IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2026, No. 4, pp. 36–61. DOI:10.46586/tches.v2026.i4.36-61

Turtle Walk: Stopping Timing Attack on NoC

Dipesh[1], Amit Kumar Singh[2] and Urbi Chatterjee[1]

[1] IIT Kanpur, India, {dipesh,urbic}@cse.iitk.ac.in
[2] University of Essex, UK, a.k.singh@essex.ac.uk

Abstract. In recent years, several countermeasures such as Gossip NoC, Ironhide, SurfNoC, Adaptive Routing, Random Arbitration, and Guard NoC have been proposed to mitigate micro-architectural timing attacks on Network-on-Chip (NoC) architectures. However, the security of these countermeasures has not been thoroughly validated due to the lack of a standardized testing framework. In this work, we first propose a generalized framework that enables seamless integration of NoC timing-attack countermeasures into the gem5 simulator, upon which representative timing attacks can be launched for systematic evaluation. The throughput values of adversarial packets in the network are then collected as *traces* and analyzed using Test Vector Leakage Assessment (TVLA) to detect the presence of timing leakage. Our analysis shows that the majority of existing countermeasures such as Adaptive Routing, Gossip NoC, and Random Arbitration are vulnerable to timing attacks as they have high TVLA score. Although countermeasures such as Ironhide and SurfNoC are secure, they restrict their applicability to hybrid topologies and routing algorithms, hence making them impractical for modern NoCs. Moreover, colluding timing attacks are not considered in the design of these countermeasures.
To overcome these limitations, we next introduce a novel lightweight countermeasure, *Turtle Walk*[1] to prevent a multi-PE attack, which leverages existing NoC hardware resources to ensure minimal overhead. It can be deployed on any modern NoCs, addressing the shortcomings of prior defenses. In *Turtle Walk*, hardware performance counters (HPCs) are used to prevent the attack. Existing HPC-based countermeasures suffer from two major limitations: (1) High false-positive rates due to the absence of a strong correlation between attack behavior and HPC values, and (2) Provide no capability to prevent attacks, even when such attacks are successfully detected. We overcome both limitations by establishing a strong correlation between the necessary conditions of a timing attack on NoC and HPC values, and by triggering the NoC flow-control mechanism based on these values to prevent the attack. Experimental results demonstrate: (1) minimal area overhead of 0.9% for ASIC and 0.3% for FPGA compared to baseline NI hardware without any countermeasure, and (2) no impact on *zero-load latency*, along with a remarkable $\approx 3\times$ reduction in average packet latency compared to state-of-the-art defenses.

Keywords: Multiprocessor System-on-Chip (MPSoC) · Network-on-Chip (NoC) · Timing Attacks on NoC · Countermeasure

1 Introduction

As modern Multi-Processor Systems-on-Chip (MPSoC) integrate hundred of cores, traditional bus architectures struggle to meet the high bandwidth demands of these massively parallel systems. Network-on-Chip (NoC) has emerged as a scalable and efficient interconnect solution, enabling seamless data movement among these cores. However, as NoCs

[1]The name is derived from the turtle, which provides strong protection through its hard shell, and reflects the mechanism of the countermeasure: slowing down the attacker's injection rate to a pace as slow as a turtle's walk.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2026-04-15 Accepted: 2026-07-15 Published: 2026-09-11

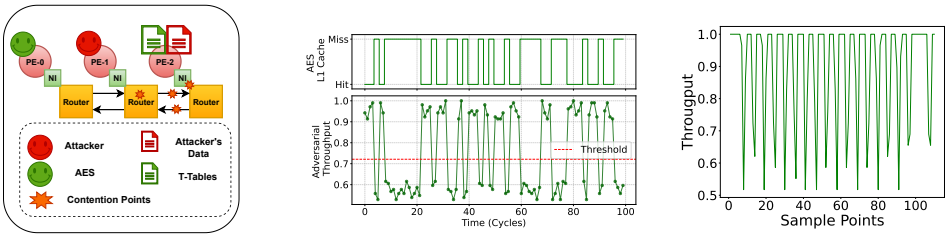


Figure 1: Timing attack on NoC: (a) Attack setup, (b) Correlation between AES L1 hit/miss and adversarial packet's throughput, and (c) Visible L1 hits of AES first round on adversarial packet's throughput at sample point 100

become the backbone of MPSoC communication, they also introduce new security vulnerabilities, particularly timing-based side-channel attacks (SCAs) [CM21, PLF21, WBZL22]. The processing elements (PEs) of the NoC can be categorized into two domains [NMB+16]: secure domain and non-secure domain, based on whether they are running a sensitive application like Advanced Encryption Standard (AES) [DR02] or not. Timing SCAs on NoC leak the sensitive information from the secure domain PE by creating contention on the shared NoC.

In a recent attack [DC25], as illustrated in Fig. 1, the attacker is able to extract the L1 hit/miss pattern of the private L1 cache of the secure domain PE running AES by observing the throughput of adversarial packets (packets injected by the attacker). A standard trace-driven strategy is then used to reduce the key search space from 2^{128} to 2^{8}. The attack is possible because the AES T-Table implementation performs memory operations based on the secret key, and these memory-request packets are injected into the NoC only when an L1 miss occurs. These packets create contention with the adversarial packets due to the shared hardware resources of the NoC, which in turn affect the throughput of the adversarial packets (See the large white gap in Fig. 1c at sample point 100, representing L1 cache hit). The attacker observes these variations in throughput to infer the hit-and-miss pattern of the private L1 cache of AES and subsequently recovers the secret key .

To mitigate timing attacks, existing countermeasures aim to disrupt the correlation between adversarial packet throughput and a victim's access to the NoC. These countermeasures can be broadly classified into three categories: (i) Isolation [WS12, OK20, SG11, WGO+13], (ii) Adaptive Routing [BK18, RSBS16], and (iii) Randomization [SDSG14, RAH+20, SFS+16]. However, a comprehensive security analysis of these countermeasures has not been conducted. To evaluate their security, we develop a generalized evaluation framework to confirm the presence or absence of timing leakage. First, the countermeasure under evaluation is integrated into the gem5 simulator by modifying the Garnet NoC module [AKPJ09]. Subsequently, a simulated timing attack based on existing attacks [DC25, RFZS18, RSBS16] is performed, and adversarial throughput values are collected as traces. These traces are then analyzed using Test Vector Leakage Assessment (TVLA) to statistically confirm the absence or presence of timing leakage (see Sec. 2, **#Contribution-1**). TVLA [SM16], a well-established methodology endorsed by the National Institute of Standards and Technology (NIST), provides statistical confidence regarding side-channel leakage [GGJR+11]. Our evaluation reveals the presence of timing side-channel leakage in countermeasures based on *Adaptive Routing* and *Randomization* [SDSG14, RAH+20, SFS+16, BK18, RSBS16]. While Isolation-based countermeasures [OK20, WGO+13, SG11] are secure against timing attacks, they suffer from significant practical limitations that render them unsuitable for modern NoC architectures (see Sec. 2.1.1).

This leads to a fundamental research question: *Can we design a countermeasure that guarantees the absence of timing side-channel leakage while incurring negligible performance degradation and minimal hardware overhead?*

We answer the above question by proposing a countermeasure named *Turtle Walk*, which detects and prevents the attack using existing hardware resources, ensuring minimal overhead while passing TVLA testing to confirm the absence of side-channel leakage (see Sec. 3, **#Contribution-2**). We identify two necessary conditions for a timing side-channel attack on the NoC: (1) communication with a secure-domain PE and (2) a high injection rate by the attacker to congest the path used by the victim's packets. These conditions are detected using hardware performance counters (HPCs) and the destination-address lookup features available in the processor and Network Interface (NI) of the MPSoC. To prevent the attack, we break the high-injection-rate condition using the NI's flow-control mechanism. *The main crux of this work is to demonstrate that complex hardware modifications are not necessary to mitigate a timing attack on the NoC if the root cause of the attack is effectively addressed.*

Although there are extensive work on the use of HPC for the detection of SCAs, majorly categorized as anomaly-based or signature-based. Anomaly-based methods use machine learning models to detect the attacks, whereas signature-based methods compare the application's behaviour with previous attack patterns. *These existing methods either fail to detect new attacks or have a high rate of false negatives as they are unable to establish a correlation between micro-architecture events and the malicious behavior of an application as shown in [KFC+24].* In NoC architecture specifically, we can overcome this limitation by establishing a strong correlation between the Misses Per Kilo Instruction (MPKI) of the L1 private cache and the high injection rate (see Sec. 3.4, **#Contribution-3**).

To provide a broader perspective, we would like to note that microarchitectural timing attacks can be broadly categorized into two classes: (i) eviction-based attacks, which target shared resources such as data and instruction caches, TLBs, BTBs, and LLCs; and (ii) contention-based attacks, which exploit contention on shared resources such as execution ports, functional units, and the NoC [PLF21]. As discussed above, the scope of this work is limited to contention-based timing attacks on the NoC. Accordingly, the MPKI metric used in *Turtle Walk* is intended only to detect such contention in NoC architectures and cannot serve as a universal detector for all microarchitectural timing channels.

To the best of our knowledge, this work is the first to prevent colluding timing attacks by directly addressing the root cause of timing side-channel leakage in NoCs by leveraging existing hardware components in MPSoCs and avoiding the need for complex hardware modules such as Guard NoC *and* SurfNoC.

In summary, our work has following contributions:

1. We develop a generalized timing-attack detection framework to evaluate the security of NoC countermeasures and systematically identify security limitations in existing defenses.
2. We propose a lightweight and effective countermeasure to detect and prevent colluding timing attacks on NoCs by establishing a strong correlation between microarchitectural events and timing SCAs on NoC. Our approach overcomes the limitations of existing HPC–based detection mechanisms designed for bus-based architectures. The security of the proposed countermeasure is validated using our framework.
3. We demonstrate both functional correctness and practical viability of *Turtle Walk* through implementation on real hardware using an *AMD Virtex UltraScale FPGA VCU108 Evaluation Development Kit.* Our results also show that *Turtle Walk* outperforms state-of-the-art countermeasures [RSBS16, AKPJ09, WGO+13, SDSG14, WS12, OK20, SG11, SFS+16] by effectively mitigating timing attacks while incurring exceptionally low overhead on ASIC platforms (only 0.9% area and 2.3% power) compared to a baseline network interface without countermeasures.

2 Timing Attack Detection Framework

The proposed generalized framework takes a NoC integrated with a countermeasure and validates the robustness of the countermeasure by performing timing attacks on the given NoC architecture. To validate the robustness of a countermeasure, we must confirm the absence of timing side-channel leakage on the NoC. In existing literature, the detection of side-channel leakage is typically confirmed using TVLA [SM16] or analyzed using machine learning (ML)-based classifiers [WMM19]. Although ML models are capable of detecting non-linear leakage with fewer traces, they suffer from overfitting issues due to the unavailability of standardized datasets and are not considered an industry standard, unlike TVLA, which is endorsed by the NIST [GGJR+11].

Despite the ability of ML techniques to detect non-linear leakage with fewer traces, TVLA is preferred because NoC timing-attack leakage is linear in nature [WS12]. The attacker attempts to establish a correlation between the throughput of its own packets and the bandwidth consumption of the victim by creating contention between adversarial packets and victim packets. Since the NoC bandwidth is fixed at design time, an increase in the victim's bandwidth consumption results in a decrease in the bandwidth available to the attacker, thereby reducing the throughput of adversarial packets. This establishes a linear correlation between the victim's NoC utilization and the adversarial packet throughput [WS12]. To confirm such a correlation, NIST also adopts TVLA over ML-based models due to the statistical rigor and interpretability of TVLA [GGJR+11]. Moreover, minimizing the number of traces is not a concern in our case, as our goal is to evaluate the robustness of a countermeasure. Therefore, TVLA is the more appropriate choice in this context. The TVLA is also classified into two types: non-specific and specific. *Non-specific TVLA* partitions traces based on public inputs, while *specific TVLA* partitions traces based on intermediate key-dependent variables, offering deeper insight into the source of leakage [RBG+16]. We adopt specific TVLA to rigorously assess the presence of leakage, making the evaluation of the countermeasure more stringent.

2.0.1 Threat Model

The framework is created based on the following threat model, adopted from existing works [DC25, RFZS18, RSB+17]. There is an attacker operating in a non-secure domain with user-level privilege and a victim (AES) residing in a secure domain. In addition, the adversary is assumed to know the NoC architecture parameters, including topology, routing algorithm, and deployed countermeasures. The adversary can inject packets into the NoC via memory accesses while measuring the throughput of their own packets. The goal of the attacker is to infer the victim's NoC utilization by conducting a timing attack on the NoC. The threat model can have the following variants:

1. **Single-PE or Multi-PE Threat Model:** In the single-PE model, only one PE of the MPSoC is under the attacker's control, whereas in the multi-PE model, two or more PEs are compromised and perform a colluding timing attack.
2. **Throughput or Latency Probing:** In throughput probing, the attacker computes the throughput by counting number of packet replies over a fixed interval, whereas in latency probing, the attacker can measures the round-trip time (RTT) of each of its packets.

The existing literature [DC24, RFZS18, AK21, RSB+17] considers only a single-PE threat model with throughput probing. However, in this work, we are the first to consider a multi-PE threat model with throughput probing, in which multiple processing elements inject low-rate traffic to collectively congest the path traversed by the victim. In a multi-PE attack, two scenarios are possible: (1) only one PE monitors its throughput for analysis, or (2) all PEs monitor their respective throughput values in a synchronized manner, after which the measurements are combined and aligned for analysis. As shown in Fig. 2a,

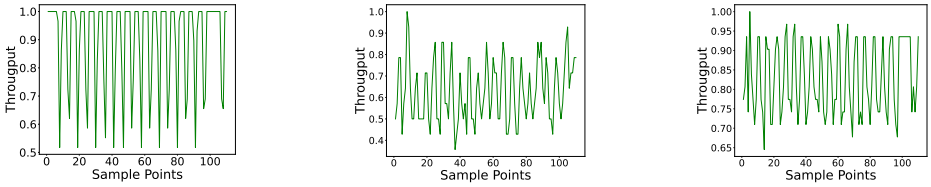


Figure 2: Adversarial packet's throughput during first round of AES: (a) Single PE with injection rate 1, (b) Multiple PE non-synchronized traces, and (c) Multiple PE synchronized traces

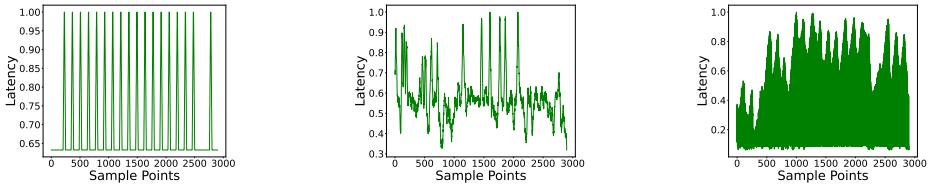


Figure 3: Adversarial packet's latency during first round of AES: (a) Single PE with injection rate 1, (b) Multiple PE non-synchronized traces, and (c) Multiple PE synchronized traces

under ideal conditions, the effect of an L1 cache hit is clearly observable in the throughput of adversarial packets (at sample point 100). This effect is not visible under the same configuration when multiple PEs monitor their throughput in a non-synchronized manner (see Fig. 2b). However, when multiple PEs coordinate to monitor their throughput in a synchronized manner, the impact of the L1 cache hit becomes observable again (refer to Fig. 2c). This highlights the necessity for all participating PEs to monitor their respective throughput values in a synchronized fashion. The latency probing is not considered for the following reasons:

- Measuring the latency of each packet requires extremely high temporal resolution, which is constrained by hardware capabilities [HRY24, PTV21]. Accurate per-packet latency measurements require a dedicated measurement thread per injected packet, and running many concurrent measurement threads on a single core is impractical due to hardware constraints. In contrast, throughput measurements can be obtained using only two concurrent threads (one to measure the interval and another to count replies).
- Combining latency measurements from multiple processing elements in a multi-PE attack is significantly less feasible than combining throughput measurements. A similar pattern is observed in throughput probing under both single-PE and multi-PE synchronized traces (see Fig. 2); however, this similarity is absent in latency-based probing (see Fig. 3), even under ideal conditions for the attacker.

2.1 Leakage Assessment Framework

① **Deployment:** To construct a leakage analysis framework, three components are required: (i) a mechanism to integrate the countermeasure into the NoC architecture, (ii) the ability to program PEs to replicate the attack scenario, and (iii) an automated process to collect traces for leakage detection. To integrate countermeasures with the NoC, we modify the Garnet module of the gem5 simulator, which models the behavior of the on-chip interconnection network within gem5. The required modifications to the Garnet module for integrating

existing countermeasures are discussed in Sec. 2.1.1. To program the PEs and replicate the attack, we modify the files in the `src/cpu/testers/garnet_synthetic_traffic` directory in gem5, which control the packet injection patterns of PEs in the NoC. Based on the node ID, one PE is designated as the victim operating in the secure domain. The victim executes AES with a fixed secret key and plaintext and injects packets into the NoC upon observing an L1 cache miss. One or more other PEs act as attackers by continuously injecting packets and monitoring the throughput of their own traffic. The throughput values of adversarial packets are collected at regular time intervals. To model background traffic, we introduce a `noise` parameter for the remaining PEs, which inject packets at random times and random destinations according to the specified noise level. To automate the leakage analysis, we develop a Python-based framework that simulates the NoC with the integrated countermeasure, executes the attack, and analyzes the collected throughput data of adversarial packets.

② **Leakage Analysis:** The analysis proceeds in multiple stages. First, we test the variation in adversarial throughput with and without the countermeasure using the same AES key. If no variation is observed, the countermeasure is considered ineffective. In the second stage, the attacker repeats the experiment using two or three random AES keys and checks for observable variations; if no variation is present, the countermeasure is deemed secure. Otherwise, run-time TVLA [SM16] is employed to determine whether a correlation exists between adversarial packet throughput and the victim's L1 cache hit/miss pattern, as exploited in [DC25]. Specifically, a run-time TVLA is computed using Algo. 1 to assess this correlation. The throughput traces are divided into two sets: one set where the number of L1 cache hits equals a chosen value i, and another set where the number of L1 hits differs from i. If the TVLA exceeds 4.5 for 10k traces, leakage is confirmed with 99.99% confidence as shown in [SM16]. Otherwise, the number of measurements is increased. At any point beyond 10k traces, if the TVLA value exceeds 4.5, the analysis is terminated, the countermeasure is declared insecure, and the corresponding TVLA plot is generated. To ensure robustness, a countermeasure is considered secure only if the TVLA remains below 4.5 even after collecting up to 1 million traces.

③ **Validation:** To validate the proposed framework, we conduct an initial study by executing the same experiment on both the gem5 simulator and an FPGA platform (Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit). A 4×4 standalone NoC is implemented in Verilog and deployed on the FPGA. The NoC comprises 5-port routers employing XY routing, FIFO buffers with a depth of 4, and round-robin arbitration. Request–reply mechanism-based PEs are used instead of RISC-V cores, which can send requests to any other PE in the NoC and generate replies to incoming requests. The framework is validated by generating synthetic traffic, where one PE mimics the behavior of an AES module by emulating an L1 hit/miss pattern. Additionally, another PE acts as an attacker by injecting packets into the NoC at a controlled injection rate and monitoring the throughput of its own packets. The objective is to validate that the variation in adversarial packet throughput, caused by contention with victim packets, is consistent between the hardware platform and the proposed framework. The impact of victim traffic on adversarial throughput in both platforms is illustrated in Fig. 4. The observed behavior is largely consistent across the two environments, with only minor deviations on the FPGA due to hardware-intrinsic factors such as Dynamic Voltage and Frequency Scaling (DVFS) and power gating. These factors introduce throughput fluctuations even in the absence of contention, thereby making the attack more challenging [DC25]. Based on these observations, we conclude that gem5 is suitable for security analysis, as it ensures that timing variations in adversarial traces arise solely from contention with victim packets rather than from hardware-intrinsic effects.

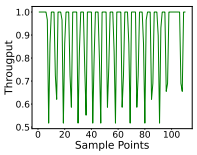
Algorithm 1 TVLA Test

Require: `Data Set`
1: **for** x **in** `Data Set` **do**
2: **if** `L1 hits` $= i$ **then**
3: **Bin_A.append(**$Average_Throuput$**)**
4: **else**
5: **Bin_B.append (**$Average_Throuput$**)**
6: **end if**
7: **end for**
8: $TVLA = \left| \frac{X_A - X_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}} \right|$
9: ▷ (X_A, σ_A^2, N_A) and (X_B, σ_B^2, N_B) represent the mean, variance, and number of points for bins A and B, respectively.

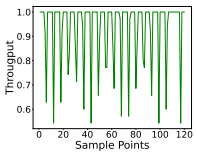
2.1.1 Security Analysis of Existing Countermeasures using the Framework

To evaluate the security of existing countermeasures, we integrate each defense in our proposed framework by modifying the Garnet network and analyze its effectiveness. To rigorously assess the resilience of a countermeasure, we further consider a best-case scenario for the attacker by assuming a *noise-less environment*. Specifically, the noise parameter is set to zero, and all processing elements except the attacker and the victim are assumed to behave ideally. Since many countermeasures rely on injecting noise to mitigate timing side-channel attacks [ASK23], eliminating all external noise sources ensures that any observed variation in adversarial throughput arises solely from contention with the victim's packets. Consequently, no other processing element interferes with the adversarial traffic. These conditions are essential for validating the absence of timing side-channel leakage in the NoC. The analysis is independent of the routing algorithm, topology, and network size. However, for clarity, only results obtained from an 8×8 NoC with a Mesh topology and XY routing are presented, for the following reasons:

- The initial analysis is conducted on NoCs ranging from 3×3, 4×4, to 8×8 with Mesh topology using both XY and YX routing algorithms, as well as a Point-to-Point topology. Similar results are obtained across all configurations because contention is created at the output port of the router or NI, which behaves similarly across architectures.
- Existing attacks [RFZS18, RSB+17, DC25, DC24] and countermeasure evaluations are also presented for Mesh topology with XY routing, as it is widely adopted in commercial NoCs [Sod15, WGH+07].
- gem5 does not support configurations beyond this scale.



(a)



(b)

Figure 4: Adversarial packet's latency during first round of AES on: (a) gem5 (b) FPGA

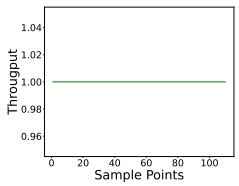


Figure 5: Isolation-based countermeasure

The security analysis of existing countermeasures is as follows:

ⓐ **Isolation-Based Countermeasures:** These countermeasures work by partitioning the hardware so that, at any given time, only packets from a single domain can utilize the hardware resources. Reversed Priority with Static Limits (RPSL) [WS12] prioritizes non-secure domain packets over secure ones. As a result, the throughput of the attacker's packets is not affected by secure domain traffic, causing the attack to fail. However, this prioritization can lead to denial-of-service (DoS), as the attacker may monopolize the available

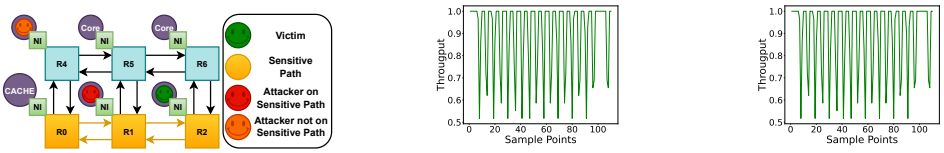


Figure 6: Adaptive routing: (a) Attack setup, (b) Attacker on sensitive path, and (c) Attack not on sensitive path

bandwidth. To mitigate this, bandwidth quotas are enforced, allocating 80% to non-secure and 20% to secure domains. *Ironhide* [OK20] allocates separate NoC zones to each domain. In [SG11], Time Division Multiple Access (TDMA) is implemented, allowing packets from different domains to use the NoC at distinct time intervals. *SurfNoC* [WGO+13] proposes a more efficient version of TDMA, where instead of reserving all the routers for a single domain at a time, routers are scheduled in a cyclic order across domains. This approach improves zero-load latency.

- *Implementation:* We implement Ironhide [OK20] by bifurcating the NoC into two zones and constraining packets to remain within their respective zones. To implement RPSL [WS12], TDMA [SG11], and SurfNoC [WGO+13], we modify the `Network Interface.cc` file of the Garnet module, assigning different virtual networks: `vnet-1` for secure domain packets and `vnet-0` for non-secure domain packets. Additionally, we modify `Switch Allocator.cc` by updating the `arbitrate_inports()` function (refer to Listing 1).
- *Observation*: As shown in Fig. 5, the isolation-based countermeasures effectively prevent contention, as hardware resources can only be utilized by packets from a single domain.

```
// RPSL
while (((get_vnet(invc)==0 && m_router->curCycle()%10>=2) || (get_vnet(invc)==1 &&
    m_router->curCycle()%10<2))) { invc++; }
// TDMA
while (((get_vnet(invc)==0 && m_router->curCycle()%2==0) || (get_vnet(invc)==1 &&
    m_router->curCycle()%2==1))) { invc++; }
// SurfNoC
if (m_router->get_id()%2==0) {
    while ((get_vnet(invc)==0 && m_router->curCycle()%2==0) || (get_vnet(invc)==1 &&
        m_router->curCycle()%2==1)) { invc++; }}
else {
    while ((get_vnet(invc)==0 && m_router->curCycle()%2==1) || (get_vnet(invc)==1 &&
        m_router->curCycle()%2==0)) { invc++; }}
```

Listing 1: Countermeasure Implementation of Isolation-Based or Priority-Based Schemes

ⓑ **Adaptive Routing:** Adaptive routing aims to prevent contention by removing the attacker from the path used by packets of the secure domain PE *(sensitive path)*. In *Gossip NoC* [RSBS16] and *Non-Interference based Adaptive Routing (NIBR)* [BK18], routers detect abnormal congestion and re-route packets to avoid contention between secure and non-secure domain traffic.

- *Implementation:* We evaluate adaptive routing-based countermeasures [RSBS16,SDSG14, BK18] by considering two positions of attacker, as shown in Fig. 6a. One attacker position is on the sensitive path, while the other is not.
- *Observation*: As shown in Fig. 6, we observe no difference in the throughput drop of adversarial packets, regardless of whether the attacker is on the sensitive path or not, as long as there is a shared output port accessed by packets from both domains. This condition is always possible since, according to the adversarial model in existing attacks [RFZS18, RSB+17, DC25], the attacker can request the same destination as the victim. Hence, adaptive routing fails to prevent timing attacks on NoC.

ⓒ **Randomization**: This technique introduces unpredictability to foil attackers. In [SDSG14],

random arbitration is implemented to prevent the attack by introducing randomness into the attacker's throughput data. *Guard NoC* [RAH+20] delays responses for fixed *(worst-case binding variant)* or random duration *(masking variant)* by holding them at the NI. *SER* [SFS+16] uses a global buffer pool with randomized allocation, ensuring buffer availability and throughput stability.

- *Implementation:* The SER [SFS+16] is implemented by creating extremely large buffers at the input port, where the size of each virtual channel (VC) has been increased from 4 to 25 flit buffers per VC. Guard NoC *(worst-case binding)* is implemented by delaying packet replies by 100 clock cycles, while Guard NoC *(masking)* introduces a random reply delay ranging from 0 to 256 clock cycles [RAH+20]. We try to implement a 15-bit masking technique, which is considered more secure. However, introducing a random reply delay ranging from 0 to 32,768 clock cycles results in a deadlock, causing the simulation to fail. Moreover, it is impractical to halt a system for such an extended period to ensure security, making these highly secure systems infeasible for practical use. Random Arbitration [SDSG14] is implemented by adding a random delay ranging from 0 to 15 clock cycles at each router through modifications to the `arbitrate_outports()` function in `SwitchAllocator.cc` of Garnet module.
- *Observation:* We observe that extremely large buffers at the input port have no impact on mitigating the attack, as they eventually fill up over time due to the high injection rate of the attacker (see Fig. 7a). The Guard NoC *(masking)* and Random Arbitration try to disrupt the correlation between throughput data and secret key values (refer to Fig. 8a). To validate leakage, we conduct a TVLA test, and the results are shown in Fig. 8b confirm the leakage as $TVLA > 4.5$ and attacker is still able to recover the key with a large set of traces. However, the attack is not possible in presence of Guard NoC *(worst-case binding)* (see Fig. 7b).

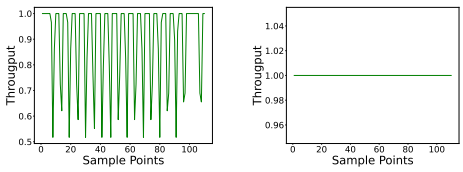


Figure 7: Adversarial throughput variations with countermeasure: (a) SER (b) Guard NoC (worst-case binding)

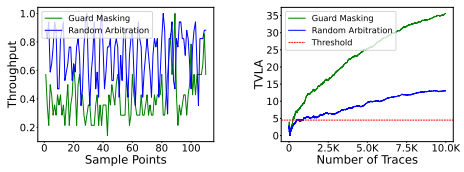


Figure 8: Countermeasure Guard NoC (masking) and random arbitration: (a) Throughput variations (b) TVLA

3 Turtle Walk

The security analysis (refer to Sec. 2.1.1) of existing countermeasures shows that *adaptive routing* and *randomization* techniques fail to prevent timing attacks. Guard NoC *(worst-case binding)* and Isolation-based countermeasures are the only ones that remain effective; however, they inherit significant limitations. For example, *RPSL* permits DoS attacks with up to 80% system degradation; *Ironhide* assumes only two applications with pre-profiled MPKI, which is impractical for dynamic workloads; *SurfNoC* is topology-specific, limiting its applicability in hybrid topologies; and *TDMA* suffers from severe latency explosion as the number of domains increases. Moreover, the *Guard NoC (worst-case binding)* [RAH+20] increases the memory access time equivalent to fetching data from the main memory and completely negating the benefits of having a cache. This significantly degrades overall NoC performance. These observations indicate that complex hardware modifications used in adaptive routing and randomization–based countermeasures are

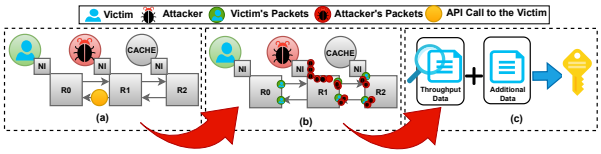


Figure 9: Generalized Attack Phases: a) Setup, b) Attack, and c) Analysis

ineffective, while simpler solutions are more effective but suffer from deployability issues. To overcome these limitations, countermeasures should meet the following design goals:

a) **No Edge to the Attacker:** The countermeasure should not allow the attacker to degrade system performance.
b) **No Constraint on Use Case:** The design should support multiple applications and handle unknown workloads.
c) **Minimal Overhead:** The hardware and performance overhead should be negligible.
d) **No Architecture Dependency:** The countermeasure should integrate seamlessly with any NoC architecture.
e) **Robustness:** The robustness of the countermeasure should be validated using statistical tests such as TVLA.

To achieve these goals, it is essential to understand the attack process, as each attack follows a well-defined sequence of phases. In this section, we create a generalized attack process based on existing works [RFZS18, RSB+17, DC25] and then identify how each phase of the attack can be detected and prevented accordingly.

3.1 Generalized Attack Process

A generalized attack process based on the threat model and existing works [RFZS18, RSB+17, DC25] is shown in Fig. 9 (a), (b), and (c), and their working is as follows:

a) *Setup Phase:* In this phase, the attacker infects the desired PEs on the MPSoC by running a malicious program [RFZS18, RSB+17, DC25]. Since timing attacks on the NoC follow a chosen-plaintext (CPA) or chosen-ciphertext (CCA) model, a crafted plaintext or ciphertext must be sent to the victim by one of the attacker-controlled PEs. This plaintext/ciphertext is sent through the API call provided by the AES module for encryption or decryption before the start of the attack.
b) *Attack Phase:* Next, the attacker congests the network path used by the victim's packets by injecting traffic into the NoC at a very high rate. The attacker can use a single PE or multiple PEs to create this congestion. It aims to generate meaningful contention with the victim's packets while monitoring its own throughput. This throughput data is used to in the analysis phase.
c) *Analysis Phase:* Finally, the attacker analyzes its throughput data along with additional information from the crafted plaintext or ciphertext to recover the secret key. The throughput data is first converted into an L1 hit/miss pattern [DC25], after which standardized cache-attack techniques are applied to derive the secret key.

3.1.1 Necessary Conditions for Timing Attack

Based on the threat model (refer to Sec. 2.0.1) and the attack process, there are three necessary conditions that must hold true for the attack to succeed:

1) There must be an initial communication between the attacker and the victim before the start of the attack to send the crafted plaintext or ciphertext.
2) The attacker must be able to inject packets into the NoC at a high injection rate to create meaningful contention with the victim's packets. This contention establishes a

direct correlation between the throughput of adversarial packets and the victim's L1 hit/miss pattern (refer to Sec. 3.4).

3) In the case of a multi-PE attack, all compromised PEs must communicate with each other at regular intervals or be controlled by a master PE to monitor their throughput values in a synchronized manner. Asynchronous monitoring results in misaligned traces that fail to reveal any meaningful information (refer to Fig. 2). This requires a communication through shared memory. Communication through a covert channels is slower and may lead to misalignment of traces and failure of the attack.

3.2 Detection and Prevention

We use the functionality of the NI and HPCs in *Turtle Walk* to effectively detect and prevent these necessary conditions, thereby stopping timing attacks on the NoC.

3.2.1 Detection

① **Initial Communication:** To detect the initial communication between the attacker and the victim, we use the Destination Address Lookup feature available inside the NI. The victim resides in the secure domain of the MPSoC, and according to the system's security policy, API calls to the victim must be encrypted since the NoC is vulnerable to eavesdropping and spoofing attacks [CM21]. A lightweight block cipher is present in the NI to facilitate the encryption/decryption of packets destined for a secure-domain PE. The Destination Address Lookup feature is used to identify such destinations, and all packets matching these addresses are encrypted using the block cipher. *Turtle Walk* is also activated once the block cipher of the PE is invoked, as it is considered the end of the setup phase in case the PE is malicious.

② **High Injection Rate:** To detect the high injection rate, the HPCs present inside each PE are used. A PE generates multiple packets into the NoC for coherence protocol messages, system calls, interrupts, peripheral access, and remote memory accesses. However, the attacker is merely a malicious program running inside a PE, and to mount a timing attack it must not only generate packets at a high rate but also monitor the throughput of its own injected packets. Instructions such as load/store operations, non-temporal load/store, `CLFLUSH`, and cache-coherence messages can all be used for this purpose. However, based on prior work [DC25, WBZL22, PLF21], the only practical approach for conducting timing attacks is to generate memory accesses using load/store instructions and measure the time taken to receive their responses. A detailed analysis of why other instruction types are not suitable for effective timing attacks is provided in Sec. 3.4.

Crucially, the corresponding load/store instruction data must reside in the L2 cache bank, which also serves as the last-level cache (LLC); if the data is present in the L1 cache, no packet is injected into the NoC and the reply is served directly from the L1 cache. Therefore, to generate packets into the NoC at a high rate while being able to monitor throughput, the attacker must generate repeated memory requests, each of which must result in an L1 cache miss.

The attacker achieves this by allocating an array larger than the L1 cache size and accessing indices that map to different cache blocks. To evade the cache prefetcher, the attacker further increases the array size beyond a page (4 KB) and accesses elements from alternate pages. As a result, the attacker injects packets at a very high rate. This leads to an exceptionally high increase in HPC values corresponding to the number of load/store instructions and the number of L1 misses within a short time. This behavior is not possible for a benign application (refer to Sec. 3.4). We use the rate of L1 misses per thousand instructions, or Misses Per Kilo Instructions (MPKI), to determine whether an application is injecting packets at a high rate.

③ **Synchronized Trace Collection:** To detect synchronized trace collection in a multi-PE attack, we identify communication between groups of PEs using kernel support. To achieve synchronization, the PEs must employ either hardware or software barriers and begin collecting throughput measurements only after all PEs have reached the barrier. Such barriers can be implemented using standard primitives such as `pthread_barrier_wait()`, or via custom synchronization code. In both cases, synchronization requires a shared variable that is accessed by all participating PEs to signal arrival at the barrier and to wait until all PEs have reached the synchronization point. The kernel tracks all PEs whose tasks that have a shared variable through shared-memory structures `(mm_struct)` and thread groups `(TGID)`. We use this information to group all PEs belonging to the same application, and the combined injection rate (or MPKI) of these groups is then used to distinguish malicious groups from benign ones.

3.2.2 Prevention

To prevent the timing attack, at least one of the three necessary conditions must be eliminated. However, Condition 1 cannot be eliminated because many benign applications legitimately communicate with secure-domain PEs for encryption and decryption. Similarly, Condition 3 cannot be eliminated, as many benign applications share address spaces. Therefore, to prevent the attack, Condition 2 must be eliminated, since no benign application exhibits such a high injection rate or MPKI (refer to Sec. 3.4).

The attacker relies on a high injection rate to create meaningful contention. To prevent the attacker from creating this contention, *the injection rate of the PE is throttled using the source-throttling mechanism* of the NI. Source throttling controls congestion on the NoC by adjusting the PE's injection rate via the flow-control module. This is achieved by comparing the current injection rate of the PE with the allowed injection rate. If the current injection rate exceeds the allowed threshold, packet injection is temporarily blocked until the rate falls below the threshold. To prevent the attack, attacker's injection rate remains throttled until the adversarial application terminates on the PE. In the case of a multi-PE attack, the injection rates of all malicious PEs detected by the kernel are reduced below the threshold, with each PE receiving an equal portion of the permitted bandwidth.

To determine the throttled injection rate that prevents the attacker from creating meaningful contention, we gradually decrease the injection rate from 100% to 0% in intervals of 10%, and compute the TVLA using Algo. 1. If the TVLA remains below 4.5 for one million traces, we confirm that the attacker is unable to create meaningful contention or extract any information via timing attacks (refer to Sec. 2.1). In the single-PE attack, an injection rate below 60% yields a TVLA below 4.5 (refer to Fig. 10). Multi-PE attacks are less synchronized than single-PE attacks, as each PE generates packets according to an independent distribution, which introduces variability in the aggregated throughput data (see Fig. 2a and Fig. 2c). To validate the impact of colluding parties on the leakage level, we vary the number of colluding parties while keeping the injection rate the same. The TVLA analysis (see Fig. 11a) shows that the quality of traces degrades as the number of colluding parties increases, making the attack more difficult. In Fig. 11d, TVLA is computed for three colluding parties (based on the attack in [DC25]), showing that an injection rate below 70% is sufficient to achieve a TVLA below 4.5. Therefore, we set the injection-rate threshold to 60% to protect against both single-PE and multi-PE attacks. This threshold is computed based on the latencies of routers and NIs, modeled as 2 and 5 clock cycles, respectively, following prior work [RSB+17]. The difficulty of inducing contention depends on these architectural parameters, and variations in them can shift the required injection-rate threshold.

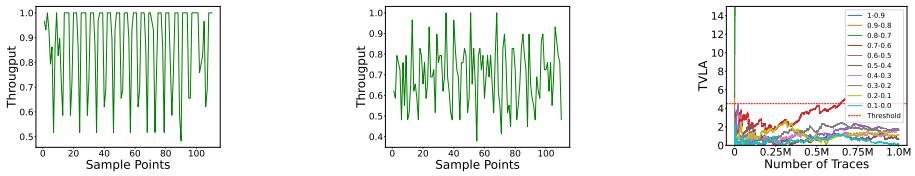


Figure 10: Turtle Walk under singe-PE attack: (a) Throughput variation above threshold, (b) Throughput variation below threshold, and (c) TVLA

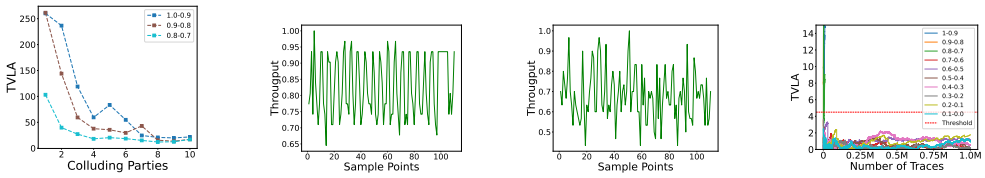


Figure 11: Turtle Walk under multi-PE attack: (a) TVLA vs colluding parties (a) Throughput variation above threshold, (b) Throughput variation below threshold, and (c) TVLA at different aggregate injection rates

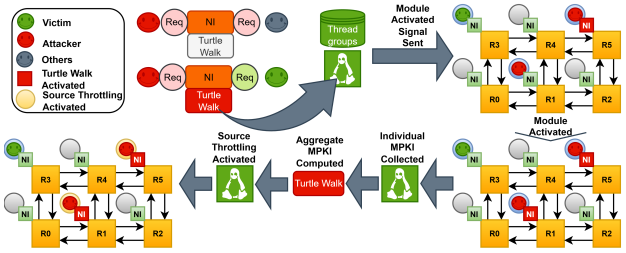


Figure 12: Turtle Walk Workflow

3.2.3 Workflow

The high-level overview of *Turtle Walk* is shown in Fig. 12, and the algorithmic workflow corresponding to the implementation is provided in Algo. 2. The *Turtle Walk* module is activated once the block cipher of any PE is invoked to send an encrypted request to a PE in the secure domain (`Line 3-5`). In a multi-PE attack, *Turtle Walk* issues a system call to invoke the kernel, which identifies all PEs running applications that share memory with the PE whose block cipher has been invoked, and activates the *Turtle Walk* modules on those PEs accordingly (`Line 10-11`).

Each *Turtle Walk* module computes the MPKI of its respective PE at regular intervals (`Line 12-14`). The kernel then collects these individual MPKI values and returns them to the *Turtle Walk* module that initiated the system call. This step is not required for a single-PE attack (`Line 7-9`). The net MPKI is computed from the aggregated individual MPKI values (`Line 13`). If the net MPKI exceeds a predefined threshold, another kernel signal is issued to throttle the injection rates of all activated PEs, ensuring that each receives an equal share of bandwidth (`Line 17-21`). In contrast, for a single-PE attack, only the injection rate of the corresponding PE is throttled. The *Turtle Walk* module is automatically deactivated upon a context switch, and all module parameters are reset to their default values.

Algorithm 2 Turtle Walk

Require: `Enable`, `L1 Misses`, `Instruction Retired`, `Threshold`
1: `Block Flag` $\leftarrow 0$
2: **if** `Enable` $= 1$ **then**
3: **if** *Block Cipher is Invoked* **then**
4: `Block Flag` $\leftarrow 1$
5: **end if**
6: **if** `Block Flag` $= 1$ **then**
7: **if** `Threat Model` $=$ `Single PE` **then**
8: $MPKI \leftarrow \frac{\texttt{L1 Misses}}{\texttt{Instruction Retired}} \times 1000$
9: **else if** `Threat Model` $=$ `Multi PE` **then**
10: Raise Interrupt to Kernel
11: `PE Group` $\leftarrow$ `GetPEsSharingMemory(mm_struct, TGID)`
12: **for all** `PE` in `PE Group` **do**
13: `MPKI[PE]` $\leftarrow \frac{\texttt{L1 Misses[PE]}}{\texttt{Instruction Retired[PE]}} \times 1000$
14: **end for**
15: `MPKI` $\leftarrow$ `Aggregate(MPKI[PE Group])`
16: **end if**
17: **if** `MPKI` $\geq$ `Threshold` **then**
18: Dropped injection rate
19: **else**
20: Default Injection Rate
21: **end if**
22: **end if**
23: **end if**

3.3 Architecture

The *Turtle Walk* is integrated with the PE and other sub-modules of the NI for its functioning (as shown in Fig. 13a). We propose four variants of the *Turtle Walk* module (as shown in Fig. 13b) based on whether the MPKI is computed by the processor or within the module and whether the system is vulnerable to a single-PE or multi-PE threat model. These also provide alternatives to choose based on importance of performance or hardware area (cost).

As shown in Fig. 13b, each *Turtle Walk* variant consists of the following components: ① `Enable Signal:` Controlled by the secure kernel to activate or deactivate the module. ② `Block Flag:` Identifies the invocation of the block cipher using the *Destination Address Lookup* module. ③ `MPKI:` A submodule that computes the MPKI using the `L1 Miss` and `Instruction Retired` values from the HPCs, as shown in Algo. 2 `(Line 8)`. ④ `Comparator:` Used to check whether the MPKI value exceeds a given threshold. ⑤ `Adder:` Included in the `V3` and `V4` variants to compute the net `MPKI` for multi-PE attacks. The details of each variant are as follows:

1) *Turtle Walk (V1)*: The `V1` variant is for single-PE threat model. It fetches the values of HPC from the processor and computes the MPKI within the module. As the MPKI is computed within the module, there will be hardware overhead, but the module will not interrupt the processor, ensuring no degradation in processor throughput.
2) *Turtle Walk (V2)*: The `V2` variant is also for single-PE threat model. It generates an interrupt signal to the processor to compute the MPKI. The `Counter` value of the flow control module is used to generate the interrupt signal after a fixed interval. As the MPKI is not computed within the module, there will be less hardware overhead, but the throughput of the processor is degraded.
3) *Turtle Walk (V3)*: The `V3` variant targets the multi-PE threat model. It is similar to the `V1` variant, with the addition of an adder within the module to compute the net MPKI of all compromised PEs. This variant also includes a kernel interrupt signal used to activate the *Turtle Walk* modules of all compromised PEs in order to fetch their respective MPKI values. Although this variant incurs the highest hardware overhead due to the inclusion of the MPKI module and the adder, it does not interrupt the processor, similar to the `V1` variant, thereby ensuring no degradation in processor throughput.
4) *Turtle Walk (V4)*: The `V4` variant also targets the multi-PE threat model and is based

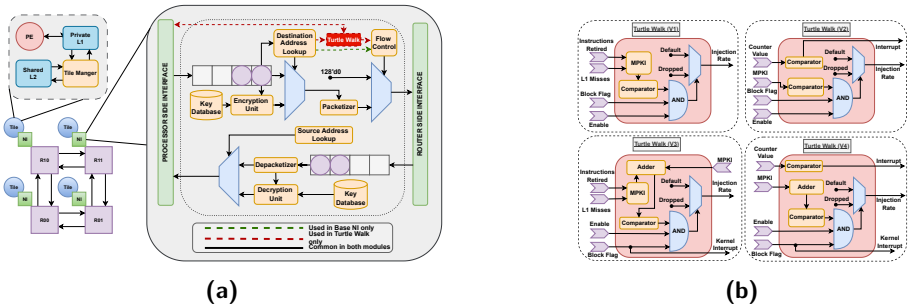


Figure 13: Turtle Walk: (a) Integration inside NI of MPSoC, and (b) Variants

on the `V2` variant, where MPKI is computed by the processor through an interrupt signal. In addition to the kernel interrupt signal, an adder is included, similar to the `V3` variant to fetch and compute the net MPKI of all compromised PEs. This variant has a higher hardware overhead than the `V2` variant due to additional adder but lower than the `V3` variant. However, it results in processor throughput degradation similar to the `V2` variant.

3.4 Correlation Between HPC Values and NoC Timing Attack

To ensure the detection of no false negatives, it is essential to establish a correlation between HPC values and NoC timing-attack behavior. Although HPC-based countermeasures are widely used to detect micro-architectural attacks on CPUs and GPUs, they face several limitations. Initially, threshold-based HPC countermeasures were proposed, where an application is considered malicious if an HPC value exceeds a predefined threshold [Pay16]. However, attackers can evade this strategy by slowing down the attack to keep HPC values below the threshold [KFC+24].

Subsequently, ML-based models were introduced, trained using HPC values to detect SCAs. Although, as shown in [KFC+24, ZGJ+18, DWA+19], these models are often ineffective because they fail to establish a direct correlation between HPC patterns and actual attack behavior. In [KAGPY17], an effective attack is demonstrated that learns the detection mechanism of the ML model itself and evades it by adapting the attack behavior. To overcome these limitations, [AMC+22] proposes a framework to generate adversarial examples by constructing vectors of HPC counters for training ML models, but it still suffers from false positives. Existing countermeasures typically utilize HPC values such as `cache accesses` and `cache misses` to predict attacks, but they fail to establish a clear and reliable correlation with cache-based timing attacks [KFC+24].

In this work, we overcome these limitations by establishing a direct correlation between HPC values and attack behavior. Moreover, unlike [OTB+24], we do not degrade the performance of the entire system to stop the attack by identifying the location of the attacker. To identify the attacker's core, we use core-specific counters instead of uncore counters. We use the characteristics of the SPEC 2017 benchmarks [LA18] and the PARSEC benchmark suite [BKSL08] to understand the behavior of benign applications and compare it with the behavior of a malicious application attempting a timing attack on the NoC.

The SPEC 2017 benchmark suite consists of 43 diverse applications designed to evaluate system performance across real-world scenarios. These applications, written in C++, C, and Fortran, include integer workloads, floating-point workloads, memory-intensive codes, and branch-heavy codes. However, SPEC 2017 does not include multithreaded workloads representative of modern ML or cloud environments. For this reason, we also consider the PARSEC benchmark suite, which includes 9 workloads from domains

such as computer vision, media processing, computational finance, enterprise servers, and animation physics. We analyze the behavior of applications from both suites from various previous studies [LA18, BKSL08, KAL22, GIK+10] and compare them against the behavior of an attacker executing a timing attack on the NoC. As discussed in Sec. 3.2.1, to monitor the throughput of its own packets, the attacker must perform load/store memory instructions that result in L1 cache misses. The details of each component are as follows:

- **Load/Store Instructions:** In the SPEC 2017 benchmark suite, the total number of executed instructions ranges from 76 billion to 22,180 billion, with only 33.93% of them being load/store operations on average. `507.cactuBSSN_r` exhibits the highest load/store ratio at 48.375%. In the PARSEC benchmark suite, the total instruction count ranges from 2 billion to 33 billion, with load/store instructions ranging from 0.19 billion to 12.12 billion, averaging 39.24%. `freqmine` has the highest load/store ratio in PARSEC at 49.47% [BKSL08, LA18].
- **L1 Miss Rate:** Although L1 miss rates vary with cache size and associativity, several studies [BKSL08, KAL22] evaluate these workloads across different cache configurations. For SPEC 2017, the average L1 miss rate is 3.424%, with `507.cactuBSSN_r` exhibiting the highest miss rate at 19.485% on a 32 KB L1 cache with a 64-byte line size [LA18]. For PARSEC, the L1 miss rate remains under 5% for most applications on the same cache size, with `radix` showing the highest miss rate at 6%. Although PARSEC includes multithreaded programs affected by coherence protocols, a similar trend is observed across MSI and MESI [KAL22].

■ *Observation:* The `507.cactuBSSN_r` exhibits the highest MPKI across all benchmarks, with a load/store instruction ratio of 48.375% and the highest L1 miss rate of 19.485%, resulting in a total MPKI of:

$$\texttt{MPKI}_{max} = \frac{\texttt{Memory instructions}}{\texttt{Total instructions}} \times \frac{\texttt{L1 misses}}{\texttt{L1 accesses}} \times 1000 = \frac{48.375}{100} \times \frac{19.485}{100} \times 1000 \approx 94.26$$

However, the average load/store instruction ratio across all benchmarks is 36.5%, and the average L1 miss rate is 4%. The average MPKI for a benign workload is therefore:

$$\texttt{MPKI}_{avg} = \frac{\texttt{Memory instructions}}{\texttt{Total instructions}} \times \frac{\texttt{L1 misses}}{\texttt{L1 accesses}} \times 1000 = \frac{36.5}{100} \times \frac{4}{100} \times 1000 \approx 14.6$$

In contrast, an attacker requires an injection rate of at least 60% to create meaningful contention (refer to Sec. 3.2.2). This implies that 6 out of every 10 instructions must be load/store operations that results in a L1 cache miss, yielding:

$$\texttt{MPKI}_{\text{attacker}} \geq \frac{6}{10} \times 1000 = 600$$

This reveals a clear distinction between the HPC values of benign and malicious applications. This correlation between the HPC values `L1 Misses`, `Instructions Retired`, and the high-injection-rate condition is exploited by *Turtle Walk* to detect a malicious application. Instructions such as non-temporal loads/stores, `CLFLUSH`, and cache-coherence messages can generate NoC traffic and induce contention without generating L1 misses. However, prior works [DC25, WBZL22, PLF21, RFZS18, RSB+17] demonstrate effective timing attacks using load/store instructions only, instead of *non-temporal load/store, `CLFLUSH`, and cache-coherence messages*, due to the following reasons:

- **Non-Temporal Loads/Stores:** Intel instructions such as `MOVNTDQA/MOVNTQ` are considered non-temporal loads/stores, as they bypass the cache hierarchy and directly access main memory. These instructions have two major limitations: (i) their access latency is comparable to DRAM latency, resulting in low temporal resolution and excessive congestion, which can lead to attack failure. A similar effect is observed in the worst-case variant of *Guard-NoC* countermeasure, where the response latency increases

from 5 cycles (L2 latency) to 100 cycles (DRAM latency), causing the attack to fail due to excessive endpoint contention (see Fig. 7b); and (ii) Since non-temporal loads/stores are directed to the memory controller, they do not provide fine-grained control over specific NoC paths, thereby limiting their effectiveness for targeted contention on AES memory accesses to a shared LLC, as considered in [DC25, RFZS18, RSB+17].

- `CLFLUSH`**:** This instruction is used in attacks such as Flush+Flush [GMWM16]. However, it is not feasible to accurately count the number of completed `CLFLUSH` operations within a fixed interval, as it does not produce a verifiable data-dependent outcome like load/store instructions. Consequently, it is better suited for latency probing rather than throughput probing and inherits the associated limitations discussed in Sec. 2.0.1.
- **Coherence Messages:** Prior work [WBZL22] demonstrates the use of coherence-based probes for timing attack on NoC. However, the attacker does not explicitly generate cache coherence messages and instead relies on load/store instructions that trigger coherence traffic. This attack behavior results in L1 cache misses, as two PEs repeatedly write to the same shared variable. When one PE writes to the variable, the corresponding cache line is invalidated in the other PE's private cache. Subsequently, when the other PE attempts to write to the same variable, it finds that the data is no longer present in its L1 cache due to invalidation, resulting in a cache miss. This process repeats continuously, generating coherence-induced cache misses. These events leave observable signatures in HPC counters and can be detected by the proposed *Turtle Walk* countermeasure.

Moreover, the primary reason mainly load instructions are used in timing attacks is that the victim performs secret-dependent memory accesses using loads like AES, and commercial NoCs often employ separate physical links for different traffic types [Lem11]. *Contention across these separate links is not feasible since they do not share the same physical resources.* Therefore, the attacker must generate traffic of the same type as the victim to create reliable contention. While additional HPC events such as `TOR_OCCUPANCY` [YXD+25] can capture a broader class of traffic, existing attacks [DC25, RFZS18, PLF21, WBZL22] along with our experimental analysis suggests that constructing a practical timing side-channel attack on NoC without inducing L1 misses is not feasible. Hence, monitoring L1 miss behavior via HPCs is sufficient to detect and mitigate existing attacks. Moreover, combining multiple metrics, such as MPKI, to detect all forms of timing attacks is challenging because different attack classes exhibit fundamentally different behaviors and correlations with HPC. The key distinction lies in the attack methodology. In eviction-based attacks, the attacker first brings a shared microarchitectural resource into a known state, allows the victim to execute, and subsequently probes the resource to infer secret-dependent activity. In contrast, contention-based attacks continuously monitor the latency of accessing a shared resource and infer secrets from contention-induced latency variations. Consequently, contention-based attacks require the attacker and victim to execute concurrently, with the attacker continuously monitoring its own requests.

This distinction has important implications for detection. As discussed above, in eviction-based attacks the attacker can often reduce its activity or perform intermittent probing while still extracting information, thereby weakening the correlation between attack behavior and HPC measurements and potentially evading both threshold-based [KFC+24] and ML-based HPC detectors [KAGPY17]. In contrast, contention-based attacks inherently require sustained monitoring and contention throughout the attack. As observed in NoC timing attacks, the attacker must continuously inject packets and measure their latency to extract secret-dependent information. This persistent activity produces a sustained impact on HPC values that is generally absent in eviction-based attacks targeting caches, TLBs, and BTBs.

Furthermore, even within contention-based attacks, attack characteristics depend on the targeted resource. For example, contention-based attacks on functional units do not

require memory requests, whereas NoC timing attacks rely on continuous packet injection through memory requests. As a result, designing a single HPC-based countermeasure capable of effectively detecting all timing attacks remains a challenging problem.

4 Evaluation

In this section, we analyze the effectiveness of *Turtle Walk* in handling false positives and false negatives. Additionally, we evaluate our countermeasure against state-of-the-art solutions in terms of *hardware and performance overhead*.

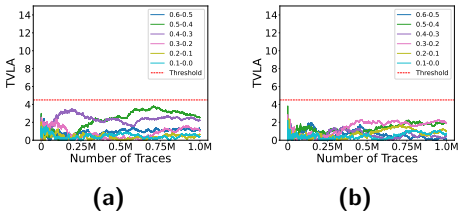


Figure 14: Turtle Walk under workloads: (a) Single-PE attack (b) Multi-PE attack

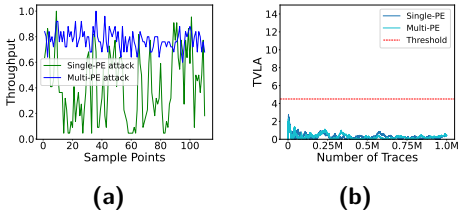


Figure 15: Adversarial packets under high noise: (a) Throughput variations (b) TVLA

4.1 Effectiveness

We test the effectiveness of *Turtle Walk*, where one PE acts as the victim, one/three PEs are considered malicious (following the attack model in [DC25]), and the remaining PEs inject traffic at different injection rates (0%–60%). The results are shown in Fig. 14, and the corresponding TVLA results confirm that *Turtle Walk* remains effective under this setting.

Although we do not observe any cases of false positives, in the event that a false positive occurs, our countermeasure does not terminate the process outright. Instead, *Turtle Walk* selectively degrades the performance of the specific application running on the non-secure-domain PE. This results in some performance loss for that application; however, the reduced contention may lead to performance improvement for other applications, ensuring that overall system efficiency remains unaffected.

Moreover, if a user wishes to run an application with high MPKI without experiencing performance degradation, we provide an enable/disable signal controlled by the secure kernel. This signal allows overriding the countermeasure and enables a benign application to run without throttling. *We assume that the privilege to override the enable signal is not buggy, and the analysis of this aspect is out of scope for this work.*

Additionally, under rare circumstances when the injection rates of a benign PE fall above 60%, the attacker is unable to retrieve the key due to high noise (refer to Fig. 15), making the attack infeasible. In such cases, the kernel can disable the *Turtle Walk* module using the enable signal, and no countermeasure is required.

To test the countermeasure in different attack variations, we also consider a scenario in which the attacker is aware of the countermeasure and attempts to evade it by injecting traffic in a bursty manner while keeping the overall MPKI below the threshold during each interval. The results shown in Fig. 16 reveal that bursty traffic also fails to create meaningful contention, and *Turtle Walk* remains effective under such conditions as no throughput variation is observed by different L1 hit/miss pattern created by different AES keys on the throughput data of adversary.

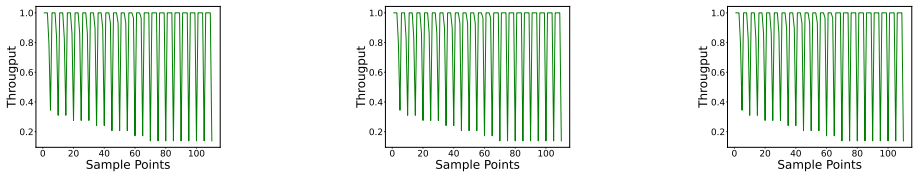


Figure 16: Throughput variation with busty traffic in presence of Turtle Walk during AES: (a) Key-1, (b) Key-2, and (c) Key-3

4.2 Overhead Analysis

The overhead of *Turtle Walk* in comparison with existing countermeasures is computed in terms of *(a)* Hardware Overhead: Area and Power, *(b)* Performance Overhead: Average Packet Latency, Zero Load Latency, and Processor Speed-up.

4.2.1 Hardware Overhead

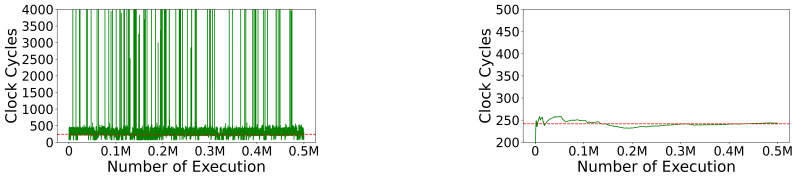
The hardware overhead is computed on both ASIC and FPGA platforms. An standalone NI (as shown in Fig. 13a) is implemented in RTL using Verilog HDL. This design choice ensures that the reported overhead reflects the true cost of the proposed modifications. For example, consider a NoC-based MPSoC with a total area of x and a reported overhead of 1% of the entire system (area overhead: $\frac{x}{100}$). However, if the modification is performed only in the NI, which constitutes 2% of the total area (i.e., $\frac{x}{50}$), this corresponds to an effective increase of $\frac{\frac{x}{100}}{\frac{x}{50}} \times 100 = 50\%$ in the NI area. To avoid such misleading interpretations, we adopt a standalone NI implementation, enabling a more transparent and precise evaluation of area overhead. We also wants emphasize that we keep our NI lightweight to demonstrate the minimal impact of *Turtle Walk* in terms of hardware overhead by integrating only the Prince block cipher for packet encryption/decryption [SGJN23]. Otherwise, an NI typically includes additional modules, such as an asymmetric cryptographic primitive like Elliptic Curve Cryptography for key exchange and a Message Authentication Code unit for packet integrity verification [SGJN23].

The ASIC synthesis is performed using the Yosys synthesis tool and a 45 nm NanGate Open Cell library [Knu08], while FPGA implementation is carried out on the AMD Virtex UltraScale FPGA VCU108 Evaluation Kit. Please note that existing works only demonstrates the countermeasure on ASIC, so we are only comparing the hardware overhead of ASIC platform with the existing works.

■ *Observation:* The area and power overhead of the base NI *(without Turtle Walk)* in comparison with all variants of *Turtle Walk* is shown in Table 1. The area and power overhead of `V1` over the base NI is 1.2% and 3.3%, respectively, whereas `V2` has an area and power overhead of 0.9% and 2.3%, respectively, in ASIC. The `V3` and `V4` variants have higher overhead compared to `V1` and `V2` due to the inclusion of an additional adder and interrupt signal, resulting in area and power overheads of 5.1%, 7.4% and 1.2%, 2.7% respectively, compared to the base NI. In FPGA, we observe a similar trend, where the area and power overhead of `V1` and `V3` is comparatively larger than that of the `V2` and `V4` variants. The higher hardware overhead in `V1, V3` compared to `V2, V4` is due to the computation of MPKI within the module. However, despite this, there is no degradation in processor throughput in `V1` and `V3` variants.

Table 1: Hardware Overhead of *Turtle Walk*

Varinat	Platform	Area (μm^2)/ LUT Count	Power (W)	Area Overhead	Power Overhead
Base	AISC 45 nm NanGate Open Cell library	29576.01	0.483	–	–
Turtle Walk (V1)		29937.50	0.499	1.2%	3.3%
Turtle Walk (V2)		29852.12	0.494	0.9%	2.3%
Turtle Walk (V3)		31087.42	0.519	5.1%	7.4%
Turtle Walk (V4)		29939.89	0.496	1.2%	2.7%
Base	AMD Virtex UltraScale FPGA VCU108 Evaluation Development Kit	4242	1.409	–	–
Turtle Walk (V1)		4295	1.415	1.2%	0.4%
Turtle Walk (V2)		4254	1.406	0.3%	−0.2%
Turtle Walk (V3)		4437	1.414	4.5%	0.4%
Turtle Walk (V4)		4251	1.387	0.2%	−1.5%



(a) **(b)**

Figure 17: Clock cycle count for one round of AES: (a) Individual traces, and (b) Average

4.2.2 Performance Overhead

The performance overhead is evaluated using the characteristics of the SPEC 2017 benchmark [LA18] and the PARSEC benchmark [BKSL08].

ⓐ *NoC Performance Evaluation of Turtle Walk:* The performance of *Turtle Walk* in terms of zero-load latency and average packet latency is tested using the benchmark characteristics on gem5. We simulate an 8×8 NoC with a Mesh topology and XY routing, where the PEs' injection rate is set between 0% and 20%. Half of the PEs belong to the secure domain, while the other half belong to the non-secure domain. We consider two scenarios: (1) Under no-attack condition and (2) When the attacker congests the NoC to recover the secret key from the victim.

■*Observations:*

- We do not observe any effect on Zero Load Latency in any variant of *Turtle Walk*.
- Under no-attack conditions, when the NoC is not congested, the average network packet latency in all variants is 32.52 cycles. However, when the attacker induces congestion in the NoC during the attack, the average network packet latency in all variants increases to 38.82 clock cycles.
- The total packet latency (network latency and queuing latency) under no contention and contention in all variants is 43.63 cycles and 419.8 cycles, respectively.

ⓑ *Processor Performance Evaluation of Turtle Walk:* There is no degradation in processor throughput due to `V1` and `V3`, whereas the degradation caused by `V2` and `V4` depends on the frequency of interrupt generation to fetch the MPKI value from the processor. The interrupt interval should not exceed the attack duration, nor should it be so short that it significantly degrades processor throughput. Existing attacks [RFZS18, RSB+17] target AES-128, so we use an interval equal to the execution time of one round of AES operation. Although this interval can vary based on the type of attack, it can be determined by finding the minimum time in clock cycles required to conduct a single execution of the attack and setting the interval accordingly.

■*Observation:* The execution time *(in clock cycles)* for $500k$ executions of AES on an *Intel i7-8700* is computed by implementing AES using the OpenSSL library *(T-Table based AES)* (refer to Fig. 17). The number of executions is chosen such that the average execution time stabilizes around a constant value at this trace count (see Fig. 17b). The time is measured using the hardware timestamp counter (`rdtsc()`), and the `clflush` instruction is used to flush any data from the cache. The cache needs to be initially empty, as a necessary condition for the attacker to execute the attack is that all cache levels are empty before each AES execution. The execution time for one round of AES is shown in

Table 2: Comparison with Existing Works

Countermeasure	Hardware Overhead		Processor Throughput Degradation	NoC Evaluation									Security Evaluation
	Area	Power		Relative Zero Load Latency (Worst Case)	Average Packet Latency (Not Under Attack)				Average Packet Latency (Under Attack)				Passed the Statistics Test
					Total	Network	Average Hop Count	Per Hop Latency	Total	Network	Average Hop Count	Per Hop Latency	
No Countermeasure	/	/	/	1	43.6399	32.5272	5.2205	6.230668	1901.76	39.6744	5.41527	7.326394	✗
RPSL [WS12]	2%	4%	0%	2	63.1179	49.3335	5.19073	9.504155	2944.65	49.2642	5.27556	9.338193	✓
Random Arbitration [SDSG14]	11%	9%	0%	15	758.692	180.635	4.94647	36.51796	3221.28	249.967	4.74644	52.6641	
Random Routing [SDSG14]	5%	8%	0%	1	43.6399	32.5272	5.2205	6.230668	1901.76	39.6744	5.41527	7.326394	✗
SER [SFS+16]	9%	8%	0%	1	43.6694	32.5528	5.22975	6.224542	1955.26	39.3485	5.35441	7.348802	
Gossip NoC [RSBS16]	21.16%	16.2%	0%	1	43.6399	32.5272	5.2205	6.230668	1901.76	39.6744	5.41527	7.326394	
NIBR [BK18]	/	/	0%	1	43.6399	32.5272	5.2205	6.230668	1901.76	39.6744	5.41527	7.326394	
Guard NoC *(Masking)* [RAH+20]	16%	18%	0%	256	12118.2	956.871	3.6781	260.1536	8866.91	750.771	3.73225	201.1577	
Guard NoC *(worst-case binding)* [RAH+20]	16%	18%	0%	100	17433.3	2558.72	3.21722	795.3202	18516.3	2535.65	3.32904	761.676	
Ironhide [OK20]	/	/	50%	1	37.8717	26.7855	3.8508	6.955827	1811.1	34.0756	4.15708	8.197004	
TDMA [SG11]	2.8%	/	0%	2	47.123	34.4656	5.1925	6.637573	1561.94	39.2969	5.15257	7.62666	
Surf-NoC [WGO+13]	161%	146%	0%	2	58.8048	44.8868	5.28955	8.485939	1969.92	49.3403	5.33038	9.256432	✓
Turtle Walk (V1)	1.2%	3.3%	0%	1	43.6399	32.5272	5.2205	6.230668	419.807	39.8207	5.37728	7.405361	
Turtle Walk (V2)	0.9%	2.3%	1.7%	1	43.6399	32.5272	5.2205	6.230668	419.807	39.8207	5.37728	7.405361	
Turtle Walk (V3)	5.1%	7.4%	0%	1	43.6399	32.5272	5.2205	6.230668	419.807	39.8207	5.37728	7.405361	
Turtle Walk (V4)	1.2%	2.7%	1.7%	1	43.6399	32.5272	5.2205	6.230668	419.807	39.8207	5.37728	7.405361	

Fig. 17a. The average execution time of one AES round is approximately 235 clock cycles, which we choose as the interval to trigger the interrupt for MPKI computation.

To compute the MPKI, only one division and one multiplication are required since the necessary values already reside in the HPCs of the PE. In a pipeline, division has a two-cycle throughput using the `RCP` instruction, and multiplication has a one-cycle throughput [int], resulting in a total of 4 clock cycles. This leads to a negligible performance degradation of $\approx 1.7\%$.

4.3 Comparison with the Existing Countermeasures

The existing countermeasures are analyzed in terms of hardware overhead *(area and power)*, its impact on processor throughput degradation, and the overall effect on NoC performance metrics such as zero-load latency, queuing latency, and network latency under the same conditions in which *Turtle Walk* is tested (refer to Sec. 4.2.2). The comparison is conducted both under normal conditions and when an attacker attempts to congest the network to recover the key (refer to Table 2).

The columns of Table 2 list the countermeasures, area overhead, power overhead, the countermeasure's effect on processor throughput, relative zero-load latency *(worst case)*, which is computed as the ratio of worst-case zero-load latency (with countermeasure) to worst-case zero-load latency (without countermeasure), the average packet latency (total = network + queuing latency), network latency alone, average hop count, average delay observed per hop under both attack and normal scenarios, and whether the countermeasure passes the security test.

■ Observations:

- Countermeasures such as [RSBS16, RAH+20, SDSG14, BK18, SFS+16] fail to prevent timing attacks, as attackers can still infer key bits by creating meaningful contention. *These countermeasures also incur significantly higher hardware overhead (ranging from 4× to 23×) and performance degradation (between 4.7× and 28×) compared to* Turtle Walk. Since they fail to prevent the attack, we do not discuss them further.
- RPSL [WS12] incurs relatively lower overhead but performs poorly in NoC performance metrics, exhibiting more than 1.5× the per-hop delay under both attack and normal scenarios when compared to *Turtle Walk*. This is due to the uneven distribution of bandwidth between secure and non-secure domain PEs.
- SurfNoC [WGO+13] introduces an extremely high overhead (178×), while TDMA [SG11] suffers from nearly double zero-load and per-hop latency compared to *Turtle Walk*. Moreover, these solutions cannot be integrated with adaptive routing or hybrid topologies [WGO+13].

- Ironhide [OK20] wastes nearly half of the core resources due to strict isolation, and the process speed-up is linearly dependent on the number of active cores [BKSL08], causing substantial performance degradation. Additionally, it cannot support the execution of unknown applications [OK20], which is a critical limitation.
- SurfNoC [WGO+13], TDMA [SG11], and Ironhide [OK20] also show poor average packet latency when the attacker congests the network to extract the key. Their latencies increase by approximately 4.5×, 3×, and 4.5× respectively, compared to *Turtle Walk*.
- Guard NoC *(worst-case binding)* [RAH+20] exhibits the highest per-hop latency due to backpressure created by holding packets at the destination NI with a increase of latencies approximately by 45×. Moreover, it nullifies the efficiency gains of caches and other architectural optimizations by holding packets for such long durations, making it an impractical countermeasure. In addition, the area and power overheads are significantly high, with increases of 16% and 18%, respectively.
- The *Turtle Walk* can be deployed with any NoC architecture as it is integrated inside the NI, not inside the router, making it a feasible choice for hybrid topology. Moreover, no precomputed data is needed, so it can work with both known and unknown applications.
- All variants of *Turtle Walk* exhibit low hardware overhead and outperform other countermeasures in NoC performance. Even under attack scenarios, packets experience the lowest latency. This is because *Turtle Walk* not only prevents the attack but also decongests the network by reducing the attacker's injection rate through source throttling.
- There is no effect on the functionality of *Turtle Walk* with an increase in the number of domains, as all other domains where the adversary does not operate are considered secure domains. *Turtle Walk* functions similarly to the two-domain strategy without incurring any additional overhead.

These observations and evaluations indicate that the *Turtle Walk* countermeasure has 50% lower hardware overhead compared to any state-of-the-art countermeasure, has minimal impact on both system and NoC performance, successfully prevents attacks, and passes the TVLA test, ensuring its robustness.

5 Conclusion

This work shows that complex hardware modifications are not required to prevent an attack if the root cause is properly addressed. We introduce a new method to safeguard NoC-based MPSoC architectures from timing side-channel attacks using existing hardware. This is achieved by monitoring the MPKI values of PEs to detect potential attacks. Once an attack is detected, source throttling is applied to prevent the attacker from creating meaningful contention. The method incurs negligible area and power overhead compared to the baseline hardware without a countermeasure. Additionally, the countermeasure demonstrates robustness against timing attacks, as confirmed by TVLA analysis over one million traces, thereby ensuring its correctness.

Acknowledgements

We acknowledge the support received from the ISEA Project sponsored by the Ministry of Electronics and Information Technology (MeitY), the TCS Research Scholar Program, the Qualcomm Innovation Fellowship, and the Royal Society International Exchanges 2024 Global Round 3 (Grant No. IES\R3\243109).

References

[AK21] Usman Ali and Omer Khan. Connoc: A practical timing channel attack on network-on-chip hardware in a multicore processor. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 192–202. IEEE, 2021.

[AKPJ09] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 33–42. IEEE, 2009.

[AMC+22] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. Evax: Towards a practical, pro-active & adaptive architecture for high performance & security. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1218–1236. IEEE, 2022.

[ASK23] Usman Ali, Sheikh Abdul Rasheed Sahni, and Omer Khan. Characterization of timing-based software side-channel attacks and mitigations on network-on-chip hardware. *ACM Journal on Emerging Technologies in Computing Systems*, 19(3):1–23, 2023.

[BK18] Travis H Boraten and Avinash K Kodi. Securing nocs against timing attacks with non-interference based adaptive routing. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE, 2018.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[CM21] Subodha Charles and Prabhat Mishra. A survey of network-on-chip security attacks and countermeasures. *ACM Computing Surveys (CSUR)*, 54(5):1–36, 2021.

[DC24] Dipesh and Urbi Chatterjee. Door knock: Reverse engineering the mpsoc layout through timing attack on noc. *IEEE Embedded Systems Letters*, 16(4):449–452, 2024.

[DC25] Dipesh and Urbi Chatterjee. N-tracer: A trace driven attack on noc-based mpsoc architecture. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 1127–1140, 2025.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael*, volume 2. Springer, 2002.

[DWA+19] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE symposium on security and privacy (SP)*, pages 20–38. IEEE, 2019.

[GGJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GIK+10] Zvika Guz, Oved Itzhak, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C Weiser. Threads vs. caches: Modeling the behavior of parallel workloads. In *2010 IEEE International Conference on Computer Design*, pages 274–281. IEEE, 2010.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: A fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[HRY24] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-scope: Cache probing at cache speed. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 109–123, 2024.

[int] Intel® 64 and ia-32 architectures optimization reference manual volume 1. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[KAGPY17] Khaled N Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. Rhmd: Evasion-resilient hardware malware detectors. In *Proceedings of the 50th Annual IEEE/ACM international symposium on microarchitecture*, pages 315–327, 2017.

[KAL22] V KALAVATHI. *Performance of different Cache Coherence Protocols on different Multi-core Architecture Systems*. PhD thesis, INDIAN INSTITUTE OF TECHNOLOGY MADRAS, 2022.

[KFC+24] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. Sok: Can we really detect cache side-channel attacks by monitoring performance counters? In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 172–185, 2024.

[Knu08] Jesper Knudsen. Nangate 45nm open cell library. *CDNLive, EMEA*, 2008.

[LA18] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158. IEEE, 2018.

[Lem11] Oded Lempel. 2nd generation intel® core processor family: Intel® core i7, i5 and i3. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–48. IEEE, 2011.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[OK20] Hamza Omar and Omer Khan. Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 111–122. IEEE, 2020.

[OTB+24] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F Oliveira, and Onur Mutlu. {ABACuS}:{All-Bank} activation counters for scalable

and low overhead {RowHammer} mitigation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1579–1596, 2024.

[Pay16] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings 8*, pages 138–154. Springer, 2016.

[PLF21] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 645–662, 2021.

[PTV21] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+ scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2906–2920, 2021.

[RAH+20] Cezar Reinbrecht, Abdullah Aljuffri, Said Hamdioui, Mottaqiallah Taouil, Bruno Forlin, and Johanna Sepulveda. Guard-noc: A protection against side-channel attacks for mpsocs. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 536–541. IEEE, 2020.

[RBG+16] Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Annelie Heuser, Sikhar Patranabis, and Debdeep Mukhopadhyay. Leak me if you can: Does tvla reveal success rate? *Cryptology ePrint Archive*, 2016.

[RFZS18] Cezar Reinbrecht, Bruno Forlin, Andreas Zankl, and Johanna Sepúlveda. Earthquake—a noc-based optimized differential cache-collision attack for mpsocs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 648–653. IEEE, 2018.

[RSB+17] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. Timing attack on noc-based systems: Prime+ probe attack and noc-based protection. volume 52, pages 556–565. Elsevier, 2017.

[RSBS16] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, and Johanna Sepúlveda. Gossip noc–avoiding timing side-channel attacks through traffic management. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 601–606. IEEE, 2016.

[SDSG14] Martha Johanna Sepulveda, Jean-Philippe Diguet, Marius Strum, and Guy Gogniat. Noc-based protection for soc time-driven attacks. *IEEE Embedded Systems Letters*, 7(1):7–10, 2014.

[SFS+16] Johanna Sepúlveda, Daniel Flórez, Mathias Soeken, Jean-Philippe Diguet, and Guy Gogniat. Dynamic noc buffer allocation for mpsoc timing side channel attack protection. In *2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS)*, pages 91–94. IEEE, 2016.

[SG11] Radu Stefan and Kees Goossens. Enhancing the security of time-division-multiplexing networks-on-chip through the use of multipath routing. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, pages 57–62, 2011.

[SGJN23] Syam Sankar, Ruchika Gupta, John Jose, and Sukumar Nandi. Sec-noc: A lightweight secure communication system for on-chip interconnects. *IEEE Embedded Systems Letters*, 16(2):214–217, 2023.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology: Extended version. *Journal of Cryptographic Engineering*, 6:85–99, 2016.

[Sod15] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24. IEEE, 2015.

[WBZL22] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1506–1524. IEEE, 2022.

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, 27(5):15–31, 2007.

[WGO+13] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3):583–594, 2013.

[WMM19] Felix Wegener, Thorben Moos, and Amir Moradi. Dl-la: Deep learning leakage assessment. *IACR Cryptology ePrint Archive*, 2019.

[WS12] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 142–151. IEEE, 2012.

[YXD+25] Yujie Yang, Lingfeng Xiang, Peiran Du, Zhen Lin, Weishu Deng, Ren Wang, Andrey Kudryavtsev, Louis Ko, Hui Lu, and Jia Rao. Architectural and system implications of cxl-enabled tiered memory. *arXiv preprint arXiv:2503.17864*, 2025.

[ZGJ+18] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 457–468, 2018.