Wang et al., 2020, 2022; Sinha et al., 2021; Galimberti et al., 2024; Wang et al., 2024] require invasive modifications to the core router architecture. They embed security components, such as traffic monitors and ML detectors, directly into the router's internal pipeline. If the router is a 3PIP, it can't be modified because it is typically licensed and delivered as a protected "black box". Furthermore, a threat model may assume the router is malicious or hosts a Hardware Trojan (HT). Yet, the security mechanism is placed within that same untrusted component. This violates the security principle of a trusted computing base, as the component being monitored is also responsible for performing the monitoring.

The second issue, **confidence**, arises from the adoption of synthetic NoC traffic that does not reflect real application scenarios and from the use of high-level simulations that do not accurately reflect the systems' actual characteristics. The following confidence limitations were identified in the state-of-the-art:

***Limitation* 6. MCSoC model confidence:** 11 works [Sinha et al., 2021; Sudusinghe et al., 2021, 2022; Wang et al., 2020, 2022; Hu et al., 2023; Ding et al., 2024; Hathal et al., 2024; Galimberti et al., 2024; Wang et al., 2024; Wang and Halak, 2025] use gem5 to model their architecture, which can provide an arbitrary accuracy depending on the implemented models [Butko et al., 2012]. Six of those works [Sudusinghe et al., 2021, 2022; Hu et al., 2023; Ding et al., 2024; Hathal et al., 2024; Wang et al., 2024; Wang and Halak, 2025] use the gem5's Garnet NoC high-level model. Wang et al. [Wang et al., 2020, 2022] do not detail the NoC model used. Three works [Madden et al., 2018; Sinha et al., 2021; Abdollahi et al., 2024] use Noxim to model the NoC. Experiments conducted by the Author's research group with several NoC models show that results obtained with Noxim deviate significantly from those obtained with a clock-cycle-accurate simulator. Yao et al. [2020] use OPNET, a computer network simulator, thus neglecting both the characteristics of NoCs and embedded systems. Vashist et al. [2019] model their work using an unspecified system-level simulator. System-level simulators provide accuracy similar to or worse than that of other simulators, such as gem5. Some works assessed with system-level simulators claim area and power overhead estimates obtained with synthesis tools [Vashist et al., 2019; Sinha et al., 2021; Sudusinghe et al., 2022; Wang et al., 2020, 2022; Galimberti et al., 2024]. This gap prevents assessing the true performance impact of the proposed solution. A similar issue is present in Wang et al. [2024], which assess performance with gem5 and overheads with FPGA. Only Kulkarni et al. [2016] uses an RTL model, evaluated on an FPGA, although with only five routers.

***Limitation* 7. Predictive performance confidence:** two works only provide accuracy as the metric of their predictive performance results [Madden et al., 2018; Wang et al., 2020], which alone cannot represent the actual performance of the ML model. Accuracy alone is unsuited to scenarios with unbalanced class distributions, as in threat detection.

***Limitation* 8. Synthetic traffic:** half of the works generate synthetic traffic to assess their models [Kulkarni et al., 2016; Madden et al., 2018; Vashist et al., 2019; Yao et al., 2020; Sudusinghe et al., 2022; Hu et al., 2023; Ding et al., 2024; Abdollahi et al., 2024]. Using synthetic traffic does not accurately model a system because it neglects the complexity of an MCSoC, which can include a PE with a multitasking OS running on a processor and multiple NoCs. Furthermore, the evaluation with a single benchmark [Sudusinghe et al., 2021; Hathal et al., 2024] also suffers from drawbacks, as it learns to detect threats relative to that application's specific traffic pattern but likely fails to generalize, leading to false positives or negatives when faced with other applications.

***Limitation* 9. Attacks easy to detect:** 6 works propose detecting flooding DoS [Madden et al., 2018; Yao et al., 2020; Sinha et al., 2021; Sudusinghe et al., 2021; Ding et al., 2024; Wang et al., 2024]. Due to the nature of flooding DoS, which overwhelms its target with packets, it is easy to detect using static thresholds. Therefore, applying ML may not be the most suitable approach. Likewise, Wang et al. [2022] and Ding et al. [2024] also propose detecting packet drops, which is unusual in NoCs and is frequently ignored, as a packet drop would cause the system to enter an undefined state.

***Limitation* 10. NoC-only systems:** 6 works [Madden et al., 2018; Vashist et al., 2019; Yao et al., 2020; Hu et al., 2023; Ding et al., 2024; Abdollahi et al., 2024] propose threat detection in NoCs without accounting for the complexity of MCSoCs, which can include multiple NoCs, heterogeneous processors, and a complete software stack. Furthermore, Wang et al. [2022] and Hathal et al. [2024] do not provide details on their MCSoC architecture. Moreover, 4 works [Wang et al., 2020; Sinha et al., 2021; Galimberti et al., 2024; Wang et al., 2024] use complex processors in their systems with out-of-order, x86 architecture, or multi-level caches, which do not apply to embedded systems.

Table 3.5 summarizes how this Thesis seeks to mitigate the limitations found in current state-of-the-art research. The first column groups the limitations into two main issues: applicability and confidence. The second column lists the 10 limitations identified in the state-of-the-art, and the final column presents the corresponding mitigation approach. The applicability mitigations highlight the features of the proposed security threat detection approach, while the confidence mitigations highlight the features of the comprehensive methodology used to validate that approach.

Table 3.5: Mitigation approaches to address the state-of-the-art limitations. Source: the Author.

Issue	Limitation	This Thesis
Applicability	Scalability	**Lightweight**: Employs distributed, per-application models to divide computation and monitoring loads, ensuring scalability.
	Model Overhead	**Lightweight & Real-time**: Uses a software-based XGBoost model with a small memory footprint and low computational cost, enabling real-time inference.
	Monitoring Overhead	**Lightweight & Non-invasive**: A distributed Management Application and DMA-based monitoring framework (Chapter 4) minimize overhead.
	Offline Inference	**Real-time**: The solution is implemented and evaluated at runtime, providing low-overhead inference that enables real-time detection.
	Router Modification	**Non-invasive**: The approach requires no router modifications, interfacing non-invasively via the PE's Network Interface (NI).
Confidence	MCSoC Model Confidence	**Synthesizable Platform**: High confidence is achieved by using the Memphis-V platform (Chapter 4), a complete, synthesizable RTL MCSoC model.
	Predictive Performance Confidence	**Trustworthy Results**: High confidence is achieved by evaluating performance with appropriate metrics (e.g., Precision, Recall, F1-score).
	Synthetic Traffic	**Trustworthy Results**: Evaluated using realistic traffic from a suite of 5 MCSoC benchmarks, avoiding synthetic workloads.
	Easy-to-Detect Attacks	**Subtle Security Threats**: The threat model focuses on subtle, hard-to-detect attacks (e.g., temporary HTs, software-based collisions), not simple flooding.
	NoC-only Systems	**Synthesizable Platform**: Evaluation is performed on the complete MCSoC, including processors, OS, DMA, and a full software stack, not just an isolated NoC.

4. MEMPHIS-V: A NOC-BASED MCSOC PLATFORM

This Chapter presents the first original contribution of the Thesis: the RTL-modeled *Memphis-V* MCSoC. Memphis-V stands for Many-core Modeling Platform for *Phivers*. It enables the design of a NoC-based MCSoC with peripherals, supported by available debugging tools to verify hardware and software simultaneously. The MCSoC itself has been under development in the research group since 2009 [Carara et al., 2009], with continuous updates. In the context of this Thesis, the original contributions are:

- co-development of the RS5 RISC-V processor (Section 4.1.1);
- integration of the RS5 processor into the Memphis-V platform (Section 4.1);
- a new software stack (Section 4.2);
- a non-invasive monitoring method that uses the DMA-enabled NI to reduce the monitoring packet overhead (Section 4.3), designed to avoid penalizing the ML methods to be detailed later.

The guiding principle during platform development was to avoid modifying the NoC to implement security mechanisms in a non-invasive manner, mitigating Limitation 5 of the state-of-the-art. The state-of-the-art presented in Chapter 3 also revealed that ML techniques for threat detection in NoC-based MCSoCs are limited in confidence. This limited confidence is mainly due to the use of simplified MCSoC models (Limitation 6) in high-level simulations, often implemented with system-level simulators, which limit the accuracy of the results. Another limitation, frequently associated with high-level simulators, is assessing just the NoC instead of a complete MCSoC when implementing a threat detection method (Limitation 10). When only the NoC is assessed, the complexity of an MCSoC is ignored, and the traffic is frequently modeled synthetically (Limitation 8). Furthermore, system-level simulations can create unrealistic assumptions that compromise the applicability of the security threat detection approach, in terms of scalability (Limitation 1) and monitoring overhead (Limitation 3). Therefore, it is necessary to use an accurate MCSoC model to obtain reliable data.

Figure 4.1 shows an overview of the Memphis-V platform stack. The bottom of the stack is the hardware, called Phivers, which stands for Processor Hive for *RS5*. The middle of the stack is the OS kernel, called MAestro. On top of the stack is the software, composed of a set of benchmark applications (in orange) and the *Management Application* (MA, in red). Outside the stack are the generation and debugging tools provided by the Memphis-V framework, which help validate the platform. *The Memphis-V platform stack provides a complete MCSoC model that mitigates Limitation 10.*

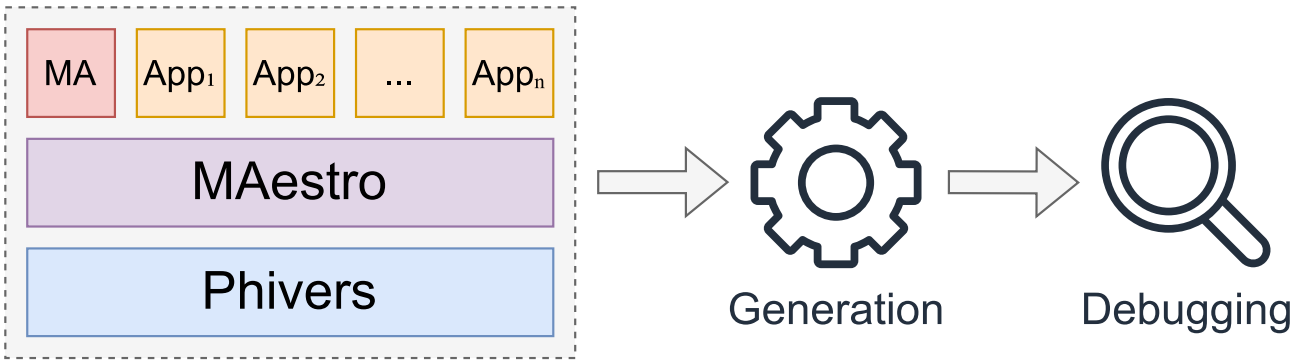


Figure 4.1: Memphis-V platform stack. Source: the Author.

The Management Application (MA) is responsible for managing Memphis-V [Ruaro et al., 2021; Dalzotto et al., 2021b]. MA transforms the management problem into a distributed application via a set of privileged tasks, enabling management to truly benefit from the high parallelism of MCSoCs. The **Observe-Decide-Act (ODA)** control loop [Hoffmann et al., 2013] divides the MA tasks into three different roles:

- **Observers**: extract information about the system or task status, such as deadline violations, communication volume, temperature, and communication latency;
- **Deciders**: decide what resources and when they will be reconfigured to meet application constraints or system budget. A multi-objective decision can be made based on multiple data sets from the *observation* phase;
- **Actuators**: enforce the decision made from the previous step by integrating the ODA loop into the hardware or the OS through techniques such as task migration and changes in the priority of the task scheduler.

Figure 4.2 presents the Memphis-V generation flow guided by YAML-formatted files and a generator program command line interface. The flow is divided into three phases: (*i*) *test case* creation (tc); (*ii*) application build (apps); and (*iii*) scenario definition (sc). The first phase, test case creation, builds Phivers, MAestro, and MA. The second phase, the application build, creates the application binaries to be evaluated in the test case. Finally, the third phase, scenario definition, combines the hardware and software definitions into a single scenario for evaluation. Multiple scenarios can be assessed using a single testcase and application set.

Memphis-V debugging features allow data extraction at simulation time. This data includes NoC traffic, instructions executed by the processor, memory access logging, scheduling status, and task logging. The debugging step is aided by a graphical tool [Ruaro et al., 2014] that allows visualization of the extracted data and provides several system metrics.

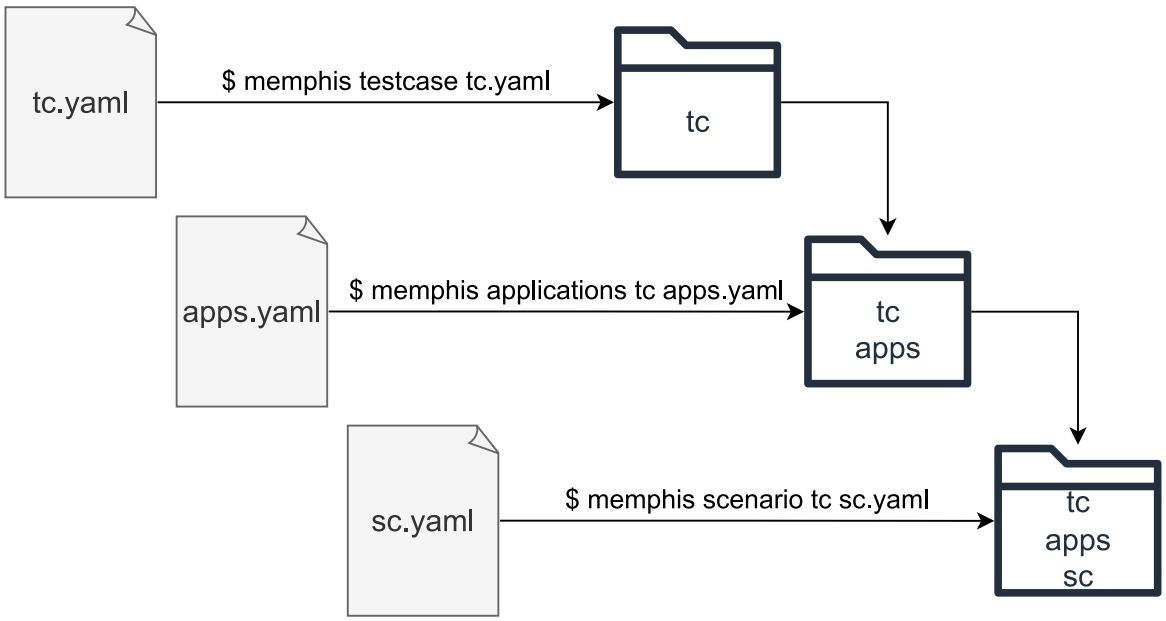


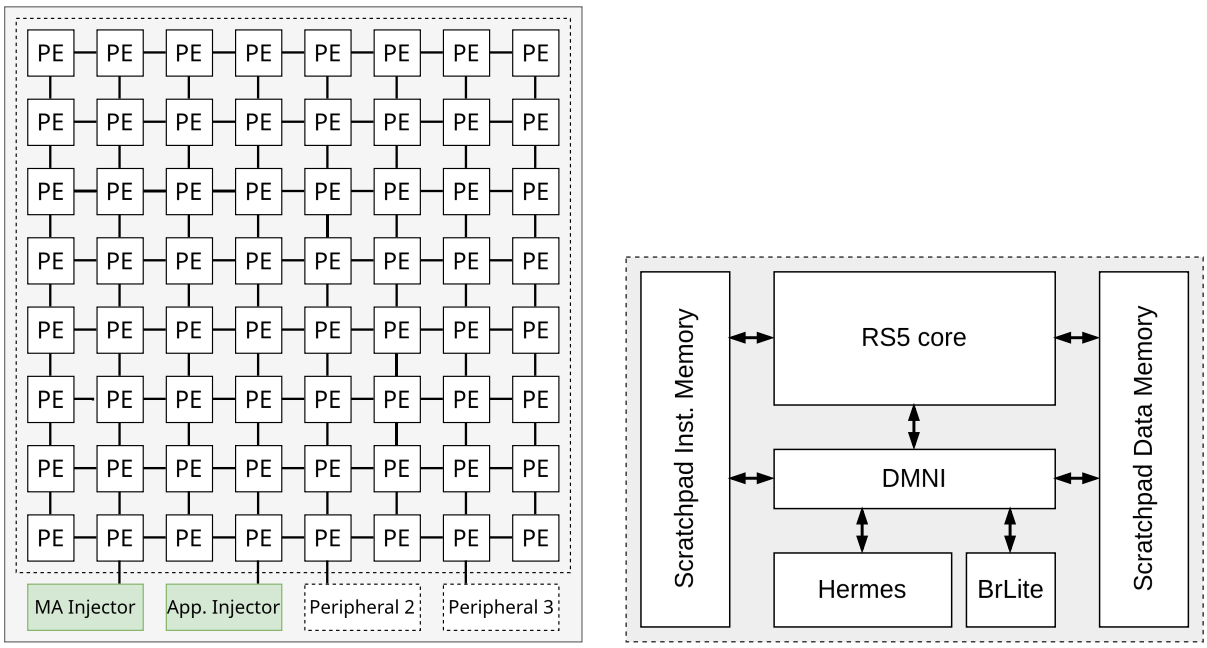
Figure 4.2: Memphis-V platform generation flow overview. The commands starting with $ transform the YAML description into the model. Source: the Author.

This Chapter is organized as follows. Section 4.1 details Phivers, the hardware model of Memphis-V. Section 4.2 details the software stack, comprised of the Memphis-V OS (MAestro), the application libraries, and the Management Application. Section 4.3 describes the main contribution of this Chapter, a non-intrusive monitoring framework for the Memphis-V MCSoC. Finally, Section 4.4 summarizes the contributions of Memphis-V to this Thesis and in relation to the state-of-the-art.

4.1 Phivers: Processor Hive for RS5

Figure 4.3 details the Memphis-V MCSoC. Figure 4.3a shows an overview of the homogeneous region of Memphis-V with PEs and the external region with peripherals connected to the routers at the borders. There are two default peripherals shown in the Figure, *Application Injector* and *MA Injector*. Both peripherals are responsible for dynamic loading tasks into the MCSoC.

Figure 4.3b details the organization of each PE. The PE core is the **RS5** processor [Nunes et al., 2024], detailed in Section 4.1.1. The processor is connected to two dual-port **scratchpad memories**, for instruction and data. Two NoCs are responsible for the MCSoC communication: a packet-switching **Hermes** router, and a broadcast **BrLite** router. Hermes [Moraes et al., 2004] has XY routing, round-robin arbitration, and input buffering. Routing is done in 5 clock cycles, and afterwards, the credit-based wormhole flow control transmits one flit per clock cycle. In Memphis-V, Hermes has 32-bit flits and input buffers that can store 8 flits.



(a) Memphis-V overview. (b) Memphis-V PE.

Figure 4.3: Memphis-V MCSoC overview and PE organization. Source: the Author.

BrLite is derived from BrNoC [Wachter et al., 2017], without backtracking[1], resulting in a smaller area footprint. BrLite transmits small control packets in a single flit. Its broadcast transmission exhibits low latency and fault tolerance due to its flooding behavior, occupying an area less than half that of a Hermes router. A BrLite flit has a 16-bit source address and a 20-bit payload. Additionally, it transmits 5 bits of packet ID and 1 bit for the *clear* flag, fields used to avoid retransmission and to clear internal buffers after the packet transmission, respectively. In summary, BrLite broadcasts 42-bit single-flit packets.

In Figure 4.3b, Hermes and BrLite interfaces to the RS5 core are abstracted by the Direct Memory Network Interface (**DMNI**). The DMNI joins the NI with DMA functions. Therefore, it is also responsible for exchanging flits directly between routers and memories. The DMNI has 16-flit input buffers, which frees Hermes wormhole control-flow while the DMNI interrupts the kernel to configure the memory address to receive a packet. The DMNI also provides a DMA-based monitoring framework detailed in Section 4.3.

Phivers is modeled at RTL with a SystemVerilog description, granting clock-cycle accuracy, thus mitigating Limitation 6.

[1] backtracking is a method for defining paths for source routing

4.1.1 RS5 processor

RS5 is a modular RISC-V processor based on the *RV32IZicntr_Zicsr_Zihpm* instruction set. It includes a base 32-bit integer ISA (RV32I) with performance counters and timers (Zihpm and Zicntr, respectively) and Control and Status Register (CSR) instructions (Zicsr). Furthermore, RS5 provides M- and U-mode, allowing an embedded operating system running in Machine-mode (M-mode) to handle traps and manage unprivileged applications executing in User-mode (U-mode).

RISC-V is an extensible ISA, and RS5 optionally implements the following Instruction Set Extensions (ISEs):

- **M**: integer multiplication and division;
- **A**: atomic instructions performing read-modify-write operations that enable locking primitives;
- **C**: compressed instructions;
- **V**: vector operations;
- **Zicond**: integer conditional operations;
- **Zcb**: additional compressed instructions;
- **Zkne**: instructions for accelerating the encryption and key-schedule functions of the AES block cipher;
- **Xosvm**: custom memory address translation extension.

The **Xosvm** extension (“X”: custom extensions, “osvm”: *offset and size virtual memory*) provides a straightforward Memory Management Unit (MMU). This extension enables paged memory organization by dividing memory into contiguous, statically sized pages for each process. It was introduced to address two significant issues: firstly, the standard RISC-V *Sv32* extension for 32-bit virtual memory systems requires the Supervisor-mode (S-Mode) alongside M- and U-Mode, leading to increased hardware overhead with the inclusion of extra CSRs and a Translation Look-aside Buffer (TLB); secondly, while RISC-V may offer a Physical Memory Protection (PMP) extension for securing embedded applications, it lacks virtual memory support needed by dynamic application loading.

The Xosvm extension requires three additional CSRs per memory, plus an additional register for enabling the extension. The `offset` registers control the page offset for each memory, restricted to powers-of-two values. Consider, for example, all page sizes equal to 4 *KB*. The offset for the first four user pages are `0x1000`, `0x2000`, `0x3000`, `0x4000`.

Equation (4.1) describes how to obtain the Physical Memory Address (*pma*) from the Virtual Memory Address (*vma*), i.e., the address generated by a code running in the processor. The *or* operation is equivalent to adding the page offset to *vma* (assuming proper alignment), but at a lower hardware cost.

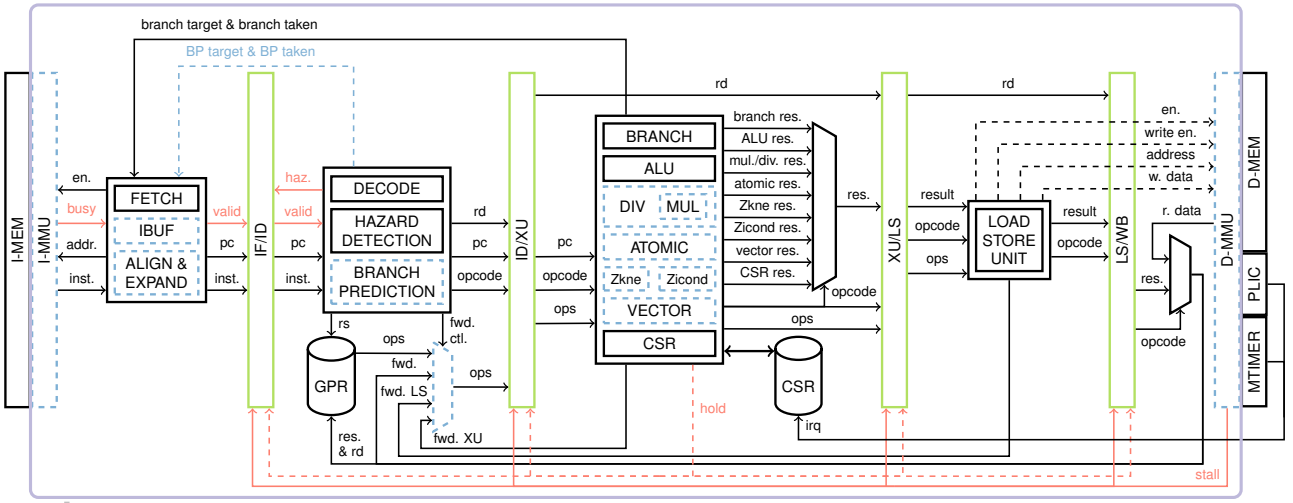
$$pma = vma \vee offset \tag{4.1}$$

The `size` registers contain the page size of each memory minus one. In the above example, for a page size equal to 4 *KB*, `size` register is equal to `0x0FFF`. The `mask` registers contain the allowed bits outside of the *size* register that can be set. For example, if the data memory starts in the address `0x01000000`, this value can be used in the `mask` register of the data MMU.

In the context of the Xosvm extension, memory isolation between pages is ensured, thus safeguarding the contents stored on each page. This is achieved through Equation (4.2). In this equation, *vma* is bitwise-AND-masked with the negated *mask* register. Thereafter, the result is bitwise-AND-masked with the negated *size* register. A memory access beyond the allowed range results in a value different than zero in Equation (4.2), and an exception is raised to the processor core.

$$exception = ((vma \wedge \neg mask) \wedge \neg size) \neq 0 \tag{4.2}$$

Memphis-V uses RS5 with *RV32IMACZicntr_Zicond_Zicsr_Zihpm_Zcb_Xosvm* instruction set. Figure 4.4 presents the RS5 organization, where green rectangles separate the four pipeline stages: instruction fetch (**IF**), instruction decode (**ID**), execution unit (**XU**), and load/store unit (**LS**).



The purple line delimits the core. Components in dashed blue lines are optional. Green components are temporal barriers. Signals that control the pipeline stall or generate bubbles are marked in red, and the hold signal is dashed because it occurs only when *M*, *A*, *V*, or *Zkne* ISEs are present.

Figure 4.4: RS5 Organization. Adapted from: Nunes et al. [2024].

The **IF** stage controls the reception of instructions from the instruction memory (I-MEM). This stage operates on virtual memory addresses and, optionally, connects to the *Xosvm* Instruction MMMU (I-MMU) to translate them into physical memory addresses. Performance can be enhanced through an optional instruction buffer (IBUF) that stores fetched instructions that are not passed to the following stage, which can occur due to a branch misprediction or a sequence of compressed instructions. If the *C* ISE is present, the IF stage adds a mechanism to align instruction addresses to 4-bytes while allowing compressed instructions placement in 2-byte aligned addresses. In this case, IF also includes expanding the compressed instructions before registering them for the following stage.

The **ID** stage extracts the opcode and operands from the instruction. Since the source operands are already decoded at this stage, it fetches them from the General Purpose Register (GPR) bank and performs data-hazard detection. An optional forwarding mechanism allows obtaining source operands directly from the results of the XU or LS stages, or from data memory reads, reducing the number of hazards and thereby enhancing performance. Furthermore, an optional static branch predictor can be included to reduce the branch penalty from 2 cycles to 1 cycle.

The **XU** stage is responsible for the computation using the Arithmetic Logic Unit (**ALU**). It is also responsible for branch control and for executing the *VECTOR*, *ATOMIC*, *Zkne*, and *Zicond* operations. The multiplication (MUL) unit can be included independently, while the inclusion of the division (DIV) unit implies the inclusion of MUL, resulting in the *M* ISE. XU stage also executes CSR operations, enforces privileges, and handles traps.

The **LS** stage interacts with the data memory (D-MEM). It also controls the *stall* signal when a memory operation is pending. An optional Data MMU (D-MMU) can virtualize D-MEM addresses when using the Xosvm ISE. Finally, after the memory operation is complete, the result is written back to the GPR bank. The data from the writeback is always forwarded (*fwd.* signal), as opposed to LS and XU forwardings (*fwd. LS* and *fwd. XU*, respectively), which are optional.

RS5 uses the Platform Local Interrupt Controller (PLIC), an interrupt treatment approach defined by SiFive, Inc. [2020] that manages global or external interrupts generated by peripheral devices. PLIC is configurable and provides an interface via Memory-Mapped Registers (MMR). RS5 also supports timer interrupts through a standardized machine timer (MTIMER).

Memphis-V uses the RS5 organization with branch prediction, forwarding, and the instruction buffer capable of storing 2 instructions. The NoC interrupts managed by the DMNI are routed to the RS5 through the PLIC in Memphis-V. Furthermore, MAestro uses the MTIMER to generate timer interrupts to control the scheduling time slice.

4.2 Software stack

Memphis-V provides a complete software stack composed of: (*i*) the MAestro kernel (Section 4.2.1); (*ii*) support libraries; (*iii*) a Management Application (Section 4.2.2); and (*iv*) a set of benchmark applications.

The support libraries are *libmemphis* and *libmutils*, leveraged by the complete C standard library (*libc*) provided by *newlib-nano* [Red Hat, 2024]. The Memphis-V library, called *libmemphis*, comprises four procedures to obtain information about the many-core, such as the PE address where the application is executing, and six message-passing functions, alongside procedures to aid the implementation of MA tasks. Furthermore, the Memphis-V utilities library, *libmutils*, provides additional data structures for both MAestro and unprivileged applications.

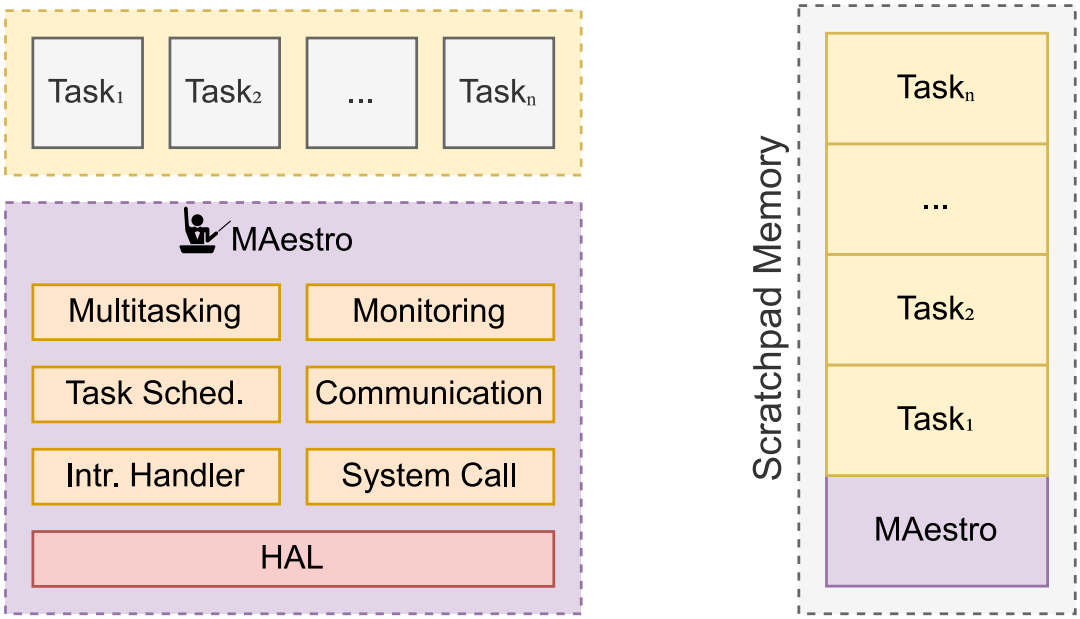
Memphis-V bundles 14 standard benchmarks to evaluate the many-core architecture. *The standard set of benchmarks provided by Memphis-V, along with the possibility of developing more applications through the support libraries, mitigates the Limitation 8 by providing realistic workloads to generate traffic in the NoC.*

4.2.1 MAestro: the Management Application Operating System

Figure 4.5 illustrates MAestro, an embedded OS that is loaded into all PEs. MAestro uses a microkernel design, where all non-core functionality, such as management, executes in separate processes at a lower privilege level. Furthermore, this microkernel approach results in a kernel memory footprint of only 20 KB.

The core functionalities of MAestro, highlighted inside the purple rectangle in Figure 4.5a are:

- **Hardware Abstraction Layer (HAL)**: the HAL abstracts the RS5 processor, providing CSR access, memory management, and context switching;
- **System calls**: MAestro provides 6 POSIX system calls to handle standard output, dynamic memory allocation, and process control, along with MCSoC-specific system calls;
- **Interrupt handling**: MAestro handles DMNI interrupts via PLIC. The DMNI abstracts both Hermes and BrLite routers. MAestro also handles MTIMER interrupts for scheduling timeslices;



(a) MAestro executes in M-mode, with purple background. User tasks execute in U-Mode, with a yellow background.

(b) Memory layout. The memory is divided into instruction and data.

Figure 4.5: MAestro overview. Source: the Author.

- **Multitasking**: through Xosvm (Section 4.1.1), MAestro supports multitasking with dynamic application loading. Figure 4.5b illustrates the contiguous memory pages when multiple tasks are allocated into the scratchpad memory;
- **Task scheduling**: MAestro supports preemptive scheduling via timer interrupts from the MTIMER. The built-in schedulers support best-effort tasks via round-robin scheduling and soft Real-Time (RT) tasks via Least Slack Time (LST);
- **Communication**: MAestro provides drivers and a protocol stack to support communication through the Hermes NoC. A Message-Passing Interface (MPI)-like Application Programming Interface (API) provides communication access to user applications. This mechanism also supports sending and receiving messages in both directions: from kernel-to-task, from peripheral-to-task, and from peripheral-to-kernel, resulting in a message-passing Inter-Process Communication (IPC) mechanism;
- **Monitoring**: MAestro monitors the system through Low-Level Monitors (LLM), which fetch monitored metrics without complex computation. The OS can periodically invoke the LLMs, collecting data from, e.g., instruction counters for power and temperature estimation, or from Quality of Service (QoS) monitors. LLMs can also be invoked as reactions to system events, such as upon packet reception. Data generated by the LLM is handled by the monitoring framework (Section 4.3).

The MAestro's communication API has two main message classes: *request* and *delivery*. A task that wants to send a message generates a *delivery* only when it has received a *request*. The *request* indicates that the receiver has allocated buffers and is ready to receive a message from the sender task. The receiving task is blocked until the message is delivered, thus preventing it from receiving messages from any source other than the intended one.

An alternative is to wait for an *available* message before sending the *request*. This way, a task can receive a message without indicating its source, i.e., it requests a message from the first task that sends an *available*. This communication mechanism enables a message-passing Inter-Process Communication (IPC).

Figure 4.6 shows the messaging API diagram of a producer-consumer application, containing all three steps: *available*, *request*, *delivery*. When the producer wants to send a message, a *data available* packet is generated containing the producer task identifier and location. When the consumer calls the receive message function, it checks a *data available* queue for senders trying to send messages. The consumer then sends a *message request* to the producer, using the producer's task identifier and location from the *data available* packet. Finally, the kernel at the PE executing the producer task dispatches the stored message inside a *message delivery* packet.

4.2.2 Management Application

Figure 4.7 presents an example of an MCSoC managed by the MA approach. Each gray tile in the Figure is a PE, either free or occupied by user tasks. The blue, red, and green tiles represent Observers, Deciders, and Actuators tasks, respectively. The MA tasks can be mapped to different system positions, and their number can also be defined based on workload requirements. Tasks can be mapped closely together or allocated to strategic regions of the chip where the monitoring load is higher.

MAestro provides three services that are necessary to support MA:

- **Low-Level Monitors (LLM)**: periodically, or triggered by events, pull raw data from hardware and redirect to Observers. LLM examples include task execution profiling (whether computation- or communication-intensive), RT constraints, power, thermal, communication latency, and core utilization;
- **Actuation Enforcer (AE)**: implements drivers and provides APIs to physically apply the requests from Actuation tasks. An example is the task migration API, which requires privileged memory access;
- **Communication API**: the OS provides a secure communication method for MA tasks, ensuring user tasks cannot tamper with the system management.

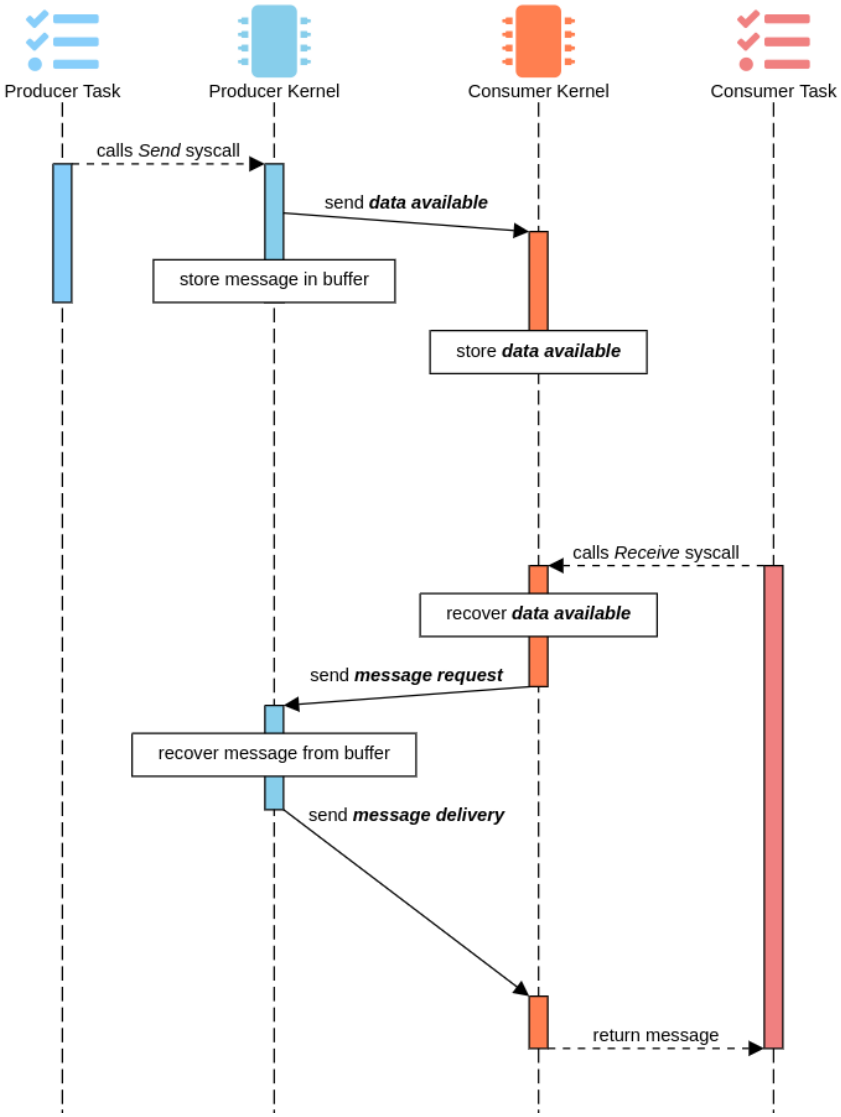


Figure 4.6: Sequence diagram of the message-passing API. *Data available* packet is optional. Source: the Author.

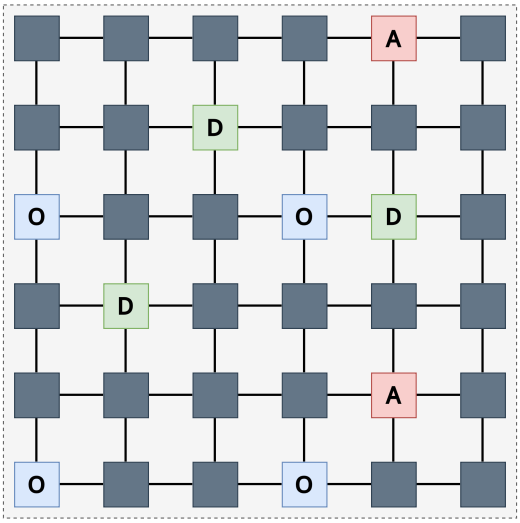


Figure 4.7: MA organization. **O**: Observers; **D**: Deciders; **A**: Actuators. Source: the Author.

Figure 4.8 depicts the MA framework model proposed by Ruaro et al. [2021]. In summary, the LLM running at the OS of *each* core generates messages periodically. Observers related to task performance, system budgets, and user commands (to force actuations directly) receive and abstract the monitored data to achieve awareness of the system status. Observers know the system *goals* and can convert raw **Monitoring data** into **Objectives**. Objectives are sent to the Deciders, that convert them into **Actions** using algorithms that detect when and which resource needs adaptation. If necessary, the Decider triggers an Actuator, which implements the protocols to dynamically change the resources by interacting with the AE at the OS level.

A goal is a high-level, loosely defined plan of the system, such as a demand for more bandwidth or less power [Rahmani et al., 2018]. Objectives are the concrete actions to meet such demand, such as establishing a dedicated network path or reducing the system frequency and voltage.

Memphis-V is bundled with an MA consisting of: (*i*) a Mapper Task, responsible for application mapping [Dalzotto et al., 2021a] with reactive defragmentation, alongside playing a role as the task migration Actuator [Dalzotto et al., 2022a]; (*ii*) a RT QoS Decider, capable of requesting task migrations when RT tasks miss deadlines beyond a threshold; and (*iii*) an RT Observer, capable of monitoring RT metrics.

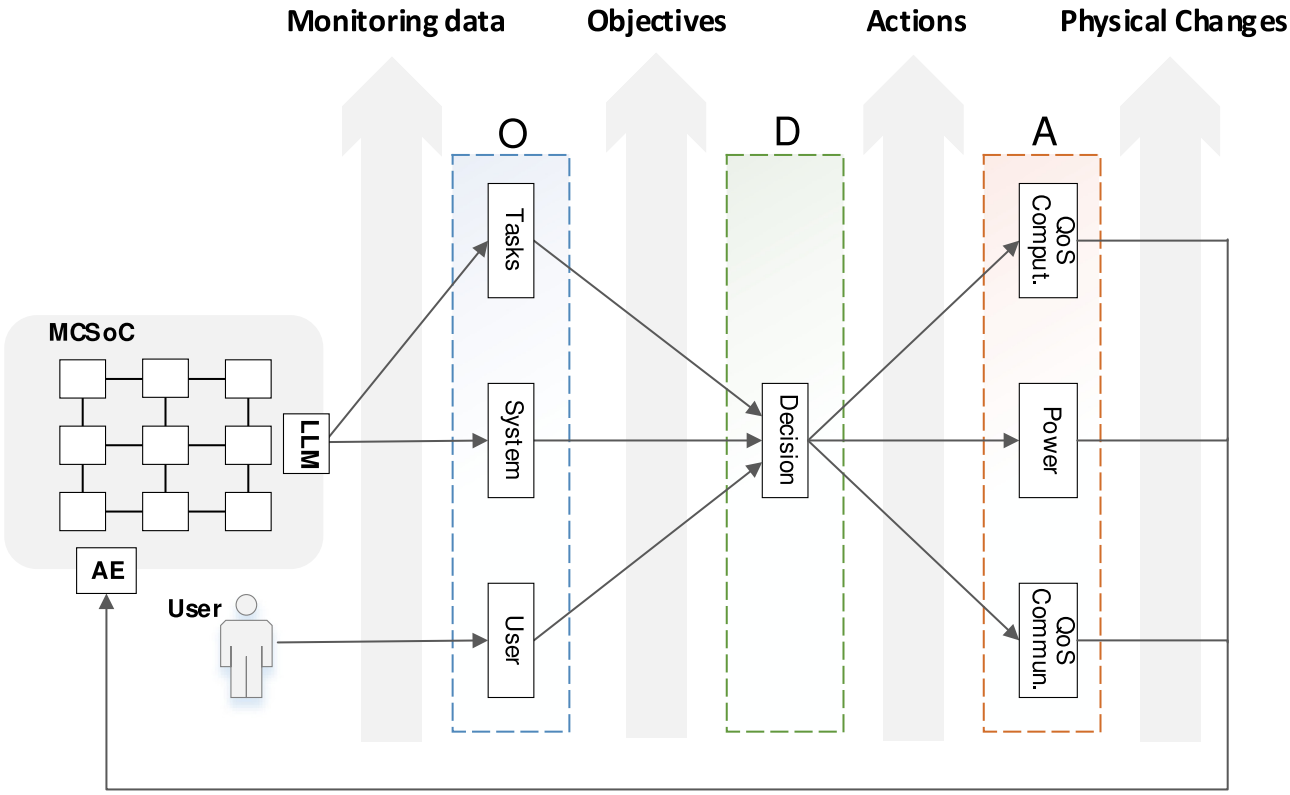


Figure 4.8: ODA Model used by the MA paradigm. There is one LLM for each core. The AE is implemented by the OS running in each core. Arrows represent the communication between the entities. Source: Ruaro et al. [2021].

The distributed MA implemented by Memphis-V reduces management message volume by 20% and increases management throughput by at least 11.7% compared to a clustered[2] approach, while providing management scalability by dividing management roles within the ODA control loop [Dalzotto et al., 2021b]. *The distributed MA approach mitigates Limitation 1, providing scalability to management goals.*

4.3 Monitoring framework

Memphis-V offers a monitoring framework leveraged by the DMNI. It uses DMA to transfer monitored packets directly into a First-In-First-Out (FIFO) structure in memory, reducing messaging protocol overhead and avoiding interrupts. It is based on previous work [Dalzotto et al., 2022b], in which single-flit monitoring packets were sent via BrLite and inserted directly into a monitoring table in memory.

Figure 4.9 shows the setup procedure of the monitoring framework executed by an Observer task. In the first step (1), the Observer task allocates a memory region that acts as the monitoring FIFO. The `memphis_mkfifo` function calls the PE kernel with the Observer task (blue background), which allocates the memory for the FIFO in kernel space and configures the DMNI. The DMNI configuration involves setting the allocated memory pointer, the number of flits of the monitoring packet, and the number of packets the FIFO can store.

Step 2 involves the monitoring announcement of the Observer task. An Observer task calls the `mon_announce` function, indicating the *type* of the metric it can monitor. The type defines both the format of the monitoring packet and the LLM's monitoring trigger condition. For example, the QoS type periodically monitors RT metrics during task execution and sends a 3-flit monitoring message.

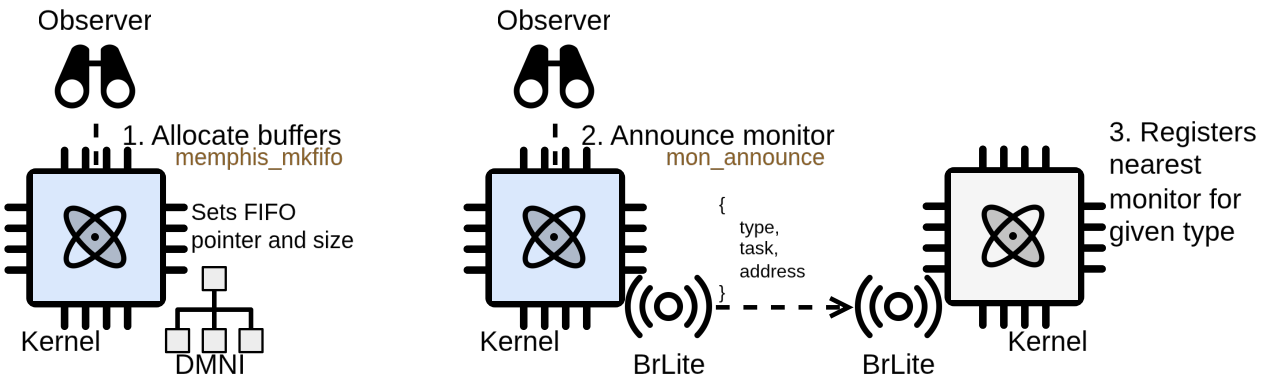


Figure 4.9: Monitoring framework setup procedure. Source: the Author.

[2] clustered management is a strategy that divides the MCSoC into regions, named clusters, with a dedicated management PE for each cluster.

In Figure 4.9, the monitoring announcement is propagated by broadcast with Br-Lite. Upon reception (step 3), the PE verifies whether the announced Observer is the nearest one for the given type. For example, if the PE at address 0×0 has registered the nearest QoS Observer at PE 4×4, and a new QoS monitor announcement arrives indicating its presence at PE 2×2, then PE 0×0 registers it as the nearest QoS Observer.

Figure 4.10 shows the monitoring framework during execution after the setup phase. The LLM at the monitored PE generates a monitoring message at a configured period or in response to an event (step 4) and sends it to the nearest Observer. The monitoring packet is prepended by the Hermes header flit. This flit indicates a 16-bit target address and the 8-bit flags for peripheral message control. The monitoring framework uses the remaining 8 bits of the 32-bit flit to encode instructions to the DMNI right at the first flit of the packet.

When the PE executing an Observer task receives a monitoring packet, the DMNI writes it directly to the memory location at the configured address, using the FIFO structure (step 5). To write the packet to memory, the DMNI first checks whether the FIFO has free space. The DMNI implements two MMR-accessible semaphores to count free and occupied space in the FIFO, avoiding a race condition. If the FIFO is full, the DMNI drops the monitoring packet to avoid stalling the NoC. This data loss is acceptable, as losing monitoring packets does not cause system failure. Furthermore, properly tuning the FIFO size and the generation rate of monitoring packets reduces the likelihood of packet drops.

The Observer task retrieves monitoring packets (step 6) through the `memphis_receive_any` function. While this interface is consistent with the standard Memphis-V message-passing API, the underlying kernel mechanism bypasses the control overhead of the *available*, *request*, and *delivery* messages. Furthermore, the DMNI generates a dedicated interrupt request for monitoring packets, which can be configured to trigger only when the PE is in an idle state. This strategy reduces context-switching overhead and has been shown to decrease interrupt frequency by up to 47% [Dalzotto et al., 2022b], significantly lowering the performance impact on the management layer.

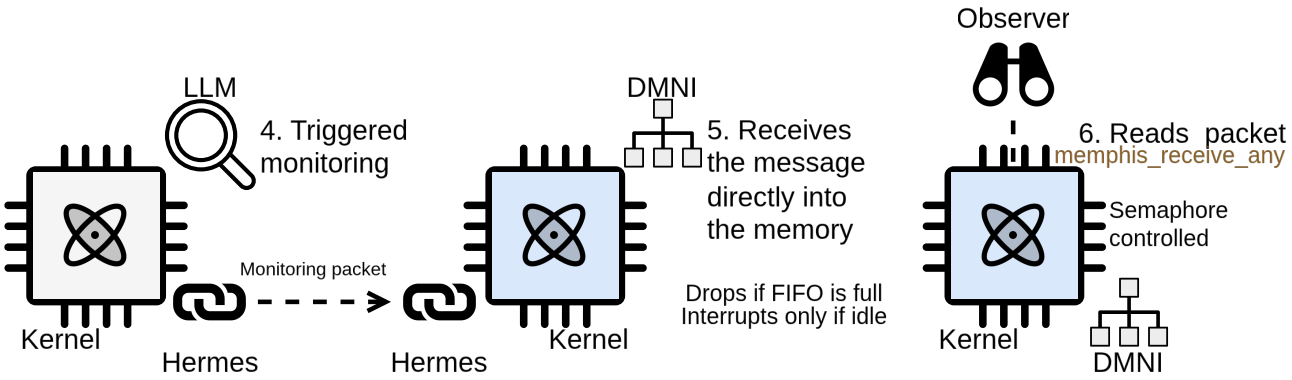


Figure 4.10: Monitoring framework execution. Source: the Author.

Beyond monitoring, this framework serves as a low-latency communication channel for management tasks. Through a specialized system call, an Observer can send packets directly to a Decider task's monitoring FIFO, bypassing the standard messaging protocol. This direct-transfer approach allows management packets to be queued without requiring an immediate interrupt at the target PE. In the context of this Thesis, leveraging the monitoring framework for inter-task communication within the Management Application enabled real-time security threat detection, reducing the total detection latency by up to 95%. For example, the detection latency for threats affecting the *Dijkstra* benchmark (described in Section 6.1), went from 1123.7 μs to 52.6 μs.

This monitoring framework mitigates the Limitation 3 due to its low performance overhead design, and the Limitation 5, as it does not require router modification.

4.4 Final Remarks

This Chapter presented Memphis-V, a MCSoC platform designed to enable research in security and management. The platform comprises: (*i*) the Phivers hardware model, featuring the modular RS5 RISC-V processor and a heterogeneous NoC infrastructure; (*ii*) the MAestro operating system, which provides multitasking and message-passing support; and (*iii*) a set of generation and debugging tools.

The development of Memphis-V directly addresses the confidence limitations identified in the state-of-the-art (Chapter 3). Unlike works relying on high-level simulators (e.g., gem5, OPNET) or standalone NoC models (e.g., Noxim, BookSim), Memphis-V provides a complete **synthesizable platform** modeled at RTL. This ensures that the evaluation of security mechanisms accounts for the cycle-accurate behavior of the entire system, including processor pipelines, OS overhead, and peripheral interactions. Furthermore, integrating the Management Application with the hardware-supported monitoring framework addresses the applicability limitations of scalability and monitoring overhead. By providing a realistic, full-stack environment, Memphis-V serves as the foundation for this Thesis.

5. FRAMEWORK OVERVIEW

The state-of-the-art presented in Chapter 3 revealed that ML techniques for threat detection in NoC-based MCSoC are limited in applicability. The primary barriers are the trade-offs between scalability (Limitation 1) and the associated implementation costs, specifically ML model overhead (Limitation 2) and monitoring overhead (Limitation 3). Consequently, a significant challenge lies in balancing detection performance with the strict resource constraints of embedded MCSoCs.

This Chapter provides an overview of the main contribution of this Thesis: a lightweight, real-time framework for detecting security threats. Unlike state-of-the-art approaches that monitor low-level network metrics, this proposal uses application-level behavior. By learning the expected temporal communication signature of an application, in terms of end-to-end latency, the proposed framework can detect deviations caused by malicious interference. This strategy is justified because the communication behavior of a legitimate application is bounded by the platform's hardware characteristics and system software overheads. Therefore, any significant deviation from this learned profile serves as an indicator of a security threat. Modeling this behavior is enabled by the Memphis-V platform (Chapter 4), which provides the necessary high-fidelity hardware model, OS, and benchmark applications to generate realistic training data.

The remainder of this Chapter is organized as follows. Section 5.1 describes how security threats are modeled and implemented in the Memphis-V platform. Section 5.2 details the communication behavior of applications on the target platform and analyzes their profiling with and without security threats, demonstrating the limitations of static thresholding. Section 5.3 presents the overview of the proposed ML flow, from dataset generation to regression model training and testing. Finally, Section 5.4 presents the final remarks on the proposed approach.

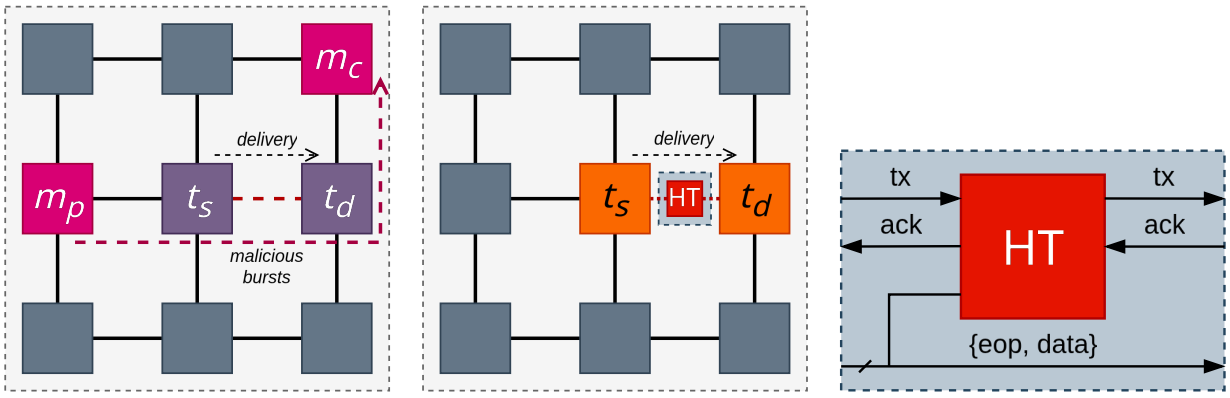
5.1 Threat Model Implementation

While Section 2.9 defined the theoretical threat model and the types of attacks relevant to MCSoCs, this section details the specific implementation of the threats adopted in this Thesis. The aim is to detect subtle traffic anomalies caused by threats originating from either hardware or software, provided their malicious activity manifests as a disturbance in NoC traffic that affects the performance of a legitimate application. Figure 5.1 illustrates the two specific threat models implemented in the Memphis-V platform to generate such disturbances: a **Malicious Application** (m_{app}) and a **Hardware Trojan** (HT). These represent two distinct classes of attacks: one that exploits NoC links via legitimate arbitration, and the other that manipulates the underlying hardware protocol.

Figure 5.1a presents the MCSoC with m_{app} interfering a legitimate application represented by a source task (t_s) that sends messages to a destination task (t_d). The m_{app} consists of a malicious producer (m_p) sending messages to a malicious consumer (m_c). m_c continuously injects small packets at randomized intervals (e.g., between 100 and 200 μs). This specific traffic pattern, consisting of small packets and random intervals, is designed to create intermittent, non-deterministic collisions on links shared with the legitimate application. Unlike a flooding attack, this behavior does not saturate the network, making it difficult to detect with rule-based approaches while still degrading the Quality of Service (QoS) of the victim application. The example in Figure 5.1a highlights a link (red-dashed) where this contention occurs due to the deterministic XY routing algorithm.

Figure 5.1b shows the HT inserted at a NoC link. This HT also aims to be stealthy by causing subtle performance degradation. Figure 5.1c details the HT implementation. The HT inspects the flit data as it traverses the link. It is triggered only when it identifies a packet header corresponding to a *message delivery* (Section 4.2). Upon activation, the HT introduces a temporary communication halt. It masks both the transmission request (*tx*) and reception acknowledgment (*ack*) signals, effectively blocking the link and stalling packet flow. To ensure stealth, the HT is parameterizable to activate only probabilistically (e.g., with a 25% chance on a *message delivery*) and for a random number of clock cycles. This sporadic activation mimics natural link competition, making the attack challenging to distinguish from legitimate system variability.

The interference caused by both threat models results in traffic anomalies that are difficult to isolate using static thresholds. As will be demonstrated in Section 5.2, the latency impact of these threats often falls within the standard deviation of valid traffic scenarios (e.g., similar to that of a longer path). This necessitates a dynamic, learning-based detection mechanism capable of distinguishing between valid system variability and malicious interference. *The proposed threat model, designed to be stealthy and hard to detect, directly addresses Limitation 9.*



(a) Malicious application competing for a link.

(b) HT inserted at a NoC link.

(c) Link with HT.

Figure 5.1: Security threats implementation. Source: the Author.

5.2 Application Modeling and Profiling

Legitimate applications executing on NoC-based MCSoCs exhibit distinct **temporal signatures** in their communication patterns. These signatures are derived from the periodic nature of the application execution, which dictates the sequence and timing of message exchanges. Profiling these signatures provides a baseline of expected behavior, including communication patterns and performance characteristics, which is essential for distinguishing valid executions from those under attack.

Figure 5.2 presents the CTG model of a pipelined *Moving Picture Experts Group* (MPEG) decoding application composed of five tasks (t_i). Task t_0 initiates the application by producing frames to be decoded by the pipeline stages (t_1, t_2, t_3) and outputted by t_4. This structural dependency results in a consistent, periodic pattern of message exchanges represented by the edges $e_{0,1}$, $e_{1,2}$, $e_{2,3}$, and $e_{3,4}$.

To demonstrate these temporal signatures, two distinct mappings of the MPEG application are shown in Figure 5.3. Figure 5.3a illustrates an optimal contiguous mapping where all communicating tasks are adjacent, with 1 hop of distance. Figure 5.3b illustrates a fragmented, non-contiguous mapping, resulting in an average distance of 2.5 hops between communicating tasks.

Figure 5.4 analyzes the impact of these mappings of the MPEG application on packet latency, where each graph represents a given edge. The x-axis represents the time, and the y-axis the end-to-end packet latency. The curves labeled C (green) represent the behavior of the optimally mapped application (Figure 5.3a). The curves labeled NC (brown) correspond to the non-contiguous mapping (Figure 5.3b).

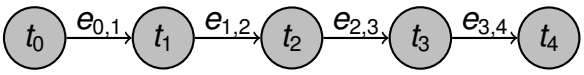
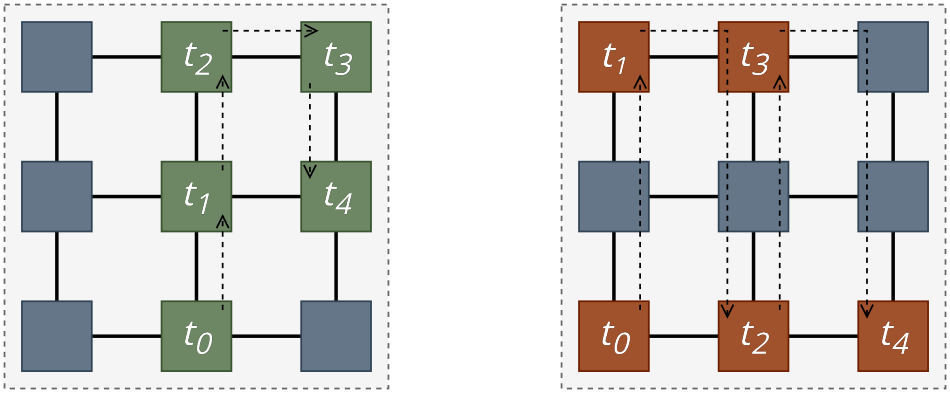


Figure 5.2: CTG of the MPEG application. Source: the Author.



(a) Contiguous mapping.

(b) Non-contiguous mapping.

Figure 5.3: Mappings for the MPEG application. Source: the Author.

In Figure 5.4, a comparison between C and NC reveals that while the increased hop count raises the absolute latency value, it does not alter the temporal profile, i.e., the shape of the curves. Hence, the application's temporal signature is maintained across system-level variations. While absolute latency depends on placement and link competition, the relative pattern of communication over time remains a consistent identifier of the application.

Figure 5.5 introduces the security threats described in Section 5.1, along with a mapping with a high hop count for $e_{2,3}$ (Figure 5.5c). Figure 5.5a shows the MPEG application under attack from m_{app}, where it competes for the links used by edge $e_{2,3}$. The malicious packets are small, with 80 flits, taking approximately 2.5 μs to traverse the three hops between m_p and m_c using Hermes at 100 MHz. This application generates traffic at random intervals between 100 and 200 μs to introduce unpredictable jitter.

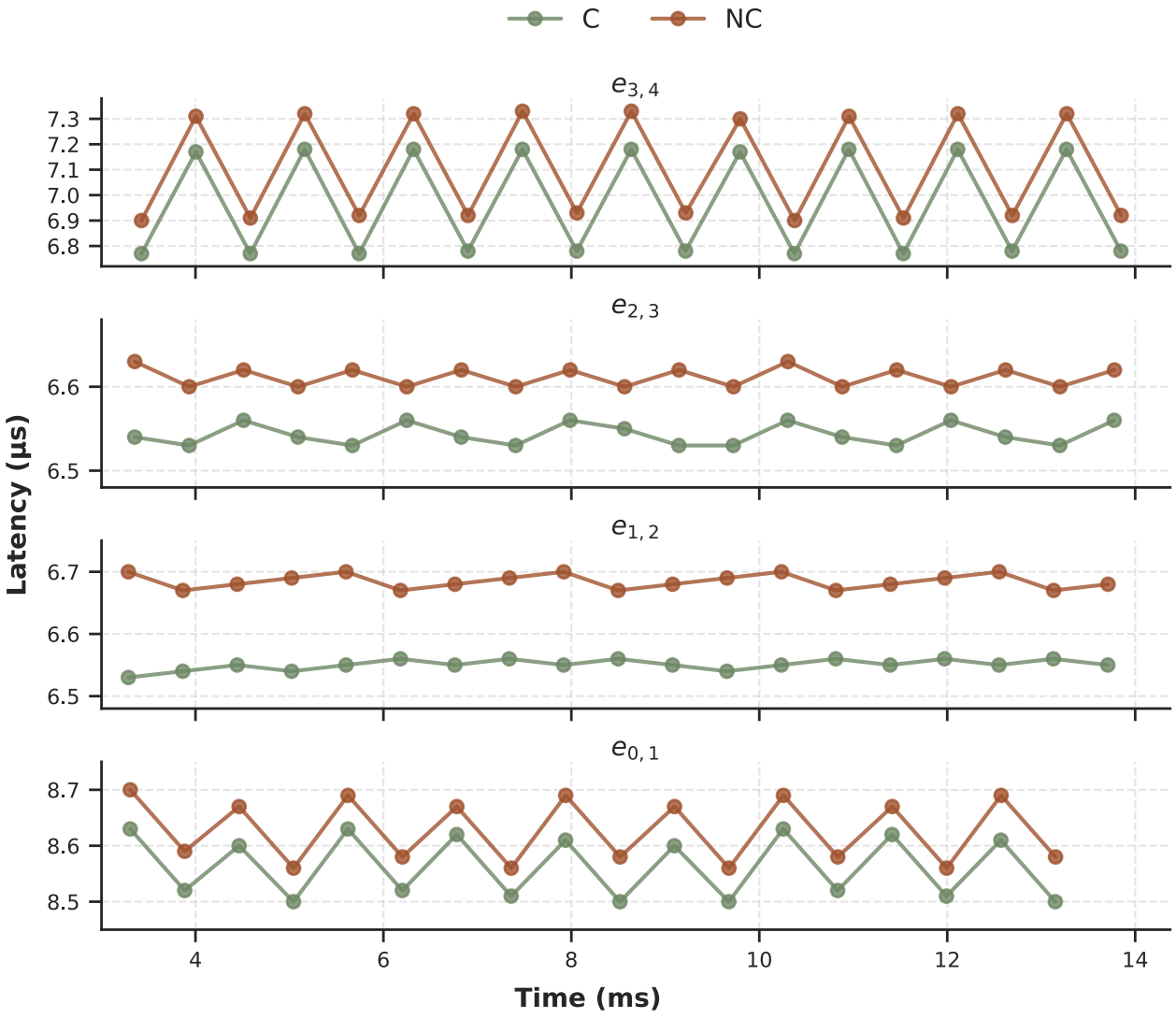
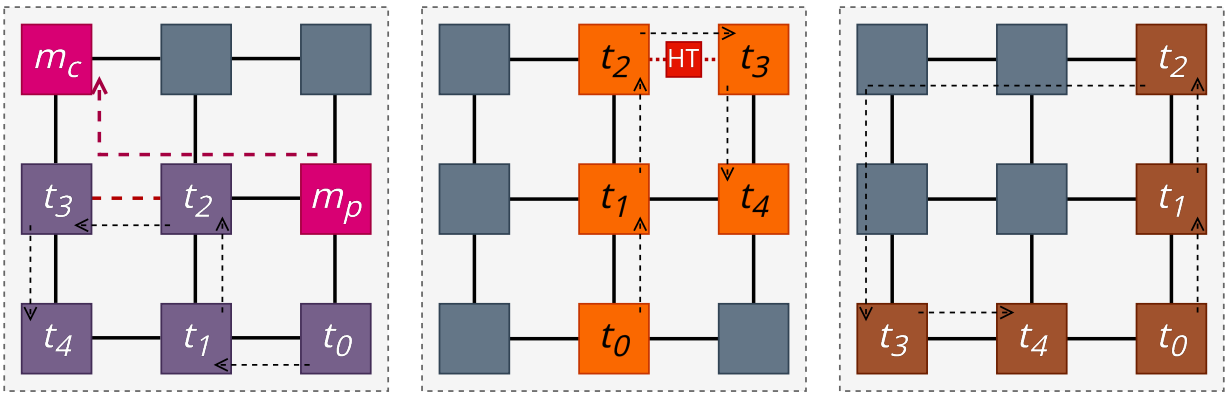


Figure 5.4: Latency of transferred messages through the application execution considering different mappings. Warm-up is omitted from the graph. Source: the Author



(a) Infected with malicious application.

(b) Infected with HT.

(c) Non-infected with high hop count for $e_{2,3}$.

Figure 5.5: MPEG mappings with and without security threats affecting $e_{2,3}$. Source: the Author.

Figure 5.5b shows the application under attack from an HT inserted into the link of edge $e_{2,3}$. The HT is configured to block message delivery packets with 25% chance for a short duration of 16-32 clock cycles. Figure 5.6 isolates the latency of edge $e_{2,3}$ across four scenarios to highlight the detection challenge:

- $\#h = 1$: optimal mapping with 1 hop of distance (Figure 5.3a);
- $\#h = 4$: distant mapping with 4 hops of distance (Figure 5.5c);
- $\#h = 1w/m_{app}$: optimal mapping with 1 hop of distance with m_{app} competing for its link (Figure 5.5a);
- $\#h = 1w/HT$: optimal mapping with 1 hop of distance with the HT (Figure 5.5b).

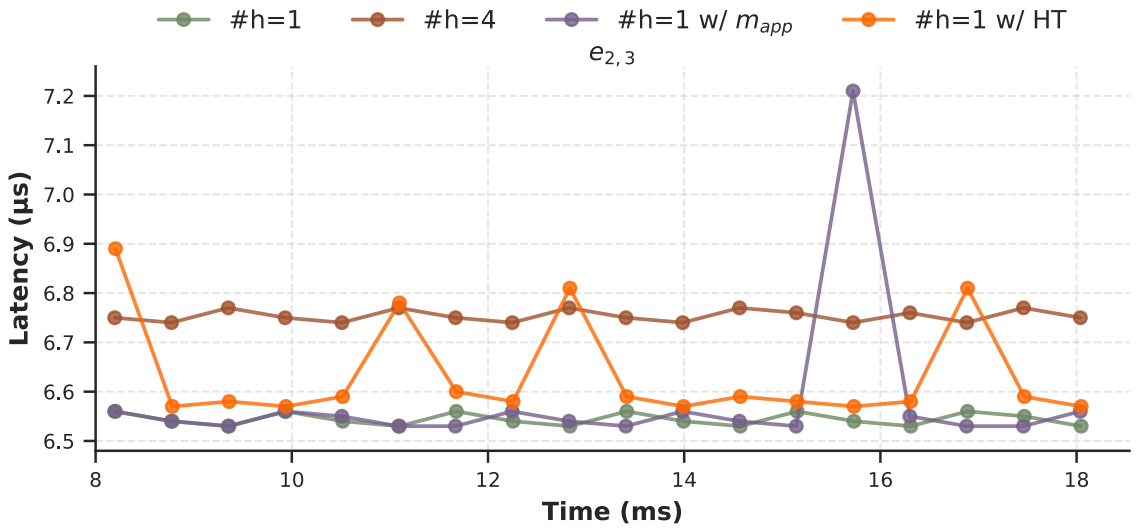


Figure 5.6: Latency of transferred messages through the application execution with and without security threats. Source: the Author

In Figure 5.6, when the optimal mapping ($h = 1$) is compared to the distant mapping ($h = 4$), an increase in latency magnitude is observed while the temporal signature remains unchanged. This result confirms that system variations may affect latency magnitude but do not alter the application's temporal signature.

The purple line in Figure 5.6 ($\#h = 1\ w/\ m_{app}$) shows that the malicious application creates subtle deviations. Aside from a collision peak at $\approx 15.8\,ms$, the impact is masked by the NoC's high bandwidth, meaning that the threat could not affect the legitimate application. However, the HT attack ($\#h = 1\ w/\ HT$, orange line) produces distinct spikes (e.g., at $\approx 11.2\,ms$). These spikes represent anomalies that expose the failure of static thresholding.

Considering a simple threshold-based detector, setting a strict threshold (e.g., $6.7\,\mu s$) to detect minor anomalies would incorrectly flag the valid $\#h = 4$ mapping as an attack, as its legitimate latency exceeds the threshold due to distance. Conversely, raising the threshold (e.g., to $6.8\,\mu s$) to accommodate the distant mapping would drastically reduce detection sensitivity, causing the system to miss 3 of the 4 anomaly spikes of $\#h = 1\ w/\ HT$.

To address this issue, this Thesis proposes a regression model that learns to predict expected packet latency as a function of dynamic system parameters. This allows the detection mechanism to adapt its expectations to the current context, flagging only latencies that deviate from the *learned* baseline.

5.3 ML-driven Security Threat Detection Framework

Figure 5.7 illustrates the proposed ML-driven security threat detection workflow, designed to bridge the gap between offline model training and runtime threat detection in resource-constrained MCSoCs. The methodology is divided into four distinct phases, which are executed for each profiled application of the MCSoC: (*i*) Dataset Creation; (*ii*) Regression Model Creation; (*iii*) Runtime Detection; and (*iv*) Testing.

The first phase, **Dataset Creation**, focuses on generating a dataset that captures the behavior of a target application under various system conditions and without security threats. It begins with **Population Creation**, where a set of *Training scenarios* is defined. A distinct set of *Testing scenarios* is held out for the final evaluation. These scenarios encompass different mappings of the target application tasks onto the MCSoC, ensuring the model generalizes across different hop counts and link competition levels. The scenarios are executed on the Memphis-V platform via **RTL Simulation**, generating raw execution logs. The **Data Extraction** and **Data Pre-processing** steps parse these logs to isolate the relevant application messages into a structured *Training Dataset*. The Dataset Creation phase is detailed in Chapter 6.

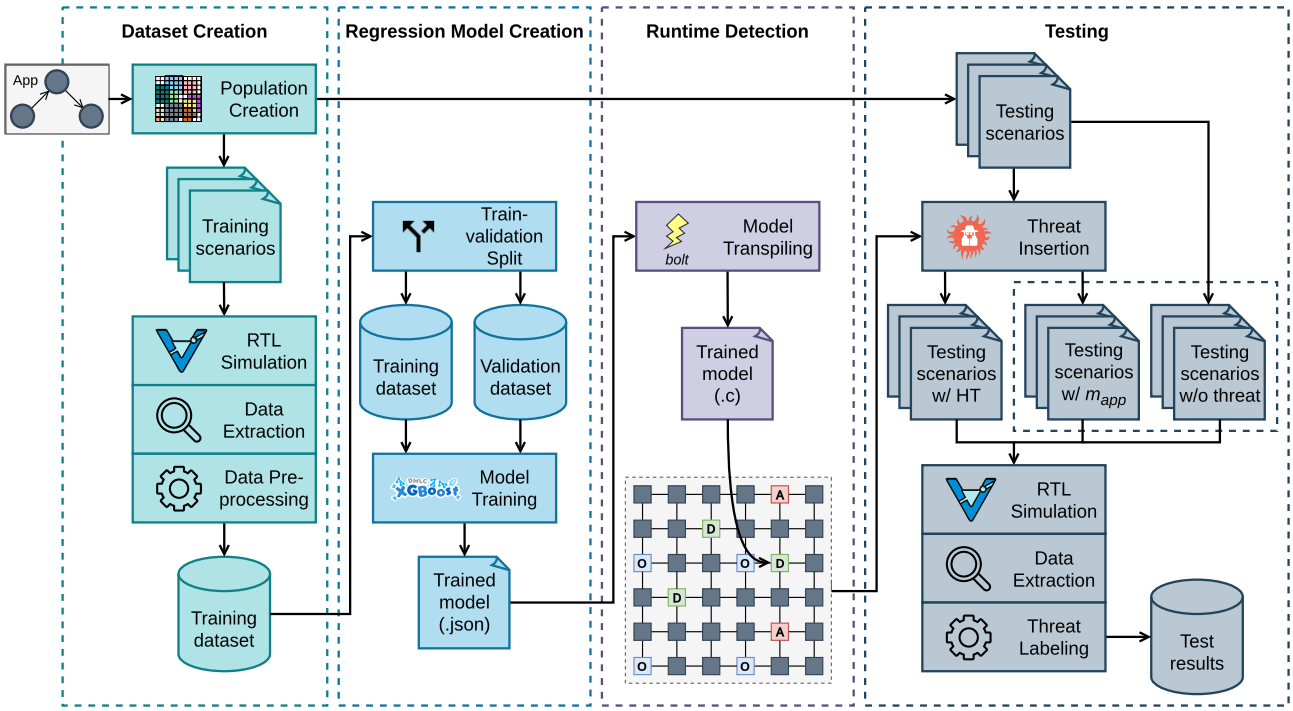


Figure 5.7: Overview of the proposed ML-driven security threat detection flow, spanning from dataset generation to runtime deployment and testing. Source: the Author.

In the second phase, **Regression Model Creation**, the *Training dataset* is split into *Training* and *Validation* subsets to train and validate the model. The core of this phase is the **Model Training** step, where the XGBoost algorithm is employed to learn an application's temporal signature. The objective is to create a regression model that predicts the expected end-to-end latency of a packet based on its monitored attributes. The output of this phase is a trained model serialized in the JavaScript Object Notation (JSON) format, representing the learned legitimate behavior of the application. The Regression Model Creation phase is detailed in Chapter 7.

The third phase, **Runtime Detection**, represents the transition from offline training to runtime deployment and addresses the scalability and overhead limitations identified in the state-of-the-art. Since standard ML libraries, such as Python-based XGBoost, cannot run natively on embedded cores due to memory and runtime constraints, the serialized JSON model undergoes **Model Transpiling**. This step converts the decision tree structures into optimized, dependency-free C code. The resulting transpiled model is compiled directly into the Decider (D) tasks' binaries within the Management Application (Chapter 4). These Deciders can be distributed across the MCSoC, enabling scalable runtime inference. The Runtime Detection phase is detailed in Chapter 7.

The final phase, **Testing**, evaluates the detection performance of the deployed model against the security threats described in Section 5.1. The set of *Testing scenarios* used in this phase is distinct from those used in training. The **Threat Insertion** injects anomalies into these scenarios, creating three test groups: scenarios with HT, scenarios

with m_{app}, and baseline scenarios without threats. After a new round of RTL simulation and data extraction, the **Threat Labeling** determines whether each extracted packet was affected by the security threats and compares the model's runtime predictions against the ground truth, producing the final *Test results*. The Testing phase is detailed in Chapter 8.

5.4 Final Remarks

This Chapter introduced a framework for detecting security threats in NoC-based MCSoCs. By shifting the monitoring from low-level network metrics to application-level profiling, the proposed approach addresses the limitations of static thresholding, which fails to distinguish between legitimate system variability (e.g., different mappings) and subtle malicious interference.

The framework is designed to explicitly address the key limitations identified in the state-of-the-art. While existing works often rely on high-overhead models, such as DNNs, or on methodologies restricted to offline inference, this proposal establishes a lightweight, real-time detection capability. By prioritizing low computational complexity and non-invasive monitoring, the proposed flow ensures that security mechanisms remain feasible for resource-constrained embedded MCSoCs, directly addressing the state-of-the-art challenges of applicability and confidence.

The ML workflow presented in Section 5.3 serves as the structural guide for the remainder of this Thesis. The following chapters provide a detailed breakdown of each phase illustrated in the overview:

- Chapter 6 details the Dataset Creation phase, focusing on the generation of realistic traffic baselines;
- Chapter 7 covers the Regression Model Creation, detailing the training and optimization of the XGBoost predictors, and the Runtime Detection implementation, which includes the model transpiling joined with the monitoring framework;
- Finally, Chapter 8 presents the experimental results derived from the Testing phase.

6. DATASET CREATION

This Chapter details the Dataset Creation phase, illustrated in Figure 6.1. This step serves as the foundation for the proposed ML-based security threat detection framework. The primary objective of this phase is to construct a training dataset that captures the temporal behavior of the target application under varying system conditions, without threats. This phase has as input a target application and yields two outputs: (*i*) a **Training dataset**, used to train the model on the baseline behavior; and (*ii*) a set of **Testing scenarios**, reserved to assess the detection method against unseen data and injected threats (Chapter 8).

The process is structured into four sequential steps that are executed for every application profiled for threat detection:

- Section 6.1 presents the benchmarks applications chosen to apply the proposed approach;
- Section 6.2 details the **Population Creation** step, where the mapping space is defined and permuted to generate diverse execution scenarios;
- Section 6.3 describes the **RTL Simulation**, **Data Extraction**, and **Data Pre-processing** steps, detailing how raw simulation logs are transformed into a structured dataset suitable for ML;
- Finally, Section 6.4 summarizes the size of the generated datasets and presents the final remarks for this Chapter.

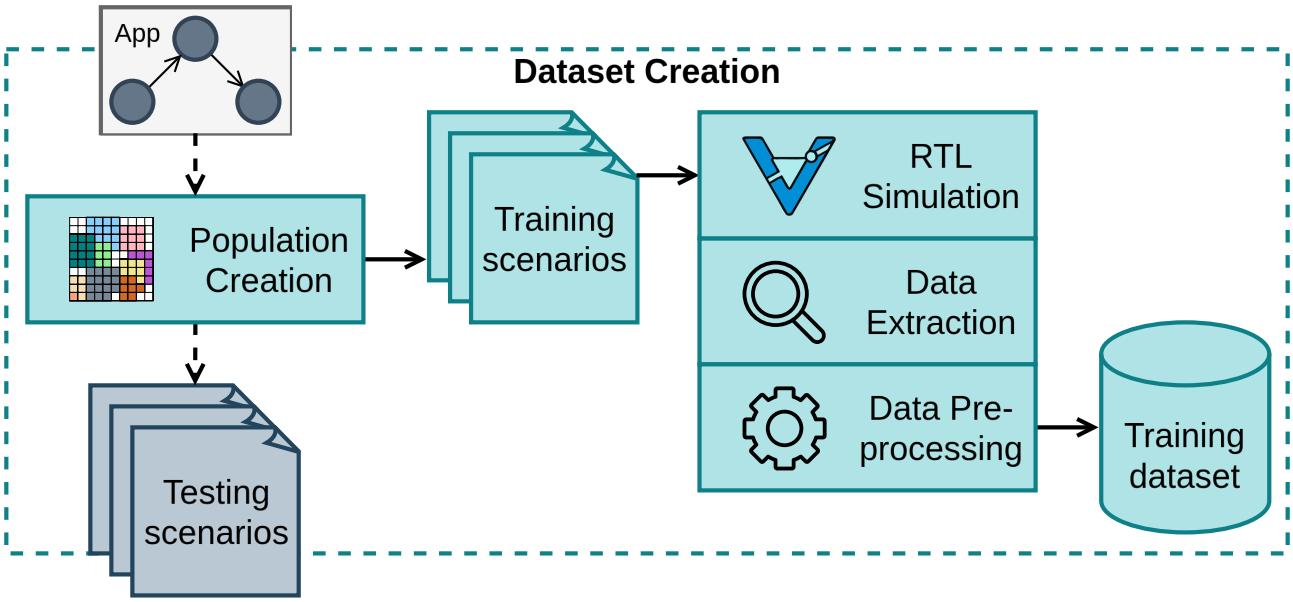
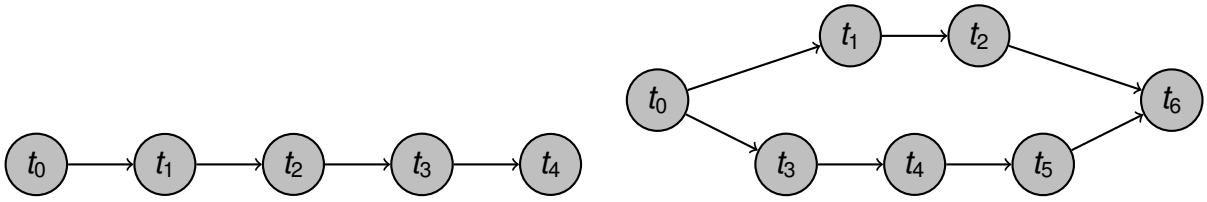


Figure 6.1: Dataset Creation flow. Source: the Author.

6.1 Benchmark Selection

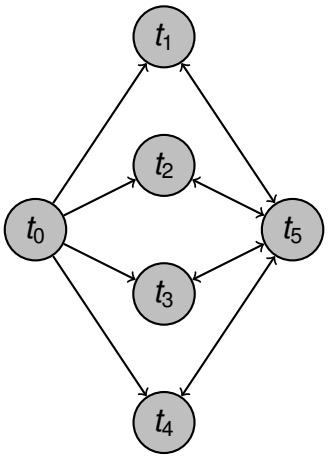
A diverse set of benchmarks was selected to execute the security threat detection workflow. This set aims to represent distinct parallel application patterns, with distinct numbers of tasks and edges, computation and communication loads, and a balance between computation- and communication-bound characteristics. This diversity ensures the proposed flow can learn and accurately predict temporal signatures, regardless of the application's complexity or traffic characteristics.

Figure 6.2 presents the CTGs of the selected benchmarks with the edge labels omitted for simplicity. Figure 6.2a reintroduces the CTG of the **MPEG** application with a pipeline pattern, in which the tasks execute sequential steps in parallel. As detailed in Section 5.2, it generates a steady, periodic communication flow.

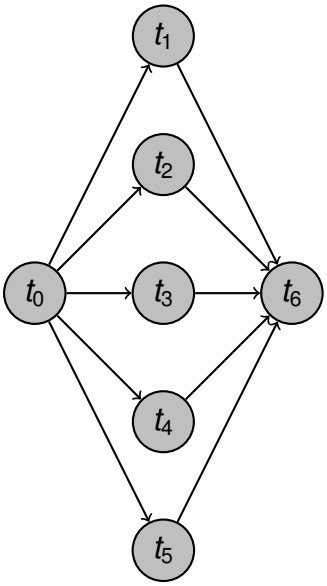


(a) MPEG.

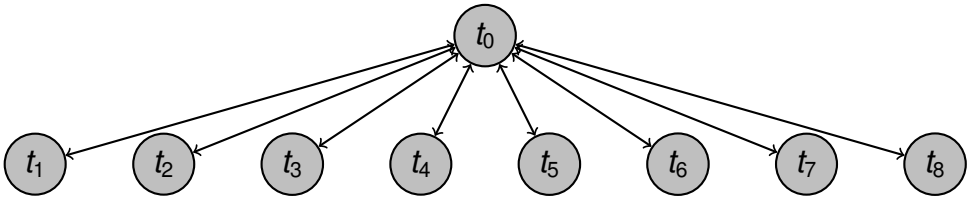
(b) Audio/Video.



(c) DTW.



(d) Dijkstra.



(e) AES.

Figure 6.2: CTGs of the profiled applications. Source: the Author.

Dynamic Time Warping (DTW) (Figure 6.2c) exhibits a bag-of-tasks pattern. The master task (t_0) periodically distributes test matrices to workers (t_{1-4}), which aggregate results in a collector (t_5). During warm-up, t_5 sends a base matrix to t_{1-4}, which is compared to the test matrices. This application creates traffic bursts, characterized by periods of network idleness followed by link competition during communication phases.

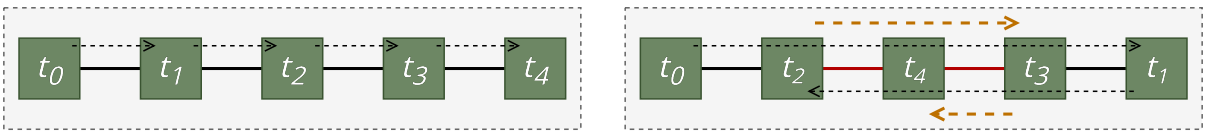
Dijkstra (Figure 6.2d) implements a parallel shortest-path algorithm, representing a fork-join pattern. Task t_0 divides graph nodes between workers t_{1-5} that execute the algorithm. Worker tasks (t_{1-5}) exchange frequent, small control messages to synchronize path updates with t_6, which joins the results. This behavior creates a temporal signature with high variance.

Audio/Video (Figure 6.2b) presents characteristics of both fork-join and pipeline patterns, consisting of two independent pipelines ($t_1 \rightarrow t_2$ and $t_3 \rightarrow t_5$) synchronizing at t_6. This benchmark executes audio and video decoding, where t_0 splits audio frames for t_1 and video frames for t_2.

AES (Figure 6.2e) represents an implementation of the Advanced Encryption Standard (AES), following a bag-of-tasks parallel pattern. In this application, a master task (t_0) distributes independent data blocks to eight worker tasks (t_{1-8}) for encryption. This benchmark is characterized by a high computational-to-communication ratio, as tasks t_{1-8} execute intensive cryptographic operations.

The selected benchmarks cover parallel patterns ranging from the linear flow of the MPEG and Audio/Video pipeline to the independent parallelism of the bag-of-tasks in DTW and AES, and to the synchronization-heavy fork-join of Dijkstra. *This ensures that the detection flow can generate accurate baseline models regardless of an application's specific structural characteristics, mitigating Limitation 8 of the state-of-the-art.*

Figure 6.3 shows how traffic on network links may also vary due to task mapping. For example, the MPEG application may generate either a single flow per link or multiple flows on the same link. In Figure 6.3a, there is no link competition. In contrast, as shown in Figure 6.3b, simply changing the task order within the PEs induces link competition. In this example, to communicate from t_0 to t_1, the packets must traverse the links (in red) used to communicate from t_2 to t_3. Likewise, to communicate from t_1 to t_2, the packets traverse the link also used to communicate from t_3 to t_4. The trained model must capture such variability.



(a) No link competition.

(b) Link competition.

Figure 6.3: MPEG under two different mappings showing possible link competition. Source: the Author.

6.2 Population Creation

The initial step of the Dataset Creation phase is **Population Creation**. This step defines the MCSoC configuration used to generate both training and testing scenarios. To ensure the dataset captures the variability inherent to the system's constraints, the MCSoC size is defined by the minimum area required to execute the workload without multitasking, while keeping the MCSoC geometry as close to a square as possible. Five criteria determine this size: (*i*) the number of tasks in the profiled application; (*ii*) the mapping task (*map*), which is required for management; (*iii*) two PEs reserved for the potential insertion of a malicious application (m_{app}); (*iv*) two PEs reserved for the management tasks that implement the runtime detection (Chapter 7); and (*v*) the number of PEs in the X dimension should be equal to or 1 greater than the number of PEs in the Y dimension.

Figure 6.4 illustrates the MCSoC setup for profiling the MPEG application. MPEG requires 5 PEs for its tasks. The *map* task (orange) is statically mapped to the top-left corner (PE 0×2). The PEs reserved for the malicious producer and consumer (m_p and m_c) are shown in magenta at the bottom-left and top-right corners, respectively. The runtime detection requires an additional 2 PEs to map its management tasks during the Testing phase (Chapter 8). The *Observer* (*O*, in blue) is reserved at the center of the MCSoC, and *Decider* (*D*, in green) is reserved at the PE on top of *O*. For the 10 required tasks, the minimum MCSoC configuration that meets the geometry constraint is a 4×3 grid (12 PEs).

The m_{app} tasks (m_p and m_c) and runtime detection tasks (*O* and *D*) have a static mapping to reduce the search space for mapping the profiled application, but they have not yet been included in the scenarios. While these PEs are **reserved** to maintain a consistent grid geometry between training and testing, no malicious or runtime detection tasks are inserted during this phase, ensuring that the generated dataset represents only the baseline behavior without threats or the proposed security threat detection approach.

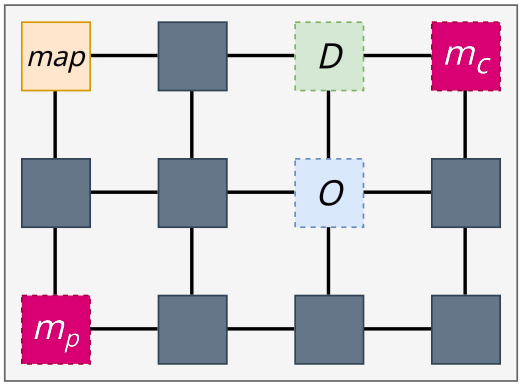


Figure 6.4: MCSoC mapping space for MPEG profiling. The gray PEs represent the available mapping space for the application. Source: the Author.

The MPEG application tasks may map to any of the available gray PEs shown in Figure 6.4. To capture the full range of valid communication behaviors, such as variations in hop count, flow direction, and link competition, the Population Creation step generates all possible mapping permutations within this space. The total number of generated scenarios is given by Equation (6.1):

$$P(n, r) = \frac{n!}{(n - r)!} \tag{6.1}$$

Where n is the number of available PEs (the gray blocks in Figure 6.4) and r is the number of application tasks. For the MPEG application ($n = 7$, $r = 5$), this yields 2,520 distinct mapping scenarios.

The rationale for using the **minimum** valid MCSoC size is to optimize the trade-off between coverage and computational cost. While modern computing resources allow for extensive RTL simulations, they remain time-intensive. By constraining the space to the minimum necessary to map the application and potential threats, the simulation computational cost is reduced while still generating a large dataset.

One limitation of this approach is that it does not consider fragmented mapping, i.e., it does not account for crossing traffic from applications other than the profiled one. However, this is consistent with the system's optimization goals: the mapping heuristic is tuned to avoid fragmentation [Dalzotto et al., 2021a] or to defragment at runtime [Dalzotto et al., 2022a]. Therefore, training on these scenarios accurately reflects the system's intended operational state.

The generated scenarios are randomly split into two subsets: (*i*) 80% are designated as **Training scenarios**; and (*ii*) 20% are held out as **Testing scenarios**. This strict separation ensures that the Testing phase (Chapter 8) assesses the model's ability to generalize to valid mappings it has never used during training.

6.3 Simulation and Data Processing

The second step of the Dataset Creation phase involves the **RTL Simulation** of the generated training scenarios. The RTL Simulations are executed on the Memphis-V platform using Verilator [Snyder et al., 2025]. Verilator is an open-source simulator that translates RTL descriptions into a native C++ model. Simulating with Verilator results in an average speed-up of 2.5× relative to Memphis's previous SystemC versions and at least 1.38× relative to Memphis-V simulated with commercial tools, such as Xcelium or Questa. In the example with the MPEG application, simulating all 2,016 scenarios in the training dataset took approximately 20 minutes on a host machine with dual AMD EPYC 7453 processors (56 cores, 112 threads) and 256 GB of RAM.

Simulation probes are inserted into the DMNI to capture the necessary traffic data without modifying the router architecture, thereby mitigating Limitation 5. Upon the reception of a *message delivery* packet, the DMNI logs the following attributes:

- **Sending time (t_{send})**: The timestamp inserted into the packet by the kernel immediately before the transmission request;
- **Receiving time (t_{recv})**: The timestamp when the DMNI completes the write operation of the packet into the local memory;
- **Packet size**: The total number of flits in the packet;
- **Sender ID**: The unique identifier of the source task, encoding both the application ID and the task number;
- **Receiver ID**: The unique identifier of the destination task.

Following the RTL Simulation, the **Data Extraction** step gathers the raw logs. In addition to the DMNI packet logs, this step extracts the output from the *map* task, which provides the application start time and the mapping location of every task in that specific scenario.

Finally, the **Data Pre-processing** step transforms the extracted data into a structured format suitable for training. First, packets unrelated to the profiled application, such as management messages, are filtered out based on the sender and receiver IDs. Second, timestamps are normalized: absolute system time is converted to application time, i.e., relative to the application's start time. This normalization allows the ML model to learn the temporal signature of the application regardless of when it begins execution. Third, the mapping locations are used to calculate the hop count for each packet, based on the Manhattan distance between the source and destination PEs.

Table 6.1 presents the header of the MPEG training dataset. This resulting Training Dataset is a structured table containing the following integer features:

- **scenario**: Identification number of the training scenario. This column is used for training-validation split (Chapter 7), but is absent in the model training.
- **timestamp**: The packet injection time relative to the application start, in μs;
- **size**: The packet payload size, in flits;
- **hops**: The distance traversed by the packet;
- **sender** and **receiver**: Categorical identifiers for the edge;
- **latency**: The target variable for the regression model, calculated as $t_{recv} - t_{send}$, in clock cycles.

scenario	timestamp	size	hops	sender	receiver	latency
0	316	133	3	4	2	857
0	867	69	3	2	1	825
0	888	133	3	4	2	872
0	939	69	2	1	0	817
0	1014	69	1	0	3	817

Table 6.1: MPEG training dataset header. Source: the Author.

By selecting these specific features, the dataset meets the requirements for a **lightweight** and **non-invasive** approach. All features are derived from the packet header or standard system events, requiring no deep packet inspection or internal router monitoring.

6.4 Final Remarks and Dataset Size

This chapter detailed the Dataset Creation phase, which establishes a baseline for the proposed ML-driven security threat detection. The systematic approach using permutation-based scenario generation, minimal MCSoC sizing, and non-invasive DMNI probing ensures that the resulting datasets capture the full range of legitimate system variability while remaining clean of malicious influence. This process addresses the **confidence** limitation in the state-of-the-art by providing a high-fidelity reference for the regression model.

The Dataset Creation phase was applied to a set of applications beyond the MPEG decoding example. These applications were selected to represent diverse communication patterns in NoC-based MCSoC. Table 6.2 summarizes the generated datasets, listing the application size in number of tasks and edges, the number of distinct mapping scenarios generated, and the size, in lines, of the resulting dataset in terms of total packet transactions captured. The training dataset size depends on the number of packets exchanged between the application's edges during execution. The Population Creation for AES was modified to exclude the m_{app} threat, as assessing the malicious application outcomes is limited relative to HT (Chapter 8), and including m_{app} would result in millions of simulation scenarios.

The resulting training datasets provide the necessary volume and diversity for the subsequent machine learning steps. The detailed implementation of the regression model training, which utilizes this data to learn the application's unique temporal signatures, is presented in Chapter 7.

Table 6.2: Dataset size for profiled applications. Source: the Author

Application	**Tasks (r)**	**Edges**	**PEs (n)**	**Scenarios (P)**	**Training Dataset Size**
MPEG	5	4	7	2,520	161,280
DTW	6	12	7	5,040	661,248
Audio/Video	7	7	7	5,040	564,480
Dijkstra	7	10	7	5,040	907,200
AES	9	16	9*	362,880	19,595,520

*AES is assessed without m_{app}, requiring 2 PEs less, resulting in a 4×3 MCSoC size. This avoids simulating millions of scenarios in a 4×4 MCSoC.

7. REGRESSION MODEL CREATION AND RUNTIME DETECTION

This Chapter details the development of the regression models and their subsequent integration into the MCSoC for runtime security threat detection. Following the generation of the application-level datasets in Chapter 6, the workflow transitions from baseline traffic characterization to the deployment of a predictive framework on the Memphis-V platform.

The methodology presented in this Chapter addresses the primary limitations identified in the state-of-the-art (Chapter 3). Unlike existing solutions that utilize high-overhead models (Limitation 2) or are restricted to offline inference (Limitation 4), this Thesis provides a lightweight, embedded implementation suitable for resource-constrained MCSoCs. By employing an XGBoost regression approach and a specialized transpilation process into dependency-free C code, detection is performed natively on the Memphis-V PEs. Furthermore, the distributed ODA architecture and the monitoring framework (Section 4.3) avoid invasive modifications to the NoC hardware (Limitation 5), ensuring a non-invasive approach that maintains the scalability of the MCSoC.

The remainder of this chapter is organized as follows. Section 7.1 presents the Regression Model Creation phase, detailing the training of the XGBoost model to learn the temporal signatures of application communication. Section 7.2 describes the Runtime Detection phase, detailing how the trained models are integrated into the Memphis-V platform to perform runtime inference on monitored traffic attributes. Finally, Section 7.3 provides a summary of the implementation results and final remarks regarding the proposed phases.

7.1 Regression Model Creation Phase

Figure 7.1 illustrates the **Regression Model Creation** phase, which is responsible for generating a predictive model of an application's temporal communication signature. In this phase, the Training dataset (Chapter 6) is processed to train an XGBoost regression model, which is then serialized to JSON.

The first step is the **Train-validation Split**. To ensure the model's ability to generalize to unseen mapping configurations, the dataset is partitioned by scenario rather than by individual packet transactions. From the scenarios designated for training, 25% are separated into a *Validation dataset*, leaving the remaining 75% for the *Training dataset*. Consequently, the total population of scenarios (Section 6.2) is distributed as follows: 60% for training, 20% for validation (hyperparameter tuning and early stopping), and 20% reserved exclusively for testing (Chapter 8).

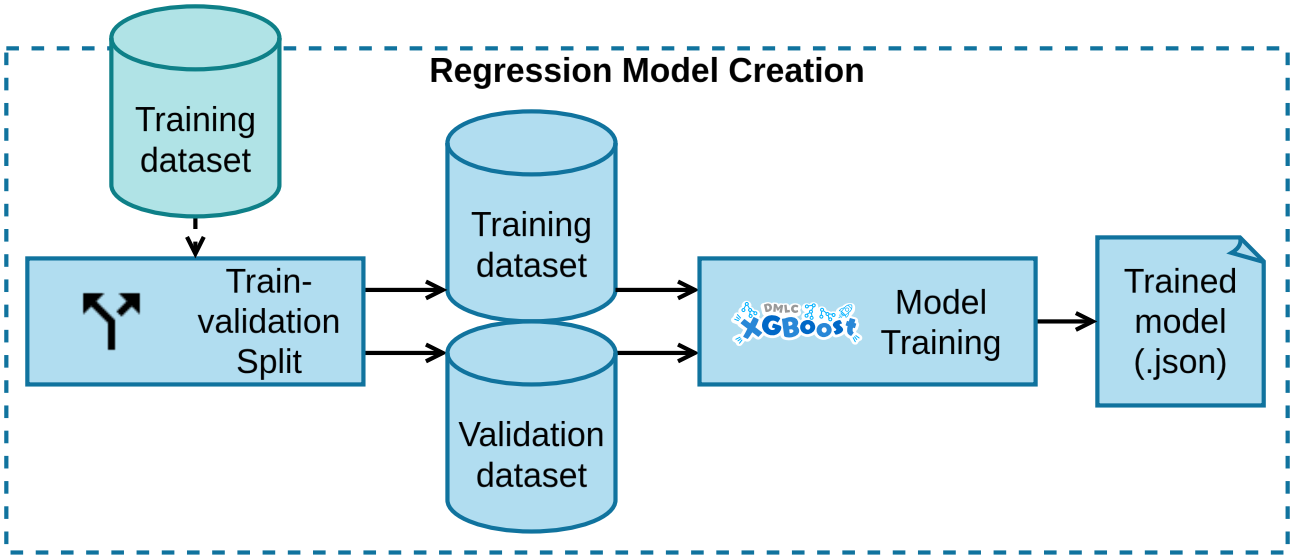


Figure 7.1: Regression Model Creation flow. Source: the Author.

The **Model Training** step uses the XGBoost regression algorithm from the *dmlc* (Distributed (Deep) Machine Learning Community [2014]) to learn the application's legitimate latency patterns. A grid search is performed to optimize hyperparameters, aiming to achieve a balance between predictive performance and implementation cost. The following hyperparameters are evaluated to control model complexity:

- **#estimators**: Controls the number of estimators (trees) that form the XGBoost ensemble. This Thesis uses 33 as the maximum number of estimators, rather than the default 100, to limit the ensemble size and reduce the memory footprint. Still, training employs early stopping, terminating if the model stops improving after 10 consecutive trained trees.
- **Learning Rate (Eta – η)**: Controls the contribution of each subsequent tree to the ensemble. The search evaluates values of 0.3 (default), 0.4, and 0.5. While higher values accelerate convergence, they may lead to locally optimal solutions. However, in this Thesis, higher values are prioritized if they result in an ensemble with fewer trees.
- **Minimum Loss Reduction (Gamma in Equation (2.9))**: Specifies the minimum reduction in loss required to split a leaf node. The search spans values from 0.5 to 3.0 (default is 0). Increasing Gamma penalizes the creation of deeper trees, which reduces the model size.
- **Minimum Child Weight**: Defines the minimum sum of instance weight needed in a child node of a tree. Values of 3, 4, and 5 are evaluated (default is 1). Higher weights prevent the model from learning highly specific, noisy patterns, thereby mitigating overfitting and limiting tree depth.

Other parameters, such as *Maximum Depth* and *Subsample*, were also assessed, but ultimately set to their default values (6 and 1.0, respectively), as the Gamma and Minimum Child Weight constraints sufficiently control the model variance.

The final selected model is the one with the smallest Root Mean Squared Error (RMSE) across all grid search candidates. The RMSE represents how far, on average, the predicted latency is from the actual latency. For example, an RMSE of 5 means that, on average, the predicted latency deviates by 5 clock cycles from the actual latency, corresponding to 0.05 μs at the system's 100 *MHz* operating frequency. Finally, the optimized model is serialized into a JSON file for the subsequent transpilation stage.

7.2 Runtime Detection phase

Figure 7.2 illustrates the **Runtime Detection** phase, which enables the deployment of predictive regression models in the MCSoC environment. This phase converts the high-level model description (the serialized JSON file) into an optimized C function that executes on the PEs. These functions are integrated into the Decider (D) tasks of the Management Application (MA), which use the Observe-Decide-Act (ODA) loop to monitor security threats in runtime.

The phase is structured into two primary components. Section 7.2.1 describes the Model Transpiling step, which converts the XGBoost ensemble into a lightweight, dependency-free format suitable for embedded execution. Section 7.2.2 describes the integration of these transpiled models into the Memphis-V monitoring framework (Section 4.3), specifically focusing on the execution of the inference logic within the ODA management structure.

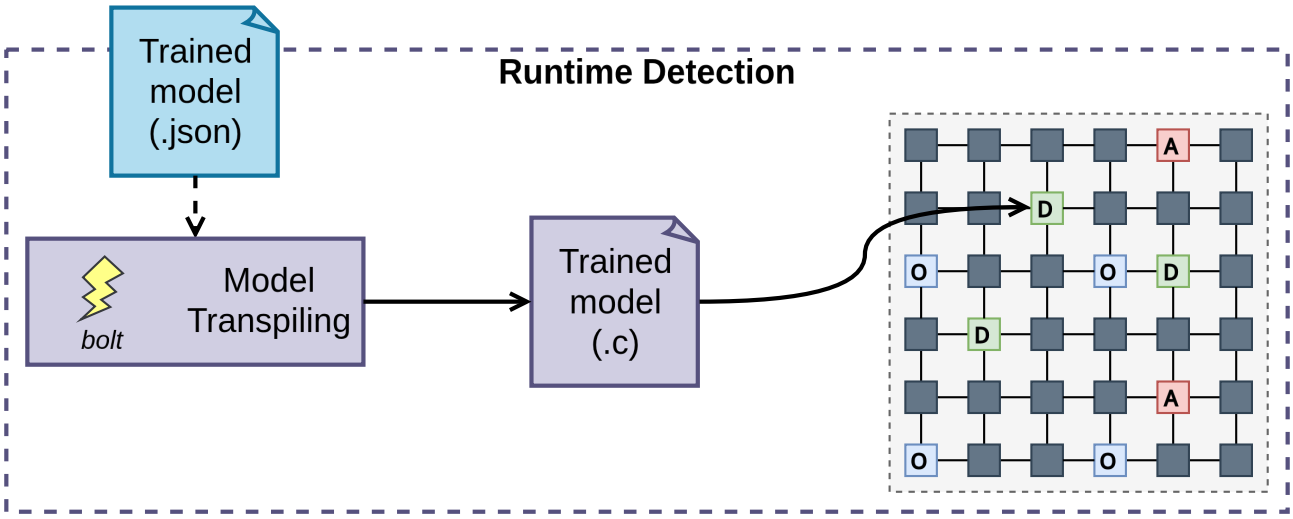


Figure 7.2: Runtime Detection flow. Source: the Author.

7.2.1 Model Transpiling

The deployment of ML in embedded MCSoCs requires evaluation beyond predictive performance, as ML integration is often compromised by limited memory, computational performance, and strict energy budgets [Batzolis et al., 2023]. Thus, feasibility depends on achieving a balance between predictive performance and computational overhead. Reducing the computational overhead of an ML model typically involves two stages: (*i*) training-time, when models are constrained through techniques such as pruning; and (*ii*) deployment-time, when the inference is optimized for execution efficiency.

Optimizing implementation efficiency is a core objective of the proposed threat-detection framework. While Section 7.1 detailed model training aimed at minimizing size through hyperparameter tuning and early stopping, these regression models must also be adapted for the MCSoC runtime environment. Compiling the trained model into native instructions is the preferred approach for embedded MCSoCs. This method eliminates external dependencies and runtime file Input/Output (I/O), significantly reducing inference latency and memory footprint. By embedding the compiled model directly into the application binary, the system yields a self-contained executable that operates without a filesystem, thereby minimizing startup overhead.

The tool *bolt* (Boost Learning Transpiler) was developed to perform **Model Transpiling** (parsing, transforming, and generating source code) directly into a C function. Standard tools for such conversions, such as *TL2cgen* [dmlc, 2023], typically support only optimizations that strictly preserve the trained model's original logic. In contrast, this Thesis implements lossy optimizations within *bolt* that modify the model structure to enhance performance. Furthermore, unlike general-purpose tools like *m2cgen* [Zeigerman et al., 2019], which default to floating-point arithmetic for all internal operations, *bolt* maps XGBoost data types directly to corresponding C types to minimize floating-point arithmetic.

Figure 7.3 illustrates a tree t_1 described in the XGBoost JSON model. Split nodes (red rectangles) compare a learned attribute X_i to a learned condition C_j. Leaf nodes (green triangles) assign a resulting weight w_k to the tree. This weight is already multiplied by the learning rate η and is represented as a floating-point number.

Figure 7.4 presents the tree t_1 converted using **bolt**. The feature vector is passed as pointers to appropriate data types (line 1), and the tree logic is implemented as nested if-else statements (lines 4–5). The comparison operators are less than (<) for numeric values and not equal to (!=) for boolean values. The tree inference reaches one of its leaf nodes (lines 6,8,11), assigning a learned weight as its result. In this implementation, split conditions (e.g., `C_t1_s0`) and tree weights (e.g., `t1_w0`) are defined as literals. This allows the compiler to often encode such values as immediates, eliminating the memory access cycles required to load constants from memory.

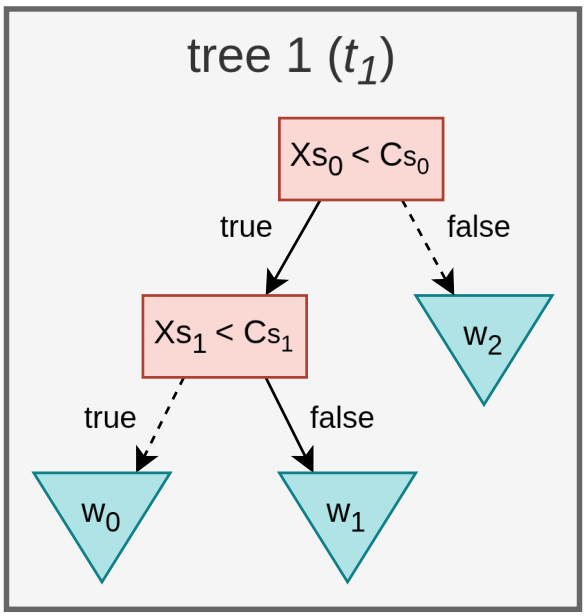


Figure 7.3: Conceptual structure of a trained XGBoost decision tree. Source: the Author

In Figure 7.4, the first split (line 4) assumes that a null attribute is sufficient to take the left path, while the second split (line 5) requires a non-null attribute to take the left path, meaning that the right path is the default one. The default paths are identified by non-dashed lines in Figure 7.3. Figure 7.5 shows an optimized transpiled model with **null-check elimination**. This Thesis assumes that the monitoring framework (Section 4.3) ensures that all attributes are non-null. By removing pointer verification in splits (lines 4–5,14–15) and passing the feature vector by value (line 1), features are kept in registers more often, reducing memory accesses due to pointer dereferences throughout the entire ensemble inference.

```
1  float model(typeX_0 *X_0, typeX_1 *X_1, ..., typeX_n *X_n)
2  {
3      float t1;
4      if (X_t1_s0 == NULL || *X_t1_s0 < C_t1_s0) {
5          if (X_t1_s0 != NULL && *X_t1_s1 < C_t1_s1) {
6              t1 = t1_w0;
7          } else {
8              t1 = t1_w1;
9          }
10     } else {
11         t1 = t1_w2;
12     }
13     ...
14 }
```

`typeX_i` is defined to the corresponding attribute `X_i` type, such as float, int, or bool. Split attributes, such as `X_t1_s0`, are defined as the corresponding attribute `X_i`. Split conditions, such as `C_t1_s0`, and tree weights `t1_wk` are defined as literals.

Figure 7.4: Example tree conversion with bolt. Source: the Author.

```
1  float model(typeX_0 X_0, typeX_1 X_1, ..., typeX_n X_n)
2  {
3      float t1;
4      if (X_t1_s0 < C_t1_s0) {
5          if (X_t1_s1 < C_t1_s1) {
6              t1 = t1_w0;
7          } else {
8              t1 = t1_w1;
9          }
10     } else {
11         t1 = t1_w2;
12     }
13     float t2;
14     if (X_t2_s0 < C_t2_s0) {
15         if (X_t2_s1 < C_t2_s1) {
16             t2 = t2_w0;
17         } else {
18             t2 = t2_w1;
19         }
20     } else {
21         t2 = t2_w2;
22     }
23     return (bias + t1 + t2 + ... + tn);
24 }
```

This figure assumes no leaf-weight quantization. If such optimization is enabled, the tree weights are defined as integer literals, and the result of the final summation (line 23) is right-shifted by the binary logarithm of the quantization factor.

Figure 7.5: Example XGBoost ensemble conversion with bolt using null-check elimination. Source: the Author.

The final step of XGBoost inference is the summation of the individual tree weights with the bias (Figure 7.5, line 23). This arithmetic is performed using floating-point. To further mitigate floating-point overhead, **bolt** can perform **leaf-weight quantization**, scaling weights by a power of 2 and truncating them to integers. In this case, the final prediction is computed via integer summation and a bitwise right shift. Since the trained models target packet latency measured in clock cycles, the loss of decimal precision is acceptable.

Finally, Figure 7.6 details the **categorical collapsing** optimization. In Figure 7.6a, a categorical variable, `cat`, is one-hot encoded, significantly increasing the number of function arguments (line 1). In Figure 7.6b, **bolt** collapses boolean indicators into a single integer variable, allowing the compiler to keep more arguments in registers rather than spilling them to the stack. This also enables a uniform model declaration across different applications regardless of the number of CTG edges. The optimized model compares the collapsed variable against its label, i.e., whether integer `cat` is different from `0`, rather than comparing boolean `cat_0` against `true` (lines 4–5). There is no performance drawback in comparing an integer instead of a boolean, as the resulting RISC-V instruction is the same in both cases.

The combination of native C inference with specific optimizations, namely null-check elimination, leaf-weight quantization, and categorical collapsing, enables a lightweight runtime inference capability that addresses the implementation overhead constraints (Limitation 2) identified in the state-of-the-art.

```
float model(bool cat_0, bool cat_1, ..., bool cat_n)
{
    float t1;
    if (cat_0 != true) {
        if (cat_1 != true) {
            t1 = t1_w0;
        } else {
            t1 = t1_w1;
        }
    } else {
        t1 = t1_w2;
    }
    ...
}
```

(a) Categorical as one-hot.

```
float model(int cat)
{
    float t1;
    if (cat != 0) {
        if (cat != 1) {
            t1 = t1_w0;
        } else {
            t1 = t1_w1;
        }
    } else {
        t1 = t1_w2;
    }
    ...
}
```

(b) Categorical collapsed as integers.

Figure 7.6: Comparison of conversion with and without categorical collapsing. Source: the Author.

7.2.2 Model Integration into the MA

The integration of the transpiled regression models into the Memphis-V platform allows scalable, runtime detection of security threats. This process extends beyond compilation and includes a distributed monitoring framework that implements the ODA control loop, with specialized **Security Observer** and **Security Decider** tasks.

Figure 7.7 shows the setup of the Security Observer. Initially, the Security Observer must discover the Security Deciders executing within the MCSoC. The Security Observer queries the Mapper task, which returns a list of all Security Deciders task IDs (Step 1). The identification of these MA tasks, e.g., as Security Deciders, is defined in their binaries and stored by the Mapper during task admission.

Since each regression model is trained for a single application (Section 7.1) and each Security Decider implements a specific model instance, a matching process is required. In the second step, the Security Observer queries each Security Decider returned by the Mapper for its unique *model ID*. Once the relationship between task ID and model ID is established, the Security Observer initializes its monitoring buffers and announces its monitoring capability (Section 4.3).

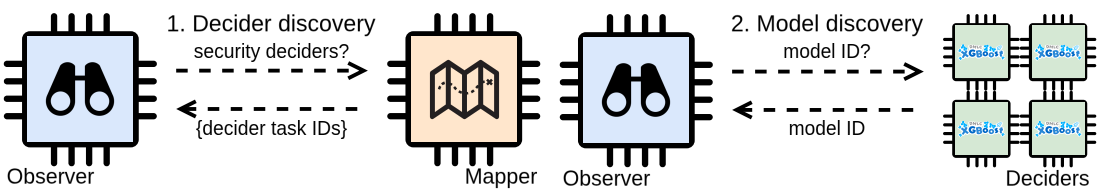


Figure 7.7: Startup procedure for runtime security threat detection. Source: the Author.

Following the startup, the MCSoC LLMs enable security monitoring after the announcement mechanism. Figure 7.8 demonstrates the interaction between the kernel and the management layer. The monitoring is triggered whenever a *message delivery* packet between two user tasks reaches the PE of the receiving task. The LLM within the kernel at this PE extracts the following attributes to build a monitoring packet:

- **Sending timestamp**: The instant the monitored packet was sent, embedded into the message header by the source kernel;
- **Size**: The total flit count of the monitored packet;
- **Hops**: The Manhattan distance between source and target addresses. The source address is embedded in the monitored packet;
- **Application, sender, and receiver IDs**: The identifier of the application and the edge to which the monitored packet belongs;
- **Latency**: The end-to-end latency, computed as the difference between the sending and receiving timestamps. The receiving timestamp is retrieved from a DMNI MMR that records the completion time of the last packet reception.

The monitoring packet is small, with a size of 5 flits, allowing it to fit entirely within the input buffers of the Hermes routers. Consequently, the control flow of such packets can exhibit a behavior similar to store-and-forward. As the routing is made, the previous router transmits the packet in its entirety to the buffer, releasing the link. This mechanism avoids link-level collisions and mitigates contention between user application traffic and monitoring traffic. Furthermore, the monitoring packet is written directly to memory upon reception using the DMNI monitoring mechanism described in Section 4.3.

Figure 7.9 details the operation of the Security Observer upon receiving a monitoring packet. To perform the inference, the Security Observer must select the correct Security Decider task. In step 3, the Security Observer queries the Mapper for the model ID and the start timestamp corresponding to the monitored application. To minimize communication overhead, this metadata is cached locally using a Least Recently Used (LRU) replacement policy, ensuring that step 3 is executed only during the first monitoring event for a specific application instance.

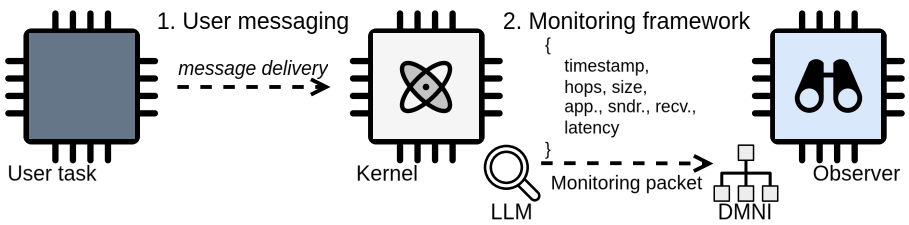


Figure 7.8: Monitoring packet generation. Source: the Author.

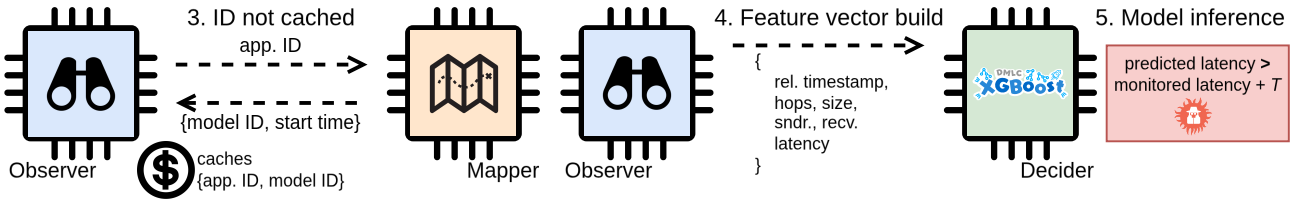


Figure 7.9: Feature vector build and inference logic within the MA tasks. Source: the Author.

In step 4 (Figure 7.9), the Security Observer constructs a feature vector message for the Security Decider. The monitored packet's sending timestamp is converted into application-relative time by subtracting the application's start time. This normalization ensures this feature is consistent with the learned during the training phase.

Finally, step 5 encompasses the model inference. Upon receiving the message, the Security Decider executes the transpiled XGBoost model. If the monitored latency exceeds the predicted value by a threshold, the packet is flagged as affected by a security threat. The threshold used for this Thesis is 45 clock cycles, representing $0.45\,\mu s$ at $100\,MHz$. This value is derived from a detection threshold of $0.5\,\mu s$, minus $0.05\,\mu s$ to compensate for model error and increase recall at the expense of precision.

This distributed approach enables efficient runtime detection by parallelizing communication and computation across the MCSoC. Multiple Security Observers can handle localized traffic, while specialized Security Deciders distribute the memory and computational costs of the XGBoost ensembles. Actuation upon detection is considered out of the scope of this Thesis.

7.3 Final Remarks and Model Size

This Chapter detailed the transition from application-specific datasets to the deployment of a functional, runtime security framework. By combining specialized model training with the *bolt* transpilation tool and a distributed ODA management architecture, the proposed framework bridges the gap between ML and the operational constraints of MCSoCs. The following results quantify the effectiveness of these phases in terms of predictive performance and resource usage.

Table 7.1 presents the outcome of the Regression Model Creation phase for the five selected benchmarks. The first column lists the application. Columns 2–4 list the hyperparameters found by the grid search of the model training. The fifth column shows the number of estimators (decision trees) of the trained model. Finally, column 6 presents the resulting RMSE of the selected model.

Application	eta	gamma	min. child weight	#estimators	RMSE
MPEG	0.5	1	3	33	1.037
DTW	0.3	3	3	31	6.550
Audio/Video	0.4	0.5	3	31	6.785
Dijkstra	0.5	0.6	3	33	4.089
AES	0.5	3	4	33	0.258

Table 7.1: Regression Model Creation results. Source: the Author.

The models demonstrate that XGBoost can learn temporal signatures of diverse parallel patterns with a limited number of trees, achieving low RMSE. *By maintaining a low number of estimators, it mitigates the high computational overhead (Limitation 2).*

To evaluate the feasibility of these models on the Memphis-V platform, Table 7.2 summarizes the memory footprint of the transpiled C functions, optimized with categorical collapsing and null-check elimination. Columns 1–2 list the application and the number of estimators of its model. The third column shows the model's memory footprint, in kilobytes. The fourth column shows the model's memory footprint after further optimization that quantizes the leaf weights to integers. Both footprints are from the model and its auxiliary functions, such as those that emulate floating-point operations, without considering the overhead of the decider task in which the model is inserted.

In Table 7.2, the footprint is a result of the number of estimators and their depth. Although all models produced a similar number of estimators, the depth of each tree is determined during training, leading to models such as the DTW one being smaller. The reduction in memory footprint achieved through leaf-weight quantization is significant, as it minimizes the binary size mainly by eliminating floating-point emulation libraries. Reducing memory requirements eases the models' fitting into the PEs' local memory.

The results presented in this Chapter demonstrate that the proposed non-invasive monitoring and distributed inference approach provides a scalable solution for MCSoC security. The lightweight nature of the transpiled models, combined with the ODA management, addresses the trade-off between predictive performance and computational cost. This implementation provides the necessary infrastructure for the experimental evaluation presented in Chapter 8.

Application	#estimators	Memory footprint (w/o quantization)	Memory footprint (w/ quantization)
MPEG	33	27.1 kB	19.7 kB
DTW	31	14.4 kB	11.1 kB
Audio/Video	31	24.3 kB	17.8 kB
Dijkstra	33	29.1 kB	21.6 kB
AES	33	31.3 kB	22.0 kB

Table 7.2: Transpiled model memory footprint. Source: the Author.

8. TESTING AND RESULTS

This Chapter presents the Testing phase, which constitutes the final stage of the proposed ML workflow, along with its results. In this phase, the framework is evaluated against the threat model described in Section 5.1: HT and m_{app}. The results quantify the detection effectiveness and performance impact on the Memphis-V platform.

Figure 8.1 shows the Testing phase flow, using the Testing scenarios generated in the Population Creation phase (Chapter 6) as input. Section 8.1 describes the Testing phase, which includes the Threat Insertion and the simulation of the Testing scenarios that yield the Test results. Section 8.2 presents and discusses the Test results from the Testing phase. Section 8.3 compares the Test results against alternative detection methods. Section 8.4 analyzes a scenario in which the MCSoC contains 4 applications executing simultaneously with HTs and the proposed framework. Finally, Section 8.5 presents the final remarks on the experimental results.

8.1 Testing phase

The Testing phase uses 20% of the scenarios held out during the Population Creation (Chapter 6), which the trained regression models (Chapter 7) have never encountered, to evaluate the framework's ability to detect security threats. The process is divided into four steps: Threat Insertion, RTL Simulation, Data Extraction, and Threat Labeling.

The Testing phase begins with **Threat Insertion**, in which the threat model (Section 5.1) is integrated into the baseline *Testing scenarios* to generate two sets of compromised workloads:

- *Testing scenarios with HT*: These scenarios involve the insertion of an HT within the NoC links. The output ports used by a task at a PE to send a packet have the HT inserted at its link. The HT is configured to actuate with a 33% probability, halting communication for 50–75 clock cycles. This set evaluates the model's sensitivity to hardware-level threats.
- *Testing scenarios with m_{app}*: These scenarios involve the mapping of a Malicious Application (m_{app}) alongside the legitimate user application. The m_{app} generates packets of 50 flits every 100-500 μs. This set evaluates the framework against software-induced network contention.

A third set, Testing scenarios without threat, is maintained to provide a baseline for indicating when the m_{app} affects a packet during Threat Labeling. All of the three sets of testing scenarios contain the same mapping instances.

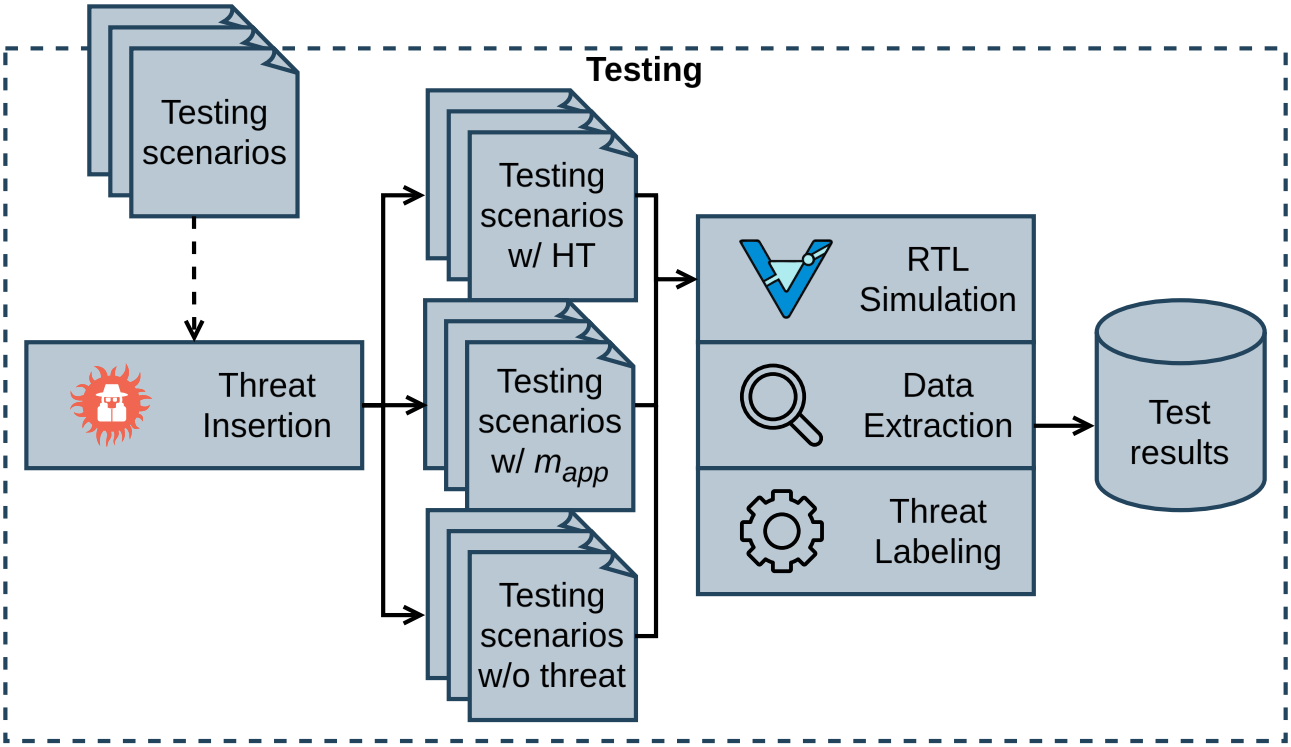


Figure 8.1: Testing phase flow. Source: the Author.

Following Threat Insertion, every Testing scenario is executed through **RTL Simulation** on the Memphis-V platform. During execution, the detection framework described in Section 7.2 operates in runtime. The Security Observers collect traffic attributes, and the Security Deciders execute the transpiled XGBoost models to perform runtime inference.

The **Data Extraction** step captures logs from the DMNI simulation probes, yielding a dataset of packets containing the sending time and their sender and receiver IDs. This dataset is filtered to include only packets exchanged between user tasks, excluding those exchanged between kernels or the MA.

The final step in compiling the results is **Threat Labeling**. Simulation logs from the Security Deciders record every instance where the monitored latency exceeds the model's predicted latency. Using the sending timestamp, sender, and receiver IDs of such records allows flagging whether the packets extracted from the DMNI were detected as affected by a security threat.

Simulation probes within the HTs log when an activation occurs, allowing precise flagging of the packets affected by the threat. Differently, flagging packets affected by m_{app} is done by comparing the latency of a packet in the Testing scenarios with m_{app} to that in the Testing scenarios without threats, considering the packet as affected by the threat when its latency increases. This approach provides an estimate of whether a packet is indeed affected by the m_{app}, since specific details on the packet collision are unfeasible to track within the NoC.

Finally, the Test results dataset consists of 5 columns:

- *Sending timestamp*;
- *Sending and receiving task IDs*;
- *Threat affected*;
- *Threat detected*.

8.2 Test results

Table 8.1 presents the predictive performance results of the Testing phase using the HT threat model. The first column lists the modeled application tested for security threats. The results were assessed using the same scenarios, with and without the model's leaf quantization optimization, as indicated in the second column. Columns 3–5 show the predictive performance metrics: recall, precision, and F1-Score, respectively.

In Table 8.1, all models presented high recall, indicating their ability to find 83%–100% of the threats occurring in the Test dataset. DTW, which has the smallest memory footprint (Section 7.3), also achieved the highest recall, showing that it's easy to predict the temporal signature of such an application and thus easily detect threats. Conversely, Dijkstra's frequent communication with high variance made threat detection more difficult, resulting in lower recall.

Precision indicates the quality of threat detection, i.e., a high precision indicates that the instances detected as threats were really threats, with false positives decreasing this metric. In Table 8.1, most models presented high precision, with 87.2%–100% of the threats correctly detected. The Audio/Video and AES models showed lower precision, as crossing traffic flows were incorrectly detected as threats.

Table 8.1: Predictive performance for runtime HT detection. LQ: Leaf Quantization. Source: the Author.

Application	LQ	Recall	Precision	F1-Score
MPEG	✗	0.919	1.000	0.958
	✓	0.919	1.000	0.958
DTW	✗	1.000	0.904	0.949
	✓	1.000	0.904	0.950
Audio/Video	✗	0.923	0.711	0.804
	✓	0.924	0.711	0.804
Dijkstra	✗	0.830	0.872	0.851
	✓	0.830	0.872	0.851
AES	✗	0.923	0.717	0.807
	✓	0.923	0.717	0.807

F1-Score is the harmonic mean between recall and precision, signifying the model's ability to minimize both false positives and false negatives. In Table 8.1, every model presented an F1-Score higher than 80%. Leaf weight quantization had a negligible impact on predictive performance. This lack of impact is attributed to the scale of the target variable. Since the target variable, latency, is measured in clock cycles, the decimal places in the floating-point weights are unnecessary to maintain a small prediction error.

Table 8.2 presents the computational performance and overhead of the threat detection. The first column lists the modeled application tested for security threats. The second column indicates the presence of leaf weight quantization. The third column shows the average inference latency, measured as the time taken to execute the transpiled model. The fourth column shows the average detection latency, measured as the time between the HT activation and the threat flagged by the Security Decider. The fifth column presents the average increase in application execution time due to the proposed approach.

In Table 8.2, the model inference presented high performance, with inference latency below 65 μs with floating point, and below 11 μs with leaf quantization. Leaf quantization decreased the inference latency by 83.6%–89.4%, while maintaining the same predictive performance of the floating point model (Table 8.1). *The high inference performance mitigates the Limitation 2 of the state-of-the-art (model overhead).*

The detection latency across floating-point models was consistent across MPEG, DTW, and Audio/Video models, at an average of 148 μs to detect an HT after its activation. In contrast, Dijkstra and AES exhibited high detection latency. This high latency occurs due to communication bursts that queue monitoring packets in the Security Observer and Security Decider tasks, accumulating communication and inference latency while the packets are in the queue. With leaf quantization, such a problem does not arise. Decreasing inference latency with leaf quantization reduced the average detection latency by 98% in Dijkstra and 97% in AES, enabling real-time threat detection for these models.

Table 8.2: Runtime detection performance overhead. LQ: Leaf Quantization. Source: the Author.

Application	LQ	Avg. inf. latency	Avg. det. latency	Avg. exec. time inc.
MPEG	✗	64.9 μs	152.6 μs	0.84%
	✓	10.0 μs	53.7 μs	0.84%
DTW	✗	53.6 μs	128.4 μs	0.05%
	✓	5.7 μs	53.0 μs	0.05%
Audio/Video	✗	57.9 μs	162.5 μs	0.87%
	✓	9.5 μs	53.3 μs	0.87%
Dijkstra	✗	60.6 μs	2814.7 μs	5.09%
	✓	9.8 μs	52.6 μs	5.09%
AES	✗	61.3 μs	1786.5 μs	5.43%
	✓	10.7 μs	60.4 μs	5.43%

In Table 8.2, the execution time increase is the overhead of the detection framework on MCSoC performance due to increased NoC traffic by the monitoring packets. This overhead is negligible in DTW due to its sparse communication pattern. In Audio/Video and MPEG, the performance impact is small ($<$ 1%). Although Dijkstra and AES showed higher overhead due to intensive communication, the $\approx$ 5% overhead is acceptable. Leaf quantization had no impact on execution time, indicating that this overhead is a function of the monitoring framework. *The low-overhead monitoring framework mitigates the Limitation 3 of the state-of-the-art.*

Table 8.3 presents the predictive performance metrics of the Test results using m_{app} threat model. The m_{app} detection was assessed only with the leaf quantization model, as the HT threat model showed that the floating-point model has no predictive performance advantage and also compromises real-time detection due to increased detection latency. Furthermore, the AES application was not tested to avoid simulating millions of scenarios (Chapter 6), as the m_{app} assessment is also limited in relation to the HT one.

In Table 8.3, the recall is high across all applications and is similar to the HT one for MPEG, Audio/Video, and Dijkstra. For DTW, recall is lower because (*i*) the number of affected instances is lower than in the HT threat, as the NoC has sufficient bandwidth to minimize the packet collisions that constitute the m_{app} threat; and (*ii*) the actual occurrence of a packet collision by m_{app} is an estimate, revealing a limitation in this particular test.

The analysis of precision under the m_{app} threat model reveals that MPEG, DTW, and Dijkstra were equal to or higher than those observed in the HT detection (Table 8.1). In contrast, the Audio/Video benchmark yielded a low precision of 0.432. This performance drop suggests that the model is unable to properly handle scenarios in which the application's temporal signature is severely disrupted by long packet collisions, particularly when m_{app} traffic affects multiple communication edges of the monitored application. Such a limitation might be mitigated by including model features that quantify the expected traffic collision rate, thereby allowing differentiation from m_{app}. Likewise, this limitation can also be addressed by an Actuation mechanism: if the application's temporal signature is disrupted at a point that results in numerous false positives, it may indicate that the application is actually under attack, and a countermeasure must be applied.

Table 8.3: Predictive performance for runtime m_{app} detection. Source: the Author.

Application	Recall	Precision	F1-Score
MPEG	0.963	1.000	0.981
DTW	0.889	0.958	0.922
Audio/Video	0.870	0.432	0.577
Dijkstra	0.882	0.988	0.932

The results in this Section presented high recall and precision across most benchmarks, demonstrating the feasibility of using application-specific temporal signatures to identify subtle traffic anomalies arising from both hardware- and software-level threats. The negligible impact of leaf quantization on predictive performance, alongside its minimal execution overhead, validates the approach as a lightweight, real-time solution. While specific communication patterns may introduce challenges, the overall performance mitigates the limitations 2 and 3 of the state-of-the-art.

8.3 Comparison with alternative methods

This Section compares the Test results using the HT as a threat model with two alternative approaches: LR and average latencies. To create and assess such approaches, the same Training and Test scenarios with HT were used. Those alternative approaches predict latency differently, but the detection threshold remains unchanged.

The LR model was trained using *LinearRegression* from Scikit-learn [Pedregosa et al., 2011]. The learned coefficients were exported from the trained model to implement the function in the Security Decider, replacing the transpiled XGBoost model.

The average latencies approach computes the average latency for each application's edge across all Training scenarios. These averages are inserted into the Security Decider, which identifies the monitored edge and retrieves its average latency in place of the transpiled XGBoost model.

Table 8.4 presents the Test results compared to the alternative approaches with the HT. The first column lists the application modeled. The second column indicates the detection method: XGBoost, LR, or Average. Columns 3–5 present the predictive performance in terms of recall, precision, and F1-Score, respectively. Columns 6–7 show the computational performance in terms of average inference and detection latencies, respectively.

The LR presented high detection performance for MPEG and DTW, with high precision and recall, resulting in an F1-score $>$ 90% for both applications, meaning that the temporal traffic signature can be approximated by a linear function across all their edges. However, for the remaining applications, LR detection performance was unacceptable, with an F1-score $<$ 70%. Despite high recall in Dijkstra and AES, its precision is too low, indicating it cannot differentiate normal traffic variations from the HT, leading to a high number of false positives. Furthermore, the LR model, which is not quantized due to its reliance on arithmetic operations, incurred, on average, 2.3$\times$ the inference latency compared to XGBoost. Nevertheless, the increase in inference latency wasn't sufficient to meaningfully increase the detection latency that could occur when monitoring packets are kept in the queue while inference is running.

Table 8.4: Runtime detection results against alternative methods. Source: the Author.

Application	Method	Recall	Precision	F1-Score	Avg. inf. latency	Avg. det. latency
MPEG	XGBoost	0.919	1.000	0.958	10.0 μs	53.7 μs
	LR	0.945	0.919	0.932	20.2 μs	64.0 μs
	Average	0.941	0.919	0.930	2.3 μs	45.7 μs
DTW	XGBoost	1.000	0.904	0.950	5.7 μs	53.0 μs
	LR	0.993	0.902	0.945	19.6 μs	74.8 μs
	Average	0.997	0.408	0.579	2.4 μs	49.2 μs
Audio/Video	XGBoost	0.924	0.711	0.804	9.4 μs	53.0 μs
	LR	0.760	0.416	0.537	19.4 μs	67.5 μs
	Average	0.898	0.433	0.584	2.3 μs	43.8 μs
Dijsktra	XGBoost	0.830	0.872	0.851	9.8 μs	52.6 μs
	LR	0.848	0.383	0.528	19.4 μs	61.5 μs
	Average	0.613	0.208	0.310	2.4 μs	45.6 μs
AES	XGBoost	0.923	0.717	0.807	10.7 μs	60.4 μs
	LR	0.895	0.550	0.681	19.8 μs	89.3 μs
	Average	0.913	0.469	0.620	2.3 μs	47.3 μs

Using edge average latency as a predictor yields the smallest inference latency in Table 8.4, at $\approx 2.3\,\mu s$. Although this decreases detection latency by an average of 15% compared to XGBoost, it does not represent a significant change, since XGBoost already detects the HT within an average of $53\mu s$. The detection performance using the average edge latency achieved a high F1-Score for MPEG, since its edges exhibit nearly constant behavior after warm-up (Figure 5.4). Conversely, the remaining applications showed unacceptable precision using average edge latency, and Dijkstra also showed low recall.

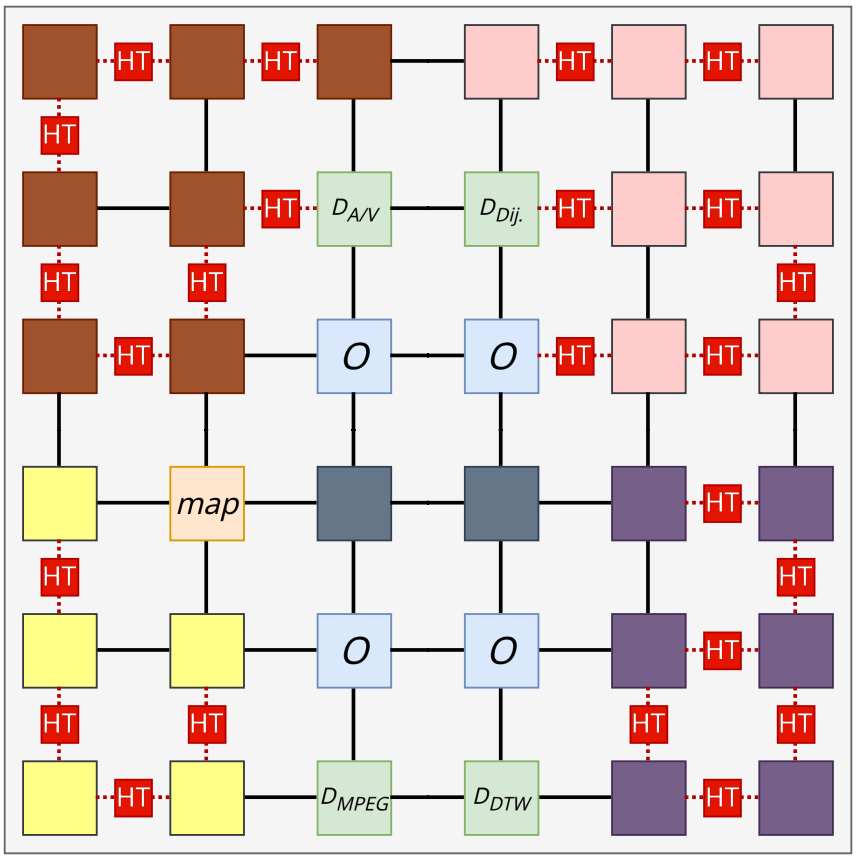
The comparison with alternative methods demonstrates that while simpler approaches, such as LR and latency averaging, can provide adequate detection performance for applications with highly deterministic communication patterns, they fail to maintain precision in more complex scenarios. The non-linear nature of the XGBoost models enables differentiation between legitimate system variability and malicious interference, significantly reducing false-positives when linear approximations yield unacceptable results. Although the averaging approach offers the lowest inference latency, the XGBoost models provide a favorable trade-off between predictive performance and computational overhead, enabling real-time detection.

8.4 Test scenario with multiple applications

This Section presents a test scenario in which multiple applications execute simultaneously with the security threat detection framework via its Security Observers and Security Deciders. Figure 8.2 shows the 6x6 MCSoC used for this test. Four applications

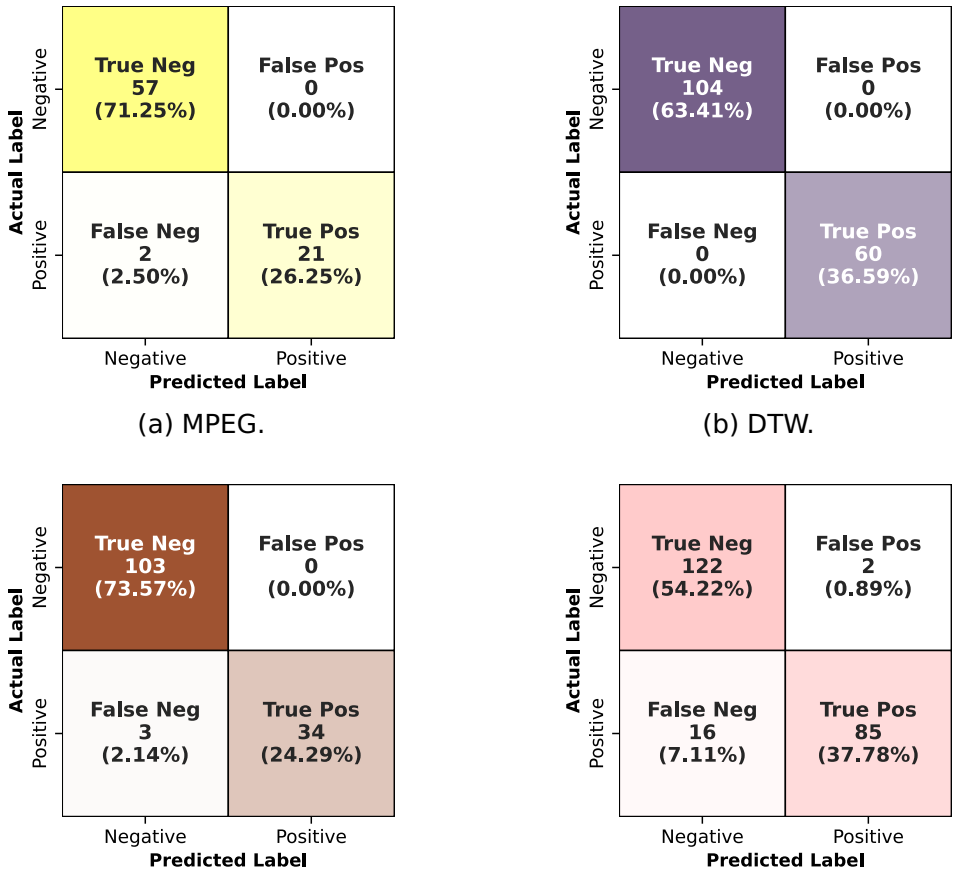
are mapped to the MCSoC: (*i*) MPEG, in yellow; (*ii*) DTW, in purple; (*iii*) Audio/Video, in brown; and (*iv*) Dijkstra, in pink. Each application has one task statically mapped to serve as an anchor for the dynamic mapping algorithm of the *map* task (in orange, at PE 1×2), which chooses the mapping locations of the remaining tasks. Likewise, the Security Observers (O) are statically mapped near each application, with Security Deciders for MPEG, DTW, Audio/Video, and Dijkstra (D_{MPEG}, D_{DTW}, $D_{A/V}$, and $D_{Dij.}$, respectively) mapped near the Observers. HTs are inserted into the links of PEs that originate packets over those links. For example, the Audio/Video task in PE 1×4 sends packets to the task at PE 2×5. Using XY routing, the originating communication link between PEs 1×4 and 2×4 includes an HT. The HTs follow the same parameters as Section 8.1, with a 33% chance of activating by 50–75 cycles.

Figure 8.3 shows the confusion matrices for the tested applications in the presented scenario. DTW (Figure 8.3b) presented no incorrect predictions, showing that 36.6% of the packets exchanged were affected by an HT, and all of them were correctly classified (TPs). Figure 8.3a shows the MPEG confusion matrix, with no FPs and 2 FNs, similar to Audio/Video (Figure 8.3c), which shows no FPs and 3 FNs. Dijkstra (Figure 8.3d) resulted in 2 FPs and 16 FNs, which remain low relative to the TPs (85) and TNs (122).



Yellow: MPEG; Purple: DTW; Brown: Audio/Video (A/V); Pink: Dijkstra (Dij.);
O: Security Observer; *D*: Security Decider.

Figure 8.2: Test scenario with four applications alongside the security threat detection framework, with HTs inserted. Source: the Author.



(a) MPEG. (b) DTW.

(c) Audio/Video. (d) Dijkstra.

Figure 8.3: Confusion matrices of threat detection for the test scenario. Source: the Author.

Table 8.5 summarizes the predictive performance presented in the confusion matrices. The first column lists the application. Columns 2–4 show the predictive performance in terms of recall, precision, and F1-Score, respectively. The fifth column shows the average detection latency. The average inference latency is not reported, as it uses the same models as in Section 8.2. The average execution time increase is also omitted, as it depends on the monitoring framework.

With the exception of Dijkstra, all models achieved 100% precision, i.e., no FPs. This particular scenario, which represents a typical MCSoC use case, maps tasks according to their application's CTG to reduce hops between communicating tasks and, consequently, the crossing traffic, thereby reducing variation in the edge's temporal signature and, therefore, resulting in fewer FPs. The recall remained high across all benchmarks, with Dijkstra showing the lowest value due to its more complex communication patterns, which make its modeling difficult with few input features in the XGBoost ensemble. The average detection latency remained under 60 μs, demonstrating the approach's real-time capability and its scalability through parallelization of monitoring among Security Observers and computation among Security Deciders, mitigating Limitation 1.

Table 8.5: Predictive performance metrics and detection latency for the scenario with multiple applications. Source: the Author.

Application	Recall	Precision	F1-Score	Avg. det. latency
MPEG	0.913	1.000	0.955	52.0 μs
DTW	1.000	1.000	1.000	59.4 μs
Audio/Video	0.919	1.000	0.958	52.1 μs
Dijkstra	0.842	0.977	0.904	52.3 μs

Figure 8.4 details the edge $e_{3,4}$ of MPEG by comparing the monitored latency against the predicted latency by the framework in runtime. The x-axis represents the system time, in *ms*, and the y-axis represents the packet end-to-end latency, in μs, The predicted latency (green curves) shows that the model learned the warm-up phase, which lasts up until $\approx 6\,ms$ of the system time. The squares in the green line highlight the predicted latency points where the framework classified the corresponding monitored latency as affected by the HT, i.e., where the monitored latency is at least 45 cycles above the predicted one. The first highlighted square shows that the corresponding monitored latency (yellow line) was actually affected by the HT and is therefore marked as a TP with a blue dot. Such behavior cannot be captured by using averaged metrics. If the first point is assumed not to be affected by the HT, its latency would be lower (e.g., near 8 μs), but using an average latency of 7 μs would still treat it as affected by the HT.

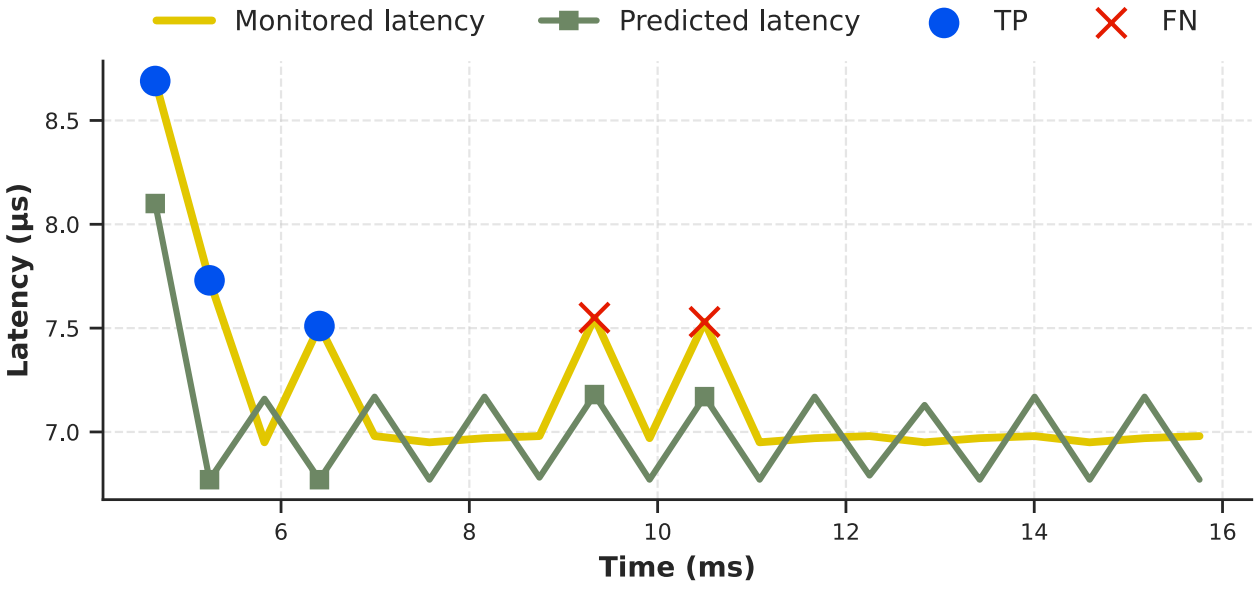


Figure 8.4: Comparison between monitored and predicted latencies for edge $e_{3,4}$ of MPEG. Source: the Author.

In Figure 8.4, the second and third highlighted squares show that, at the end and right after the warm-up, the latency should be lower than the monitored latency, thus correctly classifying those two monitored latency points as affected by the security threat.

After the warm-up, the MPEG application enters a state in which the model has learned to produce small variations. These variations are not observed in the monitored latency, although they are present in some training scenarios (as shown in Figure 5.4). This results in two False Negatives (FNs) highlighted in red $\times$ marks in the monitored latency, which are the only mispredictions for the MPEG application in this test scenario, as the threshold on the predicted latency still considers these two points as not affected by the HT, even though they actually are.

8.5 Final Remarks

The experimental results presented in this Chapter validate the effectiveness of the proposed security framework in providing a lightweight, real-time, and non-invasive security threat detection mechanism for MCSoCs. By assessing the framework across diverse communication patterns and under active security threats, the primary findings of the results are summarized as follows:

- **Detection performance:** The trained XGBoost models demonstrated high recall (83%–100%) and precision (87.2%–100%) for **most** benchmarks under the Hardware Trojan (HT) threat model. This indicates a capability to distinguish malicious anomalies from legitimate network jitter;
- **Impact of leaf quantization:** Quantizing the model's leaf weights reduces average inference latency by 82.3%–89.4% without compromising predictive performance. In complex applications like Dijkstra and AES, this optimization was necessary to prevent packet queuing and ensure real-time detection;
- **MCSoC scalability:** The distributed ODA-based management strategy, which parallelizes monitoring via Security Observers and computation via Security Deciders, maintained average detection latencies below 60 μs across a multi-application environment;
- **Minimal performance impact:** The DMNI-based monitoring framework incurred negligible overhead across most applications, with execution-time increases of less than 1% for MPEG, DTW, and Audio/Video, and approximately 5% for applications with more communication edges such as Dijkstra and AES;
- **Comparison with alternatives:** The XGBoost-based approach significantly outperformed both LR and the simple edge-average latency in terms of precision. This is evidence that non-linear modeling is necessary to capture the complex temporal signatures of parallel applications in MCSoCs.

Ultimately, the lightweight nature of the transpiled XGBoost models and the low-overhead monitoring framework successfully mitigate the limitations identified in the state-of-the-art regarding applicability. This implementation provides an infrastructure for securing MCSoCs against hardware- and software-level threats without compromising system performance.

9. CONCLUSION AND FUTURE WORK

This Thesis presented a lightweight, real-time, and non-invasive framework for security threat detection in NoC-based MCSoCs. The research was guided by the following thesis statement:

> *"This Thesis demonstrates that it is feasible to detect **subtle security threats** that cause traffic anomalies through a **lightweight**, **real-time**, and **non-invasive** Machine Learning approach, by profiling the behavior of real applications on a **synthesizable platform** modeled in RTL to achieve **trustworthy results** and mitigate the limitations in applicability and confidence found in the state-of-the-art."*

The validation of this statement is corroborated by the achievement of the research objectives. Below, the statement validation is demonstrated in parts:

*"This Thesis demonstrates that it is feasible to detect **subtle security threats** that cause traffic anomalies..."* – the proposed framework was demonstrated through subtle threat models. Such threats were implemented using hardware- and software-based approaches, with HTs and a malicious application (m_{app}), respectively. The HTs have a low activation probability (33%) and random, short communication delays (50–75 clock cycles) to evade detection. The m_{app} sends packets that can briefly collide with another application's traffic, using random intervals (100–500 μs) and a small number of flits (50). The results demonstrated that non-linear regression models, namely XGBoost, can learn temporal communication signatures of applications and use them to detect whether a monitored application is behaving as expected by comparing a monitored packet latency with its predicted value, allowing the detection of HT-caused traffic anomalies with F1-scores exceeding 80%.

*"... through a **lightweight**, **real-time**, and **non-invasive** Machine Learning approach..."* – the *bolt* tool was developed to convert trained models into C code, allowing the integration of inference logic directly into the PEs. Such models had a low memory footprint ($\leq 22\,kB$) and short inference times ($< 11\,\mu s$) when quantized, demonstrating their lightweight nature and enabling real-time detection latencies of less than $60\,\mu s$. The non-invasive monitoring integrated into the DMNI ensured that the performance impact on application execution remained negligible, below 1% for most benchmarks.

*"... by profiling the behavior of real applications on a **synthesizable platform** modeled in RTL to achieve **trustworthy results**..."* – the synthesizable RTL platform, Memphis-V, ensured that experimental results were grounded in clock-cycle accuracy, addressing the lack of confidence associated with high-level simulators. The framework was assessed at runtime using a realistic suite of benchmarks, including MPEG, DTW, Audio/Video, Dijkstra, and AES, evaluated through standard, appropriate predictive performance metrics.

"... and mitigate the limitations in applicability and confidence found in the state-of-the-art." – the following applicability and confidence limitations identified in the state-of-the-art were mitigated:

- **Limitations 1 (scalability), 2 (model overhead), and 3 (monitoring overhead):** mitigated by employing distributed, lightweight models that enabled real-time security threat detection, leveraged by the ODA management approach and a DMNI-based monitoring framework;
- **Limitations 4 (offline inference), 5 (router modification), 6 (MCSoC model confidence), 8 (synthetic traffic), and 10 (NoC-only systems):** mitigated by employing the RTL-modeled Memphis-V MCSoC with a complete software stack and realistic benchmarks, instead of high-level simulators or synthetic traffic. The experiments are assessed in runtime without requiring invasive modifications to the MCSoC;
- **Limitations 7 (predictive performance confidence) and 9 (easy-to-detect attacks):** mitigated by using hardware- and software-based threat models with random activation intervals with short durations, with results assessed using appropriate metrics (recall, precision, and F1-score).

This Thesis demonstrates that real-time security threat detection is feasible without compromising system performance or scalability by achieving high detection performance against subtle threats while maintaining a minimal memory footprint and performance overhead. The use of an RTL-modeled platform supports that these results are trustworthy and representative of actual system behavior, addressing the limitations of applicability and confidence previously identified in the state-of-the-art.

9.1 Publications

Ten publications were authored or co-authored during the period of this Thesis. Publications P1–3 are journal papers, while P4–10 are conference papers. Publications P3 and P8 are not directly related to this Thesis, but demonstrate the ODA control loop with a different MCSoC model.

Publication 10 contains early work on the monitoring framework proposed in Chapter 4. Publication 9 is not directly related to this Thesis, but it presents the task mapping defragmentation algorithm that justifies the Population Creation with minimal MCSoC size (Chapter 6).

Publication 6 presents RS5, which serves as the basis for the Memphis-V MCSoC (Chapter 4). Publications P2 and P5 are not directly related to this Thesis, but demonstrate the RS5 processor under different contexts.

Publication 7 presents early development of this Thesis, including the initial model training flow (Chapter 5) and an offline assessment of the results under the m_{app} threat. This publication was extended in P1, which refines the model training flow and presents an offline assessment of the results under the HT threat. Finally, Publication 4 won the best paper award at SBCCI'25, presenting the first results with runtime inference of the transpiled models under the HT threat.

P1. Threat Detection in NoC-based Manycores using Lightweight Machine Learning Models.
Dal Zotto, A. E.; Moraes, F. G.
IEEE Design & Test, 2025.
https://doi.org/10.1109/mdat.2025.3600352

P2. Deploying human activity recognition in embedded RISC-V processors.
Nunes, W. A.; Reusch, R. S.; Luza, L.; Bernardon, E.; **Dal Zotto, A. E.**; Juracy, L. R.; Moraes, F. G.
Springer Design Automation for Embedded Systems, v. 28, p. 187–217, 2024.
https://doi.org/10.1007/s10617-024-09288-w

P3. Chronos-V: a Many-core High-level Model with Support for Management Techniques.
Weber, I. I.; **Dal Zotto, A. E.**; Moraes, F. G.
Springer Analog Integrated Circuits and Signal Processing, v. 117, p. 57–71, 2023.
https://doi.org/10.1007/s10470-023-02190-8

P4. **Best Paper Award**
Lightweight Machine-Learning-driven Security Threat Detection in NoC-based Manycores.
Dal Zotto, A. E.; Moraes, F. G.
SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), 2025.
https://doi.org/10.1109/sbcci66862.2025.11218641

P5. RS5-SoC: A Flexible Open-Source RISC-V Platform for Embedded Systems.
Faccenda, R.; Nunes, W. A.; **Dal Zotto, A. E.**; et al.
SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), 2025.
https://doi.org/10.1109/sbcci66862.2025.11218649

P6. RS5: An Integrated Hardware and Software Ecosystem for RISC- V Embedded Systems.
Nunes, W. A.; **Dal Zotto, A. E.**; Borges, C. d.S.; Moraes, F. G.
IEEE Latin America Symposium on Circuits and Systems (LASCAS), 2024.
https://doi.org/10.1109/LASCAS60203.2024.10506171

P7. A Machine Learning Approach for Traffic Anomaly Detection in NoC-based Manycores.
Dal Zotto, A. E.; Moraes, F. G.
SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), 2024.
https://doi.org/10.1109/sbcci62366.2024.10703991

P8. A High-level Model to Leverage NoC-based Many-core Research.
Weber, I. I.; **Dal Zotto, A. E.**; Moraes, F. G.
SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), 2022.
https://doi.org/10.1109/sbcci55532.2022.9893235

P9. Leveraging NoC-based Many-core Performance Through Runtime Mapping Defragmentation.
Dal Zotto, A. E.; Borges, C. d.S; Ruaro, M.; Moraes, F. G.
IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2022.
https://doi.org/10.1109/ICECS202256217.2022.9970841

P10. Non-intrusive Monitoring Framework for NoC-based Many-Cores.
Dal Zotto, A. E.; Borges, C. d.S; Ruaro, M.; Moraes, F. G.
SBC Brazilian Symposium on Computing Systems Engineering (SBESC), 2022.
https://doi.org/10.1109/SBESC56799.2022.9965177

9.2 Future Work

This work can be further expanded in two directions: (*i*) enhancing model training; and (*ii*) enhancing the platform and management to leverage security threat detection. Some future work options are:

- **Interference Modeling:** Future iterations could incorporate features to quantify legitimate traffic contention, potentially reducing false positives in scenarios with high link competition;
- **Multitask Modeling:** The models presented in this Thesis were unable to learn temporal traffic signatures when PE-level multitasking was involved. Future work could explore additional features to train the model to support multitasking;
- **Actuation Mechanisms:** While this Thesis focused on detection, integrating countermeasure strategies, such as task migration, could be explored to mitigate detected threats in runtime;
- **NoC instrumentation**: Implementing instrumentation in the Hermes NoC routers could enable fine-grained analysis of packet collisions and improve the accuracy of threat labeling for software-induced threats (m_{app}).

REFERENCES

Abdollahi, M., Chegini, M., Hesar, M. H., Javadinia, S., Patooghy, A., and Baniasadi, A. (2024). NoCSNet: Network-on-Chip Security Assessment Under Thermal Attacks Using Deep Neural Network. In *IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–6. https://doi.org/10.1109/NoCArc64615.2024.10749907.

Abu-Mostafa, Y. S., Magdon-Ismail, M., and Lin, H.-T. (2012). *Learning From Data: A Short Course*. AMLBook. https://isbndb.com/book/9781600490064.

Agarwal, N., Krishna, T., Peh, L.-S., and Jha, N. K. (2009). GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42. https://doi.org/10.1109/ISPASS.2009.4919636.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67. https://doi.org/10.1145/1562764.1562783.

Barros, R. C., de Carvalho, A. C. P. L. F., and Freitas, A. A. (2015). *Automatic Design of Decision-Tree Induction Algorithms*. Springer. https://doi.org/10.1007/978-3-319-14231-9.

Barry, R. and The FreeRTOS Team (2024). *Mastering the FreeRTOS Real Time Kernel*. https://github.com/FreeRTOS/FreeRTOS-Kernel-Book.

Batzolis, E., Vrochidou, E., and Papakostas, G. A. (2023). Machine Learning in Embedded Systems: Limitations, Solutions and Future Challenges. In *IEEE Computing and Communication Workshop and Conference (CCWC)*, pages 345–350. https://doi.org/10.1109/CCWC57344.2023.10099348.

Bellman, K., Landauer, C., Dutt, N., Esterle, L., Herkersdorf, A., Jantsch, A., TaheriNejad, N., Lewis, P. R., Platzner, M., and Tammemäe, K. (2020). Self-aware Cyber-Physical Systems. *ACM Transactions on Cyber-Physical Systems*, 4(4):1–26. https://doi.org/10.1145/3375716.

Benini, L. and De Micheli, G. (2002). Networks on chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78. https://doi.org/10.1109/2.976921.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81. https://doi.org/10.1145/1454115.1454128.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7. https://doi.org/10.1145/2024716.2024718.

Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of Network-on-chip. *ACM Computing Surveys*, 38(1):1–51. https://doi.org/10.1145/1132952.1132953.

Butko, A., Garibotti, R., Ost, L., and Sassatelli, G. (2012). Accuracy evaluation of GEM5 simulator system. In *IEEE International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6. https://doi.org/10.1109/ReCoSoC.2012.6322869.

Caimi, L. L., Faccenda, R., and Moraes, F. G. (2021). A Survey on Security Mechanisms for NoC-based Many-Core SoCs. *SBC/SBMicro Journal of Integrated Circuits and Systems*, 16(2):1–15. https://doi.org/10.29292/jics.v16i2.485.

Carara, E. A., de Oliveira, R. P., Calazans, N. L. V., and Moraes, F. G. (2009). HeMPS - a framework for NoC-based MPSoC generation. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1345–1348. https://doi.org/10.1109/ISCAS.2009.5118013.

Catania, V., Mineo, A., Monteleone, S., Palesi, M., and Patti, D. (2016). Energy efficient transceiver in wireless Network on Chip architectures. In *ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1321–1326. https://isbndb.com/book/9783981537079.

Cerrolaza, J. P., Obermaisser, R., Abella, J., Cazorla, F. J., Grüttner, K., Agirre, I., Ahmadian, H., and Allende, I. (2020). Multi-core Devices for Safety-critical Systems: A Survey. *ACM Computing Surveys*, 53(4):1–38. https://doi.org/10.1145/3398665.

Charles, S., Lyu, Y., and Mishra, P. (2020). Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4510–4523. https://doi.org/10.1109/TCAD.2020.2972524.

Charles, S. and Mishra, P. (2021). A Survey of Network-on-Chip Security Attacks and Countermeasures. *ACM Computing Surveys*, 54(7):1–36. https://doi.org/10.1145/3450964.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. https://doi.org/10.1109/IISWC.2009.5306797.

Chen, T. and Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *ACM/SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1–10. https://doi.org/10.1145/2939672.2939785.

Coburn, J., Ravi, S., Raghunathan, A., and Chakradhar, S. (2005). SECA: Security-Enhanced Communication Architecture. In *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 78–89. https://doi.org/10.1145/1086297.1086308.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27. https://doi.org/10.1109/TIT.1967.1053964.

Dalzotto, A. E., Borges, C. D. S., Ruaro, M., and Moraes, F. G. (2022a). Leveraging NoC-based Many-core Performance Through Runtime Mapping Defragmentation. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 1–4. https://doi.org/10.1109/ICECS202256217.2022.9970841.

Dalzotto, A. E., da Silva Borges, C., Ruaro, M., and Moraes, F. G. (2022b). Non-intrusive Monitoring Framework for NoC-based Many-Cores. In *SBC Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–7. https://doi.org/10.1109/SBESC56799.2022.9965177.

Dalzotto, A. E., Ruaro, M., Erthal, L. V., and Moraes, F. G. (2021a). Dynamic Mapping for Many-cores using Management Application Organization. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 1–6. https://doi.org/10.1109/ICECS53924.2021.9665547.

Dalzotto, A. E., Ruaro, M., Erthal, L. V., and Moraes, F. G. (2021b). Management Application - a New Approach to Control Many-Core Systems. In *IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/SBCCI53441.2021.9529989.

Das, N., Pauls, F., Hasler, M., Haas, S., and Asmussen, N. (2024). Hardware Attack Models in Tiled Chip Multi-Core Processors: A Survey. In *IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 215–222. https://doi.org/10.1109/MCSoC64144.2024.00044.

Ding, R., Zhang, Y., Xia, S., Wang, Z., Deng, J., and Chen, X. (2024). Detection of Hardware Attacks in Network on Chip Based on Machine Learning. In *IEEE International Conference on Electronic Information and Communication Technology (ICEICT)*, pages 440–445. https://doi.org/10.1109/ICEICT61637.2024.10671248.

Distributed (Deep) Machine Learning Community (2014). XGBoost: eXtreme Gradient Boosting. https://github.com/dmlc/xgboost.

Distributed (Deep) Machine Learning Community (2023). TL2cgen: TreeLite 2 C GENerator. https://github.com/dmlc/tl2cgen.

Ditzel, D., Espasa, R., Aymerich, N., Baum, A., Berg, T., Burr, J., Hao, E., Iyer, J., Izquierdo, M., Jayaratnam, S., Jones, D., Klingner, C., Kim, J., Lee, S., Lupon, M., Magklis, G., Maric, B., Nath, R., Neilly, M., Northcutt, D., Orner, B., Renau, J., Reves, G., Reves, X., Riordan, T., Sanchez, P., Samudrala, S., Sole, G., Tang, R., Thorn, T., Torres, F., Tortella, S., and Yau, D. (2021). Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip. In *IEEE Hot Chips Symposium (HCS)*, pages 1–23. https://doi.org/10.1109/MM.2022.3140674.

Faccenda, R. F., Caimi, L. L., and Moraes, F. G. (2021). Detection and Countermeasures of Security Attacks and Faults on NoC-Based Many-Cores. *IEEE Access*, 9:153142–153152. https://doi.org/10.1109/ACCESS.2021.3127468.

Faceli, K., Lorena, A. C., Gama, J., and de Carvalho, A. C. P. L. F. (2011). *Inteligência Artificial: Uma Abordagem de Aprendizado de Máquina*. LTC. https://isbndb.com/book/9788521618805.

Foster, D. (2023). *Generative Deep Learning: Teaching Machines to Paint, Write, Compose and Play*. O'Reilly, 2nd edition. https://isbndb.com/book/9781098134181.

Galimberti, A., Purkait, R., Islam, N., Ganguly, A., Indovina, M., Zuzak, M., Pudukotai Dinakarrao, S. M., Zoni, D., and Fornaciari, W. (2024). ML-Assisted Attack Detection on NoC-Based Many-Cores Through On-Chip Traffic Monitoring. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 1–4. https://doi.org/10.1109/ICECS61496.2024.10848855.

Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer. https://doi.org/10.1007/978-3-642-24797-2.

Hathal, M. S., Saeed, B. M., Abdulqader, D. A., and Mustafa, F. M. (2024). Attack and anomaly prediction in networks-on-chip of multiprocessor system-on-chip-based IoT utilizing machine learning approaches. *Springer Service Oriented Computing and Applications*, 18(3):209–223. https://doi.org/10.1007/s11761-024-00393-z.

Hestness, J., Grot, B., and Keckler, S. W. (2010). Netrace: dependency-driven trace-based network-on-chip simulation. In *IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pages 31–36. https://doi.org/10.1145/1921249.1921258.

Hoffmann, H., Maggio, M., Santambrogio, M. D., Leva, A., and Agarwal, A. (2013). A generalized software framework for accurate and efficient management of performance goals. In *IEEE International Conference on Embedded Software (EMSOFT)*, pages 1–10. https://doi.org/10.1109/EMSOFT.2013.6658597.

Hu, S., Wang, H., and Halak, B. (2023). Cascaded Machine Learning Model Based DoS Attacks Detection and Classification in NoC. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. https://doi.org/10.1109/ISCAS46773.2023.10182218.

Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., and Stan, M. (2006). HotSpot: a compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513. https://doi.org/10.1109/TVLSI.2006.876103.

James, G., Witten, D., Hastie, T., Tibshirani, R., and Taylor, J. (2023). *An Introduction to Statistical Learning with Applications in Python*. Springer. https://doi.org/10.1007/978-3-031-38747-0.

Jeyasothy, A., Dora, S., Sundaram, S., and Sundararajan, N. (2022). Supervised Learning Using Spiking Neural Networks. In Angelov, P. P., editor, *Handbook on Computer Learning and Intelligence Volume 1: Explainable AI and Supervised Learning*, chapter 10, pages 423–465. World Scientific. https://doi.org/10.1142/12498.

Jiang, N., Becker, D. U., Michelogiannakis, G., Balfour, J., Towles, B., Shaw, D. E., Kim, J., and Dally, W. J. (2013). A detailed and flexible cycle-accurate network-on-chip simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96. https://doi.org/10.1109/ISPASS.2013.6557149.

Kulkarni, A., Pino, Y., French, M., and Mohsenin, T. (2016). Real-Time Anomaly Detection Framework for Many-Core Router through Machine-Learning Techniques. *ACM Journal on Emerging Technologies in Computing Systems*, 13(1):1–22. https://doi.org/10.1145/2827699.

Li, H., Liu, Q., and Zhang, J. (2016). A survey of hardware Trojan threat and defense. *ACM Integration, the VLSI Journal*, 55:426–437. https://doi.org/10.1016/j.vlsi.2016.01.004.

Liu, J., Harkin, J., Li, Y., and Maguire, L. P. (2015). Fault-Tolerant Networks-on-Chip Routing With Coarse and Fine-Grained Look-Ahead. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(2):260–273. https://doi.org/10.1109/TCAD.2015.2459050.

Madden, K., Harkin, J., McDaid, L., and Nugent, C. (2018). Adding Security to Networks-on-Chip using Neural Networks. In *SSCI*, pages 1299–1306. https://doi.org/10.1109/SSCI.2018.8628832.

Manferdelli, J. L., Govindaraju, N. K., and Crall, C. (2008). Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815. https://doi.org/10.1109/JPROC.2008.917730.

Moghimi, D. (2023). Downfall: Exploiting Speculative Data Gathering. https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi.

Monemi, A., Tang, J. W., Palesi, M., and Marsono, M. N. (2024). ProNoC: A low latency network-on-chip based many-core system-on-chip prototyping platform. *Elsevier Microprocessors and Microsystems*, 54:60–74. https://doi.org/10.1016/j.micpro.2017.08.007.

Moraes, F. G., Calazans, N. L. V., Mello, A. V., Möller, L. H., and Ost, L. C. (2004). HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. *ACM Integration, the VLSI Journal*, 38(1):69–93. https://doi.org/10.1016/j.vlsi.2004.03.003.

Mourelle, L. d. M., Nedjah, N., and Cardoso, A. N. (2024). Parallel Implementation of a Convolutional Neural Network on an MPSoC. In *ACM International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*, pages 335–347. https://doi.org/10.1007/978-981-97-4677-4_28.

National Institute of Standards and Technology (2017). An Introduction to Information Security. Technical Report Special Publication (SP) 800-12, Revision 1, U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-12r1.

Nunes, W. A., Dal Zotto, A. E., Borges, C. d. S., and Moraes, F. G. (2024). RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems. In *IEEE Latin America Symposium on Circuits and Systems (LASCAS)*, pages 1–5. https://doi.org/10.1109/LASCAS60203.2024.10506171.

Nunes, W. A., Santos, A. V., Marcon, C., and Moraes, F. G. (2025). Accelerating Machine Learning using RISC-V Vector Extension in a Manycore Platform. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–5.

OpenRISC (2012). mor1kx - an OpenRISC 1000 processor IP core. https://github.com/openrisc/mor1kx.

OPNET (2005). OPNET Network Simulator. https://opnetprojects.com/opnet-network-simulator.

Ormandy, T. (2023). Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *JMLR, Inc. Journal of Machine Learning Research*, 12(85):2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html.

Rahmani, A. M., Jantsch, A., and Dutt, N. (2018). HDGM: Hierarchical Dynamic Goal Management for Many-Core Resource Allocation. *IEEE Embedded Systems Letters*, 10(3):61–64. https://doi.org/10.1109/LES.2017.2751522.

Rajesh, J., Ancajas, D. M., Chakraborty, K., and Roy, S. (2015). Runtime Detection of a Bandwidth Denial Attack from a Rogue Network-on-Chip. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. https://doi.org/10.1145/2786572.2786580.

Rauber, T. and Rünger, G. (2013). *Parallel Programming for Multicore and Cluster Systems*. Springer, 2nd edition. https://doi.org/10.1007/978-3-642-37801-0.

Red Hat (2024). newlib 4.5.0. https://sourceware.org/newlib/.

Ruaro, M., Carara, E. A., and Moraes, F. G. (2014). Tool-set for NoC-based MPSoC debugging — A protocol view perspective. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2531–2534. https://doi.org/10.1109/ISCAS.2014.6865688.

Ruaro, M., Santana, A., Jantsch, A., and Moraes, F. G. (2021). Modular and Distributed Management of Manycore SoCs. *ACM Transactions on Computer Systems*, 38(1-2):1–16. https://doi.org/10.1145/3458511.

SiFive, Inc. (2020). *SiFive Interrupt Cookbook, Version 1.2*. https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

Sinha, M., Gupta, S., Rout, S. S., and Deb, S. (2021). Sniffer: A Machine Learning Approach for DoS Attack Localization in NoC-Based SoCs. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):278–291. https://doi.org/10.1109/JETCAS.2021.3083289.

Snyder, W., Wasson, P., and Galbi, D. (2025). Verilator v5.042. https://www.veripool.org/verilator/.

Sudusinghe, C., Charles, S., Ahangama, S., and Mishra, P. (2022). Eavesdropping Attack Detection Using Machine Learning in Network-on-Chip Architectures. *IEEE Design & Test*, 39(6):28–38. https://doi.org/10.1109/MDAT.2022.3202995.

Sudusinghe, C., Charles, S., and Mishra, P. (2021). Denial-of-Service Attack Detection using Machine Learning in Network-on-Chip Architectures. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 35–40. https://doi.org/10.1145/3479876.3481589.

Tan, P.-N., Steinbach, M., Karpatne, A., and Kumar, V. (2019). *Introduction to Data Mining*. Pearson, 2nd edition. https://isbndb.com/book/9780273769224.

Vashist, A., Keats, A., Pudukotai Dinakarrao, S. M., and Ganguly, A. (2019). Securing a Wireless Network-on-Chip Against Jamming-Based Denial-of-Service and Eavesdropping Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2781–2791. https://doi.org/10.1109/TVLSI.2019.2928960.

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

Wade, C. (2020). *Hands-On Gradient Boosting with XGBoost and scikit-learn: Perform accessible machine learning and extreme gradient boosting with Python*. Packt Publishing. https://isbndb.com/book/9781839218354.

Wang, H. and Halak, B. (2025). TampML: Tampering Attack Detection and Malicious Nodes Localization in NoC-Based MPSoC. *IEEE Transactions on Emerging Topics in Computing*, 13(2):551–562. https://doi.org/10.1109/TETC.2024.3434663.

Wang, H., Ren, J., Halak, B., and Atamli, A. (2024). GNS: Graph-Based Network-on-Chip Shield for Early Defense Against Malicious Nodes in MPSoC. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 14(3):483–494. https://doi.org/10.1109/JETCAS.2024.3438435.

Wang, K., Zheng, H., Li, Y., Li, J., and Louri, A. (2022). AGAPE: Anomaly Detection with Generative Adversarial Network for Improved Performance, Energy, and Security in Many-core Systems. In *ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 849–854. https://doi.org/10.23919/DATE54114.2022.9774693.

Wang, K., Zheng, H., and Louri, A. (2020). TSA-NoC: Learning-Based Threat Detection and Mitigation for Secure Network-on-Chip Architecture. *IEEE Micro*, 40(5):56–63. https://doi.org/10.1109/MM.2020.3003576.

Warden, P. and Situnayake, D. (2019). *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly. https://isbndb.com/book/9781492052043.

Weber, I. I., Zanini, V. B., and Moraes, F. G. (2024). Reinforcement learning for thermal and reliability management in manycore systems. *Springer Design Automation for Embedded Systems*, 29(1):1–34. https://doi.org/10.1007/s10617-024-09292-0.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The SPLASH-2 programs: characterization and methodological considerations. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 24–36. https://doi.org/10.1145/223982.223990.

Wu, L., Cui, P., Pei, J., Zhao, L., and Song, L. (2022). Graph Neural Networks. In Wu, L., Cui, P., Pei, J., and Zhao, L., editors, *Graph Neural Networks: Foundations, Frontiers, and Applications*, chapter 3, pages 27–37. Springer. https://doi.org/10.1007/978-981-16-6054-2_3.

Xue, M., Gu, C., Liu, W., Yu, S., and O'Neill, M. (2020). Ten years of hardware Trojans: a survey from the attacker's perspective. *IET Computers & Digital Techniques*, 14(6):231–246. https://doi.org/10.1049/iet-cdt.2020.0041.

Yao, J., Zhang, Y., Mao, Z., Li, S., Ge, M., and Chen, X. (2020). On-line Detection and Localization of DoS Attacks in NoC. In *IEEE Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, pages 173–178. https://doi.org/10.1109/ITAIC49862.2020.9338861.

Yoon, J. Y., Concer, N., Petracca, M., and Carloni, L. P. (2013). Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(12):1906–1919. https://doi.org/10.1109/TCAD.2013.2276399.

Zeigerman, I., Yershov, V., and Titov, N. (2019). m2cgen: Model 2 Code Generator. https://github.com/BayesWitnesses/m2cgen.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br