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

GUSTAVO COMARÚ RODRIGUES

A PROBING APPROACH FOR HARDWARE TROJAN LOCALIZATION IN NOC-BASED MANYCORES

Porto Alegre
2025

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

A PROBING APPROACH FOR HARDWARE TROJAN LOCALIZATION IN NOC-BASED MANYCORES

GUSTAVO COMARÚ RODRIGUES

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

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2025

Ficha Catalográfica

R696p Rodrigues, Gustavo Comarú

A Probing Approach for Hardware Trojan Localization in NoC-based Manycores / Gustavo Comarú Rodrigues. – 2025.
91.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. NoC-based manycores. 2. Security. 3. Hardware Trojan (HT). 4. HT Localization. I. Moraes, Fernando Gehm. II. Título.

GUSTAVO COMARÚ RODRIGUES

A PROBING APPROACH FOR HARDWARE TROJAN LOCALIZATION IN NOC-BASED MANYCORES

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

Sanctioned on April 29, 2025.

COMMITTEE MEMBERS:

Dr. Rafael Fraga Garibotti (Vector Trading)

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

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

AGRADECIMENTOS

Gostaria de aproveitar esse espaço para agradecer algumas das pessoas que tornaram esse trabalho possível.

Agradeço aos meus orientadores, Fernando Moraes, Rafael Faccenda, e Luciano Caimi, por me acolherem no grupo de pesquisa desde a graduação e me orientarem sempre com muita paciência, me emprestando um pouco de suas experiências e de suas ideias.

Agradeço à minha família, e especialmente aos meus pais, Raquel e Eduardo, por todo o apoio que me deram e que continuam me dando durante esses anos de estudo.

Agradeço ao meu namorado, Filipe, que me aguentou pelas longas noites e finais de semana que passei trabalhando e escrevendo essa dissertação.

Finalmente, agradeço às instituições CNPq e HP, que proveram apoio financeiro, permitindo que eu me dedicasse a esse trabalho.

LOCALIZAÇÃO DE HARDWARE TROJANS BASEADA EM SONDAS PARA MANYCORES BASEADOS EM NOC

RESUMO

À medida que a adoção e a complexidade de sistemas manycore aumentam, garantir a proteção de dados tornou-se um requisito crítico de projeto. Além disso, o uso generalizado de núcleos de propriedade intelectual de terceiros (3PIPs) para atender às restrições de tempo de lançamento no mercado e reduzir os custos de projeto aumenta o risco de inserção de hardware malicioso por meio de Trojans de Hardware (HTs), aumentando assim a vulnerabilidade das plataformas manycore. A rede intra-chip (NoC), devido ao seu papel central na arquitetura, torna-se um alvo atraente para inserção de HTs, pois fornece acesso a todos os outros componentes do sistema. Um HT infectando a NoC pode permitir vários ataques, como negação de serviço (DoS) e degradação de desempenho. Quando tais ataques são detectados, o sistema deve implementar contramedidas para interromper o ataque e proteger as aplicações em execução. No entanto, desconhecer a localização do HT reduz a eficácia das contramedidas. Embora a literatura ofereça técnicas para identificar a origem de ataques em NoCs, estas normalmente requerem recursos de segurança adicionais integrados ao hardware da NoC, tornando-os inadequados para NoCs baseadas em 3PIP não seguros. Esta dissertação tem por objetivo desenvolver um método não invasivo para localizar links infectados por HTs, a fim de abordar a limitação de adicionar hardware a módulos não seguros. Este trabalho apresenta uma estrutura de segurança em três fases que executam as seguintes ações: (1) monitora a comunicação entre tarefas para detectar ataques de HT; (2) emprega um algoritmo de localização para identificar os links infectados dentro da NoC; e (3) aplica contramedidas para neutralizar ou mitigar os efeitos do ataque. O algoritmo de localização de HT utiliza uma técnica chamada *path probing*, que transmite pacotes de sondagem ao longo de caminhos específicos da NoC para avaliar a integridade do link. O algoritmo envia sondagens seletivamente e analisa seus resultados, refinando a busca a cada resultado até que o HT seja localizado com precisão. O método é implementado em software, permitindo a localização de HTs sem modificar o hardware da NoC. Para validar a abordagem proposta, conduzimos uma série de campanhas de ataque nas quais HTs atacaram o manycore usando diferentes padrões de ativação. Os resultados demonstram que a estrutura de segurança identificou com sucesso a localização dos HTs, causando impacto mínimo no desempenho do sistema.

Palavras-Chave: Manycores baseados em NoC, segurança, Hardware Trojan (HT), localização de HTs.

A PROBING APPROACH FOR HARDWARE TROJAN LOCALIZATION IN NOC-BASED MANYCORES

ABSTRACT

As the adoption and complexity of manycore systems increase, ensuring data protection has become a critical design requirement. Additionally, the widespread use of third-party intellectual property cores (3PIPs) to meet time-to-market constraints and reduce design costs, raises the risk of malicious hardware insertion through Hardware Trojans (HTs), thereby increasing the vulnerability of manycore platforms. The Network-on-Chip (NoC), due to its central role in the architecture, becomes an attractive target for HT insertion, as it provides access to all other system components. An HT infecting the NoC can enable various attacks, such as denial-of-service (DoS) and performance degradation. When such attacks are detected, the system must deploy countermeasures to halt the attack and protect running applications. However, not knowing the HT's location reduces the effectiveness of countermeasures. Although the literature offers techniques for identifying the source of attacks in NoCs, these typically require additional security features integrated into NoC hardware, rendering them unsuitable for non-secure 3PIP-based NoCs. This dissertation aims to develop a non-invasive method for localizing HT-infected links in the NoC to address the limitation of adding hardware to non-secure modules. This work introduces a three-phase security framework executing the following actions: (1) monitors inter-task communication to detect HT attacks; (2) employs a localization algorithm to identify the infected links within the NoC; and (3) applies countermeasures to neutralize or mitigate the effects of the attack. The HT localization algorithm uses a technique called *path probing*, which transmits probe packets along specific NoC paths to evaluate the link integrity. The algorithm selectively sends probes and analyzes their outcomes, refining the search with each result until the HT is accurately localized. The method is implemented in software, allowing HT localization without modifying the NoC hardware. To validate the proposed approach, we conducted a series of attack campaigns in which HTs attacked the manycore using different activation patterns. The results demonstrate that the security framework successfully identified the location of the HTs while incurring minimal impact on system performance.

Keywords: NoC-based manycores, security, hardware Trojan (HT), HT localization.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. INTRODUCTION

Manycores are platforms designed to provide high performance through the use of parallelism, meeting the current demand of embedded devices with power consumption and communication constraints. A manycore contains PEs (Processing Elements) interconnected by complex communication infrastructures, such as hierarchical buses or NoCs (Networks-on-Chip) [Popovici et al., 2010]. PEs may be processors, 3PIP (third-party intellectual property) modules, memory blocks, or dedicated hardware accelerators. Examples of architectures with a large number of processors interconnected by NoCs include the Mellanox family TILE-Gx72 (72 cores) [Tecnhlogies, 2018], Intel Knights Landing [Sodani et al., 2016], Oracle M8 (32 cores) [Oracle, 2017], Kalray array (256 cores) [Dinechin et al., 2014], KiloCore chip (1,000 cores) [Bohnenstiehl et al., 2016], and Esperanto (1,100 RISC-V cores) [Peckham, 2020].

An NoC consists of routers and links and is responsible for forwarding data and control messages between PEs. Network Interfaces (NIs) connect PEs to the routers of the NoC. Whenever a PE sends a message, the NI transforms it into a packet and delivers it to the router. Then, the router sends the packet to a neighbor router through a link according to a path defined by the routing algorithm. The routers constitute the underlying communication infrastructure of the system, where multiple interconnected routers define the network topology [Hemani et al., 2000; Benini and Micheli, 2002].

As the adoption and complexity of manycores increase, the concern for data protection appears as a design requirement [Baron et al., 2013]. A manycore may be employed in scenarios where availability is critical and downtimes must be minimized. These systems may also handle sensitive information; thus, protecting this data from unauthorized access is necessary. The following seven security principles [Ramachandran, 2002] are generally accepted as the foundation of a good security solution, the first three principles being mandatory features:

- Confidentiality: the property of non-disclosure of information to unauthorized processes, entities, or users;
- Availability: the protection of assets from DoS (Denial-of-Service) threats that might impact the availability of any system resource;
- Integrity: the prevention of modification or destruction of an asset by an unauthorized entity or user;
- Authentication: the process of establishing the validity of a claimed identity;
- Authorization: the process of determining whether a validated entity is allowed to access a secured resource based on attributes, predicates, or context;

- Auditing: the property of logging the system activities at levels sufficient for the reconstruction of events;
- Nonrepudiation: the prevention of any participant denying his role in the interaction once it is completed.

A consequence of the increasing number of features and functionalities inside a single chip is the adoption of 3PIPs to meet time-to-market constraints and reduce design costs. Such IPs come from different vendors, raising the risk of having a Hardware Trojan (HT) insertion [Li et al., 2016]. Assuming HTs infect the NoC, these can perform several attacks that threaten security principles [Ramachandran, 2002]. Such attacks may affect *confidentiality* by redirecting messages to malicious agents, *availability* by dropping messages or blocking a communication path, and *integrity* by corrupting the content of a packet traversing the NoC.

The literature presents several techniques, such as cryptography [Charles and Mishra, 2020], authentication codes [Sharma et al., 2019], error correction codes [Gondal et al., 2020], creation of a communication flow profile to detect anomalous behavior [Charles et al., 2020], spatial isolation via Secure Zones (SZ) to protect communication and computation simultaneously [Fernandes et al., 2016]. Adopting these techniques makes it possible to detect violations related to security or faults in the NoC.

A particular case of SZ is the Opaque Secure Zone (OSZ) [Caimi and Moraes, 2019], which is a defense mechanism executed at runtime that focuses on finding a region on the system with free PEs to map an application with security constraints. The OSZ activation occurs by setting a link control structure at the boundaries of the rectilinear region, blocking all incoming and outgoing traffic trying to cross the OSZ. OSZ prevents attacks from outside sources, such as Denial-of-Service (DoS), timing attacks, spoofing, and man-in-the-middle [Caimi et al., 2018]. Even though the method is robust against external attacks, it still presents vulnerabilities when HTs infect routers inside the OSZ or when the application running in the OSZ needs to communicate with external peripherals.

The previous work [Comarú, 2022] addressed the vulnerabilities in IO communication by proposing the Secure Network Interface with Peripherals (SNIP) (Section 3.2), which manages access to IO devices and ensures that communication between an application and a peripheral is secure. During the MSc program, the SNIP was further modified to be fully integrated into the manycore platform (Section 3.1). New functionalities were added to send warnings when a security threat is detected.

As stated, an HT infecting the NoC makes the manycore vulnerable to several attacks, such as packet misrouting, dropped packets, data tempering, and network flooding. When such attacks are detected, the system must deploy a countermeasure to stop the attack and protect the applications. However, very limited countermeasures can be taken without knowing where the HT was implanted. The literature presents solutions to localize attacks in NoCs (Chapter 2), *but methods adopted in the literature rely on adding hardware to the untrusted router, making the security mechanism itself insecure*.

1.1 Objectives

The strategic objective of this work is to create a noninvasive algorithm to localize HT-infected links in an untrusted NoC by probing the network with test packets.

To reach the strategic objective, the following specific goals are set:

SG1 **Integrate the SNIP into the platform**
Complete the integration of the SNIP into the reference manycore platform.

SG2 **Warning generation on the SNIP**
Detect security anomalies within the SNIP and send warning messages to the Security Manager on the MPE (Manager PE) so it can detect the occurrence of attacks.

SG3 **HT insertion framework**
Build a framework to automatically insert HTs in the NoC when compiling the platform, making it possible to perform different simulations of HT attack scenarios.

SG4 **Probing protocol for localizing static HTs in the NoC**
The core of this work. This objective consists in the design and implementation of an algorithm for localizing HTs by probing the NoC with test packets. This first version of the algorithm focuses on static HTs (that have a fixed activation window and are easier to localize). The rationale for choosing this type of HT is to simplify the attack detection and focus the research on the probe method.

SG5 **Extend probing protocol to detect intermittent HTs**
Make modifications to the probing protocol to detect intermittent HTs (that have random activation windows and are more challenging to localize).

SG6 **Integration with the Security Manager**
Integrate the probing protocol with the Security Manager on the MPE. The Security Manager will acquire the information generated across the system, identify the HT attacks, and trigger the HT localization process.

1.2 Original Contributions

This section enumerates the original contributions of this dissertation.

- **Warning feature of the SNIP** (Section 3.2) – SG1 and SG2. This work expands the SNIP security feature of the baseline platform. The SNIP was modified to integrate a warning generation feature that notifies the security manager whenever a security anomaly is detected at the SNIP. This allows the security manager to recognize different types of attacks to IO communication and apply specialized countermeasures.
- **HT Insertion Framework** (Section 4.2) – SG3. An automated framework for the insertion of HTs in the NoC. The proposed framework is agnostic to the NoC and can thus be configured to work with other platforms. This framework sets the foundation for conducting extensive HT-based attack campaigns on the NoC. Additionally, it enables the exploration and development of new security mechanisms, thereby contributing to the field of security in manycore systems.
- **Fault-tolerance mechanisms for the NoC** (Section 3.3) – SG3. Although the literature acknowledges the occurrence of attacks and faults, there is no solution to their side effects on NoC. We identified four types of side effects induced by HTs that negatively affect message exchange between PEs, and we proposed solutions to keep the NoC operational during faults or HT attacks.
- **Probe API** (Section 6.1) – SG4 and SG5. Implements a protocol that sends probe packets to test specific routes of the NoC. The probes can detect HT attacks that disrupt the communication between cores by dropping or delaying packets. The probes can be configured in different ways, allowing the Probe API to detect HTs with both large and small activation windows while optimizing performance.
- **HT Localization Algorithms** (Section 6.2) – SG4 and SG5. Security mechanisms that find the location of HTs inside an infected path. The localization algorithms use the Probe API to perform systematic tests on the NoC and locate the position of HTs. We propose different algorithms to locate both static and intermittent HTs. The advantage of our approach is that the HT localization does not rely on adding security features to the NoC router, but instead uses probe packets to test paths of the NoC.
- **HT localization flow integrated in the security manager** (Section 4.4) – SG6. We propose integrating different security mechanisms into one unified HT localization flow in the security manager. This security flow is responsible for: monitoring the communication, detecting HT attacks, finding the location of the HTs, and deploying the corresponding countermeasure. Furthermore, it generates a health report about each link in the NoC and provides critical information that the system manager can use to make decisions regarding security and fault tolerance. The HT localization flow is designed to be modular; each step of this process can be swapped for other implementations according to necessity. The proposed flow is flexible and, thus, can be adapted for adoption in other platforms.

1.3 Publications During de MSc Period

During the development of this work, the author authored or co-authored four conference papers and three journal articles.

Conference papers:

Lightweight Authentication for Secure IO Communication in NoC-based Many-cores
Rafael Follmann Faccenda, **Gustavo Comarú**, Luciano Lores Caimi, Fernando Gehm Moraes
In: ISCAS 2023
https://doi.org/10.1109/ISCAS46773.2023.10181962

Secure Network Interface for Protecting IO Communication in Many-cores
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
In: SBCCI 2023
https://doi.org/10.1109/SBCCI60457.2023.10261655

Fortifying NoC-Based Manycores: Distributed Monitoring to Detect Security Threats
Rafael Follmann Faccenda, **Gustavo Comarú**, Ney Calazans, Luciano Lores Caimi, Fernando Gehm Moraes
In: ICECS 2024
https://doi.org/10.1109/ICECS61496.2024.10849315

Hardware Trojan Localization for Untrusted Network-on-chips
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
In: LASCAS 2025

Journal articles:

SeMAP - A Method to Secure the Communication in NoC-based Many Cores
Rafael Follmann Faccenda, **Gustavo Comarú**, Luciano Lores Caimi, Fernando Gehm Moraes
IEEE Design & Test, vol. 40(5), pp 42-51, October 2023.
https://dx.doi.org/10.1109/MDAT.2023.3277813

A Comprehensive Framework for Systemic Security Management in NoC-Based Many-Cores
Rafael Follmann Faccenda, **Gustavo Comarú**, Luciano Lores Caimi, Fernando Gehm Moraes
IEEE Access, vol. 11, pp 131836-131847, November 2023
https://doi.org/10.1109/ACCESS.2023.3336565

Integration of Monitoring Mechanisms in Secure Network Interfaces for Peripherals to Protect IO Communication in NoC-based Many-cores
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
Journal of Integrated Circuits and Systems (JICS), vol. 19, n. 3, December 2024.
https://doi.org/10.29292/jics.v19i3.907

1.4 Document Organization

The remainder of this dissertation is organized as follows.

- Chapter 2 discusses related work to this dissertation. This chapter includes proposals that localize attacks in NoC-based manycores or protect the system against HTs at runtime. This chapter ends by evaluating the weaknesses and strengths of the discussed works.
- Chapter 3 presents the baseline manycore system used to develop this work. The chapter is divided into four parts: **a)** an overview of the baseline platform and its main components; **b)** an introduction to SNIP, which is a security feature added to the platform to protect communication with peripherals; **c)** an explanation of faults/attacks caused by HT effects in the NoC; and **d)** a discussion about the architectural assumptions used in this work.
- Chapter 4 elaborates the problem that this dissertation aims to solve. It begins by describing how the HT works, then explains its insertion into the platform, defines the attacks considered throughout this work, and finally provides an overview of the proposed solution.
- Chapter 5 presents the monitoring phase of the proposed security flow. This chapter explains how communication is monitored and how the HT attacks are detected. This chapter also presents the main data structures used throughout the HT localization proposal.
- Chapter 6 is the core of this dissertation. It presents the localization phase of the proposed security flow, which includes the Probe API used to test individual paths of the NoC by sending probe packets, and the HT localization algorithms that systematically search the NoC to find the location of HTs.
- Chapter 7 evaluates the security flow through a set of attack campaigns.
- Chapter 8 concludes this dissertation and discusses possible future work.

2. RELATED WORK

This chapter discusses proposals that localize attacks in NoC-based manycores or propose solutions to protect the system against HTs at runtime.

2.1 Hardware Trojan Detection and High-precision Localization in NoC-based MP-SoC using Machine Learning

Wang and Halak [2023] considers that the NoC routers can be infected by HTs performing tampering attacks, modifying the packet content before injecting it into the NoC or during its transmission through the routers. Tampering the packets' content could also lead to information leakage and DoS. To tackle this issue, the authors propose a framework for detecting the tampering attacks and localizing the HT-infected router.

The proposed framework uses a machine-learning (ML) model to find packet content anomalies, thus detecting tampering attacks. Once an attack is detected, every router in the XY path taken by the anomalous packet is considered suspicious. Tables are used to keep the "security credit" of each router; every time a tampering attack is detected, the suspicious routers have their security credit decremented. Over time, a low-security credit value determines which router is responsible for the attacks.

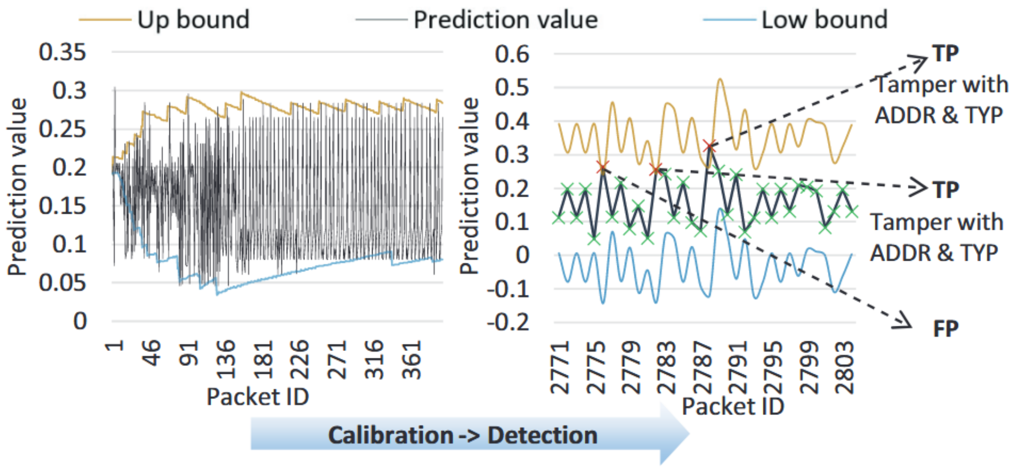
This process is divided into three steps: calibration, detection, and localization. The ML model used to detect tampering is trained offline and considers features such as source router, destination router, memory address, and packet type. The sensitivity of the ML model is adjusted dynamically at runtime to detect anomalies more accurately.

Calibration phase: the first few hundred packets sent through the NoC are considered normal and used to calibrate the sensitivity of the tampering detection. After this initial calibration, the sensitivity will remain dynamically adjusted throughout the system's execution.

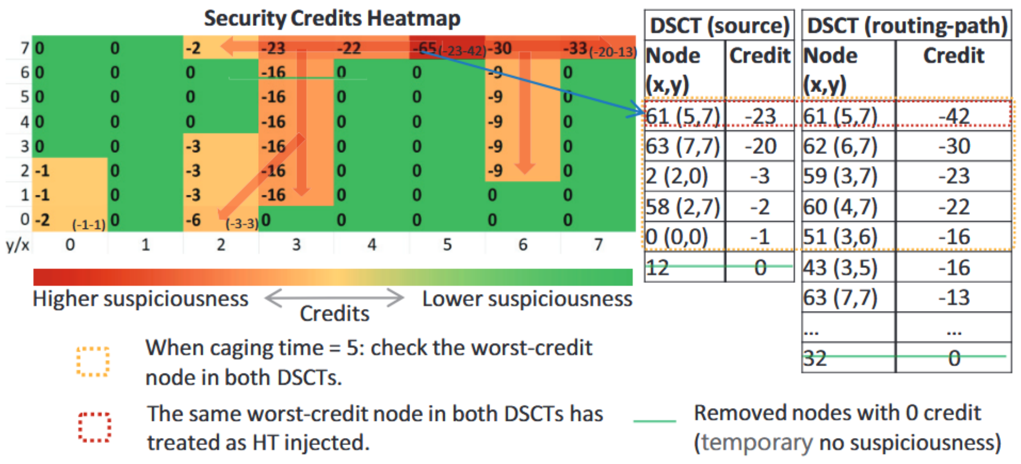
Detection phase: the ML model monitors packet parameters during this phase. They are considered anomalous if they are out of the sensitivity window defined by the calibration. Anomalous packets trigger the localization phase, while regular packets are used to calibrate the mechanism further.

Localization phase: The system updates the security credit tables whenever a tampered packet is detected. There are two different tables: one for source routers and another for path routers. When the tampered packet is found, the tables are update by: decrementing 5 credits from the source router (in the first table) and 3 credits from the other routers in the packet's path (in the second table). Every other router in the system recuperates 1 credit, as they are not involved in this attack. Two criteria are used to stop the localization: (a) there must be at least 5 routers with negative credits in each table, and (b) the worst routers must be the same in both tables. If these conditions are met, the router with the smallest credit is considered infected by an HT.

Figure 2.1 illustrates the different steps of the localization framework. Calibration and detection are shown in Figure 2.1 (a). The black line represents the output prediction of the ML. The blue and yellow lines correspond to the lower and upper thresholds for detecting anomalies. Figure 2.1 (b) shows the localization phase in a 7x7 system. The HT infects the router (5,7). The heat map shows the credit value of each router in the system. The routes taken by malicious packets are indicated with arrows. The picture shows both credit tables: source and routing path. This scenario meets the two-stop conditions and successfully localizes the HT in the (5,7) router.



(a)


Security Credits Heatmap
y/x
Higher suspiciousness
Credits
Lower suspiciousness
When caging time = 5: check the worst-credit node in both DSCTs.
The same worst-credit node in both DSCTs has treated as HT injected.
Removed nodes with 0 credit (temporary no suspiciousness)

DSCT (source)	
Node (x,y)	Credit
61 (5,7)	-23
63 (7,7)	-20
2 (2,0)	-3
58 (2,7)	-2
0 (0,0)	-1
~~12~~	~~0~~

DSCT (routing-path)	
Node (x,y)	Credit
61 (5,7)	-42
62 (6,7)	-30
59 (3,7)	-23
60 (4,7)	-22
51 (3,6)	-16
43 (3,5)	-16
63 (7,7)	-13
...	...
~~32~~	~~0~~

(b)

Figure 2.1 – Calibration, detection, and localization steps of the framework, considering a 7x7 system with one HT in the router (5,7). (Source: Wang and Halak [2023].)

A weakness of this method is that it relies solely on the anomalous packets to perform the localization. Since the framework uses the intersection between suspicious paths to determine the infected router, an attack that does not affect many different paths could remain unlocalized. As future work, the authors suggest proactively sending/receiving packets to search for hidden HTs.

2.2 EETD - An Energy Efficient Design for Runtime Hardware Trojan Detection in Untrusted Network-on-Chip

Hussain et al. [2018] divide HT detection in two categories: End-to-End (E2E) and Hop-to-Hop (H2H). E2E detection monitors only the communication endpoints, which are relatively less costly but cannot directly localize the HT. On the other side, H2H detection deploys security mechanisms in every communication router, being able to immediately localize the HT upon detection. The main problem with H2H detection is that it amounts to a greater area and power overhead. While the HT is not activated, a significant amount of energy is wasted due to unnecessary monitoring.

The authors propose an HT localization solution that leverages E2E and H2H detection features. It consists of an energy-efficient HT detection (EETD) design that uses selective activation/deactivation of detection units to reduce power overhead.

The EETD framework uses two types of detection units. The first type is the **E2E Detection Units (EDUs)**, which are attached to the cores and are always active. They authenticate the incoming packets and detect when an HT becomes active. The second type of detection unit is the **Localization Units (LUs)**. They are deployed in the NoC links and can be dynamically attached/detached from the system. To implement the framework, any state-of-the-art attack detection unit can be used as an EDU. Similarly, to locate the HT-infected router some H2H detection units can be used as LUs.

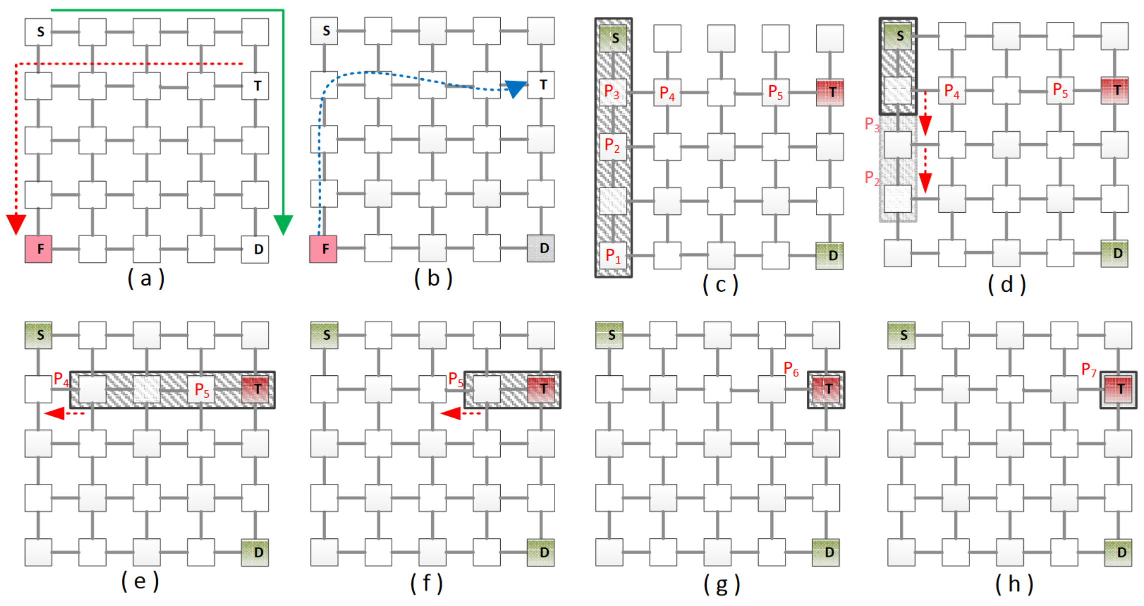
When an EDU detects the occurrence of an HT attack, it activates a worm-based algorithm that selectively enables LUs to localize the HT. The "worm" follows the direction of the tampered packets until it stops in the location of the HT-infected router. Figure 2.2 illustrates this process. The worm-based localization algorithm is composed of four steps:

Column movement: the algorithm starts by activating all LUs on the column. Each LU can capture the tampered packets from all directions. If the attacks are detected in the vertical links, the worm will move vertically along the column. Figure 2.2 (c-d).

Row activation: during this step, the worm detects the row infected by the HT and enables their correspondent LUs. This can be done either by receiving a tampered packet from a horizontal link, or if the worm is stuck at a location for too long and exceeds the threshold time. Figure 2.2 (e).

Row movement: a row search is performed, similar to the way a column search was performed in step 1. For each new detection, the worm moves forward in the direction of the HT. Figure 2.2 (f-g).

Localization: finally, if the worm stops at a location for too much time and exceeds the threshold, the worm location is regarded as the HT location. Figure 2.2 (h).



The worm-based algorithm for localizing an HT. The source router (S) sends a message to target (D), but an HT in the path (T) diverts the packet to another router (F). The EETD framework starts the worm-based algorithm to search for the HT.

Figure 2.2 – HT localization method proposed by Hussain et al. [2018].

The authors propose an HT localization mechanism that mitigates the costs of H2H detection. Enabling the LUs only when performing the localization algorithm results in saving power. A drawback is that since the detection and localization of the HT are performed at different moments, the HT can switch off before the localization algorithm is finished executing, thus remaining unlocalized.

2.3 Sniffer - A Machine Learning Approach for DoS Attack Localization in NoC-based SoCs

Sinha et al. [2021] aim to recover the NoC from DoS-flooding attacks by localizing Malicious IPs (MIPs). According to the authors, the statically configured thresholds typically used in the literature to detect anomalies are unreliable for real and dynamic systems. They propose an ML-based system called Sniffer, which automatically configures thresholds to detect malicious behavior and accurately localize MIPs.

In this framework, each router contains its own ML model, which is trained offline using NoC features such as: buffer witting time, inter-flit interval, virtual channel occupancy. These models are used to distinguish between normal and malicious flows in the NoC routers.

The localization algorithm employed by the Sniffer framework can be divided into the following steps:

Attack detection: each IP block is responsible for monitoring the incoming communication and detecting the occurrence of flooding attacks. Once the IP detects an attack, it initiates the Sniffer localization process by raising a flag to the router.

Congestion inspection: the Sniffer uses an ML model embedded within the router to inspect the congestion status of the incoming ports, determining whether they are under attack or not. If the ML model flags the status as an attack, the router creates a probing packet and sends it to the neighbor router toward the suspicious port. When the next router receives the probing packet, it will repeat the inspection step in its ports. As this process continues, the probing packet will be propagated toward the MIP.

MIP localization: the probing packet traverses through the NoC in the opposite direction of the attack path, eventually arriving at the malicious IP. When this happens, the congestion inspection will find that the local port is the source of the malicious flow, and the MIP will be localized.

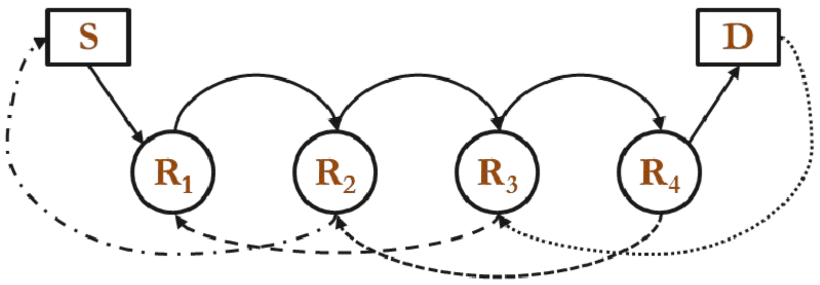
To better detect anomalous flows, the ML model uses a collective decision-making strategy: neighbor routers help to decide whether an attack is happening or not. Each router uses its ML model to detect the anomalous behavior, and then an AND/OR operation is performed between the outputs of the neighbor models. Whether an AND or OR operation is performed is decided offline, and help prevent false-positives and false-negatives, respectively.

When sending the probing packet, each router appends its own Node_ID in the packet payload. This serves the purpose of detecting loops in the path followed by the probe packet. Such loops would occur when an attack is coordinated by multiple IPs. If the probe packet returns to its origin, the Node_IDs are retrieved from the payload, and every router in the path is marked as suspicious.

2.4 Detection and Prevention Protocol for Black Hole Attack in Network-on-Chip

Daoud and Rafla [2019] consider that HTs can infect the NoC, and that its routers cannot be trusted. When a router sends a packet, it does not know if it will correctly reach its destination or if the next router will simply drop the packet. The proposed solution consists in carrying an *ack* signal from the current router to the penultimate router in the path.

Figure 2.3 considers the communication between a source router (S) and a target (D) router, passing through four routers (R1 to R4). Data transmission is represented in solid black lines, *ack* signals are shown as dashed lines. The router R1 does not know if the packet successfully reached R3 or if R2 dropped it. The proposed solution is to carry an *ack* signal from R3 to R1. The *ack* signal enables R1 to know if the communication has failed and can request to resend the packet.



> Communication between a source router (S) and target router (D), passing through four routers (R1 to R4). Solid black lines correspond to data transmission, while dashed lines are ack signals.

Figure 2.3 – HT localization method proposed by Daoud and Rafla [2019].

The security mechanism implemented in this work achieved an overhead of 10.83%, 27.78%, and 21.31%, in area, power, and performance, respectively.

This invasive security mechanism must be implemented within the untrusted router, thus also being vulnerable to attacks. The authors mention that the *ack* signal can forged by a malicious router (e.g., the R2 router in Figure 2.3). According to the authors, an authentication method is still needed to assert if the *ack* signal was forged or not. They propose doing so either by using a PNRG to generate keys inside each router, or by implementing an authentication protocol in the PEs firmware. Both options appear to have area and performance overheads, respectively.

2.5 Towards Protected MPSoC Communication for Information Protection Against a Malicious NoC

Sepúlveda et al. [2017] considers that the NoC can be tampered before its integration into the system. The network interfaces, conversely, are considered secure; their role in the network integration requires them to be built in-house. The adopted threat model is that an HT could perform the following attacks: copying, corrupting, and rerouting packets.

To tackle this issue, the proposed solution is to instrumentalize the Network Interface between the PE and the NoC to protect the communication. To ensure confidentiality, an AES-CTR module is used to derive dynamic keys, and packet encryption is performed by XORing packets with this key. Each new packet uses a different key. MACs are created using the SipHash algorithm to assert the packet's integrity.

This work relates to ours by considering that the NoC is susceptible to be infected with HTs and by proposing security mechanisms to tackle this vulnerability. But the mechanism proposed here aims to protect the packet from snooping, corruption and leakage through the usage of data obfuscation and message authentication, leaving DoS scenarios out of scope. Our work aims to develop a localization functionality to find out which NoC router and/or link is infected, thus enabling high-level decision-making to prevent or mitigate the attack, restoring secure system communication.

2.6 Real-Time Detection and Localization of Distributed DoS Attacks in NoC-Based SoCs

Charles et al. [2020] proposes a framework for real-time detection and localization of DoS attacks based on flooding caused by malicious IPs. Even though the authors do not consider the attacks coming from HTs, the localization method based on NoC presented in the paper is relevant to our work.

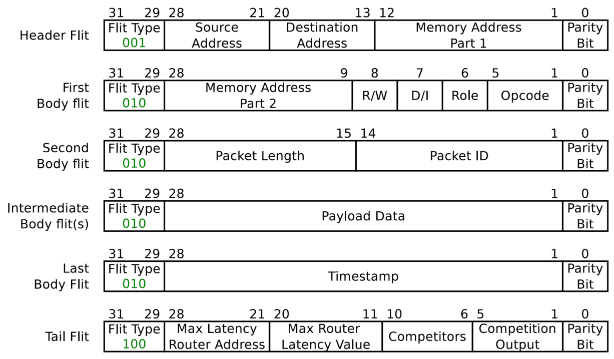
The flooding attack is detected via packet arrival curves (PAC) and destination latency curves (DLC). At design time, communication patterns are gathered from static analysis of the network traffic, generating the PACs and DLCs for each IP. Then, at runtime, IPs monitor the traffic, and when a PAC violation is detected, the IP starts the diagnosis.

First, the destination IP (D) verifies the DLCs to identify abnormal latencies and elects the source of these packets as a Malicious 3PIP (M3PIP) candidate, referred to as S. Then, D sends a *diagnostic message* to S through the routers of the congested path. Each router that receives the diagnostic message analyzes the flows of its ports and sets a flag if the ports are congested. If the next hop of the diagnostic message is congested, the message is forwarded to the next router. Otherwise, the current router is marked as a potential attacker.

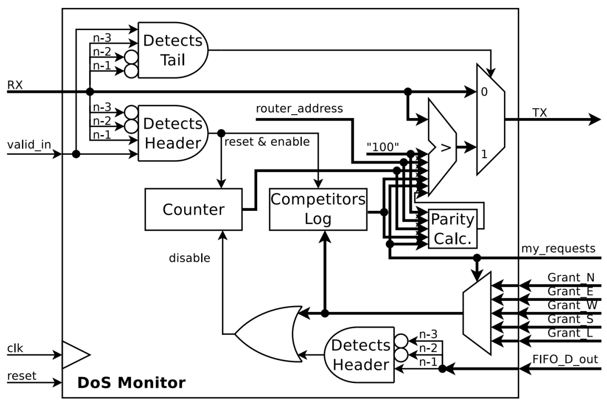
The authors prove that if the congested path contains no loops, their approach can localize at least one attacker. As a result, authors show that all attack scenarios were localized with a router area overhead of less than 6%.

2.7 DoS Attack Detection and Path Collision Localization in NoC-Based MPSoC Architectures

Chaves et al. [2019] also proposes a DoS-flooding attacker localization by path collision. The authors present two approaches: the Collision Point Router Detection (CPRD), that evolves to the Collision Point Direction Detection (CPDD). CPRD consists on equipping the data NoC routers with DoS monitors attached to every router buffer. These monitors receive the packets as well as information regarding output requests and grants. In addition to that, the data packet tail flit now carries the address of the router where the packet waited the most, and the amount of clocks it waited (Figure 2.4(a) *Tail flit* bits 28 - 11). The DoS manager is responsible for evaluating and updating these values in the packet.



(a) CPDD packet structure



(b) CPDD DoS monitor architecture

Figure 2.4 – DoS detection using Collision Point Direction Detection (CPDD). (Source: Chaves et al. [2019].)

Upon receiving the packet, if the end-to-end latency is above the acceptable range, the values are retrieved from the packet and analyzed with previous DoS reports, identifying the PE responsible for the attack.

CPDD extends the CPRD by adding more values to the tail flit of the packet: the inputs competing to enter the sensitive path and the output for which they compete (Figure 2.4(a) *Tail flit* bits 10 - 1). With this mechanism, the firmware is able to narrow down the suspects and localize the malicious PE causing the flooding.

The authors conducted simulations using various attack configurations, varying the packet length, packet injection rate, attack sources, and sensitive path. The results indicate that longer attack packets have a more significant impact. Furthermore, their proposed methods identified the direction of incoming attack flows in nearly all cases. The area overhead of the DoS monitor was 17.7% and 23.2% for CPRD and CPDD, respectively.

2.8 Final Remarks

The papers discussed in this chapter relate to ours by proposing a mechanism for either: localizing an HT inside the NoC [Wang and Halak, 2023; Hussain et al., 2018; Sinha et al., 2021; Daoud and Rafla, 2019]; localizing an attack coming from an IP connected to the NoC [Charles et al., 2020; Chaves et al., 2019]; or protecting the applications from an insecure NoC [Sepúlveda et al., 2017; Bhamidipati and Vemuri, 2024; Hossain et al., 2024]. Table 2.1 categorizes these related works considering the adopted threat model and their proposed solution to protect the system.

Table 2.1 – Classification of works related to HT localization in NoC-based manycores. (Source: the Author.)

#	Work	Attacker location	Attack type	Security mechanism	Detection	Localization	Mechanism location
1	Wang and Halak [2023]	NoC router Network interface	Packet tempering	Tempering detection (ML) Credit table	E2E	Reactive	(Not disclosed)
2	Hussain et al. [2018]	NoC router	Information leakage Flooding (DoS) Performance degradation Packet tempering	Selective activation of H2H units Worm-based algoritm	E2E H2H	Reactive	Network interface NoC links
3	Sinha et al. [2021]	IP block	Flooding (DoS) Performance degradation	Flooding detection (ML) Probe packets	H2H	Reactive	NoC router
4	Daoud and Rafla [2019]	NoC router	Packet dropping (DoS)	Ack signal	H2H	–	NoC router
5	Sepúlveda et al. [2017]	NoC router NoC links	Information leakage Packet tempering	Packet encryption Authentication codes	–	–	Network interface
6	Charles et al. [2020]	IP block	Flooding (DoS)	Diagnosis message	H2H	Reactive	NoC router
7	Chaves et al. [2019]	Application software	Flooding (DoS)	Path collision	E2E	Reactive	NoC router
8	**This work**	**NoC links**	**Information leakage** **Flooding (DoS)** **Packet dropping (DoS)** **Performance degradation**	**Probe packets** **Trust score table**	**E2E** **H2H**	**Reactive** **Preventive**	**OS (software)**

Attacker location: specifies where the malicious component is deployed – e.g., embedded within the routers or links of the NoC or in the software of a malicious application running in a core. Some works [Sinha et al., 2021; Charles et al., 2020; Chaves et al., 2019] do not consider the attacker to be specifically an HT, but their proposed solutions are still applicable for localizing HT attacks.

Attack type: defines which attacks can be performed, such as DoS-flooding, information leakage, or packet tampering.

Security mechanism: overviews the features implemented to protect the system.

Detection: relates to the type of monitoring used to detect the occurrence of an attack: it can be either Hop-to-Hop (H2H) with monitors deployed in every router, or End-to-End (E2E) with monitors at the communication endpoints. Hussain et al. [2018] uses both types of detection, enabling the more precise H2H modules only after the attack is detected by the E2E monitors. Our proposed approach employs an E2E protocol to monitor traffic and an H2H algorithm to localize the HT.

Localization: specifies when the localization mechanism is deployed. *Reactive* localization is triggered by the detection of an attack and aims to find the attacker; *preventive* localization performs exploratory searches for HTs before they interfere with system behavior. Daoud and Rafla [2019] does not propose a localization algorithm; since it uses H2H detection, the malicious router is immediately identified when performing an attack. Sepúlveda et al. [2017] aims to protect the packet's integrity and confidentiality passively, thus, it does not address detection and localization.

Mechanism location defines where the security features are implemented in the system (e.g., integrated into the NoC routers or in the network interface).

Our work stands out from the others in two main ways. Related work adopts reactive methods of localization, in which the system searches for attackers only after detecting their interference in the system's behavior. This restricts the localization scope to the NoC region currently being used by the applications, therefore allowing HTs in the unused regions to remain active and cause problems later on. Our proposal aims for a preventive localization method in which the system proactively searches for HTs throughout the NoC, leveraging unused communication resources to detect HTs before they interfere with system behavior.

The other difference in our work is that we propose a non-invasive localization mechanism. Most of the works in the literature propose adding security mechanisms to the NoC routers or links, which comes with some disadvantages. Sinha et al. [2021]; Daoud and Rafla [2019]; Charles et al. [2020]; Chaves et al. [2019] add security hardware to the NoC router. This can be a good solution for protecting the system against malicious cores, but unsuitable when the threat model is extended to include HTs infecting the NoC itself: by implementing the security hardware inside the untrusted router, the security mechanism cannot be fully trusted. Hussain et al. [2018] proposes to deploy detection units in every NoC link, which results in costly area overhead. Alternatively, our work aims for an HT localization mechanism that does not add security features to the NoC but instead uses probe packets sent by the OS to determine if the NoC has infected routers.

3. BACKGROUND KNOWLEDGE

This Chapter presents the baseline manycore system used in this work, which is a version of the Hermes MultiProcessor System (HeMPS) [Woszezenki, 2007; Carara et al., 2009] with the addition of security and fault-tolerance enhancements [Caimi, 2019; Fochi, 2019; Faccenda, 2024]. The baseline platform and this work were developed at the Hardware Design Support Group research team [GAPH, 2023].

This chapter is divided into four parts. Section 3.1 overviews the baseline platform and its main components. Section 3.2 presents the SNIP: a security feature added to the HeMPS platform to protect communication with peripherals. Section 3.3 examines the consequences of HTs in the NoC and outlines solutions to mitigate them. Finally, Section 3.4 discusses the architectural assumptions used in this work.

This chapter contains contributions that were published. The SNIP security feature covered in Section 3.2 was published in:

Secure Network Interface for Protecting IO Communication in Many-cores
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
In: SBCCI, 2023

Integration of Monitoring Mechanisms in Secure Network Interfaces for Peripherals to Protect IO Communication in NoC-based Many-cores
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
In: Journal of Integrated Circuits and Systems (JICS), 2024

The handling of effects induced by HT attacks in the NoC, covered in Section 3.3, was published in:

Hardware Trojan Localization for Untrusted Network-on-chips
Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
In: LASCAS, 2025

3.1 Architecture of the Baseline Platform

The main HeMPS platform features are:

- NoC-based system: the HERMES NoC [Moraes et al., 2004] allows multiple communications between PEs while ensuring scalability. The NoC adopts 2D-mesh topology, one physical channel, flit width equal to 32 bits, input buffer, credit-based flow control, round-robin arbitration, and XY-routing algorithm.

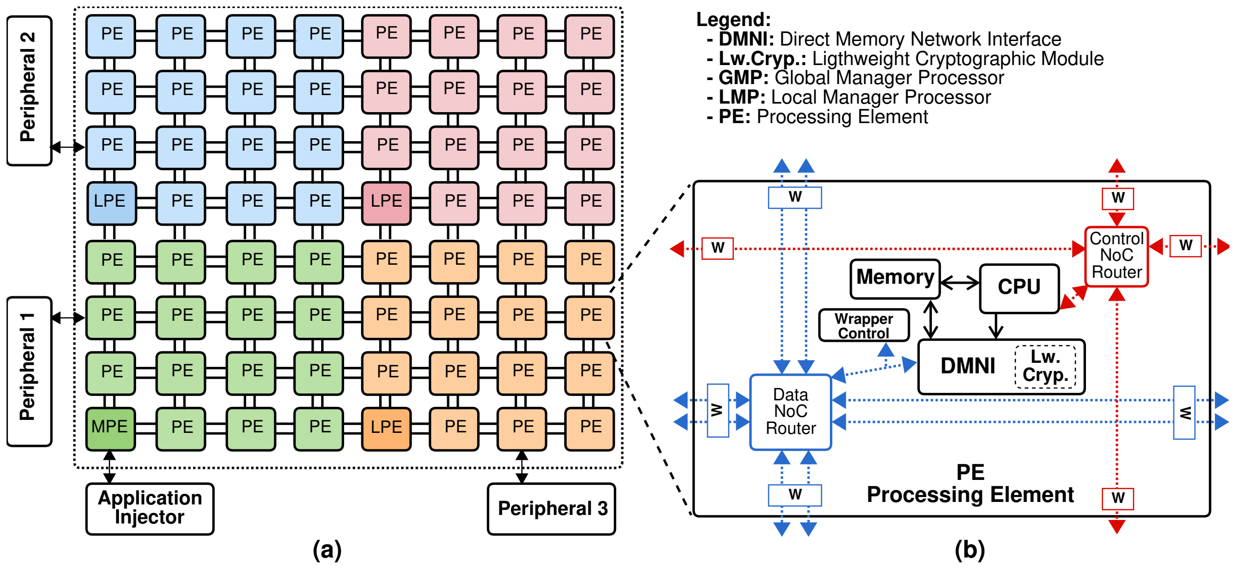
- Homogeneous system: all PEs have the same hardware architecture with a router, private memory, an MIPS-like processor, and a DMNI (Direct Memory Network Interface) module.
- Distributed memory: each PE has a true dual-port scratchpad memory for instructions and data, while message-passing performs the communication between PEs.
- Applications are modeled as a Communication Task Graph (CTG). The CTG is a model to represent functional parallelism, where an application is composed of independent parts and thus is divided into tasks [Rauber and Rünger, 2013]. A graph node represents each task in a CTG, and the graph edges represent the communication between these tasks.

3.1.1 Hardware Model

Figure 3.1 overviews the extended HeMPS manycore, supporting fault tolerance and security mechanisms. In Figure 3.1**(b)**, two mesh NoCs interconnect PEs: *data* and *control* NoC. The *data NoC* is a standard wormhole packet switching NoC without virtual channels. It has two particular architectural features. The first one is the adoption of two physical channels, acting as two disjoint NoCs. To minimize the area overhead, the flit size is 16 bits (half of the word size), and the network interface (DMNI) is responsible for serializing/deserializing the flits. The reason to adopt two physical NoC is to enable fully adaptive routing. The second feature is simultaneous support for XY (default routing algorithm) and source routing (SR). Source routing adopts the turn-based routing model, with the packet header carrying the turns that must be taken in the path. The SR is required when, e.g., it is necessary to avoid a path with a faulty or infected router.

The **data NoC** is a wormhole-switched network that uses a credit-based protocol to send flits between the routers. This protocol uses four signals: *tx*, *data*, *credit*, and *EOP*. The *tx* signal informs the receiver router that the sender is ready to transmit a flit, which is held in the *data* bus. The receiver router uses the *credit* signal to inform the sender if there is enough space in the buffer to accept the flit. If both the *tx* and *credit* signals are raised, the value of *data* is accepted by the receiver router. The last flit of the packet is marked by raising the *EOP* (i.e. end-of-packet) signal. When the first flit of a packet is received, the router executes the routing algorithm and decides in which direction to send the packet. The router remains switched until it sends the EOP flit.

The **control NoC** [Wachter et al., 2017] (also known as *BrNoC*) is a lightweight NoC, with all packets having one flit. When transmitting in broadcast (default transmission mode), packets reach all PEs of the system. Thus, this NoC can find a path from a source to a target PE if it exists, even in the presence of a fault or an HT in the data NoC. This



Link Controls, also known as Wrappers (W), are added to the control signals of NoCs links, allowing to enable/disable ports individually.

Figure 3.1 – NoC-based manycore architecture (Source: [Caimi, 2019]).

NoC may also use the unicast transmission to create a path between a source and a target PE, using a backtracking procedure. For security reasons, only the OS accesses the control NoC, avoiding its use by malicious applications.

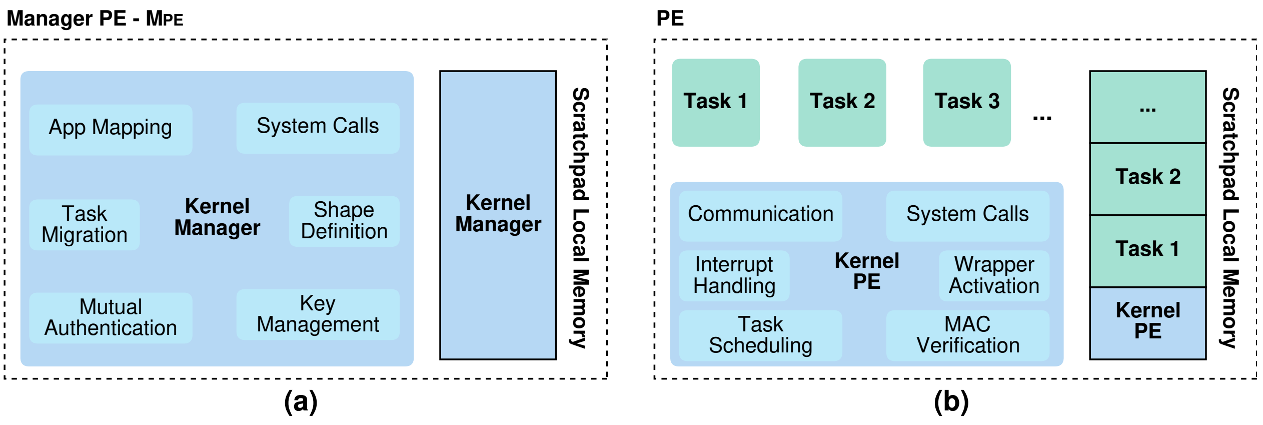
Both NoCs contain Link Controls, or *wrappers*, in the control flow signals. When activated, the wrapper enables the discard of all incoming and outgoing packets of a given port. The data NoC observes and respects the status of the wrappers. A data message arriving in an activated wrapper is always discarded, and the control NoC replies to the source of the message a new broadcast reporting that the message needs retransmission. Wrappers can be used to create secure zones [Caimi, 2019], that are isolated regions of the manycore used to execute applications with security constraints.

The control NoC has two operation modes: *global* and *restrict*. The *global* mode enables the control messages to pass through the wrappers, even if they are enabled. This mode enables the PEs inside a secure zone to exchange messages with manager PEs. The *restrict* mode observes the status of the wrappers, i.e., if a control message hits an activated wrapper, the message is discarded, which is fundamental for searching paths without traversing secure zones.

The platform is modeled at the RTL level, part in SystemC (memory, processor, DMNI) and part in VHDL (data NoC and control NoC routers).

3.1.2 Software Model

Scalability at the hardware level comes from PEs executing several tasks in parallel, using the NoC to transmit multiple flows concurrently. However, large systems require high-level management for controlling the deployment of new applications, monitoring resources usage, manage task mapping and migration, and can execute self-adaptive actions according to systems constraints. The management of HeMPS occurs in the **Manager PE** (MPE), which has a different kernel from the other PEs.



(a) Manager PE kernel controls the system and do not execute users' tasks;
(b) Regular PE kernel manage users' tasks.

Figure 3.2 – Overview of the kernels (Source: [Ruaro et al., 2019; Caimi, 2019]).

At the Manager PE level, the local memory is reserved to the kernel, without executing user's tasks. The Manager PE executes heuristics as task mapping, task migration, monitoring, authentication and key management (Figure 3.2 (a)).

At the regular PE level, a multi-task kernel acts as an Operating System. The platform adopts a paged memory scheme to simplify the kernel design. Examples of actions executed by the kernel include task scheduling, inter-task communication (message passing), interrupt handling (Figure 3.2 (b)).

Both manager kernels are written in C language. Only a small part of the code is written in assembly language, responsible for executing context saving and handling hardware and software interruptions.

Applications are written in C language. They are modeled as task graphs $A =< T, P, D, S >$, where $T = \{t_1, t_2, ..., t_m\}$ is the set of application tasks corresponding to the graph vertices; $P = \{p_1, p_2, ..., p_n\}$ is the set of peripherals corresponding to the graph vertices. The D set represents the application descriptor which contains the communicating pairs $\{(t_i, t_j), (t_i, p_r), (t_j, p_s), ..., (t_m, p_n)\}$ with $(t_i, t_j, ..., t_m) \in$ T, $(p_1, p_2, ..., p_n) \in$ P. A pair (t_i, t_j) denotes the communication from task t_i to task t_j $(t_i \rightarrow t_j)$, and a pair (t_i, p_r) denotes the communication from task t_i to peripheral p_r $(t_i \rightarrow p_r)$. The S value indicates if the applications execute in normal mode (value 0) or secure mode (value 1). Figure 3.3 presents an application following this model.

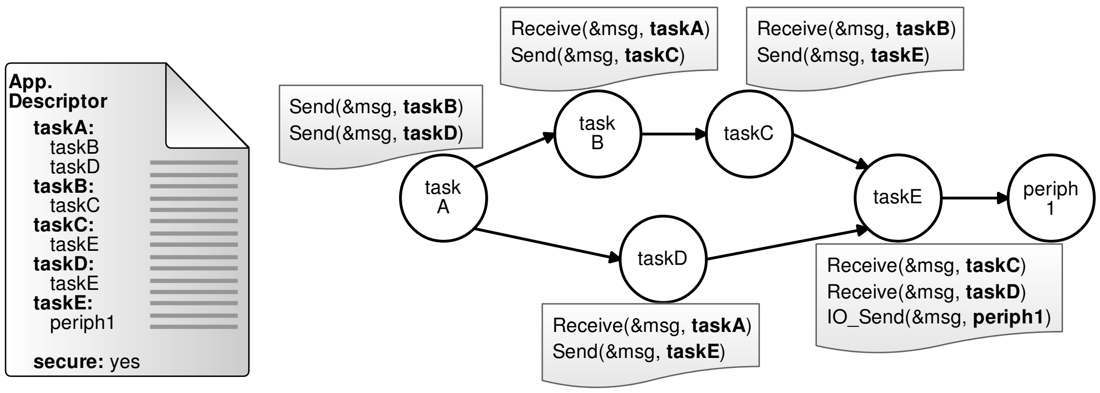


Figure 3.3 – Application task graph example (Source: [Caimi, 2019]).

Tasks communicate using message-passing (MPI-like) primitives. The API provides two primitives: a non-blocking *Send*() and blocking *Receive*(). The main advantage of this approach is that a message is only injected into the NoC if the receiver requests data, reducing network congestion. To implement a non-blocking *Send*(), a dedicated memory space in the kernel, named *pipe* [Carara et al., 2009], stores each message written by tasks. Within this work, the pipe is a kernel memory area reserved for message exchanging, where messages are stored in an ordered fashion and consumed according to it. Each pipe slot contains information about the target/source processor, task identification and the order in which it is produced.

At the lower level, the kernel communicates with the data NoC with *data_request* and *data_delivery* packets. The *pipe* and a message buffer enable packet retransmission to inter-task communication and inter-manager communication, respectively.

3.2 Secure Network Interface for Peripherals (SNIP)

The enhancement of the HeMPS system focused on establishing a secure environment for executing critical applications, requiring the development of multiple security features. One of these enhancements is the Secure Network Interface for Peripherals (SNIP), which protects communication with IO devices, integrating security mechanisms to safeguard communication with internal components in manycores. The SNIP was partly developed during the period of this MSc, and although it is not related to the premise of hardware HT localization, *it is one of the contributions of this dissertation.*

Figure 3.4 presents the IO communication model adopted in the platform. This picture highlights five important elements of the IO communication, which are described below.

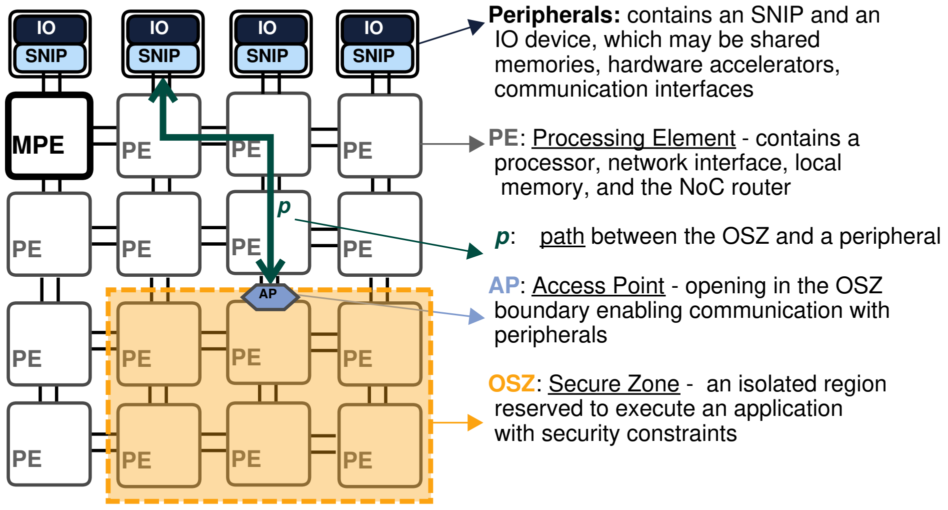

IO
SNIP
MPE
PE
p
AP
Peripherals: contains an SNIP and an IO device, which may be shared memories, hardware accelerators, communication interfaces
PE: Processing Element - contains a processor, network interface, local memory, and the NoC router
p: path between the OSZ and a peripheral
AP: Access Point - opening in the OSZ boundary enabling communication with peripherals
OSZ: Secure Zone - an isolated region reserved to execute an application with security constraints

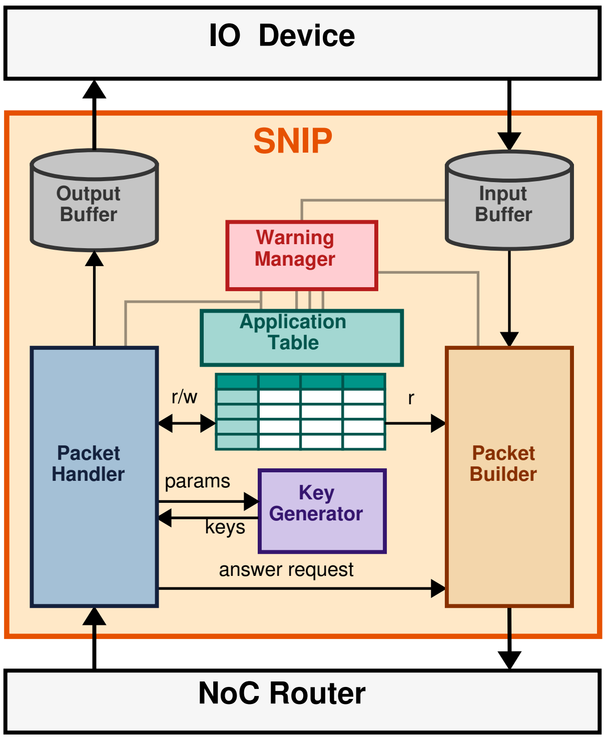
Figure 3.4 – IO communication model adopted in the HeMPS baseline platform (Adapted from [Comarú et al., 2023]).

- **Opaque Secure Zone** (OSZ [Caimi and Moraes, 2019]) – an isolated region of the manycore that executes an application with security requirements, blocking the traffic from other applications. The spatial isolation prevents attacks from other flows or tasks.
- **Access Point** (AP [Faccenda et al., 2023b]) – opening in an OSZ border that controls the entry and exit of packets. Is the only point the application can use to communicate with the exterior of the OSZ.
- **Path** *p* – the path between the AP and the SNIP. The path is defined by source routing (SR).
- **Secure Network Interface for Peripherals** (SNIP) - to secure the communication between the IO device and the application with security requirements. The SNIP will be further discussed in this section.
- **System Manager PE** (MPE) - PE reserved to execute management operations, such as application allocation and mapping, PEs and SNIPs configuration.

The packet exchange between IO devices and applications is based on the host-device model, where the PE acts as the host and the SNIPs function as the devices. Thus, communication is always initiated by the host (PE), and the device (SNIP) must send a response packet to confirm the operation.

To receive data from an IO device, the task sends an IO_READ packet, and waits for an IO_DELIVERY packet with the requested data. To send data to the IO device, the task transmits an IO_WRITE packet and waits for an IO_ACK to confirm the operation's success.

The SNIP protects the system from spoofing, DoS and misrouting attacks by implementing three main security mechanisms: (*i*) authentication; (*ii*) packet discard; (*iii*) warning generation.



Signals used by the Warning Manager (in red) are omitted for the sake of simplicity.

Figure 3.5 – SNIP architecture and interfaces (Adapted from [Comarú et al., 2023]).

The SNIP employs an **IO Authentication** protocol, presented on [Faccenda et al., 2023a], to enforce authentication and authorization principles. The protocol begins with the MPE sending a command to the SNIP (IO_CONFIG), specifying the applications authorized to interact with the IO device. Additionally, the SNIP must verify packet authenticity, send packets with the correct authentication fields, and perform key derivation. An important feature of this protocol is that the SNIP only communicates with authorized applications using a fixed source-routing path set by the MPE. As a result, the IO device is prevented from sending messages to unauthorized applications or using forged paths.

The **Packet Discard** mechanism is a countermeasure that focuses on quickly rejecting and eliminating packets that fail authentication, removing them from the NoC, and reinforcing the principle of availability.

Both aforementioned countermeasures are applied immediately upon detecting malicious actions to neutralize threats without delay. However, due to this automatic response, the MPE remains unaware of suspicious behavior on the SNIPs. Therefore, the third key countermeasure of the SNIP is **Warning Generation**, which notifies the MPE whenever a security anomaly is detected at the SNIP.

The SNIP has seven main modules, as illustrated in Figure 3.5: (*i*-*ii*) Packet Handler and Packet Builder, enable simultaneous communication to and from the NoC; (*iii*) Application Table (`ApT`), stores sensitive data to enable communication with the applications; (*iv*-*v*) FIFO buffers hold data sent to or received from the IO device until consumption; (*vi*) Key

Generator produces and updates authentication keys; (*vii*) Warning Manager detects suspicious behavior and sends packets to the MPE. Each of these components is discussed below, except the buffers.

Packet Handler: The SNIP acts as a slave to the system since it waits for incoming packets to define its action. The Packet Handler is responsible for receiving packets from the NoC and carrying out the appropriate response. It executes all the decision-making, acting as a manager to the other components. Upon receiving a packet, the Packet Handler analyzes its service code, which refers to the function of the packet. Table 3.1 displays the services the SNIP supports. The SNIP discards any received packet whose *appID* is not in the Application Table.

Table 3.1 – Services supported by the SNIP (Source: Comarú et al. [2023]).

Service code	**Packet Source**	**Function**
IO_INIT	Manager PE	Packet received at system startup with the initialization key – $k0$
IO_CONFIG	Manager PE	Configure a line of the Application Table with $\{appID, path, k1, k2, status\}$
IO_RENEW	Manager PE	Renew the *appID* keys $\{k1, k2\}$ receiving parameters $\{n, p\}$
UNBLOCK_ WARNINGS	Manager PE	Enable the SNIP to send warning packets
IO_CLEAR	Manager PE	Clear and deallocate the Application Table row indexed by *appID*
IO_WRITE	Application	Write data into an IO device Application waits an IO_ACK from SNIP
IO_READ	Application	Request data from an IO device Application waits an IO_DELIVER packet

Application Table: The SNIP uses the Application Table (`ApT`) to allow authorized applications to access the IO device connected to the SNIP. Each line of the `ApT` has the following fields: ***appID***: application identifier; ***path***: path between the SNIP and the application AP; ***k1*** and ***k2***: authentication keys, used to certify the authenticity of packets; ***status***: it may assume *free*, *pending*, and *used* values. Note that the `ApT` authenticates applications and not tasks. This “application granularity” reduces the `ApT` size and thus silicon area compared to a table with “task granularity”. The `ApT` has two interfaces, enabling the SNIP to send and receive packets simultaneously. The primary interface (read-write) is connected to the Packet Handler, and the secondary interface (read-only) is connected to the Packet Builder.

Key Generator: The Key Generator is responsible for creating and updating the keys used in the Authentication Protocol. It generates two keys, $\{k1, k2\}$, using a Linear-Feedback Shift Register (LFSR), which acts as a pseudo-random key generator. The key size is a design time parameter, in our case we use 16-bit keys. While LFSRs are not the most robust method for generating pseudo-random numbers, they offer a distributed and area-efficient solution for generating authentication keys. For the IO_CONFIG service, the LFSR uses *appID* as the seed, producing $k1$ after n rounds and $k2$ after an additional p rounds. For the IO_RENEW service, new keys are generated using $k2$ as the seed, following the same procedure. The generated $\{k1, k2\}$ keys are stored in the ApT row indexed by *appID*, with n and p being randomly generated for each IO_CONFIG and IO_RENEW service.

Packet Builder: The Packet Builder assembles and sends packets to the applications. These packets can be either *IO_DELIVERY* messages with the data requested from the IO device, or *IO_ACK* to acknowledge data from the application. Once the Packet Handler receives a valid packet from an application, it uses the *Answer Request* (Figure 3.5) interface to notify the Packet Builder to send an answer. The parameters specifying the packet to be sent are *appID*, *messageType*, and *requestSize*. Upon receiving a request, the Packet Builder registers the parameters, raises a busy signal, and generates the packet. The information required to build the packet header, such as the authentication keys and the source-routing path, is retrieved from the ApT through the secondary interface. If the outgoing packet is an *IO_DELIVERY*, data sent by the IO device is retrieved from the Input Buffer and sent in the packet payload. Since only one request can be handled at a time, if another request needs to be issued while the Packet Builder is busy, the Packet Handler stays blocked until the completion of the current request.

Warning Manager: The Warning Manager module detects suspicious behavior and generates warning packets to send to the MPE. The SNIP components alert the Warning Manager through specific warning signals when irregularities occur. The Warning Manager collects relevant data via the *Warning Parameters* interface and issues a *Warning Request* to the Packet Builder, which assembles and sends the warning packet into the NoC. The SNIP issues four types of warnings: (i) *Failed authentication*, triggered when an incoming packet fails authentication; (ii) *Write on a full table*, indicating that the SNIP received an *IO_CONFIG* request, but the table has no available space; (iii) *Row overwrite*, reporting that a slot in the SNIP table for an authenticated application was replaced; and (iv) *Abnormal peripheral*, signaling that the peripheral is not adhering to the correct communication protocol.

The proposed SNIP design addresses the security challenges in manycores, specifically protecting the communication with IO devices. The SNIP integrates security mechanisms that safeguard communication between internal components in manycores and bridges a research gap regarding the communication between manycores and peripherals.

3.3 Effects Induced by HTs in the NoC

While working with HT-infected NoCs, we observed that simple faults in the packet transmission could cause significant issues to the overall system, such as completely blocking the NoC. For instance, if the value of a control signal is modified (e.g., by an HT), it may cause flits to become permanently stuck in the router, leading to congestion and potentially blocking the manycore. Although the literature acknowledges the occurrence of attacks and faults, there is no solution for their side effects on the NoC. Thus, to continue our work, we first had to tackle the problem of preparing the system to handle “faulty packets” generated by attacks originating from HTs.

Faulty packets are created when an anomaly interferes with the packet transmission protocol, altering the structure of the packet or its transmission flow. We identified four different types of faulty packets that negatively affect the system, as presented in Table 3.2 and described below:

Packet stuck in router: a *Credit Block HT* acts on the control flow signals of the NoC links. This HT type forces the *credit_in* to ‘0’ even if the receiver buffer has space to accept new flits. As a result, the router cannot use this port to receive incoming flits, and packets arriving through this infected link have to wait until the HT becomes disabled to finally be transmitted. Non-transmitted packets occupy the NoC buffers, causing congestion in the NoC.

Packet without tail: a packet traversing the NoC without the EOP (end-of-packet) signaling. This may occur if: (1) an HT drops the tail of the packet; (2) the packet is blocked during its transmission by an HT (by a *Credit Block HT*); (3) an incorrect packet is injected into the network. As a result, all links between the HT and the packet’s target are switched (i.e., connecting an input port to an output port), preventing other flows from using the path defined by the data flow. We call this effect as ***residual switching***. Furthermore, at the target PE receiving the packet, the NI will continue to wait for an EOP-marked flit and cannot receive other packets, blocking the PE.

Packet without header: the first flits of a packet contain the information to execute the routing algorithm. When a faulty packet loses its header, the routing algorithm is executed using the wrong flits and routed incorrectly. The packet may either travel to the border of the manycore and be dropped or reach an unpredictable PE and be handled by an NI as if it were a regular packet.

Packet without header and tail: a packet traversing the network without a correct header and without an EOP flit. This may happen, for instance, if an intermittent *Black Hole HT* drops the beginning and the end of the packet, but leaves the middle part untouched.

This causes the packet to be incorrectly routed throughout the network, leaving behind residual switching. This is especially harmful since the path taken by the packet is unknown, and the affected routers cannot be identified.

Table 3.2 – Effects of faulty packets on the system and the proposed solutions (Source: Comarú et al. [2025]).

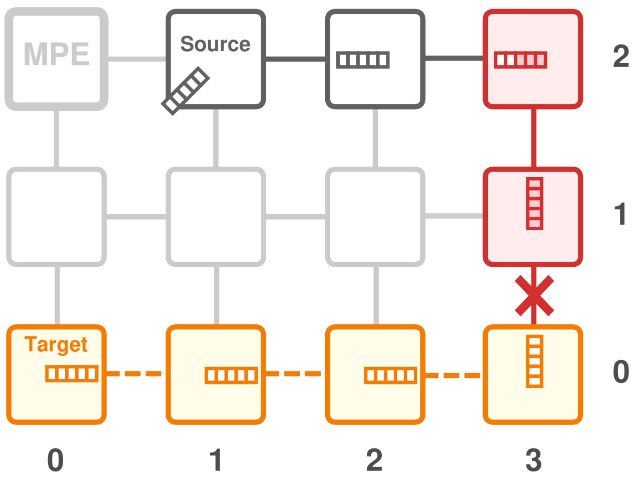
Faulty-packet issue	Cause	Effected location	Effect	Proposed solution
Packet stuck in router	–Credit Block HT	–NoC router	–Residual switching and congestion	–Router Reset
Packet without tail	–Black Hole HT –Credit Block HT –Packet Injector HT –Router Reset	–NoC router –Network interface	–Residual switching and congestion –Target NI waits for non-existing packet tail	–Router Reset –Reception Timeout
Packet without header	–Black Hole HT –Credit Block HT + Reception Timeout –Packet Injector HT –Flooding HT –Router Reset	–Network interface	–Packet is routed to incorrect target	–BOP Signal
Packet without header and tail	–Intermittent Black Hole HT –Intermittent Credit Block HT + Reception Timeout –Packet Injector HT –Flooding HT –Router Reset	– NoC router	–Packet is routed to incorrect target –Packet leaves behind residual switching	–BOP Signal

Figure 3.6 exemplifies how an HT attack can cause faulty packets. The Figure shows the *Source* sending a packet to the *Target* using the highlighted path (the figure uses source routing). The *Credit Block HT* (red X) is activated during the packet transmission, effectively breaking the packet into two parts and causing the following issues:

- ***Packet stuck in router***: the tail of the packet cannot be transmitted and occupies the buffers of the network, causing congestion (red routers).
- ***Packet without tail***: the beginning of the packet is successfully transmitted to *Target*, but the absence of the EOP leaves behind a set of switched routers (yellow routers), and other packets cannot use the same path. The Target NI does not receive an EOP and gets stuck waiting for the second half of the packet.

Table 3.2 presents the solution proposed to protect the platform from faulty packets. We implemented three mechanisms, described below:

Reception Timeout: to avoid the problem in which the NI gets stuck receiving a packet that will never arrive, we implemented a timeout counter in the reception port of the NI.



A Credit Block HT (*red X*) interferes with the packet transmission protocol causing faulty packet issues in the platform. The path between Source and Target is highlighted, and the picture presents the input buffers for each link in the path. *Gray routers* have already transmitted the packet and cleared the switching; they are now in their default state, waiting to transmit a new packet. *Red routers* contain the tail of the packet: the Credit Block HT prevents them from forwarding the last flits of the packet, causing congestion. *Yellow routers* have already delivered the packet to Target, but the absence of the EOP signal causes them to remain switched.

Figure 3.6 – Example of faulty packets caused by a Credit Block HT (Source: the Author).

When a packet flit is received, the counter is set to zero. But the counter increments every clock cycle if the NoC router stops sending new flits (i.e., the *credit* signal goes to ’0’). If the counter reaches a predefined threshold, a timeout occurs. The FSMs responsible for receiving the packet are reset, aborting the reception of the faulty packet and releasing the NI to receive other packets. In the wormhole switching, once flits start arriving at the NI, they arrive continuously. Thus, this threshold is low, set to 30 clock cycles.

Router Reset: this mechanism resets a specific router port, clearing the residual switching and flushing the flits in the buffer. When a faulty packet is detected (e.g., by the Reception Timeout), the MPE uses the control NoC to send a RESET_ROUTER_PORT packet to every router in the faulty packet path. The payload of this packet specifies which port of the data-NoC router should be reset. When the control-NoC router receives this packet, it sends a control signal to the data-NoC router, resetting the specified buffer and clearing the port.

BOP Signal: prevents packets without header to navigate through the NoC. A new BOP (i.e., beginning-of-packet) signal was created to mark the start of new packets, akin to the EOP signal. The packet is dropped if a router or NI receives a new packet that does not begin with a BOP flit. SoCIN is an example of NoC using BOP signal [Zefferino, 2003].

3.4 Architectural Assumptions for Data NoC

This section formalizes the architectural assumptions adopted for the data NoC. For our proposal to be applicable, the data NoC must implement the following requirements:

Packet signaling: the target NoC must have signals indicating the beginning and the end of the packets, i.e., beginning-of-packet (BOP) and end-of-packet (EOP) signals. Both signals are necessary due to the HT effects described in Section 3.3. For example, a packet sent through the NoC can be cropped by an HT during transmission, the BOP and EOP signals, allow to recognize and drop cut-in-half packets without impairing the network.

Router reset: the target NoC must have a functionality that allows resetting the routers' buffers. Due to HT effects discussed in Section 3.3, a packet can become indefinitely stuck in the NoC, occupying routers' buffers and propagating congestion. The buffer reset feature allows the flits stuck in each router to be flushed, freeing the network to send other packets.

Source routing: the target NoC must support source routing (SR). This routing algorithm allows sending packets using specific paths to circumvent suspicious routers. The SR feature is also necessary because our proposal for localizing HTs is based on sending probe packets to test routes of the NoC. This method is further detailed in Section 6.1.

At the beginning of the text, we mentioned that "*this work aims to build a non-invasive solution for HT localization*". That means a method that localizes HTs without adding hardware to the data NoC. Note that the necessary features listed in this section are not security mechanisms but minimum requirements for our proposal to be applicable. The **packet signaling** and **router reset** are necessary for the data NoC to remain minimally functional during faults and HT attacks, while the **source routing** is a commonly implemented routing algorithm.

Works presented in the literature often embed core security features inside the NoC router, such as communication monitors, key generation, and even machine learning models. On the other hand, our work builds on these three requirements presented above to propose a noninvasive localization solution for the data NoC model.

4. THREAT MODEL AND SECURITY FLOW

This chapter details the threat model used throughout this dissertation and the security flow. This chapter is divided into four sections. Section 4.1 introduces the HT model we used in this work, explaining different attacks and activation mechanisms. Section 4.2 presents the HT Insertion Framework that was developed to allow seamless integration of HTs into RTL simulations. Section 4.3 defines the actual threat model adopted for this work. And, finally, Section 4.4 gives an overview of the security flow we propose to tackle the threat model .

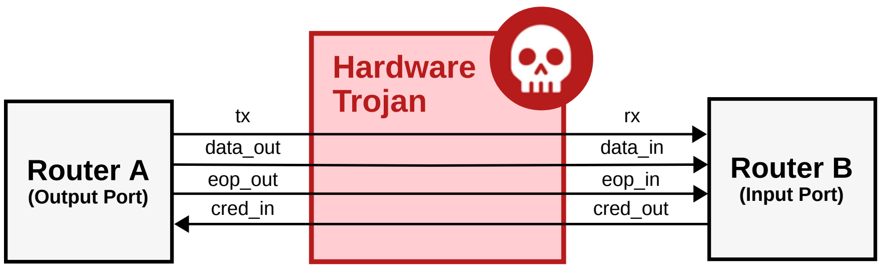
Part of this chapter, comprising Section 4.1 and Section 4.2, was published in the following conference paper:

> **Hardware Trojan Localization for Untrusted Network-on-chips**
> Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
> In: LASCAS, 2025

4.1 HT Model

This section presents the developed HT models and their corresponding activation methods. These HTs are justified based on the prior study of HT types, with a detailed taxonomy provided in Appendix A.

We model HTs as discrete hardware blocks that can be inserted into each NoC link, placed between the transmission port of a sender router and the reception port of a receiver router (Figure 4.1). Most related works assume that HTs are integrated within routers, an assumption that poses significant challenges due to the requirement for detailed knowledge of the router's internal logic. In contrast, our work adopts a distinct and non-invasive approach by inserting HTs into the NoC links. This method eliminates the need to modify the router hardware to accommodate HTs.



> HTs are inserted in NoC links and can access all link signals.

Figure 4.1 – Adopted model for HTs (Source: the Author).

Designing the HT as an external entity to the router facilitates its construction as an individual module. This approach targets minimal area and power overhead, making the HT more efficient and less intrusive in the system's overall architecture.

The HT module can access any control signal available in the NoC link, as illustrated in Figure 4.1. These signals may interfere with the credit-based flow control protocol of the Hermes NoC (Section 3.1.1).

Each HT comprises the attack payload and an activation mechanism (trigger) [Tehranipoor et al., 2023]. The ***attack payload*** implements the attack itself; it modifies the link signals to perform an attack. The ***activation mechanism*** decides when the payload is activated. These parts are implemented separately and can be combined to create different HTs. We implemented four attack payloads and three activation mechanisms.

The following ***attack payload*** were implemented:

Black Hole HT: the HT drops any packet that tries to traverse the infected link, acting as a sink for packets. It is implemented by forcing the control signal *tx* to '0'. The sender router will send the packet as normal, i.e., with its *tx=1*, but from the receiver's perspective, no flits are being transmitted, thus effectively dropping the packet.

Credit Block HT: simulates congestion in the NoC. It forces the control signal *credit_in* to '0'. When the sender router tries to forward a packet to the receiver, it understands there is no space left in the receiver to accept the packet, so the sender will hold the packet flits in its buffer, causing congestion and violations of QoS and real-time constraints. Once the HT is disabled, the packet is forwarded and continues its transmission.

Packet Injector HT: sends forged packets through the NoC. The HT can inject a forged packet in the NoC by assuming complete control over the link. This HT uses a counter to iterate over the packet's format: with each value, the HT injects the correspondent flit into the receiver's port. Then, the injected packet proceeds to be routed as a regular packet, possibly arriving at a PE and being handled by the OS. This implementation uses hard-coded values for the packet fields. Also, it is important to note that a packet injection attack can only be successfully performed if the attacker knows the packet format used by the NoC.

Flooding HT: injects a flow of flits into the NoC, causing congestion. It is implemented by forcing the *tx* signal to '1'. At each clock cycle, the receiver router assumes the transmission of a new flit from the sender. The received invalid flits are forwarded, flooding the NoC.

It is important to mention that, although all these attacks were implemented, our HT localization proposal will only consider Black Hole and Credit Block attacks. The Packet

Injector and Flooding HTs are out of scope of this work. Section 4.3 will provide more details about the threat model adopted for this work.

The following list presents the ***activation mechanisms*** we implemented to be combined with the attack payloads.

Always-on activation: it is the simplest activation method – the HT is always activated. This naive implementation makes finding the HT location easier than other activation techniques.

Time-triggered (static) activation: defines a time window for the HT activation. The HT has an internal counter that increments with each clock cycle. When the counter reaches a predetermined start time, the attack begins. The counter continues to increment until it reaches a stop time to be deactivated, stopping the attack.

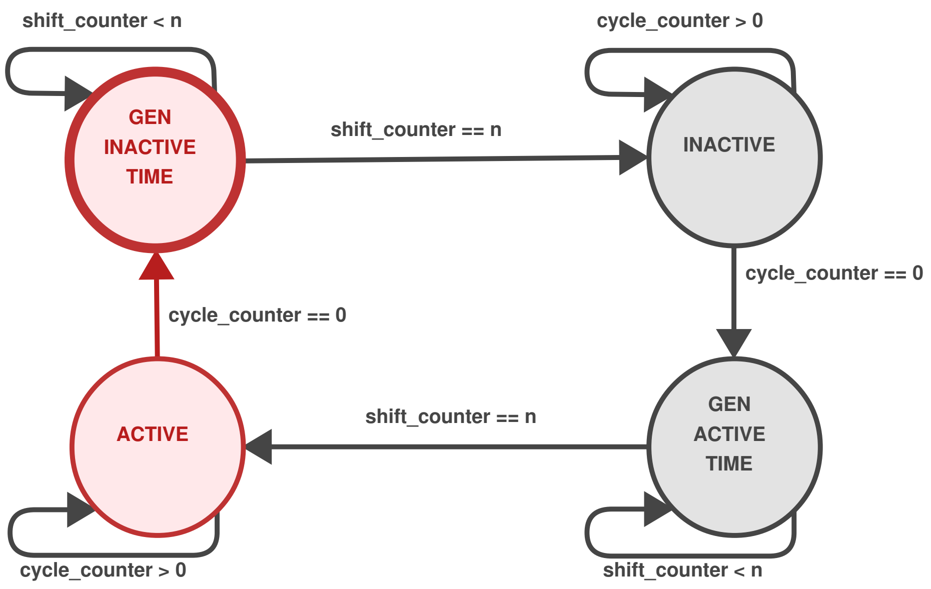
Time-triggered (intermittent) activation: activates and deactivates the HT in random time intervals. This kind of HT attacks the system in an unpredictable way, which makes it harder to locate. The intermittent HT is implemented using an FSM (Figure 4.2) and an LFSR. Although the LFSR does not generate true random numbers, it is an area-efficient way to generate a random-like sequence of numbers sufficient for making the HT hard to detect. Each time the HT needs a new random number, it shifts the LFSR *n* times. The resulting value is then masked to ensure that the activation/deactivation times are within a predetermined range. Both *n* and the range of random numbers are fixed and defined at design time.

4.2 HT Insertion Framework

The first step in developing a defense mechanism is to define a flexible environment to execute attacks. We implemented a framework that enables the placement of HTs on different NoC links. The goal of this framework is to allow the simulation of several HT attacks and then propose, test, and evaluate localization mechanisms.

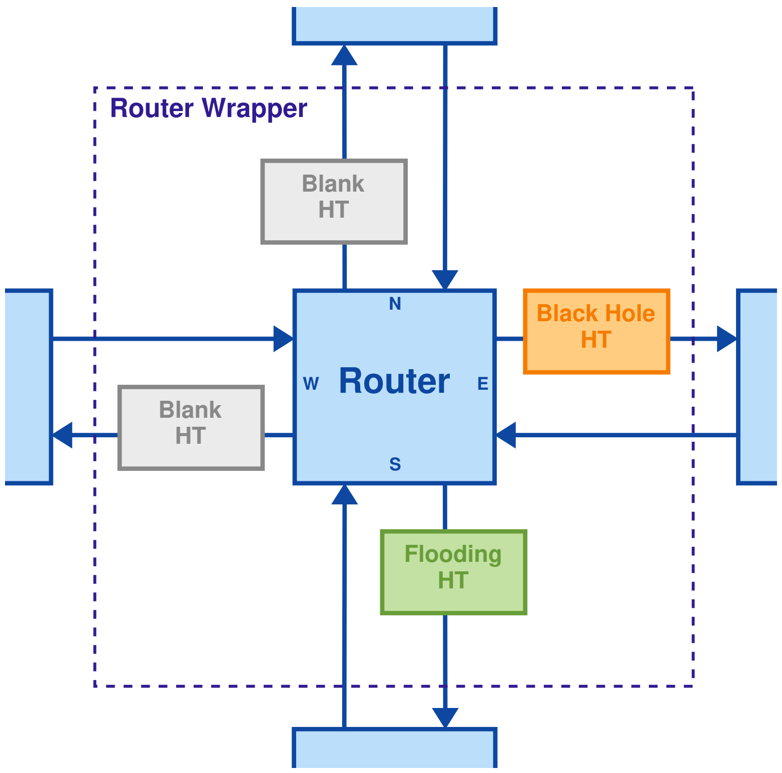
The router is placed within a wrapper with HT circuits into the NoC links, as depicted in Figure 4.3. Within this wrapper, the signals from each outgoing link pass through an HT block before proceeding to the subsequent router. The primary concept here is that the HT blocks are initially inactive, essentially acting as placeholders. They are designed for future injection of active HTs, allowing easy implementation without modifying the router.

The Router Wrapper is described in VHDL, and each outgoing link instantiates an HT block entity. There are different architectures for the HT block entity, such as Blank HT, Black Hole HT, and Flooding HT. By default, the HT blocks are instantiated as Blank HTs,



The HT is activated and deactivated at unpredictable time intervals. The states where the HT is active are shown in red, whereas those where the HT is inactive are in gray. During the ***Gen Active/Inactive Time*** states the LFSR register is shifted *n* times (*n* is defined at design-time) and generates a pseudo random-number. During the ***Active/Inactive*** states, the value on the register is decremented each clock cycle until it reaches zero. The process is cyclic, and the attack occurs at different intervals and durations.

Figure 4.2 – FSM that implements the intermittent activation mechanisms for HTs (Source: the Author).



The Figure shows a router with four physical ports. Ports E and N contain Blank HTs, which are harmless. Ports W and S instantiate HTs for Black Hole and Flooding attacks, respectively.

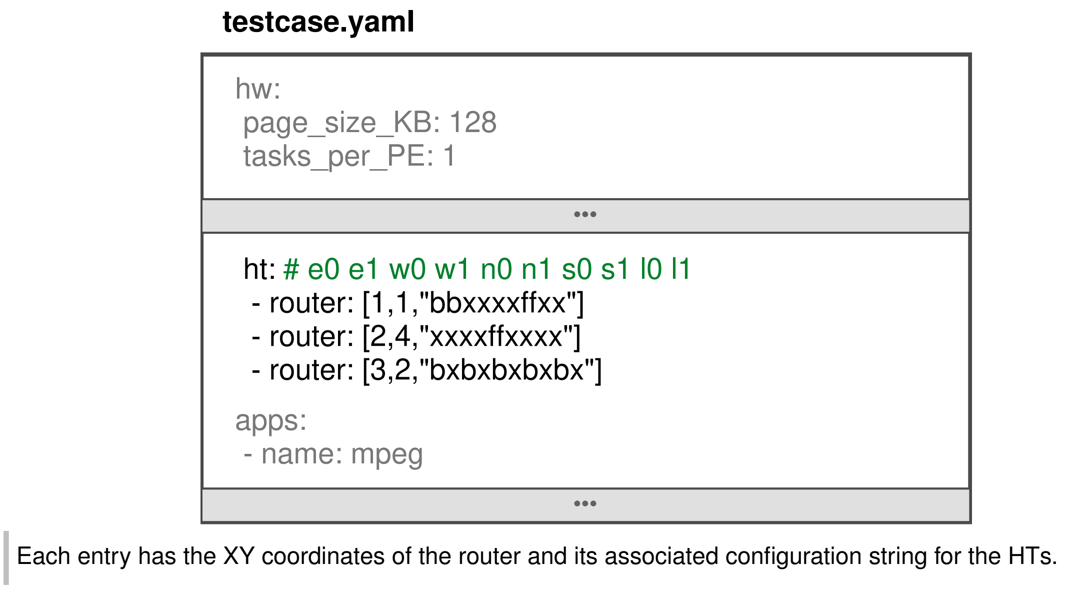
Figure 4.3 – Insertion of HTs in the NoC links (Source: the Author).

which are wires connecting the input signals to the outputs, without any malicious hardware in the link. Figure 4.3 shows the example of a wrapped router with four physical ports and each output link with an HT block (2 of which are blank and 2 implementing different attacks).

To place HTs into the system, it is necessary to define which HT block each NoC link has to instantiate. This information comes from a list provided in the test case descriptor, a YAML file specifying the platform parameters for a given simulation. This file contains information such as system dimensions, memory size, location of instantiated peripherals, and so forth. It now also specifies which links contain HTs and of which type.

Comprehensively listing each HT in the system offers several advantages. It enables precise experimentation by allowing the selection of specific links and HT implementations tailored for each scenario. This approach also retains the flexibility for broader simulations: for example, a script could generate a list of HTs distributed randomly throughout the system. A key benefit in both cases is the deterministic and repeatable nature of the experiments. Once the HT list is established, it can be used for multiple simulations. Moreover, this list can be manually fine-tuned, enhancing the precision and control over the experimental setup.

Figure 4.4 shows an example of how the HTs are listed in the YAML descriptor. Under the HT grouping, each line corresponds to a different router. The first two values provided are the XY coordinates of the router, and the third value is the configuration string for inserting HTs into its links.


testcase.yaml

```
hw:
 page_size_KB: 128
 tasks_per_PE: 1
...
 ht: # e0 e1 w0 w1 n0 n1 s0 s1 l0 l1
  - router: [1,1,"bbxxxxffxx"]
  - router: [2,4,"xxxxffxxxx"]
  - router: [3,2,"bxbxbxbxbx"]
apps:
 - name: mpeg
...
```

Each entry has the XY coordinates of the router and its associated configuration string for the HTs.

Figure 4.4 – Example of HT listing in the YAML descriptor (Source: the Author).

Each position in the string corresponds to a router link, and its character defines which implementation of the HT Block is instantiated in that link. The NoC has two physical channels, so there are 10 links per router: east, west, north, south, and local for both physical planes. The order of the links represented in the string are E0, E1, W0, W1, N0, N1, S0, S1, L0, and L1. The characters ’b’ and ’f’ denote the Black Hole and Flooding HT, respectively, while ’x’ denotes a Blank HT. Every router left out of the listing instantiates only Blank HTs.

Figure 4.4 provides three examples of configurations. The first router (1x1) is configured in the same way as in Figure 4.3, with Black Hole Trojans in the east links and Flooding Trojans in the south. The second router (2x4) is infected with Flooding Trojans in the north links. The last router (3x2) has its primary physical plane entirely covered by Black Hole Trojans.

To build the system for simulation, a Python script reads the YAML test case descriptor and generates the hardware files according to the specification. It builds an array with the HT configuration string for each router during this process. The script initializes the array with only Blank HTs for every router and then iterates over the list provided in the YAML file, updating the entered values. The signals from this array are propagated from the top file to the router wrappers, where the string characters are used to select the HT block architecture of each link.

This section presented one of the contributions of this dissertation, corresponding to the HT Insertion Framework. The modifications integrated into the system resulted in an automated framework for inserting HTs in the NoC. The proposed framework is agnostic to the NoC and can thus be configured to work with other platforms. Coupled with the modular design of our HT model, which enables the seamless integration of new HT implementations, this framework sets the foundation for conducting extensive attack campaigns on the NoC. Additionally, it enables the exploration and development of new security mechanisms, thereby contributing to the field of security in manycore systems.

4.3 Threat Model

This section details the threats we considered for this dissertation. We separate the components of our platform into two categories: trusted and vulnerable. The ***trusted*** components are considered to always work correctly, whereas the ***vulnerable*** components may be compromised by the attacker and can behave maliciously. The premise of this work is that all manycore components are trusted, with only the ***data NoC*** being vulnerable to HT attacks.

The ***data NoC*** is an attractive target for attackers aiming to insert HTs. Given its centralized role within the manycore, the NoC interconnects critical components, such as PEs and IO devices. Consequently, its extensive interactions result in a critical attack surface, making it highly susceptible to exploitation by malicious actors. Furthermore, as the NoC is often incorporated into the system as a 3PIP, we cannot fully access its implementation. Thus, we cannot completely trust that it does not contain malicious functions embedded in its circuitry.

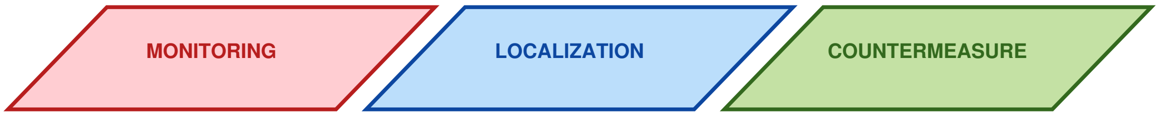
We consider the ***control NoC*** a trusted component. The control NoC is a simple network that works by broadcasting 1-flit packets, which transmit control information. The control NoC is also meant to be implemented in-house and not acquired as an IP from a third party.

In this work, we consider attacks from ***Black Hole*** or ***Credit Block*** HTs. These HTs can be triggered by any of the activation mechanisms aforementioned: ***Always-on***, ***Static*** or ***Intermittent***. These HTs follow the model explained in Section 4.1 and are positioned in the NoC links. For the scope of this work, we are also considering the NoC having a single plane.

Packet Injector and Flooding HTs are out of the scope of this Dissertation.

4.4 Security Flow

We propose the security flow presented in Figure 4.5 to tackle the threat model. This flow contains three phases: *Monitoring*, *Localization*, and *Countermeasure*. This flow aims to protect the manycore by finding and neutralizing the HTs infecting the data NoC.



The security flow comprises three phases: monitoring, localization, and countermeasure. The ***Monitoring*** phase contains a protocol to monitor the exchange of packets through the NoC and detects the occurrence of HT attacks. The ***Localization*** phase implements an algorithm to send probe packets through the NoC selectively and is responsible for finding the location of the HTs. The ***Countermeasure*** phase acts on the system to neutralize or mitigate the effects of the HT on the platform.

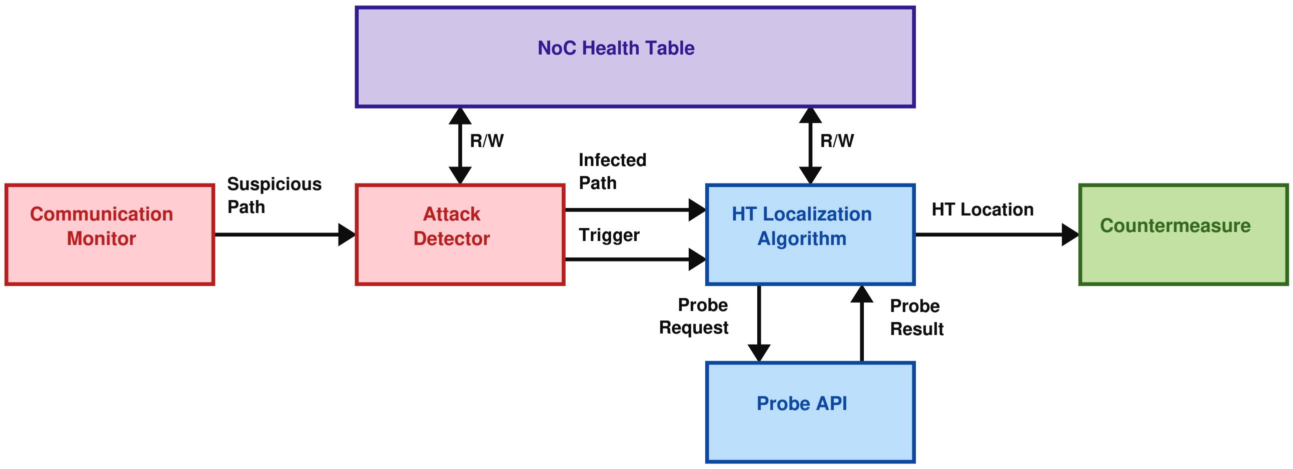
Figure 4.5 – Security flow proposed to protect the manycore from HTs (Source: the Author).

Monitoring: the first phase of the security flow starts by monitoring the communication between tasks running on the system. This monitoring aims to detect anomalous behavior affecting the packet transmitted through the NoC, thus hinting the possibility of an HT attack. The data gathered from the monitor is then evaluated to decide whether or not an HT attack is happening. This step consists of an end-to-end HT detector capable of identifying HT attacks without knowing where they occur. Chapter 5 describes the Monitoring phase.

Localization: once the Monitoring phase detects the occurrence of an HT attack, the Localization is deployed to search for its location in the NoC. The HT localization is performed using a technique called *path probing* that tests the health of the NoC by sending probing packets through specific paths. This process relies on an algorithm to select which paths must be tested to find the HT, and in a distributed mechanism to trigger a packet transmission between two arbitrary PEs. Chapter 6 details the Localization process.

Countermeasure: after the Localization discovers the actual location of the HT, the Countermeasure phase applies a security response to neutralize or mitigate the HT effect, thus resuming the correct behavior of the manycore. The execution of a countermeasure is an important part of handling attacks, but the implementation of countermeasures is not included in the scope of this work. The goal of this dissertation is developing a mechanism for finding the location of HTs within the NoC before deploying a countermeasure, thus the countermeasure phase is presented to give context to the security flow.

Figure 4.6 shows the microarchitecture used to implement the security flow. The phases presented in Figure 4.5 were broken into blocks that can be implemented separately and then connected. Each block can have multiple implementations that follow different strategies and can be swapped at design time.



Each block is colored according to Figure 4.5 to represent the phase it belongs to. The ***Communication Monitor*** and the ***Attack Detector*** blocks implement the Monitoring phase. The Localization phase is composed by the ***HT Localization Algorithm*** and ***Probe Mechanism***. The ***NoC Health Table*** is used across the security flow and tracks the suspicion level of each link. The ***Countermeasure*** step implements the functionality to neutralize the localized HT.

Figure 4.6 – The microarchitecture of the security flow (Source: the Author).

Communication Monitor (Section 5.1): is the mechanism responsible for monitoring the communication between PEs. The Communication Monitor analyzes the packets exchanged using the data NoC and verifies if the communication works properly. When the monitor detects an anomalous behavior, it warns the Attack Monitor, informing the *Suspicious Path*. The Communication Monitor may also perform ***preventive tests*** on the network, sending probe packets to evaluate if specific paths are working correctly. This allows the security flow to detect HTs proactively before they interfere with the applications. Preventive monitoring aims to perform tests in areas of the NoC without traffic, thus ensuring that a given network region is secure without affecting the performance of applications already executing in the platform. Due to time limitations, the implementation of the preventive monitoring feature is left out of the scope of this work. Section 8.1 provides directions on how this feature can be best implemented within the proposed Communication Monitor.

NoC Health Table (Section 5.2): keeps track of the suspicion level of each NoC link. The Monitoring and Localization phase information is used to evaluate how likely an HT will infect each link. It resembles a heat map in which links near an HT are hotter than links in an HT-free network region. This information helps the other modules of the security flow to make accurate decisions about the existence and location of HT attacks.

Attack Detector (Section 5.3): takes all Suspicious Paths reported by the Communication Monitor and analyzes them. This block aims to differentiate between the anomalies caused by HT attacks and the false positives caused by fluctuations in the NoC behavior (e.g., a congestion in a NoC link during a short period). Once the Attack Detector has received enough reports to decide that a given Suspicious Path is indeed infected with an HT, it marks the path as an Infected Path and activates the HT Localization Algorithm.

Probe API (Section 6.1): is responsible for actually testing (i.e. probing) paths of the data NoC. The Probe API receives a request with a source PE, a target PE, and the path to be tested. The API implements a protocol that sends probe packets between the source and target PEs using the specified path. The API returns a positive result if the packets were received successfully and a negative response otherwise.

HT Localization Algorithm (Section 6.2): investigates the Infected Path in search of the HT location. The HT Localization Algorithm considers the suspicion level of each NoC link and then chooses specific routes to probe (i.e., test). It collects the results of each probe, refining the search until it finds the HT location. This algorithm is responsible for the decision-making of the HT localization process but relies on a lower-level mechanism to probe the chosen NoC routes (i.e., the Probe API).

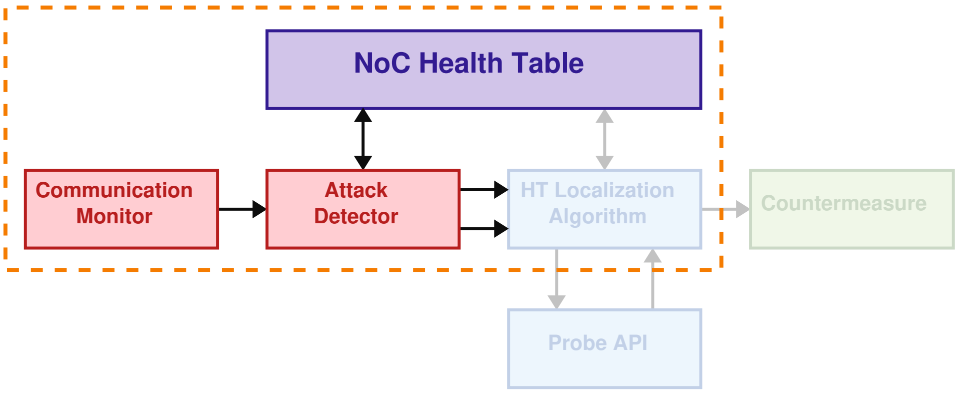
Countermeasure: This component is activated once the HT Localization Algorithm identifies the HT's location. Its objective is to neutralize or mitigate the effects of the HT. This can be achieved through various strategies, such as isolating the HT from the rest of the manycore system, rerouting data packets to avoid the infected link, or remapping the application to a region unaffected by the HT. As previously mentioned, implementing countermeasures is not in the scope of this work. Instead, we focus on the other blocks of the security flow that aim to find the location of HTs.

The security flow presented in Figure 4.6 is implemented completely in software and is contained within the operating system of the PEs.

The Communication Monitor and the Probe API blocks act directly on the communication between different PEs and, thus, are implemented in the OS version that is executed in *all the slave PEs*. On the other hand, the Attack Detector, the NoC Health Table and the HT Localization Algorithm blocks are implemented in the OS version that executes *only in the master PE*.

This section provided an overview of the security flow proposed in this work. Subsequent chapters describe each security flow phase in detail and explain each module we designed to implement the flow in the manycore.

5. MONITORING PHASE



This chapter describes the monitoring phase of the security flow outlined in Section 4.4 and introduces the NoC Health Table, which is used throughout the security flow. Section 5.1 presents the Session Manager protocol that will act as the **Communication Monitor** block. Section 5.2 details the **NoC Health Table**, a core data structure within the localization approach. Section 5.3 introduces the **Attack Detector** block responsible for initiating the subsequent phase (i.e. Localization phase) of the security flow.

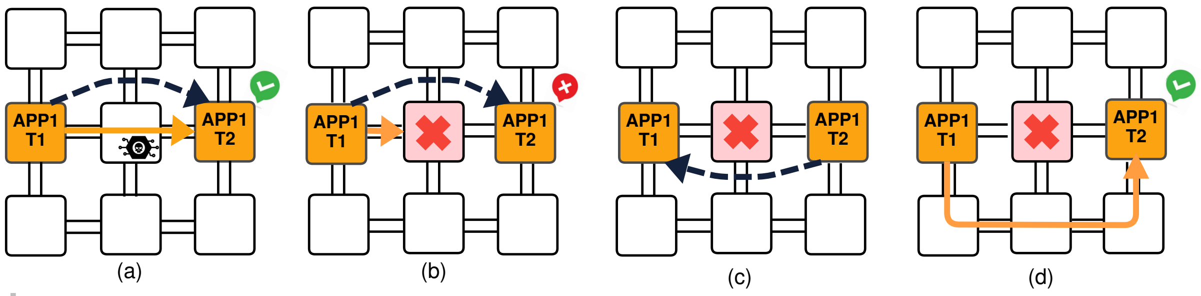
5.1 Communication Monitor

The Session Manager protocol [Faccenda et al., 2021] corresponds to the **Communication Monitor** block. The Session Manager mechanism is designed to monitor, detect, and recover the system against attacks or faults that disrupt the packet delivery via the data NoC (e.g., by a Black Hole HT). The Session Manager implements a session-based protocol that monitors the communication between tasks. Packets that are dropped or delayed result in a session timeout, triggering a recovery countermeasure that searches for an alternative route for the packets.

The objective of the Session Manager is to supervise the sending and receiving of packets in the data NoC. Every time the PE sends a data message, it also sends a control message via the control NoC at the same time to the same target. This control message carries the communicating pair unique identifier, which enables the packet receiver to verify its authenticity and confirm the data message arrival. The Session Manager verifies that the communication is working correctly if both messages are received successfully. If the target PE detects any violation of this session protocol, it activates a recovery mechanism that requests the source PE to resend the data packet, avoiding the original path. As the data NoC supports source routing, the control NoC creates a new path, circumventing the affected region.

The target PE starts the recovery process after detecting an attack or fault through a packet timeout. Here, the control NoC also plays a major role in finding a new path through its built-in path-finding algorithm. As there is no information related to the HT or fault precise location, the method searches for a new path that avoids the routers of the broken path (previous path, which did not deliver the packet correctly). The hop number of the new path can be non-minimal, as presented by the example in Figure 5.1. In addition to the recovery process, the Session Manager also sends a warning message to the Manager PE using the control NoC. This warning allows the Manager PE to make system-level decisions and apply more robust countermeasures.

Figure 5.1 illustrates the process performed by the Session Manager to recover a path broken by an HT. In Figure 5.1 **(a)** the two tasks of *App1* are communicating successfully through the data NoC. In Figure 5.1 **(b)** an HT is activated, disabling a router and breaking the path between the tasks; at this point, only the control messages sent through the control NoC can reach their destination. After a data packet timeout, task *T2* notifies *T1* through the control NoC of the failure to receive their packet (Figure 5.1 **(c)**). Lastly, in Figure 5.1 **(d)**, *T1* activates the path-finding algorithm to look for an alternative path to *T2*; once the new path is found, the packet is retransmitted through the data NoC and reaches its destination.



Task T1 communicates with task T2 sending packets through the data NoC. The path used to send this packets contains an HT-infected router that drops packets. *Dashed arrows* represent the packets transmitted in broadcast using the control NoC. *Straight arrows* are the packets transmitted through the data NoC.
(a) shows a successful packet transmission.
(b) the data transmission interrupted in the data NoC due to a fault or HT.
(c) contains the request for packet retransmission using the control NoC.
(d) the successful retransmission using source routing.

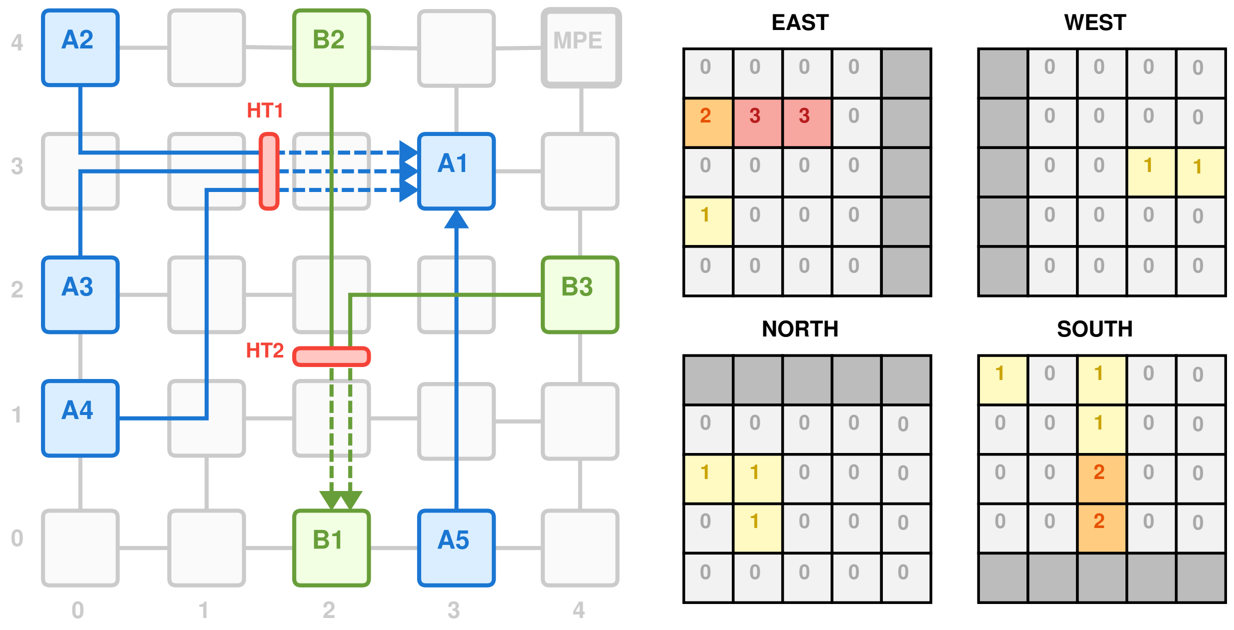
Figure 5.1 – Session Manger protocol (Adapted from Faccenda et al. [2021]).

After detecting an attack or fault, the Session Manager applies a countermeasure to recover the communication between the tasks. This ***countermeasure*** is applied ***locally*** by the Source PE and aims to offer a quick solution, establishing an alternative path and allowing the tasks to continue executing. Although this approach mitigates the threat of HT attacks, the HT remains active in the NoC and is able to perform additional attacks. To execute more robust countermeasures that neutralize the HT, we need to discover the location of the HT in the NoC (Chapter 6).

5.2 NoC Health Table

The purpose of the **NoC Health Table** is to consolidate, into a single centralized structure, all data generated in a distributed manner by the Session Manager and the Probe API. It offers a comprehensive view of the NoC state, indicating which links are operating correctly, which may be compromised, and which are effectively compromised by HTs or faults. Conceptually, it acts as a heat map: “cold” regions indicate normal operation, whereas “hot” regions signify areas exhibiting symptoms of HT-related activity.

The NoC Health Table consists of a 3D-matrix, in which each position represents a different NoC link. The width and length of the matrix are the same as the width and length of the NoC, and the matrix has one layer for each output port of the NoC router. For example, Figure 5.2(a) shows a 5x5 NoC. Each router connects to its neighbors using 4 output ports: east, west, north, and south. This NoC configuration results in a 5x5x4 NoC Health Table. Each position of the NoC Health Table contains a health report about its correspondent link. This report comprises three health metrics: (*i*) health status, (*ii*) suspicion score, (*iii*) probe counters.



EAST

0	0	0	0	
2	3	3	0	
0	0	0	0	
1	0	0	0	
0	0	0	0	

WEST

	0	0	0	0
	0	0	0	0
	0	0	1	1
	0	0	0	0
	0	0	0	0

NORTH

0	0	0	0	0
1	1	0	0	0
0	1	0	0	0
0	0	0	0	0

SOUTH

1	0	1	0	0
0	0	1	0	0
0	0	2	0	0
0	0	2	0	0

a) Platform Overview **b) NoC Health Table (Suspicion Scores)**

(a) Presents a test scenario, with two applications executing in a 5x5 manycore. The application A is depicted in blue and contains five tasks (A1–A5). Application B is green and has three tasks (B1–B3). The arrows represent the paths used to send packets through the NoC. The platform is infected with 2 HTs that drop packets, represented as red blocks.
(b) illustrates the Health Table for this scenario. The table is updated when packets are dropped. The 3D-matrix was sliced in 4 layers, each corresponding to a different router port. The values in the table are the **Suspicious Score** of each element, corresponding to the number of suspicious paths intersecting a link.

Figure 5.2 – Example of the NoC Health Table populated according to HT attacks (Source: the Author).

Health Status: indicates the current status of a link. This variable can assume 3 values: *HEALTHY*, *SUSPICIOUS*, or *INFECTED*. The ***HEALTHY*** status means that this link is behaving normally. The ***SUSPICIOUS*** status means this link has shown symptoms of being infected with an HT. And the ***INFECTED*** status marks that this link was confirmed to be infected with an HT.

Suspicion Score: it is a quantitative value of the probability of the link being infected. The higher this score is, the higher the probability that an HT is located within this link. The Suspicion Score is managed by the Attack Detector, covered in Section 5.3.

Probe Counters: keep track of the probes sent across each link. There are two probe counters per NoC link: the **Total Probes** counter indicates how many probes traversed this link; and the **Failed Probes** counter indicates how many of these probes have failed. Together, these counters give the probing success rate of the link.

Figure 5.2 provides an example scenario of the NoC Health Table. Figure 5.2(a) presents the mapping of 2 applications executing in the manycore. There are 2 Black Hole HTs infecting the NoC. The blue application has one consumer task (A1) and four producer tasks (A2, A3, A3, and A4). The three paths originating from A2, A3, and A4 are all affected by an HT in the 1x3-East link. The path between A5 and A1 is free from HTs and works correctly. The second application (in green) has one consumer task (B1) that receives messages from 2 producers (B2 and B3). Both paths from B2 and B3 are infected by an HT in the link 2x2-South.

Figure 5.2(b) presents the state of the NoC Health Table. The picture presents the Suspicious Scores of each link corresponding to the number of suspicious paths (paths that cross HTs) with an intersection in that link. For instance, consider the suspicious paths that originate from tasks A2 and A3: these paths arrive at the router 0x3 and take an east turn, traversing the link 0x3-East. If we look at the position 0x3 of the "EAST" matrix, we see that the link 0x3 contains a Suspicion Score equal to 2. This corresponds to the intersection of the paths from A2 and A3. The same behavior can be observed for the other links. Note that only paths affected by HTs impact the NoC Health Table. For example, the path from A5 to A1 had no impact on the "NORTH" matrix. The picture also shows that each slice of the NoC Health Table contains one empty row/column. This happens because routers in the border of the NoC have an open port that cannot be connected to any link (e.g., a router in the right-most column of the NoC does not have an east link).

When the system is initialized, the NoC Health Table is set to its initial state: all links are set to the *HEALTHY* status, with its Suspicion Scores and Probe Counters equal to zero. If the system starts to be attacked by an HT, the NoC Health Table begins to be progressively populated.

The HT Localization Algorithm (Chapter 6) uses the data provided by the NoC Health Table to search for HTs. Furthermore, the MPE can use this data to make decisions and perform other system functions. For instance, the MPE can use the NoC Health Table during the application mapping process to avoid placing a task on a suspicious area of the NoC, or during route configuration, to circumvent suspicious links and transmit through a safe route across the NoC.

5.3 Attack Detector

This section presents the **Suspicion Manager** module (***$S_{us}M$***), which acts as the **Attack Detector** of the security flow. The ***$S_{us}M$*** processes the "missing packet warnings" generated by the Session Manager, attributes a suspicion score for each NoC link and registers it in the NoC Health Table. An HT attack is detected when a link's suspicion score becomes high enough to exceed a given threshold. If the ***$S_{us}M$*** detects an HT attack, the next step of the security flow is to search for the HT location.

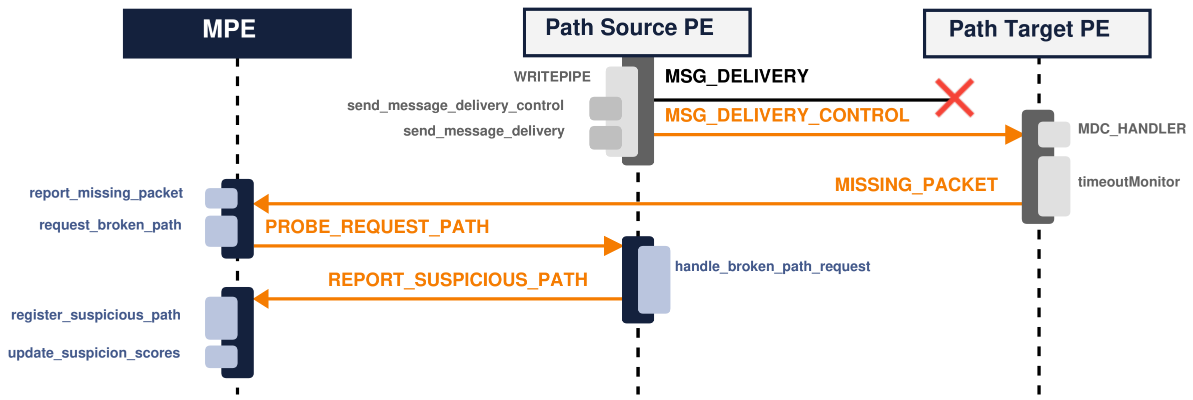
The ***$S_{us}M$*** is necessary because when an HT attack occurs, the MPE can receive several "missing packet warnings" from different sources. Furthermore, the Session Protocol may issue warnings concerning QoS violations unrelated to HT attacks. For instance, if the packets are delayed due to congestion, or if the kernel of the target PE is busy handling an interruption instead of receiving the packet from the NoC. If we triggered the HT Localization algorithm every time the Session Manager issued a "missing packet warning", the MPE would become overloaded. The ***$S_{us}M$*** acts as a filter that only allows the localization process to start once we have confidence in the existence of an HT.

The ***$S_{us}M$*** relies on two key data structures:

- ***NoC Health Table***: it is the main data structure of the security flow and is detailed in Section 5.2. The ***$S_{us}M$*** uses the NoC Health Table to register the suspicion score of each link in the NoC. This information becomes available for every other module in the security flow.

- ***Suspicious Path Table***: it is an ***$S_{us}M$*** internal table. The Suspicious Path Table stores the information of every path that an HT may infect. A path becomes suspicious when the Session Manager sends a "missing packet warning" informing that a packet has been dropped/delayed in that path. This table stores three fields for each suspicious path: Source PE address, Target PE address, and the sequence of turns that constitutes the path between the Source and Target PEs.

The ***$S_{us}M$*** activation occurs when the MPE receives a "missing packet warning". The MISSING_PACKET warning message contains the address of the Source and Target PEs of the affected path but does not contain the path used to route the packet. So, the first action executed by the ***$S_{us}M$*** when it receives a MISSING_PACKET warning is to execute a protocol to obtain the affected path from the Source PE. Figure 5.3 illustrates this protocol. The MPE sends a PROBE_REQUEST_PATH to the Source PE, requesting the path it used to communicate with the Target PE. The Source PE retrieves the path from its routing table and sends it to the MPE using a REPORT_SUSPICIOUS_PATH packet. All messages of this protocol are sent using the control NoC. This ensures that the messages arrive at their destinations despite the existence of HTs in the data NoC.



The sequence diagram displays the name of each function executed during the process. Functions related to the Session Manager are shown in gray. Functions related to the $\boldsymbol{S_{us}M}$ are shown in blue. The black and orange arrows represent packets sent through the data and control NoCs. This picture shows the Source PE sending a message to the Target PE using the Session Manager protocol. An HT attack (shown as a red "x") drops the packet. The Session Manager detects the missing packet and sends a MISSING_PACKET warning to the MPE. The MPE executes the path request protocol to acquire the path to route the missing packet (messages PROBE_REQUEST_PATH and REPORT_SUSPICIOUS_PATH). Finally, the $\boldsymbol{S_{us}M}$ registers the new suspicious path in the Suspicious Path Table and updates the suspicion score of the links in the path.

Figure 5.3 – Sequence diagram for registering a new suspicious path in the $\boldsymbol{S_{us}M}$ (Source: the Author).

Since the control NoC only transmits single-flit packets, we have limited space to encode the path within a control-NoC packet. The payload size of the control-NoC defines the maximum path size supported by the $\boldsymbol{S_{us}M}$. Our version of the manycore supports paths with a maximum of 12 hops. Each hop is encoded using a 2-bit symbol representing East, West, North, and South values. Each path position is read sequentially, starting from the first hop. The path ends when two adjacent hops contain opposite turns (e.g., a south turn followed by a north turn). This compact representation allows paths to be transmitted through the control NoC.

After receiving the REPORT_SUSPICIOUS_PATH message, the $\boldsymbol{S_{us}M}$ has all the information required to configure a new suspicious path (Source PE address, Target PE address, and the path turns). Before adding the new path to the Suspicious Path Table, the MPE first verifies that the table does not contain the same path. This is done because when a packet is dropped, the Source PE might try to resend the packet using the same infected path, causing the packet to be dropped again and resulting in multiple MISSING_PACKET warnings informing the same path.

Registering repeated paths in the table would pollute the table, making the same path appear more suspicious than it is. To avoid this, the $\boldsymbol{S_{us}M}$ always verifies if the path is repeated before registering it. The MPE iterates the table, comparing the Source PE, Target PE, and path turns. If all fields match, the MISSING_PACKET warning is ignored, and the repeated suspicious path is not registered.

Every time the $\boldsymbol{S_{us}M}$ registers a new suspicious path, it also accesses the NoC Health Table and increments the suspicion score of each path link. The suspicion score of each link begins at zero and grows depending on how many suspicious paths are crossing the link, indicating how probable it is for this link to contain an HT. The $\boldsymbol{S_{us}M}$ also verifies the resulting score of each link. If the new suspicion score exceeds a threshold value, the $\boldsymbol{S_{us}M}$ considers it an HT attack and triggers the HT Localization Algorithm. The threshold value for the suspicion score was arbitrarily set to 3 in the experiments.

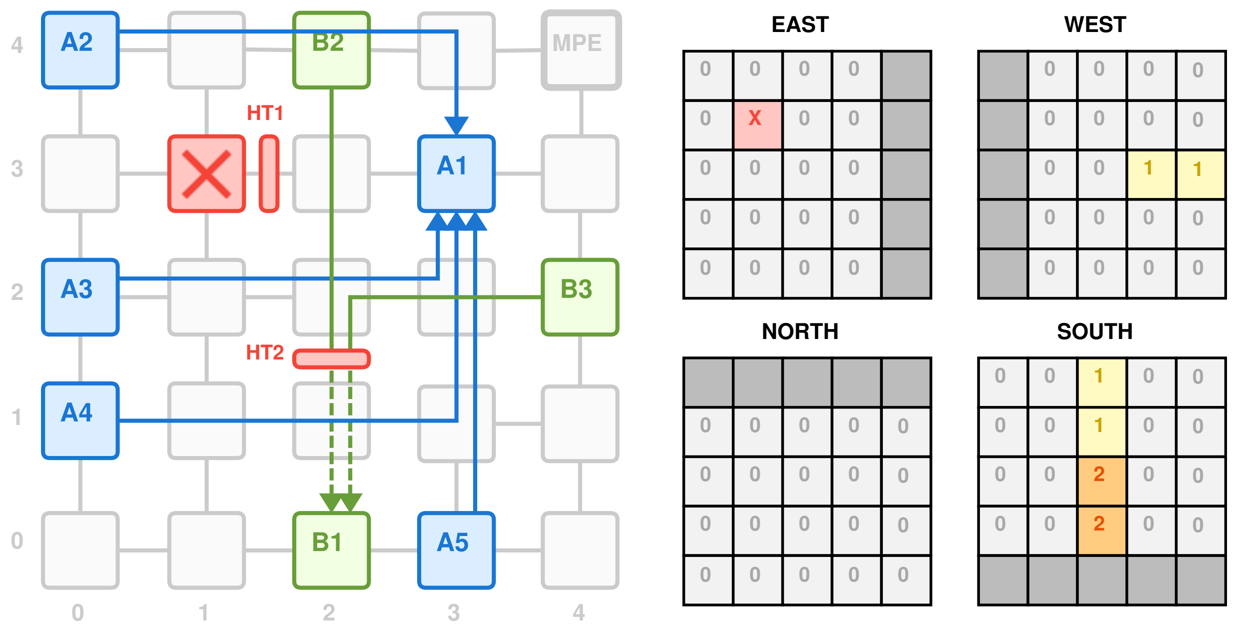
Besides registering suspicious paths and detecting attacks, the $\boldsymbol{S_{us}M}$ must also clear information no longer needed after localizing an HT. When a suspicious path is registered, it means that the path is probably infected with an HT whose location is unknown. After the HT Localization Algorithm successfully finds the location of the HT-infected link within the path, it is no longer necessary to consider the entire path suspicious. The other links used in the path can be considered healthy. The $\boldsymbol{S_{us}M}$ implements a ***clearing routine*** responsible for cleaning the suspicion links related to the detected HT.

The clearing routine is triggered after the HT Localization Algorithm finds a new HT. It works by iterating the Suspicious Path Table and searching for paths that contain the localized HT. If any link in the path matches the location of the HT, then the path is cleared. Clearing a path has two consequences: (a) the cleared path is deleted from the Suspicion Path Table, and (b) the suspicion score of each link in the cleared path is reset to zero in the NoC Health Table. The clearing routine assumes that each suspicious path is infected with only 1 HT. Suppose there happens to be more HTs infecting the same suspicious path. In that case, the $\boldsymbol{S_{us}M}$ will continue to receive missing packet warnings from the Session Manager, and the process will be repeated to localize the second HT.

Consider the scenario presented in Figure 5.2(a). The two applications running on the manycore (app A and app B) are affected by HT attacks with five affected paths. The paths originating at A2, A3, and A4 are affected by HT1, while the paths from B2 and B3 are affected by HT2. Figure 5.2(b) shows the suspicion scores registered in the NoC Health Table for this scenario. These numbers show how many suspicious paths are crossing each NoC link. The Suspicious Path Table contains all five affected paths, originated from A2, A3, A4, B2, and B3.

Let's consider that the last suspicious path to be registered was from A3 to A1. When the $\boldsymbol{S_{us}M}$ registered this path, the suspicious score of the links 1x3-East and 2x3-East were incremented to 3, reaching the threshold value and triggering the HT Localization Algorithm on this path. The HT Localization Algorithm employs methods explained in Section 6.2. Once the HT is localized, the ***S***$\boldsymbol{S_{us}M}$ executes the clearing routine to clear the suspicion of the non-infected links.

Figure 5.4(a) presents the scenario shown in Figure 5.2 after executing the Localization algorithm, which found the location of the HT in the link 1x3-East (HT1). The router containing this infected link is marked with a red "x". The paths traversing the infected link (paths beginning at A2, A3, and A4) were **rerouted** using the communication recovery countermeasure from the Session Manager. The second HT in the link 2x2-South (HT2) remains unlocalized.



EAST

0	0	0	0	▨
0	X	0	0	▨
0	0	0	0	▨
0	0	0	0	▨
0	0	0	0	▨

WEST

▨	0	0	0	0
▨	0	0	0	0
▨	0	0	1	1
▨	0	0	0	0
▨	0	0	0	0

NORTH

▨	▨	▨	▨	▨
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

SOUTH

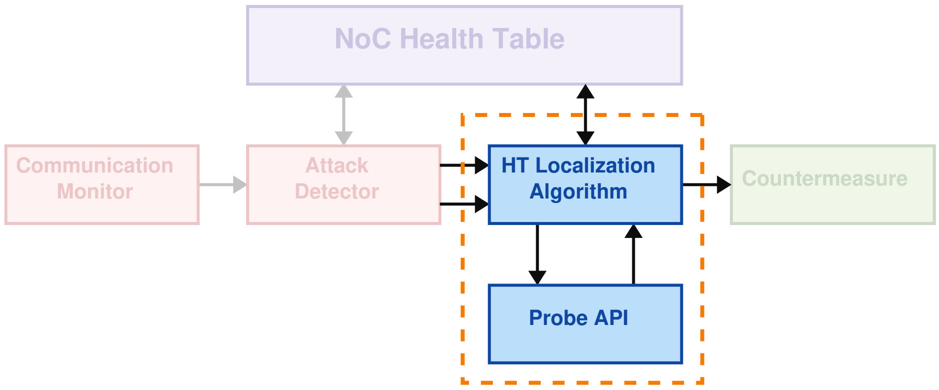
0	0	1	0	0
0	0	1	0	0
0	0	2	0	0
0	0	2	0	0
▨	▨	▨	▨	▨

a) Platform Overview **b) NoC Health Table (Suspicion Scores)**

Figure 5.4 – Example of the clearing routine of the $\boldsymbol{S_{us}M}$ (Source: the Author).

Figure 5.4(b) shows the state of the NoC Health Table after the localization of HT1 and the subsequent execution of the clearing routine. The clearing routine has cleared all suspicious paths that intersected with HT1. The suspicion scores of each link of the cleared paths were reset. The 1x3-East link that was confirmed to be infected with an HT was marked as *INFECTED* in the NoC Health Table (red "x" in the "EAST" matrix). At the end of this scenario, the Suspicious Path Table and the NoC Health Table contain only information related to the two paths affected by HT2: B2 and B3.

6. LOCALIZATION PHASE



This chapter presents the localization phase of the security flow described in Section 4.4. It begins with Section 6.1, which introduces the **Probe API** and explains one of its core features: Probe Batches. The Probe API enables the MPE to test individual paths of the NoC by sending probe packets. In Section 6.2, two **HT Localization Algorithms** are proposed. Both use the Probe API to search the NoC systematically and identify the location of HTs.

The Probe API and the HT Localization Algorithms introduced in this chapter constitute the main contributions of this dissertation. Part of this chapter, specifically those related to the Probe API (Section 6.1) and the binary search localization algorithm (Section 6.2.1), was published in the following conference paper:

> **Hardware Trojan Localization for Untrusted Network-on-chips**
> Gustavo Comarú, Rafael Follmann Faccenda, Luciano Lores Caimi, Fernando Gehm Moraes
> In: LASCAS, 2025

6.1 Probe API

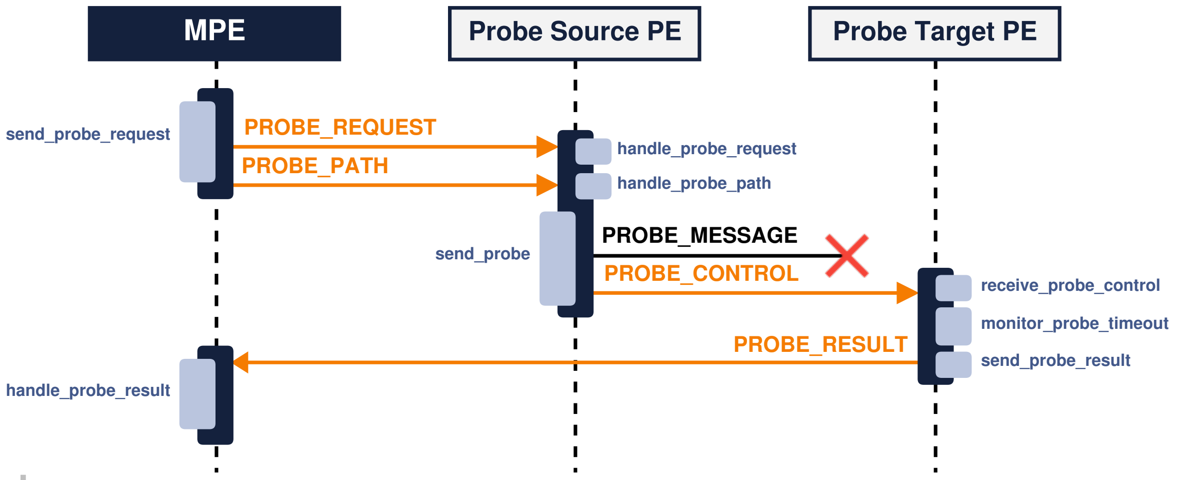
In our proposal, the task of localizing HTs is attributed to the Manager PE (MPE). The MPE receives security warnings from other components and investigates the suspicious segments of the network to find the location of HTs. It is also responsible for proactively monitoring and registering the health of the NoC routers. To fulfill these tasks, the MPE sends probe packets to test whether a specific network segment is working as expected or not. To simplify this process, we implemented the Path Probing API, which abstracts the probing mechanism for the MPE. The Path Probing API has two functions: *Send Probe Request* and *Handle Probe Result*.

Send Probe Request: when the MPE decides to initiate a new probe, it calls the *send_probe_request* function, specifying the source and target PEs, as well as the specific path to be taken by the probe packet. At this point, a unique identifier, *Probe_ID*, is created, and all the information related to the probe is stored in a local table at the MPE.

Handle Probe Result: when the probing is finished, the MPE receives a `PROBE_RESULT` packet and the function *handle_probe_result* is called. This function receives the value of *Probe_ID* and the result specifying if the probe was a success or a failure.

The unique identifier *Probe_ID* allows the MPE to perform several probes simultaneously. When the results come back, the *Handle Probe Result* function retrieves the probe parameters from the table and processes the result accordingly. From the perspective of the HT localization algorithm, the MPE only needs to execute two functions.

Figure 6.1 presents the Probing Protocol, which works underneath the Path Probing API. The Probing Protocol is implemented almost entirely by sending packets via the control NoC; only one packet is sent using the data NoC, which is the probe packet used to test the NoC.



Black and orange arrows represent packets sent via the data NoC and control NoC, respectively. The functions executed by each PE at every step are represented in blue. The Probe Source PE sends a `PROBE_MESSAGE` packet to test a specific path of the data NoC. In this example the probe is affected by an HT attack (represented by a red x) which drops the packet. As a consequence, a probe timeout occurs and the probe returns a negative result to the MPE.

Figure 6.1 – Sequence diagram representing the Probing Protocol that implements the Probe API (Adapted from Comarú et al. [2025]).

The MPE initiates the protocol by sending a `PROBE_REQUEST` packet to the Probe Source PE. This packet contains the parameters for a new probe: Probe Source Address, Probe Target Address, and Probe_ID. Due to the packet size imposed by the control NoC, the last parameter is sent in a separate packet named `PROBE_PATH`. The maximum number of flits in the control NoC payload determines the maximum path size. In the current implementation, the path has a limit of 12 hops, each represented by a two-bit number (E, W, N, S). The last hop of the path is indicated when the next hop contains the opposite direction (e.g., an E hop followed by a W).

The Probe Source PE waits for the reception of both packets from the MPE. Once it has all the parameters, it builds and sends the probe packet. This process is similar to the Session Manager protocol (Section 5.1), as it sends two packets to the Probe Target PE. The first packet, `PROBE_CONTROL` is sent via the control NoC and is guaranteed to arrive due to the broadcast nature of this network. The function of the `PROBE_CONTROL` packet is to inform the Probe Target PE that it should expect the arrival of an incoming probe packet. The second packet, `PROBE_MESSAGE`, is the probe packet itself. The `PROBE_MESSAGE` is sent through the data NoC using the Source Routing algorithm and follows the path specified by the MPE. This packet may or may not arrive at the Probe Target PE, as the path taken by the packet in the data NoC is susceptible to HT attacks. In the scenario presented in Figure 6.1, the probe is dropped by an HT and does not arrive at its destination.

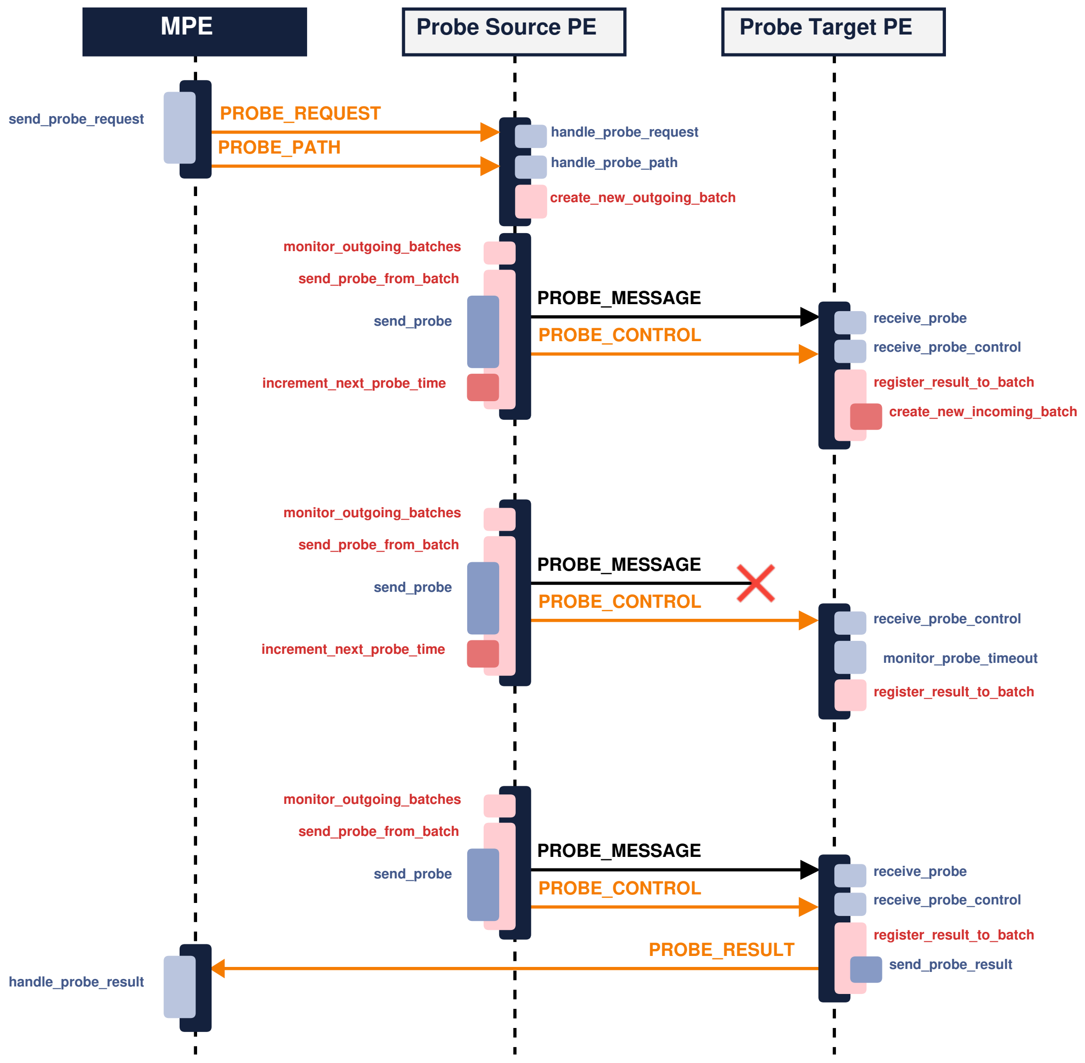
The packets may arrive at the Probe Target PE in any order; however, the `PROBE_CONTROL` packet typically arrives first, as the control NoC is faster than the data NoC. Upon receiving the first packet, the Probe Target PE monitors the delay until the second packet arrives. If the data NoC functions correctly, both packets should be received within a short interval, and the probe is deemed successful. Conversely, if the `PROBE_MESSAGE` packet does not arrive (as illustrated in Figure 6.1) or is significantly delayed, a timeout occurs, and the probe is considered a failure. A `PROBE_RESULT` packet is sent to the MPE, containing the probe result. This packet includes the *Probe_ID* and an indicator of either *SUCCESS* or *FAILURE* for the executed probe.

The Probing Protocol, abstracted by the Path Probing API, allows the MPE to execute higher-level algorithms to investigate paths and find HTs in the data NoC. An example of such an algorithm is the proposed Binary Search Localization, presented in Section 6.2.1.

6.1.1 Probe Batches

The Probing Protocol can reliably detect active HTs until the probing procedure concludes. However, in some cases, a single probe may be insufficient to detect an HT. The Intermittent HT (Section 4.1) exemplifies a type of Trojan that is difficult to detect with a single probe, as it attempts to evade detection by randomizing its activation and deactivation periods. Frequently, attempts to localize an Intermittent HT fail because it is inactive during the probing process, thereby remaining undetected. To address this challenge, we developed a probing method that monitors the NoC over an extended period, thereby increasing the likelihood of detecting such Trojans. The ***Probe Batch*** functionality initiates a session between the Probe Source PE and the Probe Target PE, enabling the execution of multiple probes within a specified time window.

The Probing Protocol sends a single probe (Figure 6.1) by default. Now we expand the Probe API to include three new parameters that can be used to configure a Probe Batch session: *(i)* ***batch size*** is the total number of probes to be sent between the Source PE and Target PE; *(ii)* ***probe delay*** is the time (in microseconds) that the Source PE will wait between sending another probe to the Target PE; and *(iii)* ***probe length*** size (in words) of the probe packet that is sent through the data NoC. The MPE chooses these parameters each time it configures a new batch.



The MPE configures a probe batch between the Probe Source PE and the Probe Target PE. This batch operates by sending a sequence of periodic probes along the same path between the source and target PEs, enabling extended monitoring over a longer time window than a single probe would allow. The MPE can adjust batch parameters to control both the duration and density of the probing process. The sequence diagram illustrates the functions executed by each PE: functions shown in blue correspond to general operations of the Probing Protocol, while those in red are specific to the Probe Batches session. Orange arrows denote packets transmitted via the control NoC, and black arrows represent probe packets transmitted through the data NoC. In this example, the batch consists of three probes. The Target PE successfully receives the first and third probes, whereas the second probe is affected by an attack, resulting in a probe timeout.

Figure 6.2 – Sequence diagram for the Probe Batches feature of the Probing Protocol. (Source: the Author.)

Figure 6.2 presents an example of the Probe Batch session. The MPE starts the process by calling the *send_probe_request* function of the Probe API and passes the *batch* *size*, *probe delay*, and *probe length* parameters. The example provided in Figure 6.2 uses a *batch size* equals to three probes. The basis of the protocol used to configure the batch is the same as explained before in Figure 6.1. First, the MPE uses the control NoC to send the `PROBE_REQUEST` and `PROBE_PATH` messages to transmit the probe parameters to the Probe Source PE. When the Probe Source PE receives both messages, it evaluates the *batch size* parameter: if the size equals 1 or 0 it sends a single probe; but if the size is larger than 1, the Probe Source PE calls the *create_new_outgoing_batch* function to configure the new Probe Batch session. The parameters configuring the probes (e.g., probe target, path, probe delay) are written on a new line in the **Outgoing Batches Table**.

Once a new batch is configured into the Outgoing Batches Table, it is responsible for sending a sequence of probes respecting the *probe delay* parameter. To perform this action, each batch registered in the table contains a field called ***next_probe_time***, which specifies the time (in clock cycles) that the next probe must be sent. The OS monitors this table by programming the interrupt handler to call the *monitor_outgoing_batches* function periodically. When the system clock counter matches the *next_probe_time*, the Probe Source PE executes the probe by sending the `PROBE_MESSAGE` and `PROBE_CONTROL` packets to the Probe Target PE and calls the ***increment_next_probe_time*** function to update the time configuration to send the next probe. This process is repeated until the Probe Source PE sends a number of probes equal to the *batch size* parameter. When all the probes are sent, the Probe Source PE finalizes the batch by clearing the line allocated in the Outgoing Batches Table.

Each `PROBE_MESSAGE` and `PROBE_CONTROL` packet is sent with its unique ProbeID, its *batch size* parameter, and also with its respective position inside the probe batch. When the Probe Target PE receives a probe from a new batch, it configures a new line in the **Incoming Batches Table**. This table keeps track of how many probes the batch has received and how many are still pending. Furthermore, every time the Target PE receives a probe belonging to a batch that is configured in the table, it registers whether the probe was a success or a fail. After receiving all the probes from the batch and evaluating their results, the Probe Target PE deallocates the line from the Incoming Batches Table and sends a `PROBE_RESULT` packet back to the MPE containing the number of successes and fails. Finally, the MPE receives the PROBE_RESULT packet and triggers the execution of the *handle_probe_result* function of the Probe API, which processes the results generated by the probe batch.

This process enables the MPE to monitor the network over a time window that exceeds the duration of a single probe. By adjusting the *batch size*, *probe delay*, and *probe length* parameters, it is possible to control both the duration and the density of the probing activity. When configuring a new probe batch, it is important to consider the trade-off between monitoring intensity and the associated system overhead. Batches that execute many probes with minimal delay between them result in high probe density, increasing the likelihood of detecting HTs, but at the cost of greater resource consumption by the PEs and the NoCs. Conversely, smaller batches or those with greater spacing between probes impose less overhead but reduce the probability of HT detection. Given the wide variety of HTs, intense and resource-intensive probing may be justified when targeting HTs that remain active for only brief periods. The Probe API, with the Probe Batches feature, provides the MPE a flexible tool that can be configured to minimize overhead when detecting easily observable HTs, or to perform intensive monitoring when attempting to identify more stealthy and transient threats.

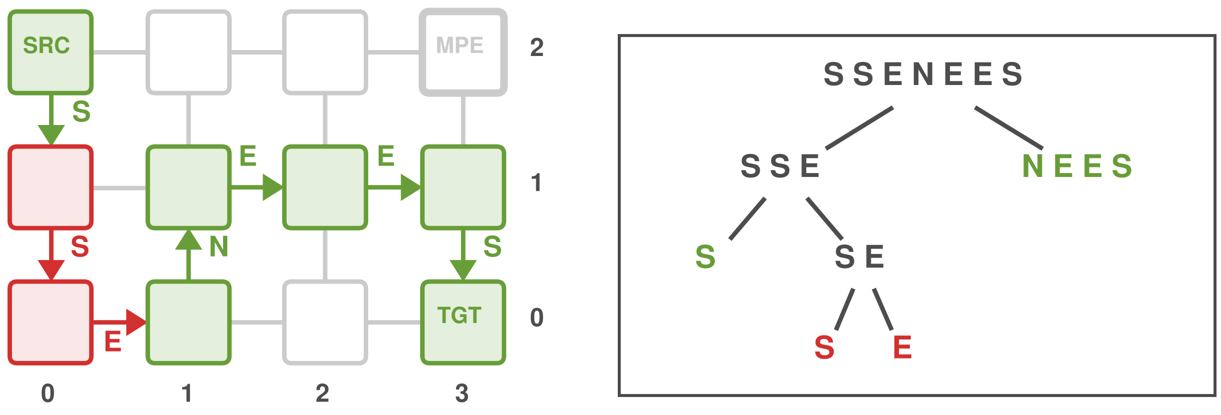
6.2 Search Algorithms

The Probe API provides the MPE with a mechanism to test individual paths within the NoC and to assess whether HTs compromise these paths. This section proposes two algorithms that use the Probe API to localize HTs within the NoC. These algorithms take as input an *Infected Path*, i.e., a path suspected of containing HTs, and operate on this path by issuing Probe API calls to determine the specific location of the HTs. The following subsections describe this process. Section 6.2.1 introduces the first localization algorithm, **BSA**, which is based on a binary tree search strategy. Section 6.2.2 presents an alternative method, the **OSA** algorithm, which prioritizes probing the most suspicious links first. Finally, Section 6.2.3 describes a **unified approach** that combines BSA and OSA to enhance localization effectiveness.

6.2.1 Binary Search Algorithm (BSA)

The Binary Search Algorithm (BSA) breaks the infected path into two segments and tests each individually with different probes. Segments that work correctly are considered to be HT-free. Segments that do not work are divided again, repeating the process. The HT is considered localized when the infected path reaches the length of only one hop. The HT is in this probed link since the path cannot be further divided.

The process employed by this method to localize an HT resembles searching a binary tree structure, hence its name. Figure 6.3 illustrates how an infected path can be divided to search for HT-infected links. When a path is divided, it produces two new paths with half of the original size: the *left-side path* and the *right-side path*. These paths correspond to the left branch of the tree (source to middle) and the right branch (middle to target), respectively. In the same way that each node of a binary tree can be considered a binary tree by itself, the Binary Search Localization considers each probe a different search. This allows the Binary Search Algorithm to take maximum advantage of the support for parallel probes: when the path is broken, the MPE sends probes to the left side and right side simultaneously, and when a `PROBE_RESULT` packet arrives, it is handled independently.



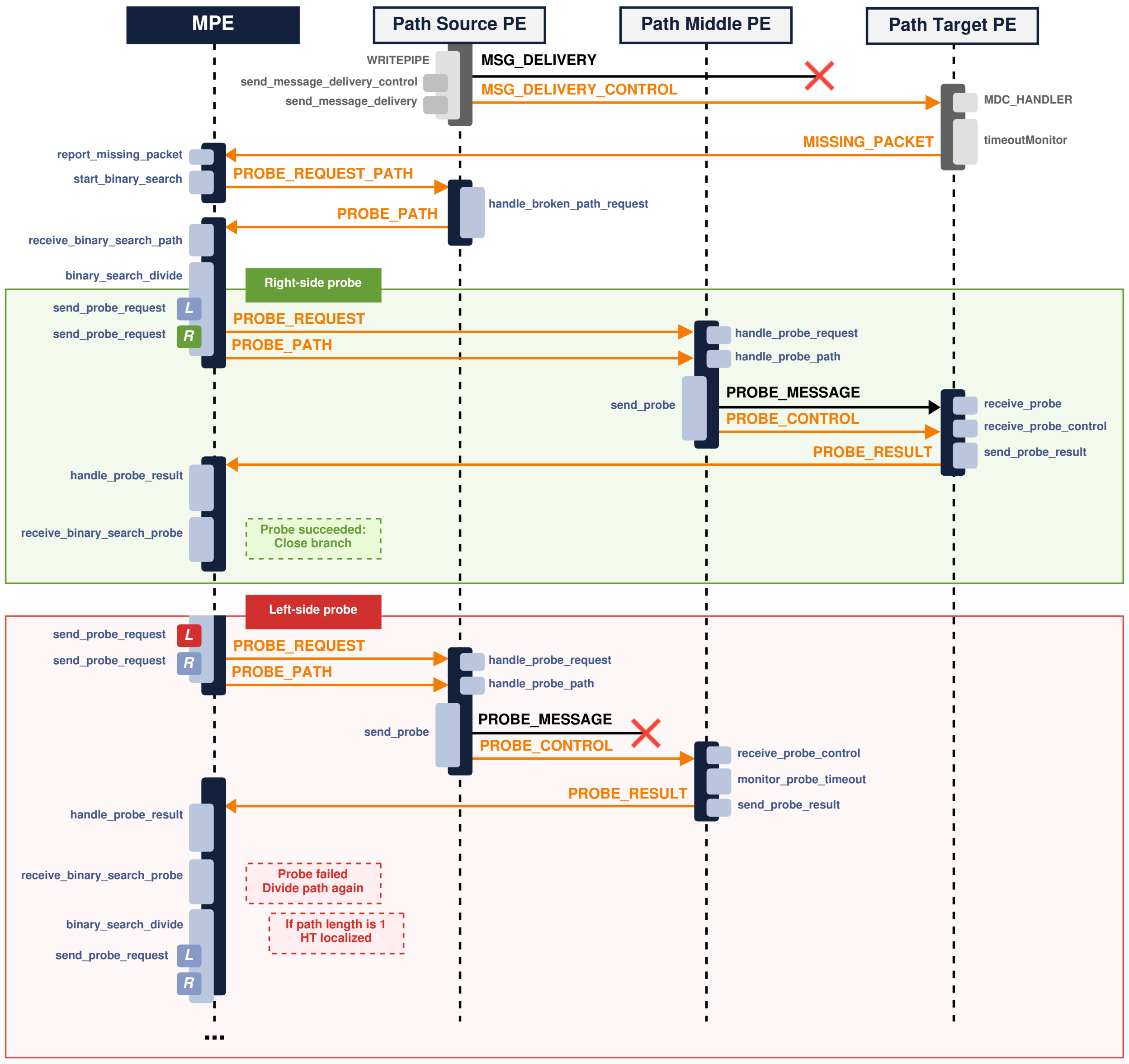
The left-side part of the figure shows the infected path between Path Source PE (SRC) and Path Target PE (TGT). Routers containing infected links are shown in red, HT-free routers in green, and routers outside the path are in light gray. Infected links (red arrows) drop any packet that tries to traverse the link. The right-side part of the figure presents a tree containing the probes performed to localize the HT. The probes in black fail and are divided in half. The green probes are successful and are not divided. The two probes in red fail but cannot be divided, so they identify the HTs localized in 0x1-South and 0x0-East.

Figure 6.3 – Example of path division to localize HTs on an infected path (Adapted from Comarú et al. [2025]).

Considering that each probe can be treated as a separate search makes the implementation of the algorithm straightforward. The MPE keeps a table called the **Binary Search Table** which stores the information for each probe used in the search: the Probe_ID and the Source Routing path. When an infected path is broken into two halves, the MPE allocates two more lines in the table (for the left-side and the right-side segments), and when the result for a probe comes back, the MPE retrieves the parameters from the table and then releases the “parent” line.

Figure 6.4 presents the Binary Search Localization for the same scenario as in Figure 6.3. The diagram begins with two PEs communicating through the data NoC: the *Path Source PE* sends a message to the *Path Target PE*, but this packet is dropped due to interference of an HT (represented as a red "x") and does not reach the target. The target Session Manager detects a timeout violation and reports the missing packet to the MPE, which initiates the Binary Search Localization. The MPE first step is to learn which path the dropped packet took (i.e., acquire the infected path). The MPE sends a `PROBE_REQUEST_PATH` packet to the *Path Source PE*, asking which path was used when sending the dropped packet. Each time the Session Manager detects a failed communication, the source saves the path in a table and thus can answer the MPE by sending a `PROBE_PATH` packet with the infected path.

After these steps, the MPE can start the localization by performing the first path division: the infected path retrieved from the *Path Source PE* is broken into two halves, and one probe is sent through each half. The left-side path starts in the *Path Source PE* (0x2)



Black and orange arrows represent packets sent via the data NoC and control NoC respectively. The functions executed at each step are denoted in light blue blocks. The green and red boxes are executed in parallel. The uppermost structures represented in gray instead of blue are not part of the Binary Search Localization.

Figure 6.4 – Sequence diagram illustrating the Binary Search Localization algorithm (Source: the Author).

and ends in the *Path Middle PE* (1x0), performing the turns S, S, and E (Figure 6.3). The right-side path starts in the *Path Middle PE* (1x0) and ends in the *Path Target PE* (3x0) performing the turns N, E, S, and S. These probes are sent in parallel. Figure 6.4 uses two boxes to represent the different probes being executed simultaneously: the right-side probe is shown first as a green box, and then the left-side probe is shown as a red box. The right-side probe is successful and returns a positive result to the MPE. The MPE closes this branch without dividing the path any further, the routers used in the right-side path are no longer considered infected. On the other hand, the left-side probe is dropped by an HT attack, returning a negative result to the MPE. The MPE divides the path once again, sending two more probes. This process continues until the path can no longer be divided and the HT is localized.

The BSA algorithm is designed to minimize its impact on system resources. It avoids probing the same segments of the infected path multiple times, focusing instead on localizing HTs that remain active for relatively long durations. Although it typically sends only one probe per tested segment—i.e., each time a new branch is explored—it can also be configured to use Probe Batches (Section 6.1.1). However, because BSA explores multiple branches in parallel, intensive use of Probe Batches may lead to excessive load on the control NoC, which operates using broadcast communication. We developed an alternative localization algorithm, presented in the following section, to support the detection of HTs that are active for shorter periods and use the Probe Batch functionality more effectively.

6.2.2 Ordered Search Algorithm (OSA)

The second localization algorithm is the Ordered Search Algorithm (OSA). The OSA conducts a selective search by probing one link at a time. This targeted approach enables the use of the Probe Batches feature without overloading the control NoC with broadcast messages. To enhance efficiency, the OSA initiates the search at the most suspicious link along the infected path and progressively examines less suspicious links. This strategy minimizes unnecessary probing, thereby conserving system resources and accelerating the localization of HTs.

When the OSA begins execution, it initializes an array structure containing information about each suspicious link along the infected path. Each element of this structure includes the following data: (a) the source address of the link, which refers to the address of the router transmitting data through the link; (b) the router port associated with the link (i.e., east, west, south, or north); and (c) the suspicion score assigned to the link. The suspicion score is retrieved from the NoC Health Table (Section 5.2) at the time the OSA algorithm is initiated. This score reflects the number of suspicious paths (those that have reported hardware HT attacks) that traverse the given link. A higher suspicion score indicates a greater likelihood that the HT is located within that specific NoC link.

The next step of the algorithm is to order the array from the highest suspicion score to the lowest suspicion score. When multiple links have the same score, the ordering prioritizes those closer to the target PE. When the links are sorted, the OSA iterates over the array, performing a separate probe batch for each NoC link. The paths that are probed by these batches consist of a single hop (H2H): they start at the ***source address*** specified for each position of the array and take one turn in the direction of the ***router's port***, and take one turn in the direction of the ***router's port***, stopping on the next router.

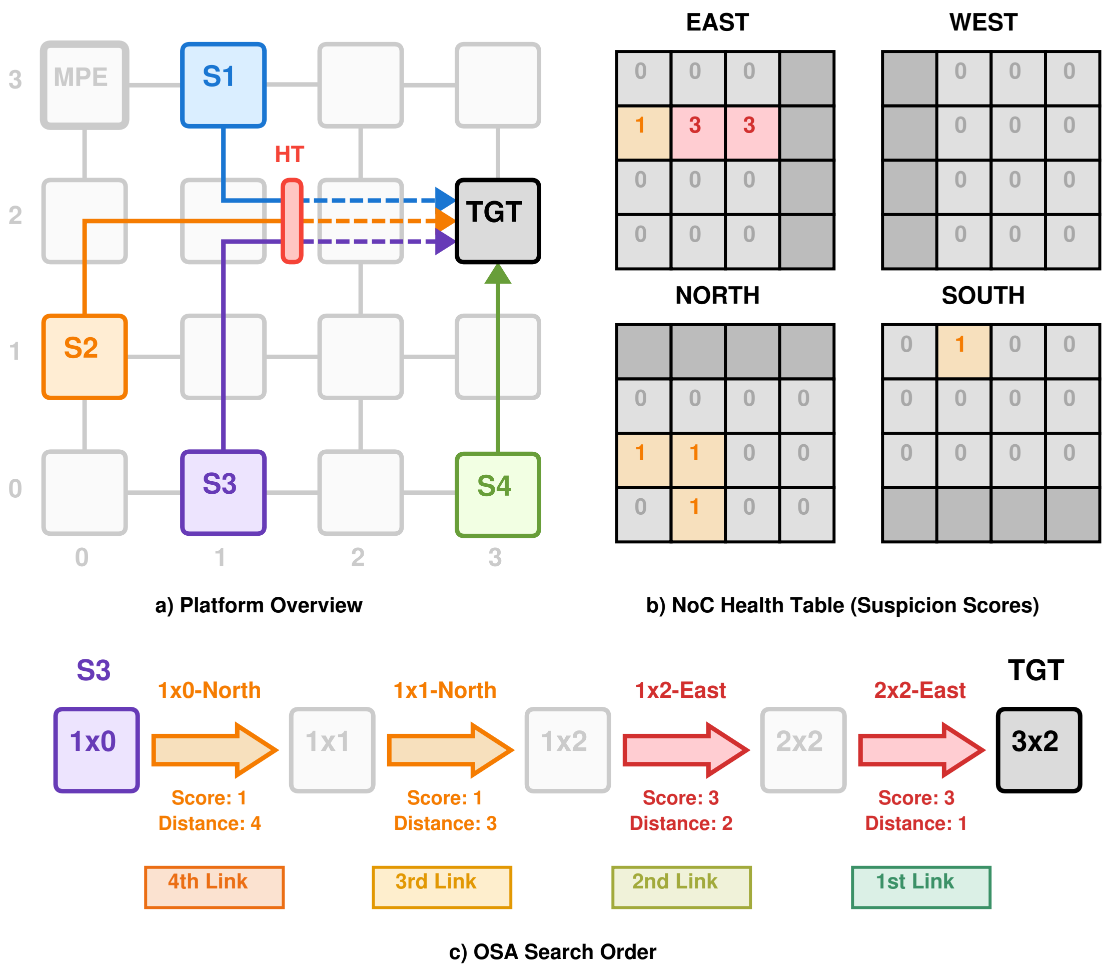
Only one probe batch is executed at a time. Once the OSA programs the batch designated to probe the most suspicious link, it waits for the batch to complete before incrementing the iterator and proceeding to the next suspicious link. Upon completing a batch, the OSA evaluates the results returned by the Probe API. If all probes are successful, it indicates that no HT attack was detected on that link during the batch's time window; in this case, the algorithm evaluates the next link in the array. Conversely, if the batch result contains at least one probe failure, an HT attack has been observed, and the corresponding link is marked as infected. The OSA terminates when all entries in the array have been evaluated or when an HT is successfully localized.

Since the OSA executes only one probe batch at a time, it can configure each batch to perform intensive probing without overloading the control NoC with Probing Protocol control packets. These batches, executed sequentially by the OSA, send a series of probes with minimal delay between them. This probing strategy enables the algorithm to detect HTs that remain active for short periods, such as intermittent HTs (Section 4.1).

If the OSA completes execution without localizing any HTs, two interpretations are possible. First, the detection may have been a false positive, indicating that the infected path contains no HT-infected link. Alternatively, the HT may have a very narrow activation window, and the probe batches configured by the OSA could not capture its activity. To address the latter possibility, the OSA can be re-executed with more stringent probe configurations, sending probes at shorter intervals and over an extended time window.

The remainder of this section presents an example illustrating the execution of the OSA algorithm. Consider the scenario depicted in Figure 6.5, where four source tasks (S1, S2, S3, and S4) attempt to communicate with a common target task (TGT). Initially, S1 and S2 send packets to TGT via the blue and orange paths, respectively. These packets are dropped by a HT located on the 1x2-East link. The attack is detected by the **Session Manager** (Section 5.1), and the **NoC Health Table** updates the suspicion scores of the links traversed by these paths. Subsequently, S3 sends a third packet along the purple path, which is also dropped. The **NoC Health Table** is updated again; this time, the suspicion scores of the 1x2-East and 2x2-East links exceed the threshold defined by the Suspicious Paths Table (Section 5.3), thereby triggering the localization algorithm to initiate the search for the HT along the purple path.

In this example, assume that the OSA is selected as the localization algorithm responsible for identifying the HT. The first step the OSA performs is to load each of the four hops in the purple path into an array. It then consults the NoC Health Table and reorders the array from the most to the least suspicious link. The links 1x2-East and 2x2-East have the highest suspicion score (3) and are therefore placed at the beginning of the array. The remaining two links, 1x0-North and 1x1-North, each have a score of one and are placed at the end. Ties are resolved by giving priority to links that are closer to the target task (TGT). As a result, the final order used by the OSA is: 2x2-East; 1x2-East; 1x1-North; 1x0-North.



(a) scenario in which several source tasks (S1, S2, S3 and S4) communicate with one target (TGT). An HT drops packets sent by the tasks S1, S2, and S3 and never reach TGT. Packets sent by S4 take a different path and arrive successfully at TGT.
(b) NoC Health Table before executing the OSA algorithm. Values correspond to the suspicion score of each NoC link.
(c) illustrates the infected path searched by the OSA. The OSA uses the metrics provided by the NoC Health Table to order the links of the infected path and search them individually. The arrows in this diagram correspond to each link of the infected path. The metrics used by OSA are represented just below each link, as well as the final order that the OSA uses to search the links.

Figure 6.5 – Example for the Ordered Search Algorithm (Source: the Author).

Interestingly, in this case, the OSA produces a link order that is the exact reverse of the original purple path. However, this outcome is specific to the current example and does not represent the algorithm's typical behavior.

The OSA then configures the first probe batch on the 2x2-East link. This initial batch sends probes that originate at PE 2x2, follow a path with only an eastward turn, and arrive at PE 3x2. Since the 2x2-East link is not infected, the result of this probe batch indicates that all probes were successful. The OSA then increments the iterator and probes the 1x2-East link using the same process applied to the first link. This new batch is configured with source PE 1x2, a path consisting solely of an eastward turn, and target PE 2x2. The batch executes by sending multiple probes through the 1x2-East link. Eventually, the HT placed on this link activates and drops one or more probes. Upon receiving the result indicating failed probes, the OSA aborts its execution, thereby localizing the HT to the 1x2-East link. Finally, the **NoC Health Table** marks the 1x2-East link as infected. All other links in the blue, orange, and purple paths are considered HT-free, and their suspicion scores are reset to zero, following the process described in Section 5.3.

6.2.3 Integration of the Localization Algorithms

The previous two sections presented distinct algorithms for localizing HTs in the data NoC. The BSA algorithm (Section 6.2.1) is lightweight, exerting minimal impact on the performance of both the PEs and the NoC. However, its localization capability is limited, with a low likelihood of detecting HTs with short activation periods, such as the intermittent HT described in Section 4.1. In contrast, the OSA algorithm (Section 6.2.2) conducts a more intensive search on selected links and can localize even more "evasive" HTs. This improved detection capability comes at the cost of higher execution resource usage on the PEs and increased communication bandwidth consumption in the NoC.

We combine the two localization algorithms to leverage the strengths of both approaches. Initially, the BSA algorithm is employed to perform a low-impact localization, minimizing disruption to the system. If the BSA completes its execution without localizing any HT, it is assumed that the HT has deactivated, and the OSA algorithm is deployed. The OSA conducts a more intensive search using the Probe Batches feature of the Probe API (Section 6.1.1). This search continues until either the HT is successfully localized or all links of the suspicious path have been searched. The combined localization strategy effectively detects HTs with long and short activation periods while avoiding unnecessary consumption of system resources.

7. SECURITY FLOW RESULTS

This chapter presents the validation and evaluation of the HT localization methods proposed in this dissertation. The security flow described in Section 4.4 was implemented using the modules outlined in Chapter 5 and Chapter 6. These methods were developed in C and integrated into the MPE and Slave PEs kernels. Following this implementation, attack campaigns were conducted on the manycore. These campaigns consisted of multiple RTL-level simulations in which tasks communicated over an HT-infected NoC. Each simulation used different combinations of applications and HTs. The HT Insertion Framework, presented in Section 4.2, enabled actual HTs in the simulations.

Two distinct attack campaigns were performed and are analyzed in the subsequent sections. The first campaign (Section 7.1) employed the BSA algorithm to localize statically time-triggered HTs. The second campaign (Section 7.2) combined the OSA and BSA algorithms to localize intermittent HTs.

7.1 Attack Campaign with the Binary Search Algorithm

The first attack campaign serves as an initial proof of concept to validate the use of probes for HT localization. This campaign used a simplified version of the security flow, excluding the $\boldsymbol{S_{us}M}$, the NoC Health Table, the Probe Batches, and the OSA algorithm.

This campaign considers only **static** HTs, those with predefined activation windows that remain active for the duration of the simulation. As a **result**, all HTs are expected to be successfully localized, since, once activated and blocking a given path, the probes must detect their presence.

The following list enumerates the steps of the simplified security flow.

1. **Communication Monitoring** – The Session Manager establishes a session to monitor the communication between a Source PE and a Target PE (Section 5.1).

2. **Warning Message** – When a packet is dropped or delayed, the Session Manager sends a missing packet warning to the MPE. The Session Manager also executes the recovery countermeasure to reestablish the communication using source routing (Section 5.1).

3. **Path Acquisition** – The MPE executes a simple protocol to obtain the suspicious path that the Source PE used to communicate with the Target PE (Section 5.3).

4. **Localization Deploy** – The MPE deploys the BSA algorithm to search for the HT in the suspicious path (Section 6.2.1).

5. **Localization** – The BSA algorithm uses the Probe API to systematically probe the suspicious path in search of the HT location (Section 6.2.1).

The first attack campaign contains five scenarios. Each scenario is executed twice: once using the Black Hole HT, and another using the Credit Block HT. The list below describes the scenarios.

Scen1 **Basic HT attack** — A synthetic producer-consumer application executes on the manycore. The producer task sends messages to the consumer task along an XY path infected with an HT. The HT activates at 300 μs.

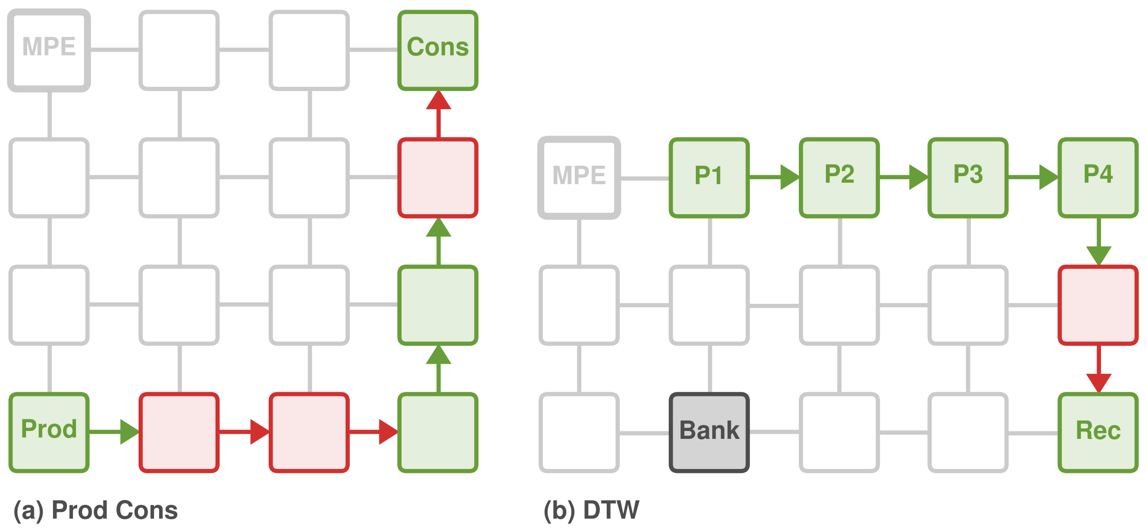
Scen2 **MPEG** — This application consists of a pipeline of five tasks mapped with a 1-hop distance between each pair of tasks. One HT blocks the path between the first and second tasks. The HT activates at 2 ms.

Scen3 **Same HT affecting multiple paths** — The DTW application executes on the manycore (Figure 7.1(b)). In this application, the Bank task distributes the workload across four slave tasks (P1, P2, P3, and P4). These slave tasks process the data from the Bank task and send their results to the Recovery (Rec) task. All four slaves use the same communication path to reach the Recovery task, which contains a single HT. The HT activates at 2 ms (during communication), simultaneously affecting all four communication paths.

Scen4 **Attack on the recovered path** — A producer-consumer application executes on the manycore, with the producer task sending messages to the consumer along a straight-line path. The NoC is infected with two HTs (HT1 and HT2). This scenario has three phases: (*i*) both HTs are initially deactivated, allowing the communication between tasks; (*ii*) HT1 activates at 600 μs, disrupting communication between the producer and consumer tasks. The Session Manager recovery mechanism is triggered, identifying an alternative path between the tasks, and communication resumes over this new path; (*iii*) HT2 activates at 2 ms, affecting the alternative path and again interrupting communication. Simultaneously, HT1 deactivates. The recovery mechanism is executed again, restoring communication via the original path.

Scen5 **Multiple HTs over one path** – Similar to **Scen1** (Figure 7.1(a)). One producer-consumer application executes in the manycore and communicates using an XY path that is infected by 3 HTs. All the 3 HTs become active at 300 μs (before the first message is sent).

Figure 7.1 illustrates two scenarios used to validate the simplified security flow. Figure 7.1(a) presents **Scen5**, in which a producer-consumer application exchanges messages through a path infected with three HTs. During the application's execution, all HTs are activated simultaneously, triggering the BSA localization algorithm. By the end of the simulation,



Labels show which task is executed in each PE. The paths compromised by HTs are colored: green routers are not infected; red routers contain HTs, and drop packets trying to cross the infected link (shown as a red arrow). (a) Producer-consumer application trying to communicate through a path with multiple HTs. (b) DTW application with four workers (P1-P4) sending messages to the Recognizer task (Rec) using the same path.

Figure 7.1 – Examples of testcases used to validate the BSA HT Localization Algorithm (Adapted from Comarú et al. [2025]).

the algorithm successfully identifies the coordinates of all HTs. Figure 7.1(b) depicts **Scen3**, which involves the DTW application. In this scenario, the tasks *P1*, *P2*, *P3*, and *P4* all communicate with *Rec* via a shared path infected by an HT. Upon activation of the HT, each affected communication pair sends a missing packet warning to the MPE, independently triggering a localization request. The MPE queues the incoming requests and processes them sequentially. After the first search concludes and the HT is localized, the MPE handles the next request. Since the same HT causes all disruptions, each BSA execution identifies the same set of HT coordinates in this scenario.

The first attack campaign executed 10 simulations with static HTs performing Black Hole and Credit Block attacks. In every scenario simulated we observed that:

i. the Session Manager detected the missing packet and recovered the communication through source routing;

ii. the BSA algorithm, together with the Probe API, localized the correct coordinates for every HT attack;

iii. the fault-tolerance mechanisms discussed in Section 3.3 successfully keep the NoC functional despite HT attacks inducing side-effects (such as packets stuck in the routers).

This initial attack campaign validates the core premise of this dissertation by demonstrating that HTs can be localized by probing the NoC with test packets.

At this stage, we have addressed only a simplified version of the security flow, explicitly designed to localize static HTs. This represents an incremental step toward fulfilling the objective of this dissertation. Subsequently, we repeated the simulations using intermittent HTs and observed that the simplified approach frequently failed to localize them. This limitation arises because intermittent HTs may not remain active long enough for the BSA algorithm to complete its execution.

7.2 Attack Campaign with the Complete Security Flow

This section presents the second attack campaign executed in the manycore, which considers the complete implementation of the security flow described in Section 4.4. This second attack campaign assesses our proposal's capability to localize intermittent HTs. We consider a more complete scenario that contains three applications executing in the manycore, with two intermittent HT attacks. This section also evaluates how different configurations of Probe Batches affect the HT localization process.

The second attack campaign uses the complete security flow, implemented with the modules presented in Chapter 5 and Chapter 6. The list below provides a short overview of the steps performed by the security flow.

1. **Communication Monitoring** – The Session Manager establishes a session to monitor the communication between a Source PE and a Target PE (Section 5.1).
2. **Warning Message** – When a packet is dropped or delayed, the Session Manager sends a "missing packet warning" to the MPE. The Session Manager also executes the recovery countermeasure to reestablish the communication using a different path (Section 5.1).
3. **Path Acquisition** – The $\boldsymbol{S_{us}M}$ executes a protocol to acquire the suspicious path used by the missing packet, registering it in the Suspicious Path Table (Section 5.3).
4. **Score Update** – The $\boldsymbol{S_{us}M}$ updates the NoC Health Table by incrementing the suspicion score of each link contained in the suspicious path (Section 5.3).
5. **Attack Detection** – The $\boldsymbol{S_{us}M}$ progressively updates the suspicion scores of the NoC Health Table whenever a new path is registered. When a suspicious score exceeds the threshold, the $\boldsymbol{S_{us}M}$ triggers the localization phase (Section 5.3).
6. **BSA Execution** – The BSA performs an initial search, using the Probe API to localize HTs in the suspicious path (Section 6.2.1). The localization phase is concluded if the BSA succeeds in localizing an HT. However, if no HT could be localized, the BSA

considers that the HT has been deactivated and triggers the OSA algorithm to execute another search (Section 6.2.3).

7. **OSA Execution** – The OSA conducts a more comprehensive search using the Probe Batches feature of the Probe API. This secondary search aims to localize intermittent HTs and those with shorter activation windows (Section 6.2.2).

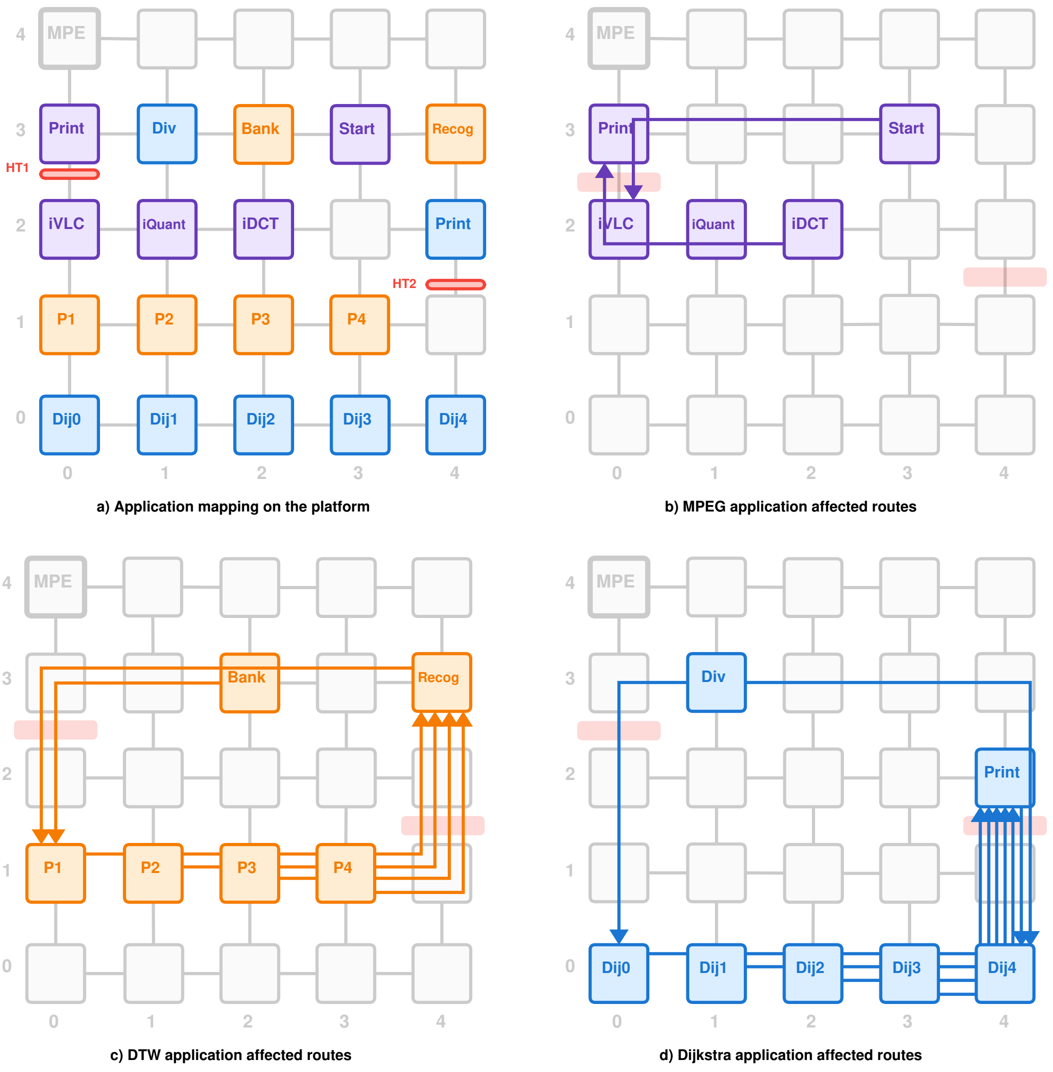
8. **Conclusion** – The OSA finishes its execution, either localizing the HT or not.

The second attack campaign employs a single scenario, **Scen6**, simulated under varying parameters. This scenario comprises three applications: **MPEG** (a pipeline consisting of five tasks), **DTW** (a master-slave model with six tasks), and **Dijkstra** (a master-slave model with seven tasks). These 18 tasks execute concurrently on a 5×5 manycore architecture and communicate via a NoC infected with two intermittent HTs, which activate and deactivate at random intervals within a predefined range.

Figure 7.2(a) presents the tasks mapped in the manycore, with the three applications displayed in different colors. Figure 7.2(b-d) consider each application individually, showing the paths that cross an HT-infected link. HT attacks can affect these paths depending on whether the HT activates when the path is being used. Note that Figure 7.2 only considers the original paths used by the applications and abstracts the alternative paths found by the Session Manager to recover the communication after an HT attack. Alternative paths may also cross HT-infected links and thus be affected by attacks.

Figure 7.3 presents the suspicious score of the NoC Health Table in **Scen6**, both before executing the localization algorithms (Figure 7.3(a)) and after executing the localization algorithms (Figure 7.3(b)). These values were taken from one of the simulations performed in this attack campaign. Figure 7.3(a) shows that some suspicion scores exceeded the threshold values at ports 0x3-South and 1x3-West, with a value of 4. That happens because the HT localization started executing from the threshold value (3); however, during its execution, the suspicion score increased due to another path affected by the HT.

In Figure 7.3(b), the table reflects the state after multiple HT localization attempts, during which both HTs were successfully identified (marked with an “X”). Even after both HTs are localized, some ports display non-zero values. This occurs because the current version of the security flow does not include a countermeasure to isolate an HT following its detection; as a result, the HT remains active and can influence other paths. Consequently, once the suspicion score reaches three again, the system initiates another localization attempt. To simulate the presence of a countermeasure, localization attempts targeting already identified HTs are discarded. Although the search is not executed in such cases, the suspicion scores are still recorded in the NoC Health Table.


MPE
Print
Div
Bank
Start
Recog
HT1
iVLC
iQuant
iDCT
Print
HT2
P1
P2
P3
P4
Dij0
Dij1
Dij2
Dij3
Dij4
a) Application mapping on the platform
b) MPEG application affected routes
c) DTW application affected routes
d) Dijkstra application affected routes

The scenario contains three applications: MPEG (purple), DTW (orange), and Dijkstra (blue). Those applications execute in a 5x5 manycore and communicate using an NoC infected with two intermittent HTs.
(a) shows the application mapping in the manycore, the name of each task is labeled over their respective PE. The 2 HTs are shown in red: HT1 infects the 0x3-South link; HT2 infects the 4x1-North link.
(b-d) represent the paths that may be affected by the intermittent HT attacks. These pictures are separated by application for the sake of clarity. The HTs are also omitted and represented by a red mark over the infected links.

Figure 7.2 – Scenario **Scen6** used in the attack campaign with the complete security flow and intermittent HT attacks (Source: the Author).

```
(a) NHT before executing the HT localization.            (b) NHT after executing the HT localization.

SOUTH:        NORTH:        WEST:        EAST:           SOUTH:        NORTH:        WEST:        EAST:
. . . . .     . . . . .     . . . . .    . . . . .       . . . . .     . . . . .     . . . . .    . . . . .
4 . . . .     . . . . .     . 4 3 2 1    . . . . .       X . . . .     . . . . .     . . . . .    . . . . .
3 . . . .     . . . . 3     . . . . .    . . . . .       . . . . .     . . . . X     . . . . .    . . . . .
1 . . . .     . . . . 3     . . . . .    1 2 2 3 .       . . . . .     . . . . 3     . . . . .    . . . . .
. . . . .     . . . . .     . . . . .    . . . . .       . . . . .     . . . . .     . . . . .    1 1 1 2 .
```

Figure 7.3 – Example of NoC Health Table (NHT) presenting the suspicion scores before and after the HT localization in **Scen6** (Source: the Author).

The second attack campaign consists of multiple simulations of the **Scen6** scenario. Each simulation employs a distinct Probe Batch configuration. The parameters used for the Probe Batches are listed below, and the results of the 12 simulations are presented subsequently.

- ***Batch Size***: 5, 10, and 30 probes.
- ***Probe Delay***: 10, 50, 100, and 250 μs.
- ***Probe Length***: 30 words (60 flits) – fixed size.

These simulations utilize the Black Hole intermittent HT, configured with activation windows ranging from 0 μs to 81.91 μs, and inactivation windows ranging from 204.80 μs to 655.35 μs. The ***S$_{us}$M*** detects HT attacks using a suspicion score threshold of 3, the default value. The timeout thresholds for the Session Manager and the Probe API are set to 655.34 μs and 150 μs, respectively.

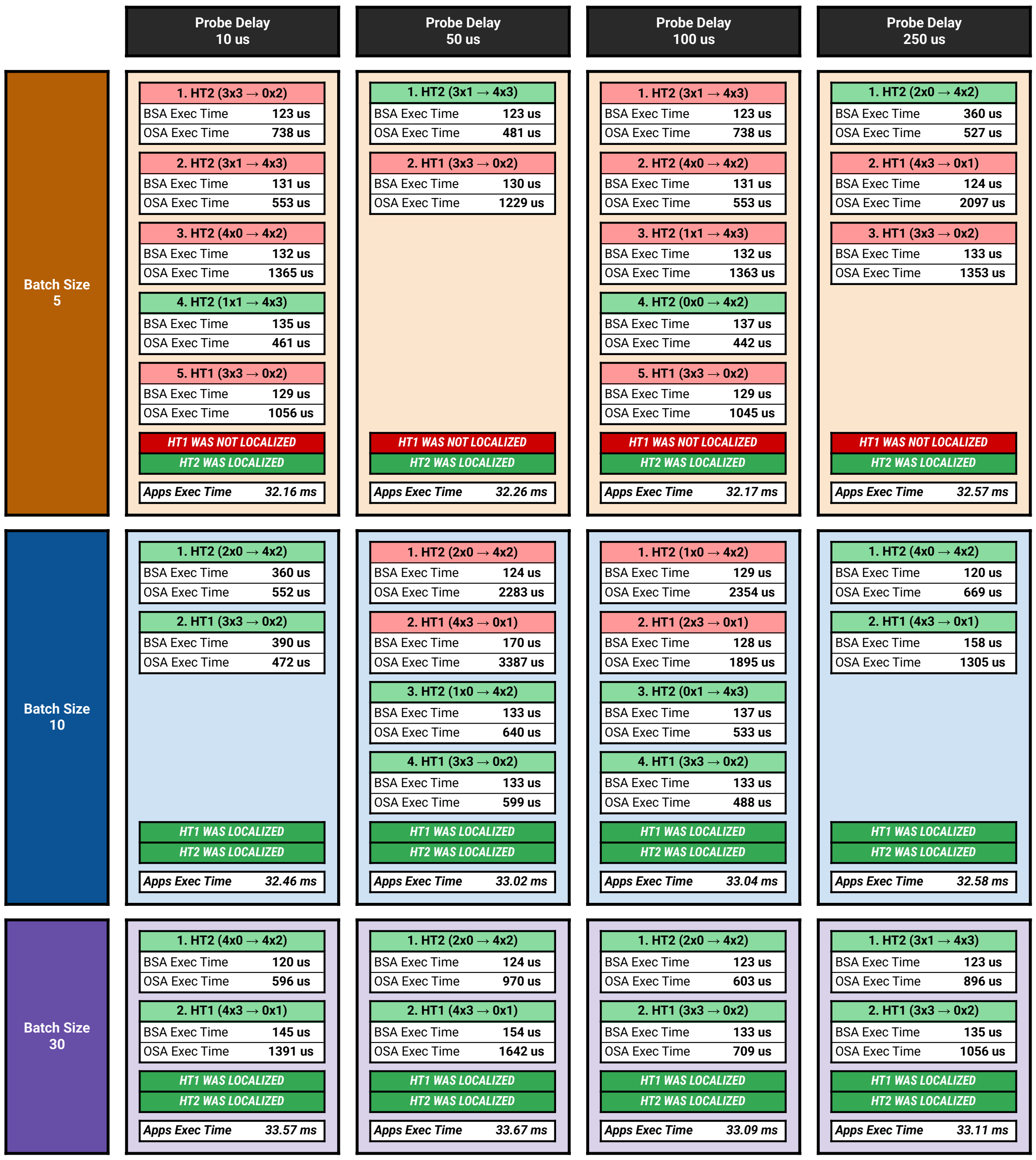
Figure 7.4 presents a table summarizing the results of the attack campaign related to the detection of the intermittent HTs, HT1 and HT2, in the **Scen6** scenario. The following observations can be made:

i. **HT2** triggers more localization attempts then **HT1**. This happens because **HT2** affects nine different communication flows, whereas **HT1** affects only four. As a consequence, there are simulations in which **HT2** is localized by subsequent searches, while **HT1** remains unlocalized after only one search (batch size = 5).

ii. Small batch sizes exhibit reduced effectiveness in localizing intermittent HTs. This is evident in the first row of the table, where the smaller batch (batch size = 5) failed to localize **HT1**. Due to their shorter execution periods, smaller batches may complete before the HT can reactivate.

iii. As the number of probe packets increases (i.e., as the batch size grows), HT localization becomes more efficient, requiring fewer searches to identify the HTs. For example, with a batch size of 30, a single search is sufficient to detect both HTs.

iv. The BSA algorithm could not detect the intermittent HTs, requiring the execution of the OSA algorithm. It is noteworthy that the execution time of BSA remains nearly constant (approximately 130 μs), except in two cases where a BSA probe was dropped by the HT, resulting in a probe timeout. In these instances, the HT became inactive shortly thereafter, preventing the BSA from identifying its location. The need for the OSA algorithm does not diminish the role of BSA within the proposed approach. Given its shorter execution time, the BSA algorithm can serve as an initial attempt to localize static HTs before executing the more resource-intensive OSA.



	Probe Delay 10 us	Probe Delay 50 us	Probe Delay 100 us	Probe Delay 250 us
Batch Size 5	🟥 1. HT2 (3x3 → 0x2) BSA Exec Time 123 us OSA Exec Time 738 us 🟥 2. HT2 (3x1 → 4x3) BSA Exec Time 131 us OSA Exec Time 553 us 🟥 3. HT2 (4x0 → 4x2) BSA Exec Time 132 us OSA Exec Time 1365 us 🟩 4. HT2 (1x1 → 4x3) BSA Exec Time 135 us OSA Exec Time 461 us 🟥 5. HT1 (3x3 → 0x2) BSA Exec Time 129 us OSA Exec Time 1056 us *HT1 WAS NOT LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.16 ms*	🟩 1. HT2 (3x1 → 4x3) BSA Exec Time 123 us OSA Exec Time 481 us 🟥 2. HT1 (3x3 → 0x2) BSA Exec Time 130 us OSA Exec Time 1229 us *HT1 WAS NOT LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.26 ms*	🟥 1. HT2 (3x1 → 4x3) BSA Exec Time 123 us OSA Exec Time 738 us 🟥 2. HT2 (4x0 → 4x2) BSA Exec Time 131 us OSA Exec Time 553 us 🟥 3. HT2 (1x1 → 4x3) BSA Exec Time 132 us OSA Exec Time 1363 us 🟩 4. HT2 (0x0 → 4x2) BSA Exec Time 137 us OSA Exec Time 442 us 🟥 5. HT1 (3x3 → 0x2) BSA Exec Time 129 us OSA Exec Time 1045 us *HT1 WAS NOT LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.17 ms*	🟩 1. HT2 (2x0 → 4x2) BSA Exec Time 360 us OSA Exec Time 527 us 🟥 2. HT1 (4x3 → 0x1) BSA Exec Time 124 us OSA Exec Time 2097 us 🟥 3. HT1 (3x3 → 0x2) BSA Exec Time 133 us OSA Exec Time 1353 us *HT1 WAS NOT LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.57 ms*
Batch Size 10	🟩 1. HT2 (2x0 → 4x2) BSA Exec Time 360 us OSA Exec Time 552 us 🟩 2. HT1 (3x3 → 0x2) BSA Exec Time 390 us OSA Exec Time 472 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.46 ms*	🟥 1. HT2 (2x0 → 4x2) BSA Exec Time 124 us OSA Exec Time 2283 us 🟥 2. HT1 (4x3 → 0x1) BSA Exec Time 170 us OSA Exec Time 3387 us 🟩 3. HT2 (1x0 → 4x2) BSA Exec Time 133 us OSA Exec Time 640 us 🟩 4. HT1 (3x3 → 0x2) BSA Exec Time 133 us OSA Exec Time 599 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.02 ms*	🟥 1. HT2 (1x0 → 4x2) BSA Exec Time 129 us OSA Exec Time 2354 us 🟥 2. HT1 (2x3 → 0x1) BSA Exec Time 128 us OSA Exec Time 1895 us 🟩 3. HT2 (0x1 → 4x3) BSA Exec Time 137 us OSA Exec Time 533 us 🟩 4. HT1 (3x3 → 0x2) BSA Exec Time 133 us OSA Exec Time 488 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.04 ms*	🟩 1. HT2 (4x0 → 4x2) BSA Exec Time 120 us OSA Exec Time 669 us 🟩 2. HT1 (4x3 → 0x1) BSA Exec Time 158 us OSA Exec Time 1305 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 32.58 ms*
Batch Size 30	🟩 1. HT2 (4x0 → 4x2) BSA Exec Time 120 us OSA Exec Time 596 us 🟩 2. HT1 (4x3 → 0x1) BSA Exec Time 145 us OSA Exec Time 1391 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.57 ms*	🟩 1. HT2 (2x0 → 4x2) BSA Exec Time 124 us OSA Exec Time 970 us 🟩 2. HT1 (4x3 → 0x1) BSA Exec Time 154 us OSA Exec Time 1642 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.67 ms*	🟩 1. HT2 (2x0 → 4x2) BSA Exec Time 123 us OSA Exec Time 603 us 🟩 2. HT1 (3x3 → 0x2) BSA Exec Time 133 us OSA Exec Time 709 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.09 ms*	🟩 1. HT2 (3x1 → 4x3) BSA Exec Time 123 us OSA Exec Time 896 us 🟩 2. HT1 (3x3 → 0x2) BSA Exec Time 135 us OSA Exec Time 1056 us *HT1 WAS LOCALIZED* *HT2 WAS LOCALIZED* *Apps Exec Time 33.11 ms*

Table with the results of the 12 simulations that use different Probe Batch parameters. The left-most column shows the Batch Size values. The top-most row shows the Probe Delay values. The colored blocks are associated with a Batch Size and a Probe Delay, representing one simulation from the attack campaign. Each simulation block shows the following information:
1) Every HT localization attempt that was executed in the simulation. Localization attempts that localized the HT are marked in green, whereas attempts that failed to localize the HT are marked in red. Together with every localization attempt, we also show the suspicious path being searched, the HT responsible for the search, and the execution time of both localization algorithms.
2) An indicator showing whether or not each HT was localized during the simulation.
3) The total execution time for the applications. This time starts when the applications are accepted into the manycore and ends when the last application finishes executing.

Figure 7.4 – Results of the second attack campaign, which uses the complete security flow to localize intermittent HTs (Source: the Author).

v. The execution time of the applications is minimally affected by the search method, varying between 32.16 *ms* and 33.67 *ms* (4.7 %).

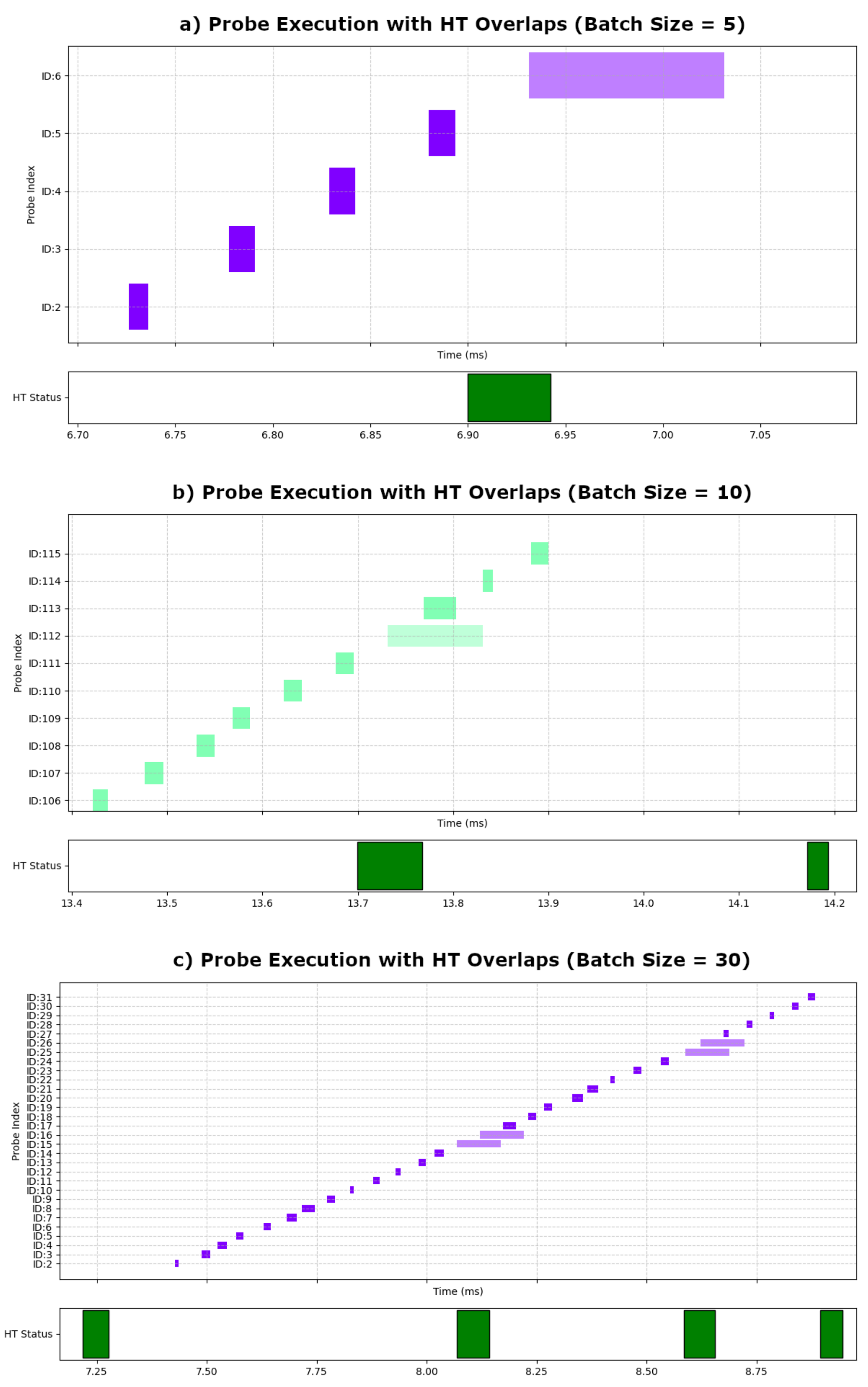
vi. The execution time of the OSA algorithm can be relatively high, with some cases exceeding 1 *ms* (e.g., 1.624 *ms* for a probe delay of 50 μs and a batch size of 30). As mentioned above, this time does not impact the execution of the applications because the link affected by the HT is no longer in use due to the rerouting countermeasure of the Session Manager.

The execution times of the HT localization algorithms presented in Figure 7.4 are influenced by several factors: (*i*) longer communication paths require more time to be searched; (*i*) if an HT is successfully localized, the search is aborted early, resulting in reduced execution time; (*iii*) the processing elements (PEs) may be interrupted to execute other functions during localization, leading to longer execution times; (*iv*) the configuration of the Probe Batches affects the duration of the OSA, larger batch sizes result in longer execution times.

The results presented in Figure 7.4 indicate that the detection of intermittent HTs depends on both the timing characteristics of the HTs and the Probe Batch configurations, as these factors influence the probability of collisions between probe packets and HT activation periods.

Figure 7.5 presents three graphs illustrating the collisions between probe packets and HT activation periods. These graphs correspond to the simulations shown in the second column of Figure 7.4 (Probe Delay = 50 μs). Each simulation involved a successful OSA search that localized **HT2** on the 4×1-North link, using different Batch Size values (5, 10, and 30 probes). The graphs in Figure 7.5 depict the Probe Batches employed in these successful OSA searches. Each graph plots the duration of individual probes (upper portion) alongside the corresponding HT activation windows (lower portion). A collision occurs when an HT is active during a probe's transmission, causing the HT to drop the packet and prevent its arrival at the target. Missed probes, resulting from such collisions, are represented as enlarged bars.

The analysis highlights a trade-off between the number of probe packets and the probability of detecting HTs during their activation intervals. Specifically, smaller batch sizes reduce the chance of collision with active HTs, potentially allowing threats to remain undetected, as demonstrated by the first row in Figure 7.4 (batch size = 5), where HT1 could not be localized using only five probes. Conversely, increasing the number of probes enhances detection probability but also demands more system resources, restricting the availability of links for other applications during testing periods. The results indicate that, for the HT configured for these particular activation times and intervals, a compromise is achieved with a Probe Delay of 10 μs and a Batch Size of 10 probes, which successfully detected both HTs during the first execution of the OSA algorithm.



This figure contains three graphs that illustrate the collision between probes and HT activation periods for different simulations. The graphs show a batch executed in the 4x1-North link, attempting to localize HT2. The upper part of each graph represents the probes sent through the link. Probes with a larger width correspond to probes that were dropped by an HT, resulting in the detection of the HT. The lower part of each graph contains green rectangles that correspond to the intervals where the HT is active — time in *ms*.
(a) Batch Size: 5; Probe Delay: 50 μs. HT was detected in the fifth probe.
(b) Batch Size: 10; Probe Delay: 50 μs. HT was detected in the seventh probe.
(c) Batch Size: 40; Probe Delay: 50 μs. Four probes failed, resulting in multiple detections of the same HT.

Figure 7.5 – Collisions between probes and HT activation periods (Source: the Author).

8. CONCLUSION AND FUTURE WORK

HTs infecting the NoC expose manycore systems to threats such as packet misrouting, packet dropping, data tampering, and network flooding, requiring effective countermeasures to protect the applications. However, deploying these countermeasures effectively depends on identifying the HT's location. Existing localization methods typically rely on embedding additional security hardware into the NoC, making them inappropriate when the NoC itself is deemed untrusted. This dissertation sought to develop a non-invasive method that does not need to integrate extra security hardware in the NoC.

This work proposed a three-phase security framework: (1) the **Monitoring Phase** establishes sessions to observe inter-task communication and detect the presence of HT attacks; (2) the **Localization Phase** executes an HT localization algorithm that searches for the HT by transmitting probe packets along specific NoC routes; and (3) the **Countermeasure Phase** applies a mechanism to neutralize or mitigate the impact of the HT on the system.

For the **Localization Phase**, we proposed a technique called *path probing*, which operates by transmitting probe packets along specific NoC paths to evaluate the integrity of the links. The HT localization algorithm selectively sends probes through the NoC and analyzes their results. Based on the result of each probe, the search is progressively refined until the HT is localized. This method is implemented in software, enabling HT localization without embedding security hardware into the NoC routers.

The implementation of the security framework resulted in additional contributions:

- **HT Insertion Framework**: an automated tool for injecting HTs into the NoC links, establishing the foundation for conducting HT-based attack campaigns on the NoC;
- **Fault-Tolerance Mechanisms for NoC Protection**: integration of mechanisms to safeguard the NoC against disruptions caused by either hardware faults or HT attacks.
- **Secure Network Interface for Peripherals (SNIP) Extension**: full integration of the SNIP into the manycore, adding new functionalities to issue warnings upon detecting security threats.

The results demonstrated the effectiveness of the probing method. The first attack campaign confirmed the BSA algorithm's efficacy in identifying permanently active HTs. Subsequently, we showed that the OSA technique can detect intermittent HTs, with minimal impact on applications' execution time (less than 5%). It was also observed that detecting intermittent HTs is not straightforward, as it depends on the collision between HT activation periods and the transmission of probe packets. We evaluated different configurations for the probe packets, and for the experiment considered, specific parameter values (batch size = 10 and probe delay = 10) enabled successful HT localization. However, these values are not universally applicable, as HTs may exhibit varying activation patterns. Therefore, the detection process implemented in software is advantageous because it allows dynamic adjustment of parameters when detection fails along suspected paths.

The implementation of the proposed security mechanisms is publicly available at https://github.com/gaph-pucrs/hemps_OSZ. Furthermore, this work has resulted in the publication of journal and conference papers listed in Section 1.3.

8.1 Future Work

Based on the work developed in this dissertation, we outline suggestions for future research below.

- ***Countermeasure Mechanisms*** — Due to the time dedicated to developing the HT localization flow, the implementation of HT countermeasures was left as future work. Our approach detects the coordinates of the HT and the suspicion scores are recorded in the NoC Health Table. Such data can support the implementation of the following countermeasures: (*i*) mapping applications to regions of the NoC that are free of HTs; (*ii*) rerouting packets through paths that avoid infected links; and (*iii*) isolating infected NoC links using Link Control units (Section 3.1.1).
- ***Preventive Monitoring*** — The security flow proposed in this dissertation includes a preventive monitoring strategy for the NoC. However, the implementation of this proactive monitor remains as future work. We suggest extending the Communication Monitor module to: (*i*) identify underutilized NoC paths for observation; (*ii*) periodically send probe packets through these paths; and (*iii*) report any failed probes to the Attack Monitor module. The security flow is compatible with this enhancement.
- ***Fault-Tolerant NoC*** — During the development of this work, we observed that both attacks and faults in the NoC may produce severe side effects, potentially rendering the NoC inoperable. Although the literature acknowledges the possibility of NoC faults, current approaches do not address their secondary effects. This dissertation proposes three mitigation mechanisms, though fault tolerance was not its primary focus. Therefore, the design of a fault-tolerant NoC remains a promising direction for future research.
- ***New Localization Algorithms*** — The modular architecture of the proposed security flow allows for enhancements through the development of new module implementations. In particular, we encourage the design of alternative HT localization algorithms. Potential directions include: (*i*) an algorithm tailored for NoCs with multiple physical

planes, and (*ii*) an algorithm that dynamically adjusts Probe Batch parameters to target HTs with narrow activation windows.

- ***Probe Batches with Randomized Delays*** — The current Probe Batch mechanism establishes periodic probe transmission between two PEs based on a fixed delay. This mechanism could be extended to support randomized delays, generated according to different statistical distributions, to improve the detection of HTs exhibiting erratic activation behavior.

- ***Extending the Security Flow to Other Attacks*** — This dissertation focused on localizing HTs that prevent packets from reaching their destinations, specifically targeting Black Hole and Credit Block attacks. Future work could broaden the threat model to include a wider range of HT behaviors. New probing strategies may be developed to detect HTs that perform attacks such as flooding, misrouting, spoofing, or data tampering.

REFERENCES

Baron, S., Wangham, M. S., and Zeferino, C. A. (2013). Security mechanisms to improve the availability of a Network-on-Chip. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 609–612. https://doi.org/10.1109/ICECS.2013.6815488.

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

Bhamidipati, P. and Vemuri, R. (2024). QA-NoCs: Quantitative Analysis for Trojan Detection in Network-on-Chips. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 757–761. https://doi.org/10.1109/ISVLSI61997.2024.00147.

Bohnenstiehl, B., Stillmaker, A., Pimentel, J., Andreas, T., Liu, B., Tran, A., Adeagbo, E., and Bass, B. (2016). A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *IEEE Symposium on VLSI Circuits (VLSIC)*, pages 1–2. https://doi.org/10.1109/VLSIC.2016.7573511.

Caimi, L. L. (2019). *Secure Admission and Execution of Applications in NoC-based Many-cores Systems*. PhD thesis, PPGCC-PUCRS. 121p, http://tede2.pucrs.br/tede2/handle/tede/8917.

Caimi, L. L., Fochi, V., and Moraes, F. G. (2018). Secure Admission of Applications in Many-Cores. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 761–764. https://doi.org/10.1109/ICECS.2018.8618021.

Caimi, L. L. and Moraes, F. (2019). Security in Many-Core SoCs Leveraged by Opaque Secure Zones. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 471–476. https://doi.org/10.1109/ISVLSI.2019.00091.

Carara, E. A., de Oliveira, R. P., Calazans, N. L., 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.

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. (2020). Securing Network-on-Chip Using Incremental ryptography. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 168–175. https://doi.org/10.1109/ISVLSI49217.2020.00039.

Chaves, C. G., Azad, S. P., Hollstein, T., and Sepúlveda, J. (2019). DoS attack detection and path collision localization in NoC-based MpsoC architectures. *Journal of Low Power Electronics and Applications*, 9(1):7. https://doi.org/10.3390/jlpea9010007.

Comarú, G., Faccenda, R. F., Caimi, L. L., and Moraes, F. G. (2025). Hardware Trojan Localization for Untrusted Network-on-chips. In *IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–5. http://dx.doi.org/10.1109/LASCAS64004.2025.10966274.

Comarú, G. (2022). Proposal of a Secure Network Interface for Protecting IO Communication in Many-cores. End of Term Work (TCC). https://fgmoraes.github.io/docs/tcc/tcc_gustavo.pdf.

Comarú, G., Faccenda, R. F., Caimi, L. L., and Moraes, F. G. (2023). Secure Network Interface for Protecting IO Communication in Many-cores. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/SBCCI60457.2023.10261655.

Daoud, L. and Rafla, N. (2019). Detection and prevention protocol for black hole attack in network-on-chip. In *Symposium on Networks-on-Chip (NoCs)*, pages 1–2. https://doi.org/10.1145/3313231.3352374.

Dinechin, B. D. D., Amstel, D. V., Poulhiès, M., and Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *IEEE/ACM Design, Automation Test in Europe Conference (DATE)*, pages 1–6. https://doi.org/10.7873/DATE.2014.110.

Faccenda, R. F. (2024). *Securing Applications in NoC-based Many-Core Systems: A Comprehensive Methodology*. PhD thesis, PPGCC-PUCRS. 140p, http://tede2.pucrs.br/tede2/handle/tede/8917.

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.

Faccenda, R. F., Comarú, G., Caimi, L. L., and Moraes, F. G. (2023a). Lightweight Authentication for Secure IO Communication in NoC-based Many-cores. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. https://doi.org/10.1109/ISCAS46773.2023.10181962.

Faccenda, R. F., Comarú, G., Caimi, L. L., and Moraes, F. G. (2023b). SeMAP – A Method to Secure the Communication in NoC-based Many Cores. *IEEE Design & Test*, 40(5):42–51. https://doi.org/10.1109/MDAT.2023.3277813.

Fernandes, R., Marcon, C., Cataldo, R., Silveira, J., Sigl, G., and Sepúlveda, J. (2016). A Security Aware Routing Approach for NoC-based MPSoCs. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/SBCCI.2016.7724054.

Fochi, V. M. (2019). *Fault-tolerance at the Management Level in Many-core Systems*. PhD thesis, PPGCC-PUCRS. 108p, http://hdl.handle.net/10923/13396.

GAPH (2023). Hardware Design Support Group. www.inf.pucrs.br/gaph/.

Gondal, H., Fayyaz, S., Aftab, A., Nokhaiz, S., Arshad, M., and Saleem, W. (2020). A method to detect and avoid hardware trojan for network-on-chip architecture based on error correction code and junction router (ECCJR). *International Journal of Advanced Computer Science and Applications*, 11(4):581–586. http://dx.doi.org/10.14569/IJACSA.2020.0110476.

Hemani, A., Jantsch, A., Kumar, S., Postula, A., Öberg, J., Millberg, M., and Lindqvist, D. (2000). Network on chip: An architecture for billion transistor era. In *Nordic Circuits and Systems Conference (NORCHIP)*, pages 166–173. https://api.semanticscholar.org/CorpusID:18064630.

Hossain, N., Buyuktosunoglu, A., Wellman, J. D., Bose, P., and Martonosi, M. (2024). NoC-level Threat Monitoring in Domain-Specific Heterogeneous SoCs with SoCurity. In *International Symposium on Computer Architecture*, pages 1–8. http://prism.sejong.ac.kr/dossa-6/dossa_paper/dossa_2024_paper2.pdf.

Hussain, M., Malekpour, A., Guo, H., and Parameswaran, S. (2018). EETD: An energy efficient design for runtime hardware trojan detection in untrusted network-on-chip. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 345–350. https://doi.org/10.1109/ISVLSI.2018.00070.

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

Moraes, F. G., Calazans, N., Mello, A., Möller, L., and Ost, L. (2004). HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69 – 93. https://doi.org/10.1016/j.vlsi.2004.03.003.

Oracle (2017). Oracle's SPARC T8 and SPARC M8 Server Architecture. Technical report, Oracle Corporation. 44p. https://www.oracle.com/a/ocom/docs/sparc-t8-m8-server-architecture.pdf.

Peckham, O. (2020). Esperanto Unveils ML Chip with Nearly 1,100 RISC-V Cores. https://www.hpcwire.com/2020/12/08/esperanto-unveils-ml-chip-with-nearly-1100-risc-v-cores.

Popovici, K., Rousseau, F., Jerraya, A. A., and Wolf, M. (2010). *Embedded Software Design and Programming of Multiprocessor System-on-Chip: Simulink and System C Case Studies*. Springer Publishing Company, Incorporated, 290p.

Ramachandran, J. (2002). *Designing Security Architecture Solutions*. John Wiley & Sons, Inc., 483p.

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.

Ruaro, M., Caimi, L. L., Fochi, V., and Moraes, F. G. (2019). Memphis: a Framework for Heterogeneous Many-core SoCs Generation and Validation. *Design Automation for Embedded Systems*, 23(3-4):103–122. https://doi.org/10.1007/s10617-019-09223-4.

Sepúlveda, J., Zankl, A., Flórez, D., and Sigl, G. (2017). Towards protected MPSoC communication for information protection against a malicious NoC. *Procedia Computer Science*, 108:1103–1112. https://doi.org/10.1016/j.procs.2017.05.139.

Shakya, B., He, T., Salmani, H., Forte, D., Bhunia, S., and Tehranipoor, M. (2017). Benchmarking of Hardware Trojans and Maliciously Affected Circuits. *Journal of Hardware and Systems Security*, 1(1):85–102. https://doi.org/10.1007/s41635-017-0001-6.

Sharma, G., Kuchta, V., Sahu, R. A., Ellinidou, S., Bala, S., Markowitch, O., and Dricot, J. (2019). A twofold group key agreement protocol for NoC-based MPSoCs. *Transactions on Emerging Telecommunications Technologies*, 30(6):1–18. https://doi.org/10.1002/ett.3633.

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.

Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. C. (2016). Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46. https://doi.org/10.1109/MM.2016.25.

Tecnhlogies, M. (2018). TILE-Gx72 Processor Overview. http://www.mellanox.com/page/products_dyn?product_, Nov. 2018.

Tehranipoor, M., Anandakumar, N. N., and Farahmandi, F. (2023). *Hardware Security Training, Hands-on!* Springer. https://doi.org/10.1007/978-3-031-31034-8_6.

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. *Microelectronics Journal*, 68:69 – 77. https://doi.org/10.1016/j.mejo.2017.08.010.

Wang, H. and Halak, B. (2023). Hardware Trojan detection and high-precision localization in NoC-based MPSoC using machine learning. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 516–521. https://doi.org/10.1145/3566097.3567922.

Woszezenki, C. (2007). Alocação de Tarefas e Comunicação entre Tarefas em MPSoCs. Master's thesis, PPGCC-PUCRS. 121p, http://hdl.handle.net/10923/1500.

Zefferino, C. A. (2003). *Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho*. PhD thesis, PPC-UFRGS. 242p, https://lume.ufrgs.br/handle/10183/4179?show=full.

APPENDIX A – HARDWARE TROJAN TAXONOMY

The main reason enabling the insertion of HTs in a design is the distributed production chain adopted in the microelectronics industry, which allows designers to build a system with IPs from different design companies (third-party IPs – 3PIPs). Such a situation raises the question: "Is this foreign IP trustworthy?". The answer to this question is not simple since most IPs do not reveal their content or design process to protect intellectual property.

Figure A.1 shows the HT taxonomy proposed by Shakya et al. [2017]. The authors present six main characteristics to classify the different types of HT: Insertion phase; activation mechanism; abstraction level; effect; location; and physical characteristics.

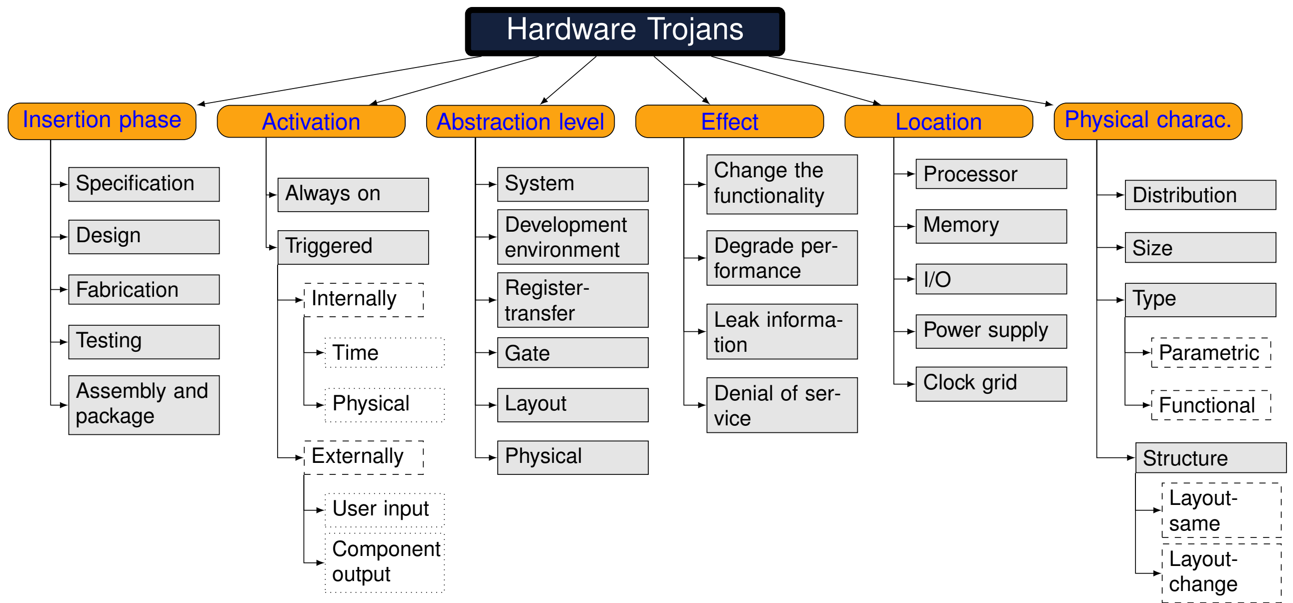


Figure A.1 – HT taxonomy presented by Shakya et al. [2017].

A.1 Insertion Phase

Inserting an HT can involve altering the design specification, such as manipulating tamperature to compromise design dependability. Tampering can occur during both the design and fabrication stages. This malicious activity can take the form of adding extra gates to a design's netlist or modifying its masks. HT insertion during the testing phase involves manipulating tests to conceal the presence of an inserted Trojan after fabrication.

Moreover, even in the presence of trustworthy individual chips, unprotected interconnections between them are susceptible to Trojan interference. An unshielded wire connection may introduce unintended electromagnetic signals, providing adversaries opportunities to exploit information leakage or inject faults.

A.2 Activation Mechanism

HTs may always function, or they get conditionally activated. Always-on Trojans start as soon as their hosting designs are powered-on while conditional Trojans seek specific triggers either internally or externally to launch. The internal triggers can be timing-based (an HT is activated after a certain time), or physical-condition-based (an HT is activated by certain internal events e.g. specific temperature or bus value). The externally triggered Trojans track user inputs or component outputs, and the Trojans get activated if activation conditions are met.

A.3 Abstraction level

The control of advisory influence on HT implementation is determined by the level of abstraction. At the system level, a design is articulated in terms of modules and their interconnections, with adversaries restricted to the modules' interfaces and interactions.

Moving to the development environment level, HTs can be inserted into modules by exploiting CAD register transfer tools and scripting languages. Each module is intricately described in terms of signals, registers, and Boolean functions at the register-transfer level, granting adversaries full access to functionality and implementation, allowing easy modifications.

Descending to the gate level, a design is depicted as a list of gates and their interconnections. At this level, adversaries can implement HTs with detailed control over gates and their interconnections. In the layout level, Trojans' impact on design power consumption or delay characteristics can be managed. Trojans can be actualized by altering the parameters of the original circuit's transistors.

Finally, at the physical level, all circuit components, along with their dimensions and locations, are determined. HTs can be inserted into the white/dead space of the design layout with minimal impact on design characteristics.

A.4 Effect

HTs exhibit distinct characteristics depending on their effects. For instance, they can alter a design's functionality by modifying elements like the data path of a processor. Additionally, HTs have the potential to diminish design performance or compromise reliability by manipulating various design parameters. In specific instances, a HT might lead to the unauthorized disclosure of a cryptographic processor's secret key or induce a denial of service for a requested service at a specific time.

A.5 Location

Any part of a design is potentially subjected to HT insertion. A Trojan may be dispersed across multiple components or concentrated within a single part. Its influence on a processor could involve tampering to gain control over its controller or data path units. In the case of a HT within memory, it has the capacity to alter stored values or impede read/write accesses to the memory. When present on a Printed Circuit Board (PCB) housing multiple chips, an I/O-inserted Trojan on the interfaces of these chips can disrupt communication. Furthermore, a HT can extend its impact to the design power supply, modifying current and voltage characteristics. Introducing interruptions to the clock grid, a HT can alter design delay characteristics. This includes freezing part of the clock tree and disabling specific functional modules.

A.6 Physical characteristics

The physical characteristics of HTs encompass diverse hardware manifestations. HTs can manifest as either functional or parametric types. Functional Trojans are brought about by the addition or deletion of transistors/gates, while parametric Trojans result from modifications to design parameters such as wire thickness. The size of a HT is determined by the quantity of transistors/gates added or removed. The distribution of HTs types signifies the spacing, either loose or tight, at which Trojan cells are positioned in the physical layout. Trojan structure pertains to potential alterations in the original physical design to accommodate the placement of HT cells.

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