Timing Interference in Multi-Core RISC-V Systems: Security Risks and Mitigations

Andreas Wrisley(✉)[1,3] [0009−0005−2664−0043], Roberto Guanciale[2] [0000−0002−8069−6495], Simin Nadjm-Tehrani[1] [0000−0002−1485−0802], and Ingemar Söderquist[3] [0000−0001−9863−9985]

[1] Department of Computer and Information Science, Linköping University, Linköping, Sweden
{andreas.wrisley,simin.nadjm-tehrani}@liu.se
[2] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden
robertog@kth.se
[3] Saab AB
ingemar.soderquist@saabgroup.com

Abstract. Modern safety-critical and real-time systems increasingly rely on multi-core architectures, which introduce shared hardware resources that can lead to inter-core interference. This interference poses risks to both security and safety, enabling timing side channels and Denial of Service (DoS) attacks. This paper presents a methodology for evaluating the memory hierarchy of hardware platforms, focusing on timing interference and side-channel leakage. Using the OpenPiton platform, we identify and characterize a cross-core covert channel and demonstrate a proof-of-concept side-channel attack exploiting the Network-on-Chip (NoC). Additionally, we evaluate the impact of NoC contention on the worst-case execution time (WCET) of safety-critical applications. Despite exploring software-based mitigations, we find that covert channels cannot be completely eliminated without significant performance trade-offs.

Keywords: Side Channels · Covert Channels · NoC · Computer Architecture · WCET · Multi-Core · RISC-V

1 Introduction

Modern safety-critical and real-time systems increasingly rely on multi-core and many-core system-on-chip (SoC) architectures to meet growing performance demands. Platforms such as autonomous vehicles, industrial controllers, and avionics systems require not only high computational throughput, but also strict guarantees of temporal isolation and predictability.

Yet, most multi-core processors incorporate multiple interconnected components shared among processing units, leading to potential inter-core interference when several cores access the same resource in parallel. Such interference undermines software predictability and poses risks to both safety and security. From a

confidentiality perspective, inter-core interference can act as a timing side channel, allowing adversaries to exploit timing variations in shared-resource accesses to construct side or covert channels. From an availability perspective, the same interference can be abused to launch Denial of Service (DoS) attacks. Therefore, the same hardware features can affect both the real-time domain and the security-critical domain, and investigating them jointly is beneficial since the two aspects are related.

Identifying threats that leverage shared hardware resources and finding effective mitigations is therefore a central security assurance activity. Although this problem has been demonstrated in high-speed off-the-shelf Intel CPUs with a ring interconnect, the study for other architectures and interconnects is still open.

Among other architectures, the OpenPiton platform is particularly interesting due to the growing interest in open RISC-V–based architectures, particularly within the European industry, as highlighted in the ECS roadmap [11]. The openness of RISC-V enables domain-specific hardware design and facilitates certification efforts in sectors such as avionics, making it an important target for analyzing the implications for safety and security.

Previous work has demonstrated side and covert channels in shared hardware resources, such as last-level caches [17, 30], memory controllers [21, 23, 27], and ring interconnects [20]. However, less attention has been paid to tiled many-core research platforms such as OpenPiton, where the combination of a distributed last-level cache (LLC) and a multi-network NoC may open new attack surfaces.

In this paper, we introduce a methodology for evaluating a memory hierarchy in the context of combined safety and security risks through timing interference or side-channel leakage. The evaluation is based on the following two questions, *Can timing variations caused by interconnect and LLC behavior leak information across cores?* and *To what extent can one core influence the execution time of another through shared resources such as the interconnect or LLC?*

To address these questions, we make the following contributions which combine a general methodology with a concrete evaluation on an open source platform. In this paper, we use a small platform with two cores for demonstration, and we see no reason why the methodology would not be scalable to larger platforms.

- A methodology for assessing the interconnect and the memory hierarchy of a hardware platform from both a security and a safety perspective, applied in a *black-box manner*. This makes the approach applicable even when detailed hardware design information is unavailable or when the trustworthiness of the final silicon cannot be assumed.
- Application of the methodology to the network-on-chip (NoC) of the OpenPiton platform.
- Identification and characterization of a cross-core covert channel on the NoC.
- A proof of concept of a side-channel attack on the NoC.
- An evaluation of the impact of NoC contention on the worst-case execution time (WCET) of safety-critical applications.

The remainder of the paper is structured as follows. Section 2 contains relevant background material. In Section 3 we present our evaluation methodology, and we apply the methodology to evaluate a hardware platform in Section 4. In Section 5 we discuss safety implications, which is followed by an evaluation of software-based mitigations in Section 6. Related work can be found in Section 7 and we conclude the paper in Section 8.

2 Background

To ground our analysis, we first describe the OpenPiton platform used in our experiments and then introduce the binary symmetric channel model used to quantify the covert channel capacity.

2.1 OpenPiton

The OpenPiton [4] open source research framework features a scalable tiled many-core system and support for the CVA6 (Ariane) application processor core [31], which is a 64-bit single-issue in-order CPU with 6 stages and compatible with RISC-V. A chip can contain up to 256 tiles in each dimension (2D mesh), and multiple chips can be connected together for a total of 500 million cores. Each tile consists of a CVA6 core (with private L1 cache), private L1.5 and a shared distributed last-level cache L2, and network-on-chip routers. External memory (DRAM) is attached to the (upper) left tile. A schematic view of our instantiation, containing two tiles (due to FPGA limitations), used for our experiments, is shown in Fig. 1.

As the L2 cache is distributed, consecutively mapped lines in the private caches may be mapped to different L2 slices. In our instantiation with two cores, this means that every other cache line goes into the same slice (all even lines in slice 0 and all odd lines in slice 1).

Fig. 1 also shows the NoC interconnect (blue numbered arrows), which consists of three physical networks, NoC1 - NoC3. The L1.5 issues requests on NoC1, receives data on NoC2, and writes back data (modified cache lines) on NoC3. The L2 receives cache miss requests from L1.5 on NoC1, sends memory requests to DRAM and response packets to L1.5 on NoC2, and receives responses from DRAM and the write-back data from L1.5 on NoC3.

This organization of the memory hierarchy and NoC creates multiple contention points that can be observed by other cores. If an adversary can distinguish between cache hits and misses through such contention, they can infer fine-grained memory access patterns of victim applications, which is a powerful primitive for extracting sensitive information such as cryptographic keys or control-flow behavior.

2.2 Binary Symmetric Channel

To evaluate the potential of a covert or side channel, it is useful to quantify how much information can be transferred over it. A common model for this purpose

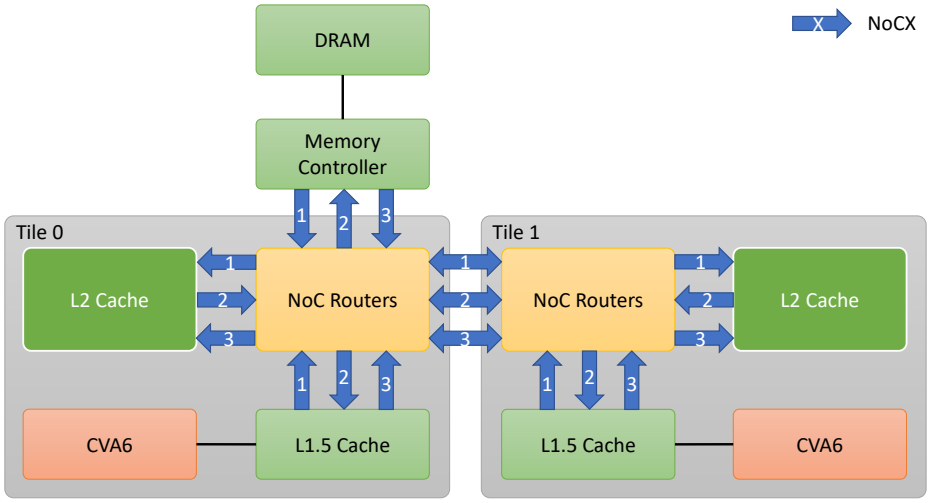

Fig. 1: OpenPiton architecture

is the binary symmetric channel (BSC). In this model, the sender sends a bit (0 or 1) and the receiver observes a binary symbol that may differ from the original due to transmission errors. The probability that a bit is flipped is p. The BSC provides a simple, yet powerful, abstraction for analyzing noisy channels, including those created by contention in shared hardware resources. Information can be transferred at any rate up to the channel capacity C, according to Shannon's noisy-channel coding theorem [24], and is defined as

$$C(p, r) = r(1 - H_b(p)) \tag{1}$$

where r is the raw transmission bandwidth, which is determined by the interval between bit transmissions. H_b is the binary entropy function defined as $H_b(p) = -p \cdot log_2(p) - (1 - p) \cdot log_2(1 - p)$.

3 Security Evaluation Methodology

Assessing a hardware platform from both a security and safety perspective often requires detailed design knowledge, which may not always be available, particularly for proprietary systems. Even when documentation or HDL code exists, it is rarely sufficient for a complete assessment. To ensure general applicability, we adopt a black-box approach that relies only on run-time behavior, making our methodology usable even when hardware details are limited or the final silicon cannot be fully trusted.

3.1 Threat Model

In the case of side-channel attacks and safety-critical tasks, we assume an adversary that can execute code on one core while a victim application runs on another. The adversary cannot rely on core-local resources, such as private caches or branch predictors, nor can it share memory or last-level cache (LLC) lines with the victim. Instead, it must exploit shared resources such as the interconnect or the memory bus using software only, and no direct interaction with the interconnect is possible. The adversary's objectives are twofold: (i) to extract information through a side or covert channel, and (ii) to interfere with the execution time of victim tasks, potentially leading to missed deadlines (with potential safety consequences). In the case of a covert channel, the adversary controls two colluding applications running on different cores, which also do not share memory or LLC lines, and attempts to establish a communication channel by leveraging the interconnect.

3.2 Overview of our approach

Our approach is staged. First, we establish the baseline behavior of the memory hierarchy without contention. Next, we induce interference from a subset of the available cores in the system, since we are interested in cross-core interference. As for the non-contended case, we execute the victim applications on a subset of the cores to identify any core-dependent effects. This gives us data on possible vulnerabilities. Given that our contention experiments show an identifiable difference in latencies, we move on to the exploitation stage where we try to establish covert and side channels. The safety aspects of any contention are analyzed by evaluating the effects of the contention on the execution time by executing (domain-specific) applications under analysis in isolation and also with contenders on other cores.

To minimize noise, we run all our initial evaluation experiments on a bare-board system without an operating system or other applications.

3.3 Memory Latency

To evaluate the potential for contention-based risks, we measure the latencies of accessing different components of the memory hierarchy with and without contention. The general idea is that we have two tasks, $measure(c_m, t_m, s_m, b_m)$ and $contender(c_c, t_c, s_c, b_c)$ where c is the core to execute on, t is the target in the memory hierarchy (e.g., private caches, shared last-level cache, shared DRAM), s is the LLC slice target and b is the cache set target.

The *measure* task runs on core c_m, performs loads targeting t_m by building a measure set of addresses in such a way that the loads will miss at the higher levels (e.g., miss in L1 if $t_m = L2$).

This is possible if the number of ways in the caches differ in such a way that not all addresses can fit in the higher level(s). For example, if the number of ways in L1 is four, we have eight addresses in the measure set and we are targeting

L2. The first four addresses will fit in both L1 and L2, but when accessing the other four, the four addresses in L1 will be replaced and ensuring an L1 miss next iteration when the first address in the measure set is accessed again, it will still be an L2 hit if the number of L2 ways is larger than the number of L1 ways or if the L2 cache is large enough.

One can also utilize an eviction set that contains addresses that map to the same cache set in the higher levels, but different cache set in the target (not equal to b_m) for which we are measuring the latency. Accessing the addresses in the eviction set will ensure that the addresses in the measure set are not in the higher-level cache. In the general case, it can be difficult to generate an eviction set and numerous algorithms have been proposed [19, 26].

The addresses in the measure set are evicted from higher levels if applicable (no need for L1) using the eviction set, and the loads of the addresses in the measure set are timed using applicable instructions (e.g., *rdcycle* on RISC-V). To ensure that the addresses in the measure set are accessed in order, the loads are serialized using pointer chasing. This will ensure that the method without the eviction set works as intended.

The *contender* is set up to create a lot of traffic from its core (c_c) to its target (t_c) and is set up in the same way as the *measure* task, but does not time its loads.

3.4 WCET Estimation

We reuse the *contender* task from our latency experiments to induce a large number of memory accesses to measure the effects of contention on relevant tasks for the intended domain-specific system. First, we perform WCET estimations for each domain-specific task using a relevant method (static analysis, measurement-based, or a hybrid variant [5]) for the domain and system. The number of memory accesses is also an interesting metric in this phase, as a higher number of memory accesses would probably mean higher sensitivity. Next, we run the *contender* in parallel with the tasks to evaluate the contention effects.

4 Hardware Platform Analysis

In this section, we apply the methodology of Section 3 to analyze our chosen hardware platform to identify potential architectural details that may provide cross-core channels. We use a Digilent Genesys 2[4] FPGA board with a synthesized two-core OpenPiton system where the cores run at 66.67 MHz (i.e., 15 ns cycle time).

4.1 Contention Potential

As discussed in Section 2.1, the DRAM is connected to tile 0 and there is an L2 slice in each tile. When core 0 accesses the data in L2 slice 0, the NoC traffic

[4] https://digilent.com/reference/programmable-logic/genesys-2/start

goes from the core through the NoC router to the L2 in tile 0, so the traffic is limited to tile 0. Even when there is an L2 cache miss, traffic will be limited to tile 0. On the other hand, if core 0 accesses the data in L2 slice 1, the NoC traffic will travel to the NoC router in tile 1 and then to the L2 in tile 1. If the access misses in L2 slice 1 there will be additional NoC traffic to tile 0 and the DRAM and then back again. The same goes for core 1 when accessing the L2 cache. In contrast, all DRAM requests from core 1 inevitably generate NoC traffic to tile 0, since the DRAM controller is attached there. To introduce contention, we must inject traffic into appropriate locations.

4.2 Latency Measurements

We use the tasks described in Section 3.3 to perform the initial latency measurements.

Implementation Notes To reduce measurement noise, we use the *fence* instruction to ensure that the relevant memory load has been retired before retrieving the timestamp (*rdcycle*). We also measure the latency of a number of loads to reduce the overhead of the *fence* instruction and the measurement code. This also helps to find a good trade-off between accuracy and granularity.

Baseline Latency To measure the baseline latencies, we set up our experiments with the following parameters, $c_m \in \{0, 1\}$, $b_m \in \{0, 1\}$ and $t_m \in \{L1, L2, DRAM\}$. The L2 slice (s_m) is determined by the cache set (b_m) and in our two-core instantiation the slice is given by $s_m = b_m \bmod 2$. This will target both L2 slices for L2 and DRAM accesses, and we run the experiment from each core in our system to identify any core-dependent latency differences.

We collect 1000 samples where each sample contains 8 accesses to the latency target (cache or DRAM). The *Isolation* column in Table 1 and Table 2 shows the average latency and standard deviation per access measured in core 0 ($c_m = 0$) and core 1 ($c_m = 1$) respectively.

As expected, L1 latency is constant at ∼5 cycles. L2 latency is ∼30 cycles across both cores and both slices, with little variation. DRAM latency is ∼100–110 cycles but shows a systematic difference: accesses to data allocated on the same slice of the accessing core are consistently ∼10 cycles faster than accessing data allocated on the other slice.

Latency under Contention Next, we measure the latency of memory accesses under contention by setting up our system with the additional task, *contender*.

From the discussion in Section 4.1 we identify four relevant cases of cross-core interference. NoC traffic is routed either through the same L2 slice for both tasks (even or odd sets for both tasks) or through different slices. When using the same L2 slice, $s_m = s_c$, we have the choice of accessing different cache sets ($b_m \neq b_c$) or the same set ($b_m = b_c$ are standard cache evictions and are not

Table 1: Average latency (cycles) for *measure* on core 0 ($c_m = 0$) with contention on L2 and DRAM respectively

Target (t_m)	s_m	s_c	Isolation		Contention (t_c)			
					L2		DRAM	
			Avg (μ)	Std dev (σ)	Avg (μ)	Std dev (σ)	Avg (μ)	Std dev (σ)
L1	-	-	5.04	0.47	5.04	0.41	5.04	0.41
L2	0	0	29.82	0.96	42.03	6.19	30.85	1.33
L2	1	1	29.50	1.28	29.57	1.26	30.55	1.66
L2	0	1	-	-	41.71	8.39	30.86	1.44
L2	1	0	-	-	29.60	1.28	30.33	1.43
DRAM	0	0	102.12	2.92	127.41	4.22	119.36	4.17
DRAM	1	1	111.24	3.52	118.21	3.15	121.67	4.52
DRAM	0	1	-	-	127.06	4.17	109.98	6.13
DRAM	1	0	-	-	118.60	3.40	116.90	4.91

Table 2: Average latency (cycles) for *measure* on core 1 ($c_m = 1$) with contention on L2 and DRAM respectively

Target (t_m)	s_m	s_c	Isolation		Contention (t_c)			
					L2		DRAM	
			Avg (μ)	Std dev (σ)	Avg (μ)	Std dev (σ)	Avg (μ)	Std dev (σ)
L1	-	-	5.04	0.47	5.04	0.40	5.04	0.41
L2	0	0	29.57	1.21	41.73	7.44	30.98	1.80
L2	1	1	29.75	1.04	29.85	1.03	30.54	1.46
L2	0	1	-	-	40.05	7.86	30.39	1.43
L2	1	0	-	-	29.86	1.03	30.95	1.39
DRAM	0	0	110.04	3.03	131.86	4.11	121.85	4.38
DRAM	1	1	103.00	3.10	110.55	3.82	121.66	3.78
DRAM	0	1	-	-	131.62	4.00	115.86	5.40
DRAM	1	0	-	-	110.22	3.92	110.67	5.42

of interest in this work). Since the L2 slice is determined by the cache set, we cannot have the same cache set when targeting different L2 slices.

The *contender* is configured with $c_c \neq c_m$, $b_c \in \{0, 1, 2, 3\}$, $b_c \neq b_m$, $t_c \in \{L2, DRAM\}$ and the *Contention* columns in Table 1 and Table 2 shows the average latency with the *contender* accessing the L2 and the DRAM in core 0 and core 1 respectively.

In the presence of contention, clear latency differences emerge compared to the baseline. As expected, L1 accesses remain unaffected, since they are core-private. In contrast, L2 accesses show noticeable slowdowns when the measurer targets slice 0 and the contender accesses L2 cache data, regardless of which slice the contender accesses or how the cores are allocated. Moreover, L2 accesses by the measurer also exhibit slight slowdowns whenever the contender performs uncached DRAM accesses. For DRAM accesses, all measurer configurations ex-

perience increased latency under contention. The slowdown is generally greater when the contender's data is cached in L2, independent of core placement. A notable exception occurs when both measurer and contender target slice 1: in this case, the slowdown is more pronounced if the contender accesses uncached data from DRAM. The analysis of interferences in this section demonstrates that there are several opportunities for timing-based covert and side channels.

4.3 Covert Channel

Using the findings of our initial memory latency experiments (Section 4.2), we demonstrate and evaluate a covert channel.

In this scenario, we manage to compromise two applications and thus we have two colluding entities, a *trojan* and a *spy* that will try to communicate outside of the channels established by the operating system or similar. For a covert channel, the *trojan* task corresponds to the *contender* and the *spy* corresponds to *measure*.

A covert channel should be possible whenever there is a difference between the average latency in isolation and in contention in Table 1 or Table 2. We conclude that no covert channel is possible when accessing L1 since it is core-local, but for all other cases there are differences, albeit very small for some cases. For example, if neither the *trojan* nor the *spy* access DRAM, it seems that the channel could be prevented. We evaluate the most important cases.

Evaluating Channel Capacity To characterize the covert channel, we use the results of Section 4.2 to set up our experiments. The *trojan* and the *spy* agree on the burst interval (i.e., the raw bandwidth), the L2 slice, and the cache set to target. During a burst interval, the *trojan* induces memory traffic for a '1' and idles for a '0'. The *spy* performs memory accesses continuously and depending on the memory traffic of the *trojan* there will be a difference in the number of memory accesses during the burst interval. This communication method has been used in previous work [20, 23, 27].

We model the channel as a binary symmetric channel and perform experiments for the cases discussed in Section 4.2 with a raw bit rate ranging from 1 kilobit per second (kbps) to 70 kbps (after which the burst interval becomes shorter than the measurement time). Equation 1 in Section 2.2 is used to assess the channel capacity.

The *trojan* sends a string of alternating bits '1' and '0', which the *spy* decodes using this difference in the number of memory accesses resulting from the latency differences identified in Table 1 and Table 2.

Fig. 2 illustrates the difference in latency between ones and zeros for a raw bandwidth of 20 kbps, and Fig. 3 shows the capacity and probability of error when varying the raw bitrates for the same case.

Table 3 summarizes the maximum capacity and error probabilities for the case where both access DRAM and the case where both access L2, respectively. The highest channel capacity is achieved in the L2 case, 68 kbps when $s_{trojan} =$

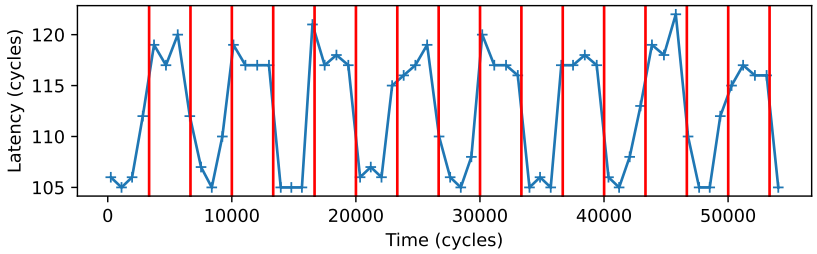

Fig. 2: Spy memory access latency ($r = 20$ kbps, $c_{trojan} = 1, c_{spy} = 0, t_{trojan} = t_{spy} = DRAM, s_{trojan} = s_{spy} = 0$)

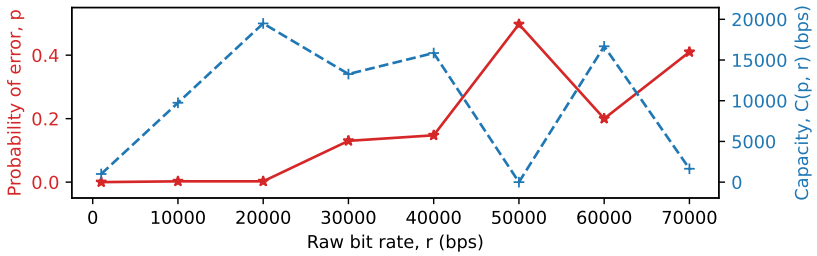

Fig. 3: Channel capacity and error probability ($c_{trojan} = 1, c_{spy} = 0, t_{trojan} = t_{spy} = DRAM, s_{trojan} = s_{spy} = 0$)

$s_{spy} = 0$. For the DRAM case, a channel capacity of 19.5 kbps is achieved ($s_{trojan} = s_{spy} = 0, c_{trojan} = 1$).

The prevention case discussed above, ($t_{trojan} = t_{spy} = L2, s_{spy} = 1, s_{trojan} \in \{0, 1\}$, in fact, provides a channel. Actually, from what we can see, there is no configuration that is secure.

4.4 Side Channel

With the successful covert channel in Section 4.3 in mind, we relax the adversary's requirements to create a side channel. In this setting, the *measure* and *contender* tasks in Section 4.2 would correspond to the *attacker* and the *victim*, respectively.

Being able to differentiate between L1 cache hit and miss would enable a trace-based [1] attack on an AES implementation using lookup tables. For example, the last round uses a separate lookup table compared to the previous rounds. There are 16 accesses, and depending on the input state to the last round, a number of these accesses will be misses, as the table has not been accessed before. Then, either a hit or a miss will occur. From this one can infer the round key, and from this it is possible to extract the master key.

Table 3: Maximum channel capacity per cache configuration case

s_{trojan}	s_{spy}	$c_{trojan} = 0$				$c_{trojan} = 1$				
		$t_{spy} = t_{trojan} = L2$				$t_{spy} = t_{trojan} = DRAM$				
		r (kbps)	p	$C(p,r)$ (kbps)	r (kbps)	p	$C(p,r)$ (kbps)	r (kbps)	p	$C(p,r)$ (kbps)
0	0	70	0.0025	68.2	20	0.1050	10.3	20	0.0025	19.5
1	1	20	0.0175	17.5	20	0.0100	18.4	70	0.2100	18.1
0	1	10	0.2925	1.3	50	0.3800	2.1	10	0.2825	1.4
1	0	70	0.3125	7.3	70	0.3375	5.4	60	0.2500	11.3

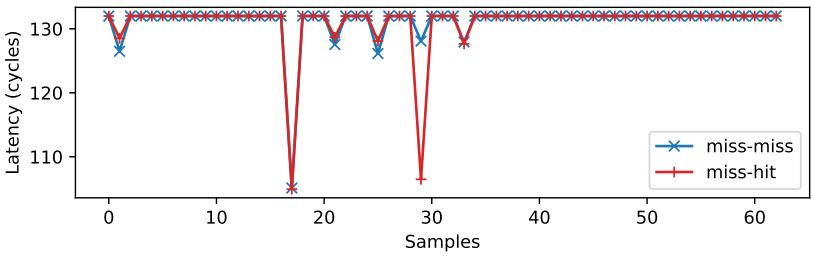

Fig. 4: Average latency for side-channel PoC ($c_{victim} = 0, s_{victim} = s_{attacker} = 0$)

We set up a proof-of-concept experiment in which the contention is located in L2 slice 0 as this is the case with the largest relative latency difference according to Table 1. The victim ($c_{victim} = 0$) performs two memory accesses while the attacker ($c_{attacker} = 1$) measures the latency of its four L2 accesses ($t_{attacker} = L2$) per sample. First, both victim accesses are L1 misses and hits in L2 ($t_{victim} = L2$), and in the second case, the first access is a miss ($t_{victim} = L2$) and the second access is a hit ($t_{victim} = L1$). Figure 4 shows the average latency over 1000 traces, and in sample 29 one can identify the case of miss-hit, which produces a much lower latency compared to miss-miss. Therefore, an attacker can use this latency difference together with Table 2 to classify the trace. There is also a latency drop in sample 17, here for both miss-miss and miss-hit, due to how the tasks are synchronized in this PoC such that the *attacker* collects samples before the *victim* performs its accesses and will be further analyzed in future work.

5 Implications for Safety

In this section, we analyze the potential implications of our identified contention scenarios that can affect safety by violating the application timeliness requirements. We analyze the variability of the execution time of applications in the presence of memory interference from the other core.

Table 4: Characterization of applications

Application	Max memory requests	Max execution time (ms)	
		Isolation	Contention
Nav	140	7410	7410
Mult	1899	471	482

5.1 Failure Model

From a safety point of view, we consider a system in which a time-critical application (victim, cf. *measure* task) must meet a strict deadline. A second, malign, application (attacker, cf. *contender* task) runs concurrently on another core trying to interfere with the time-critical application by inducing contention in the memory hierarchy. Both cores share the NoC and L2 cache. The failure condition is a missed deadline due to interference caused by the contention of shared resources.

This model reflects real-world use cases in embedded or mixed-criticality systems, where temporal isolation must be ensured to avoid cascading system failures.

5.2 Execution Time Measurements

We use two applications with differing memory demands, one that implements a navigation algorithm (nav) and one that performs a 100 x 100 matrix multiplication (mult) [18].

In this use case, we use a measurement-based WCET estimation approach, based on an existing approach [5], in which we manually insert instrumentation points that are used to derive the WCET estimate. How to do such measurements with good control over the probe effect has been documented in the literature [18].

First, we measure execution time and count the number of memory accesses for our applications isolated on core 0 ($c_{nav} = c_{mult} = 0$), and then measure execution time in a scenario with memory contention. The memory-intensive *contender* task, described in Section 3, with parameters $c_c = 1$, $s_c = c_{nav,mult}$ and $t_c = DRAM$. That is, it executes on the other core and only performs accesses to DRAM, with cache storage in the L2 slice belonging to the tile where the domain application executes, to interfere with our two applications under analysis.

From Table 4 we can see that Mult requires an order of magnitude more memory accesses than Nav and is thus more sensitive to NoC traffic and DRAM accesses from the other core, which is shown in the execution time measurements. Nav is unaffected by the memory traffic of the other core, but Mult's execution time is 11 ms longer (around 2.3 percent), and this could be enough to cause it to miss its deadline.

6 Discussion

This section discusses a number of possible mitigations of the vulnerabilities discovered, and an evaluation of select mitigations.

6.1 Software-Based Mitigations

Time-Multiplexing Memory Accesses We can arbitrate the memory accesses and only allow each process access to the memory during specified non-overlapping periods of time (e.g., Time Division Multiple Access [TDMA]). During the periods where memory accesses are not allowed, only L1 can be accessed, as evident from Tables 1 and 2 and the discussion in Section 4.3.

If there is no direct support in the hardware to schedule the memory accesses, the run-time system can make use of the Memory Management Unit (MMU) of the processor to stop the processes from accessing the memory. In Section 6.2 we perform a small evaluation of the implications for the covert channel, discussed in Section 4.3, using TDMA.

Arbitrating memory accesses with TDMA may result in under-utilization if processes do not access memory during their respective period.

Monitoring Memory Accesses The memory accesses can be monitored by using performance counters to count cache misses (the last level that does not involve the interconnect to reduce inter-core interference). Each process is assigned a periodic memory access budget and, when the budget has been depleted, the process is suspended until its next period starts. Löfwenmark and Nadjm-Tehrani [18] use this approach to study the interference aspects of multiple application processes due to shared resources and how it affects the Worst-Case Execution Time (WCET) of application processes on different processor cores.

Suspending the process after a number of memory accesses will most likely result in longer execution times for the affected processes. It may work well in a situation where a critical process is allowed to run and is guaranteed to meet its WCET estimates, and other processes are monitored.

Performance counters can also be used to detect attacks of different types by monitoring system behavior for anomalous patterns [6, 7, 9, 12]. By tracking specific hardware and software metrics, security systems can identify deviations from normal operation that might indicate malicious activity. This approach can be particularly effective in detecting stealthy attacks, such as cache-based side-channel attacks or malicious code injection. By monitoring cache access patterns, security systems can detect when an attacker is exploiting the shared cache to steal sensitive information.

Prefetching Data and Instructions Together with time-multiplexing, one could prefetch instructions and data into a private cache or to a scratchpad memory from where the instructions and data would be fetched. For private

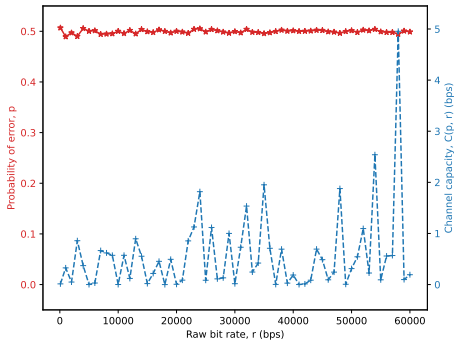

Fig. 5: Channel capacity with time multiplexing

caches, this only works if there are guarantees that there will be no cache evictions, as this could result in accesses to external memory. Also, it requires a write-back cache, as a write-through cache would write back any changes to the external memory as they happen.

This method affects the way the software applications are implemented and is difficult to use on already existing applications.

6.2 Mitigation Evaluation

The OpenPiton platform does not have support for performing TDMA arbitration in hardware, so we simulate this by giving the *trojan* and the *spy* an access quota (the optimal quota is very much system dependent and is a research area of its own). During their access period, they are allowed to access the memory, and the other is not. In this work, the *trojan* and the *spy* cooperate and do not perform any memory accesses outside of their respective access windows. As can be seen in Fig. 5, the bandwidth of the channel is greatly reduced but cannot be completely removed.

In a real-world scenario, the adversary would not cooperate, and other means would need to be implemented to ensure that it cannot access the memory. This could, for example, be changing the MMU tables, so there would only be invalid memory accesses during its off-time.

7 Related Work

Micro-architectural side-channel attacks have become more and more common, and it is an active research area where attacks, countermeasures, and detection mechanisms are discovered and developed. For example, Spectre and Meltdown [14, 16] are well-known vulnerabilities that exploit performance-enhancing optimizations such as the speculative execution of future instructions.

Power-analysis attacks use differences in the power consumption of the different hardware elements in a processor, and external equipment is usually required

to measure the consumption. This has changed with works [15, 28] that demonstrate software-based power side channels.

CPU frequency scaling is used to adjust the speed and voltage of the CPU to lower power consumption to extend, for example, battery life in mobile phones and laptops. Several works [2, 28, 29] use frequency scaling to communicate bits by inducing demanding computations, and thus increasing the CPU frequency. This can be used to signal a '1', and the absence of computations would result in a lower frequency, indicating a '0'. However, it requires direct access to the CPU frequency, but Zhu et al. [32] present IOLeak, which uses I/O latency to infer CPU frequency changes without this direct access to the CPU frequency. In this paper, we focus on safety-critical applications, where frequency scaling is not used because it would interfere with the desired deterministic behavior, and estimating the worst-case execution time (WCET) would be even more demanding.

Cache-based side channels are among the most prominent microarchitectural attacks. Techniques such as Prime+Probe [17] and Flush+Reload [30] have been demonstrated to extract cryptographic keys by exploiting cache contention and eviction patterns. This type of attack assumes that there are shared pages of memory between processes, while our work does not have such assumptions.

To be able to perform a successful real-world attack, there are several aspects of the target system that may not be known by an attacker and need to be obtained at run-time. Cache parameters, such as the total cache size, the cache block size, and the number of ways, are important for cache-based attacks. Shen et al. [25] present a method to extract these parameters. Eviction patterns are also a very important piece in performing cache-based attacks and generating eviction sets is a research area of its own; Vila et al. [26] present a theoretical analysis and an empirical evaluation, while Morgan et al. [19] improve the efficiency of eviction set construction on Intel processors utilizing non-linear hash functions for cache slice mapping.

The Network-on-Chip (NoC) is a scalable on-chip interconnect architecture that has become very popular and, therefore, also become very interesting for attacks and countermeasures [8, 13]. Ali et al. [3] use timing variations from multiple shared hardware resources to be less sensitive to noise. Their covert channel requires shared cache blocks, which we do not. Paccagnella et al. [20] demonstrate a practical side channel attack on a ring interconnect, which is an architecture used in many Intel processors. We use a similar method, but on a different platform and with a different interconnect architecture. A similar NoC has been used [10, 22], but the attack model allows the adversary to interact directly with the interconnect as it is simulated.

8 Conclusion

This work outlines a method for assessing the memory hierarchy of a hardware platform to identify security threats that can affect safety by violating application timeliness requirements. Using this assessment method, we identify and

classify a covert channel in an implementation based on the OpenPiton platform. Even in the seemingly secure case found in the contention measurements it is shown that a covert channel exists. We also show that we can distinguish an L1 cache hit from an L1 cache miss from another core by exploiting the memory hierarchy contention.

We discuss several software-based mitigation techniques and demonstrate that none of them can completely shut down the covert channel. In addition, they may have a serious performance impact.

Our current understanding after a substantial amount of experimental work on two tiles is that introducing additional tiles would make the NoC traversal longer and would provide many more opportunities for interference injection for an adversary. The experiments are nearly noise-free, and to assess the real-world exploitability of any identified channels using our method, more experiments should be performed in a more realistic scenario. Furthermore, the scalability of the methodology to larger platforms and multiple concurrent attacks needs to be demonstrated.

Future work includes expanding our side channel PoC to actually show it is viable for extracting the AES key. Another future direction is to pursue a hardened NoC to mitigate the channels identified in this paper.

Acknowledgments. This work was partially supported by Sweden's Innovation Agency under grant numbers 2023-01548 and 2024-03785.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Acıiçmez, O., Koç, c.K.: Trace-driven cache attacks on aes (short paper). In: Information and Communications Security. p. 112–121. Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11935308_9
2. Alagappan, M., Rajendran, J., Doroslovački, M., Venkataramani, G.: Dfs covert channels on multi-core platforms. In: 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). pp. 1–6 (2017). https://doi.org/10.1109/VLSI-SoC.2017.8203469
3. Ali, U., Khan, O.: Multicon: An efficient timing-based side channel attack on shared memory multicores. In: 2022 IEEE 40th International Conference on Computer Design (ICCD). pp. 97–104 (2022). https://doi.org/10.1109/ICCD56317.2022.00024
4. Balkind, J., McKeown, M., Fu, Y., Nguyen, T., Zhou, Y., Lavrov, A., Shahrad, M., Fuchs, A., Payne, S., Liang, X., Matl, M., Wentzlaff, D.: Openpiton: An open source manycore research framework. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 217–232. ASPLOS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2872362.2872414
5. Betts, A., Marref, A.: Wcet analysis of component-based systems using timing traces. In: 2011 16th IEEE International Conference on Engineering of Complex Computer Systems. pp. 13–22 (2011). https://doi.org/10.1109/ICECCS.2011.9

6. Bhade, P., Paturel, J., Sentieys, O., Sinha, S.: Lightweight hardware-based cache side-channel attack detection for edge devices (edge-cascade). ACM Trans. Embed. Comput. Syst. **23**(4) (Jun 2024). https://doi.org/10.1145/3663673
7. Bhade, P.P., Sinha, S.: Detection of cache side channel attacks using thread level monitoring of hardware performance counters. In: 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). pp. 210–217 (2021). https://doi.org/10.1109/MCSoC51149.2021.00039
8. Charles, S., Mishra, P.: A survey of network-on-chip security attacks and countermeasures. ACM Comput. Surv. **54**(5) (May 2021). https://doi.org/10.1145/3450964
9. Cho, J., Kim, T., Kim, T., Shin, Y.: Real-time detection on cache side channel attacks using performance counter monitor. In: 2019 International Conference on Information and Communication Technology Convergence (ICTC). pp. 175–177 (2019). https://doi.org/10.1109/ICTC46691.2019.8939797
10. Dipesh, Chatterjee, U.: N-tracer: A trace driven attack on noc-based mpsoc architecture. In: Proceedings of the 20th ACM Asia Conference on Computer and Communications Security. p. 1127–1140. ASIA CCS '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3708821.3736201
11. ECS strategic research and innovation agenda 2025 (ECS-SRIA). `https://ecssria.eu/2025`, accessed: 2025-08-24
12. Kapotoglu Koc, M., Altilar, D.T.: Selection of best fit hardware performance counters to detect cache side-channel attacks. In: Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems. p. 17–22. SaT-CPS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579988.3585052
13. Kar, A., Liu, X., Kim, Y., Saileshwar, G., Kim, H., Krishna, T.: Mitigating timing-based noc side-channel attacks with llc remapping. IEEE Comput. Archit. Lett. **22**(1), 53–56 (Jan 2023). https://doi.org/10.1109/LCA.2023.3276709
14. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy (S&P'19) (2019)
15. Kogler, A., Juffinger, J., Giner, L., Gerlach, L., Schwarzl, M., Schwarz, M., Gruss, D., Mangard, S.: Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In: USENIX Security (2023)
16. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
17. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy. pp. 605–622 (2015). https://doi.org/10.1109/SP.2015.43
18. Löfwenmark, A., Nadjm-Tehrani, S.: Understanding Shared Memory Bank Access Interference in Multi-Core Avionics. In: Schoeberl, M. (ed.) 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016). Open Access Series in Informatics (OASIcs), vol. 55, pp. 12:1–12:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/OASIcs.WCET.2016.12
19. Morgan, B., Horowitz, G., O'Connell, S., van Schaik, S., Chuengsatiansup, C., Genkin, D., Maennel, O., Montague, P., Ronen, E., Yarom, Y.: Slice+slice

baby: Generating last-level cache eviction sets in the blink of an eye. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 3479–3496 (2025). https://doi.org/10.1109/SP61157.2025.00264
20. Paccagnella, R., Luo, L., Fletcher, C.W.: 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). pp. 645–662. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella
21. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 565–581. USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl
22. Reinbrecht, C., Aljuffri, A., Hamdioui, S., Taouil, M., Forlin, B., Sepulveda, J.: Guard-noc: A protection against side-channel attacks for mpsocs. In: 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 536–541 (2020). https://doi.org/10.1109/ISVLSI49217.2020.000-1
23. Semal, B., Markantonakis, K., Akram, R.N., Kalbantner, J.: Leaky controller: Cross-vm memory controller covert channel on multi-core systems. In: Hölbl, M., Rannenberg, K., Welzer, T. (eds.) ICT Systems Security and Privacy Protection. pp. 3–16. Springer International Publishing, Cham (2020)
24. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal **27**(3), 379–423 (1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
25. Shen, S., Li, Z., Song, W.: Methods of extracting parameters of the processor caches. In: Cheng, C.M., Akiyama, M. (eds.) Advances in Information and Computer Security. pp. 47–65. Springer International Publishing, Cham (2022)
26. Vila, P., Köpf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 39–54 (2019). https://doi.org/10.1109/SP.2019.00042
27. Wang, Y., Ferraiuolo, A., Suh, G.E.: Timing channel protection for a shared memory controller. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 225–236 (2014). https://doi.org/10.1109/HPCA.2014.6835934
28. Wang, Y., Paccagnella, R., He, E., Shacham, H., Fletcher, C.W., Kohlbrenner, D.: Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In: Proceedings of the USENIX Security Symposium (USENIX) (2022)
29. Wang, Y., Paccagnella, R., Wandke, A., Gang, Z., Garrett-Grossman, G., Fletcher, C.W., Kohlbrenner, D., Shacham, H.: DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2023)
30. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 719–732. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom
31. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (Nov 2019). https://doi.org/10.1109/TVLSI.2019.2926114

32. Zhu, L., Wang, C.: Side-channel information leakage with cpu frequency scaling, but without cpu frequency. In: Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems. p. 69–76. HotStorage '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3736548.3737831