Statistical Analysis of HPC Filtering: Towards Reliable Use in Security

Adam Henault, Camille Monière, Philippe Tanguy, and Vianney Lapôtre

Université Bretagne Sud, UMR 6285, Lab-STICC, Lorient, France
firstname.lastname@univ-ubs.fr

Abstract. This article evaluates Hardware Performance Counters (HPCs) reliability for security in multitasking environments. We introduce a process-based filtering methodology for RISC-V architecture and analyze its efficiency across various noise levels. Results show that filtering efficiency depends on the physical nature of events. By classifying HPCs, we demonstrate that filtering reduces measurement variability by up to two orders of magnitude. Applied to buffer overflow detection, this filtering reduces false positive rates, proving that high F1-scores can mask underlying error distributions.

Keywords: Embedded Systems Security · HPC-based Detection · Hardware Performance Counter.

1 Introduction

Given the increasing diversity and sophistication of threats, the effectiveness of detection systems depends on their ability to leverage new detection metrics. While traditional software metrics (event logs, system calls) are often blind to stealthy activities or can be manipulated by attackers with elevated privileges, the exploration of low-level metrics has become interesting. It is in this context that Hardware Performance Counters (HPCs) make a major contribution. Natively integrated into modern processors, HPCs offer a new dimension of visibility by providing microarchitectural metrics (cache misses, branch prediction errors, etc.). A significant body of research thus leverages these microarchitectural traces as new signals to train machine learning models capable of distinguishing, in real time, between normal behavior and malicious activity [1], [2], [3]. However, the integration of these HPC indicators into detection systems raises a fundamental issue: the reliability of the collected data. Originally designed for debugging and performance optimization, HPCs metrics prove to be inherently noisy and non-deterministic. This noise stems from unavoidable interference related to Operating System (OS) activity: hardware interrupts, context switches, or the execution of background tasks. Without proper processing to isolate and filter these measurements, detection systems rely on biased metrics. This leads to detection models that are overly optimistic in controlled environments but whose reliability collapses in real-world conditions due to an unacceptable false positive rate.

The main contribution of this paper is to propose a method for evaluating HPC filtering for security purposes. To this end, we propose an experimental protocol that allows us, in a controlled environment representative of a real-world system, to collect both filtered and raw HPC data from the same execution. We then analyze the effectiveness of the filtering through statistical evaluation. Finally, we demonstrate the relevance of filtering through a security case study by highlighting the differences in performance when detecting malicious activity.

2 Related Work

HPCs are interesting for detecting various threats, ranging from general malware to microarchitectural attacks. Demme et al. [1] demonstrated the feasibility of an online detector by training machine learning models on specific event traces (cache misses, prediction errors, completed instructions). This approach was then extended to side-channel attacks: Carnà et al. [4] prove that kernel-level instrumentation based on cache HPCs can detect these threats at the system level, at a reasonable performance cost, while deploying dynamic countermeasures.

However, this research is based on the assumption of determinism and measurement reliability, which is a strong assumption outside a controlled environment. As early as 2013, Weaver et al. [5] demonstrated on x86_64 architecture that events assumed to be exact, such as the number of completed instructions, suffer from overcounting and variations from one execution to another. This instability was systematized by Das et al. [6], who identified several sources of non-determinism: event multiplexing on limited registers, architectural specifics of processors, and, above all, operating system activity (interrupts, context switches, background tasks). In practice, even small measurement variations are sufficient to degrade models trained under ideal conditions, making them vulnerable to load noise or scheduling variations. Furthermore, Bhade et al. [7] point out that system-wide data collection aggregates the activity of all processes; a legitimate but resource-intensive program can thus generate a signature indistinguishable from that of an attack.

To mitigate these disturbances, the literature has turned to filtering and signal processing techniques. Carnà et al. [4] introduce self-adjusting observation windows and a scoring system to absorb benign fluctuations. More recently, the focus has shifted to architectural filtering through isolation, as seen in [8] which collect HPCs per process and thereby eliminate global system noise.

However, although these filtering mechanisms are at the heart of recent architectures, these studies paradoxically fail to evaluate their actual value and impact. Indeed, key works such as [8] provide no quantitative comparison between their filtered metrics and the corresponding raw values, effectively leaving a blind spot regarding the actual effectiveness of filtering depending on the nature of the events. We thus introduce an evaluation methodology based on a systematic statistical comparison between raw and filtered HPC measurements under different levels of perturbation.

3 Experimental Setup

The main obstacle to using HPCs for security purposes is the noise caused by concurrent processes and the OS. To address this, we implemented a process-aware HPC filtering system designed to isolate the user-mode microarchitectural footprint of individual processes. Implementing this system requires integrating the filtering logic directly into the OS scheduler to accurately track HPCs during context switches. Additionally, to avoid incorrectly attributing kernel activity to the monitored process, the system must suspend the HPC as early as possible during privilege transitions (system calls or interrupts) and resume them only upon returning to user mode. Finally, to comply with the RISC-V specification, since the hardware manages these counters at a privilege level higher than that of the kernel, explicit cooperation with the underlying firmware layer is required.

3.1 Filtering Implementation

We implemented and evaluated this architecture on an embedded RISC-V platform. Since standard instrumentation tools such as `perf` are not currently fully compatible with RISC-V targets, we have designed a custom software solution. Concretely, we suspend the HPCs at the very beginning of the Linux Trap Handler, immediately after saving the context. Furthermore, since counter configuration on this architecture is restricted to a privilege mode higher than that of the kernel, we extended both the Linux kernel and the underlying firmware layer to enable seamless, software-driven counter control without requiring hardware modifications.

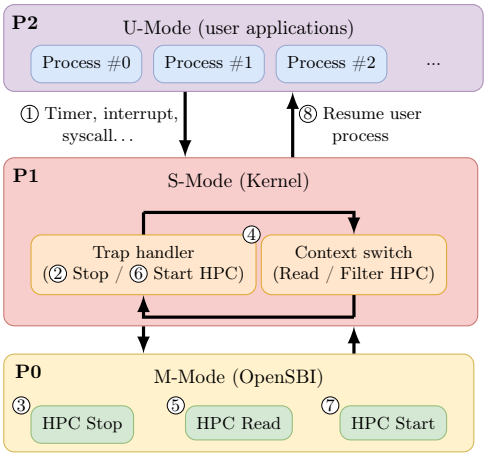

Fig. 1. Overview of HPC filtering by process

Figure 1 details the steps involved in filtering HPC by positioning them according to their respective execution privilege levels. The RISC-V ISA is organized into three distinct privilege levels:

P0. **M-Mode (OpenSBI):** The highest privilege level (Machine Mode), running the Open Supervisor Binary Interface (OpenSBI).
P1. **S-Mode (Linux Kernel):** The supervisor level, responsible for kernel operations.
P2. **U-Mode (User Applications):** The least privileged mode where user processes are executed.

The filtering process follows a chronological sequence (labeled 1 through 8) triggered by a system event:

1. **Exception/Interrupt Trigger**: The flow initiates when a running user process is interrupted by an event such as a timer, a hardware interrupt, or a system call. Forcing a transition from P2 to P1.
2. **Trap Handling**: Upon entering P0, the Trap Handler immediately issues a command to stop the HPCs before any kernel instruction can be recorded.
3. **HPC Suspension**: This request is delegated to P1, where the HPC STOP operation is executed via OpenSBI. Suspending the counters at this earliest entry point is the critical mechanism that prevents kernel-side activity from polluting the measurement.
4. **Context Switch**: The kernel in P1 proceeds to the Context Switch phase. During this phase, the system performs a "Read & Filter HPC" operation.
5. **Data Retrieval**: This requires a transition to P0 to execute HPC READ, allowing the kernel to save the HPCs values associated with the outgoing process.
6. **Process Restoration**: Once the context switch logic is complete, control returns to the Trap Handler in P1. The handler prepares to resume execution and issues a command to restart the counters.
7. **HPC Reactivation**: This triggers the HPC START operation in P0, ensuring HPCs are active for the incoming process.
8. **Return to P2**: Finally, the system exits P1 and resumes the execution of the user process in P2.

This approach ensures that user-level process monitoring is not contaminated by the activity of the OS or the other processes. The metrics collected should therefore accurately reflect the actual microarchitectural behavior of the application, making this implementation suitable for security applications. To ensure the reproducibility of our findings, a GitHub repository containing the complete experimental setup, configurations, and source code required to replicate these results is publicly available[1].

3.2 CVA6 HPC Events

HPC events are inherently heterogeneous. This distinction directly impacts their sensitivity to background noise, the effectiveness of software filtering, and their utility for security analysis. Based on the CVA6 platform [9], we classify these

[1] github.com/labsticc-arcad/HPC-Filtering-Analysis

Table 1. Classification of CVA6 HPC events based on their sensitivity to the activity of concurrent processes and the OS.

Label	Event IDs	Sensitivity	Interference Source
G_1	5, 6, 9, 12, 13	Isolated	None (Instruction-stream dependent).
G_2	10, 11	Moderate	Branch Target Predictor Pollution. Noise depends on the preceding task's control flow.
G_3	1–4, 16, 17	High	Cache and TLB Evictions (Cold start effect). Physically shared resources regardless of software filtering.

events into three distinct groups (Table 1). While our evaluation spans the entire dataset, our analysis focuses on eight core events frequently leveraged in HPC-based detection literature [10], [11], [12]: L1 I-Cache Misses (1), L1 D-Cache Misses (2), ITLB Misses (3), DTLB Misses (4), Load Accesses (5), Store Accesses (6), Branch Instructions (9), and Branch Mispredicts (10).

Architectural events (G_1) are strictly tied to the process's instruction stream (e.g., loads, stores, branches). Since the filtering mechanism resets or restores these counters at each context switch, concurrent processes cannot alter them. This temporal isolation ensures quasi-determinism, making G_1 events highly reliable metrics for behavioral profiling and anomaly detection.

Speculative state events (G_2) track microarchitectural structures, such as branch predictors, that persist across context switches to preserve performance. This structural sharing introduces an indirect inter-process coupling: a preceding process can pollute the predictor tables and artificially increase the misprediction rate of the subsequent monitored task. While this creates residual noise, it also allows G_2 events to capture abnormal environmental activity.

Memory hierarchy events (G_3) correspond to shared physical resources like caches and TLBs. Even if software filtering successfully excludes kernel-induced events, a memory-intensive concurrent process can evict the cache lines of the suspended target, causing a "cold-start" effect upon resumption. Consequently, G_3 exhibits the highest residual variability and is the hardest to isolate. However, because cache and TLB evictions underpin most microarchitectural attacks (e.g., Flush+Reload [13], Prime+Probe [14]), these events provide critical evidence of resource-contention threats.

An event's nature dictates both its filtering behavior and its security application. Instead of treating them as interchangeable features, a robust detection framework must leverage them as complementary layers: G_1 provides clean, process-local baselines, while G_2 and G_3 act as sensors for broader, system-level threats.

4 Filtering Efficiency

Building on the classification introduced in Section 3.2, we evaluate filtering effectiveness across the three event groups rather than through a single aggregate metric. Our analysis relies on a comparative statistical study between raw

and filtered HPCs using the Coefficient of Variation (CV) ($CV = \sigma/\mu$) to enable dimensionless comparisons across different magnitudes. However, real-world HPC distributions are highly asymmetric: interrupts and cache cold-start effects introduce extreme outliers that inflate the mean (μ), making the classical CV unreliable. To ensure stability against these heavy-tailed distributions, we substitute the mean with the median ($\tilde{x}$), which is inherently robust to outliers. We therefore adopt the CV_{med}, defined as:

$$\text{CV}_{\text{med}} = \frac{\sigma}{\tilde{x}} \tag{1}$$

where σ represents the sample standard deviation. The following subsections leverage this metric to quantify the variance reduction achieved by our filtering mechanism across each group and analyze their residual instability.

4.1 Experimental Protocol

The experimental protocol evaluates the filtering mechanism's resilience through simultaneous acquisition: capturing both raw and filtered HPC vectors concurrently during a single run to eliminate cross-trial variability. Here, noise is defined as any OS or concurrent process activity competing for microarchitectural resources. To test the filter under various stress levels, we employ an incremental noise strategy, sweeping the number of concurrent processes n from 1 (near-idle) to 100 (resource saturation for an embedded system). For each run, background processes are randomly selected using unique seeds to ensure diverse and unpredictable contention patterns. Each configuration is repeated for r iterations, yielding a comprehensive dataset of paired vectors $\{HPC_{\text{raw}}, HPC_{\text{filtered}}\}$ annotated with metadata (process type, noise level, seed, and iteration index). Finally, the Embench suite [15] plays a dual role: it serves as the workload pool from which background processes are randomly drawn, and it provides the two benchmarks selected as our monitored case studies (AES and matrix multiplication).

4.2 Analysis of Group 1 Events

In this first phase of the experimental evaluation, we focus on events belonging to G_1. These events include, for example, the number of load instructions (event 5), the number of store instructions (event 6), and the number of branch instructions (event 9). Since these events reflect the static structure of the program path, their values are expected to remain constant when the same program is re-executed in the same environment with identical input data. This characteristic makes G_1 events ideal candidates for assessing the effectiveness of filtering. To provide an intuitive understanding of the filter's impact, we first examine the probability density of raw and filtered HPC values. Figure 2 presents both raw and filtered HPC values for Event 5 (Load instructions), obtained during the execution of the AES benchmark with $n = 10$ and $r = 100$. The distribution of raw HPC exhibits a broad, bell-shaped profile, reflecting the substantial

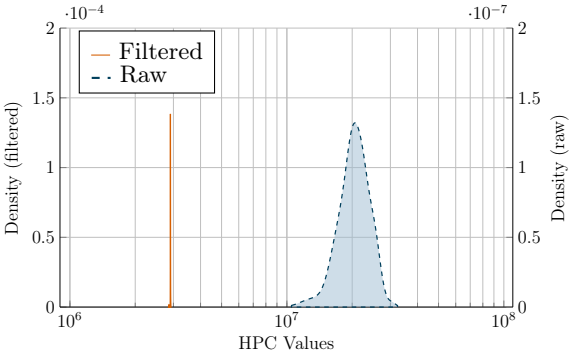

Fig. 2. Distribution of raw and filtered HPC values for the AES benchmark on Load Accesses (Event 5) with $n = 10$ and $r = 100$.

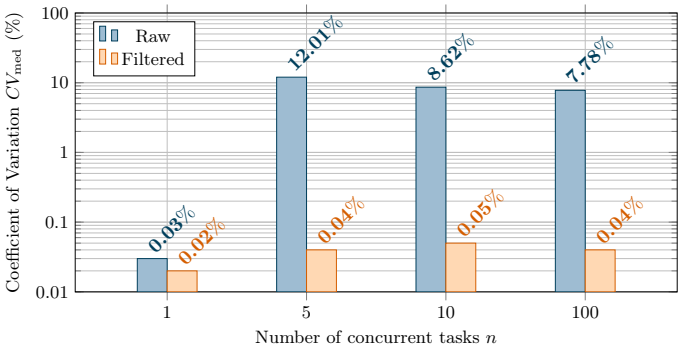

Fig. 3. CV_{med} for raw and filtered HPC for the AES benchmark on Load Accesses (Event 5) with ($n \in \{1, 5, 10, 100\}$) and $r = 100$.

and variable contamination caused by the cumulative activity of concurrent processes. This dispersion is the combined result of several factors: the interleaving of instructions from concurrent processes, the servicing of hardware interrupts, and context switches. In contrast, the distribution of filtered HPC collapses into a narrow, spike-like peak centered, with a spread that is nearly an order of magnitude smaller than that of the distribution of raw HPC. This narrowing provides strong visual evidence that the filter successfully isolates the microarchitectural footprint, effectively stripping away the noise.

Figure 3 and Table 2 present the CV_{med} for all three G_1 events for the AES benchmark with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. In a noise-free environment ($n = 1$), CV_{med} Raw for Event 5 is naturally very low (0.03%), since the OS introduces negligible noise in the absence of concurrent processes. However, as soon as five concurrent processes are introduced ($n = 5$), CV_{med} Raw surges to 12.01%, indicating that the HPC value becomes unreliable. CV_{med} Filtered, by contrast, remains at 0.04% under the same conditions. This robustness is maintained consistently regardless of the noise: CV_{med} Filtered never exceeds 0.05% for Event 5, even with $n = 100$. A similar pattern is observed for Events 6 and 9. Event 6 (Store instructions) exhibits CV_{med} Raw that climbs to 13.68% and 14.49% at $n = 5$ and $n = 10$ respectively, yet filtering constrains it to at

Table 2. CV_{med} for raw and filtered HPC for the AES benchmark across G_1 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value.

Event	Metric	Number of concurrent processes n			
		1	**5**	**10**	**100**
5	CV_{med} Raw	0.03%	12.01%	8.62%	7.78%
	med Raw	$\sim 10^6$	$\sim 10^7$	$\sim 10^7$	$\sim 10^7$
	CV_{med} Filtered	0.02%	0.04%	0.05%	0.04%
	med Filtered	$\sim 10^6$	$\sim 10^6$	$\sim 10^6$	$\sim 10^6$
6	CV_{med} Raw	0.15%	13.68%	14.49%	6.26%
	med Raw	$\sim 10^5$	$\sim 10^6$	$\sim 10^6$	$\sim 10^7$
	CV_{med} Filtered	0.13%	0.19%	0.40%	0.19%
	med Filtered	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$
9	CV_{med} Raw	1.27%	21.01%	14.76%	10.50%
	med Raw	$\sim 10^5$	$\sim 10^6$	$\sim 10^6$	$\sim 10^7$
	CV_{med} Filtered	1.06%	0.72%	0.86%	0.77%
	med Filtered	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$

most 0.40%. Event 9 (Branch instructions), which shows the highest baseline variability among the three (1.27% at $n = 1$), sees its CV_{med} Raw reach 21.01% at $n = 5$. After filtering, the CV_{med} stabilizes in the range 0.72% to 1.06%.

An interesting secondary observation concerns the non-monotonic behavior of CV_{med} Raw: it peaks at $n = 5$ for most events and then decreases slightly as the number of concurrent processes increases further. This apparent paradox can be explained by the progressive saturation of shared hardware resources (caches, execution units, memory bandwidth). At moderate noise levels, each additional process introduces new interference patterns, maximizing variability. At high noise levels, the system reaches a steady state of saturation where the marginal impact of each additional processes is statistically smoothed out. This saturation effect is also reflected in the median order of magnitude of the raw HPCs values, which shifts from 10^6 ($n = 1$) to 10^7 ($n = 100$), confirming that the raw HPCs are inflated by extraneous activity.

To verify that the observed filtering effectiveness is not specific to the AES benchmark, we repeat the entire analysis using the matrix multiplication benchmark with identical experimental conditions. Table 3 presents the corresponding results. The matrix multiplication results corroborate the findings obtained with the AES benchmark. CV_{med} Raw values are comparable in magnitude, and CV_{med} Filtered consistently remains below 0.26% for Events 5 and 6, and below 0.25% for Event 9. The same non-monotonic trend in CV_{med} Raw with increasing the number of concurrent processes is observed, further supporting the saturation hypothesis.

4.3 Analysis of Group 2 Events

We now extend our evaluation to the second group of events, G_2. The principal event of this category is the number of branch mispredictions (Event 10). Unlike the isolated events of group G_1, the HPCs events in G_2 depend on shared microarchitectural structures that maintain a dynamic internal state. The branch

Table 3. CV_{med} for raw and filtered HPC for the matrix multiplication benchmark across G_1 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value.

Event	Metric	Number of concurrent processes n			
		1	**5**	**10**	**100**
5	CV_{med} Raw	0.03%	16.15%	10.57%	6.16%
	med Raw	$\sim 10^6$	$\sim 10^7$	$\sim 10^7$	$\sim 10^7$
	CV_{med} Filtered	0.03%	0.04%	0.05%	0.05%
	med Filtered	$\sim 10^6$	$\sim 10^6$	$\sim 10^6$	$\sim 10^6$
6	CV_{med} Raw	0.18%	14.89%	11.54%	6.30%
	med Raw	$\sim 10^5$	$\sim 10^6$	$\sim 10^6$	$\sim 10^7$
	CV_{med} Filtered	0.15%	0.18%	0.23%	0.26%
	med Filtered	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$
9	CV_{med} Raw	0.11%	17.01%	9.85%	7.30%
	med Raw	$\sim 10^5$	$\sim 10^6$	$\sim 10^6$	$\sim 10^7$
	CV_{med} Filtered	0.09%	0.25%	0.19%	0.17%
	med Filtered	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$

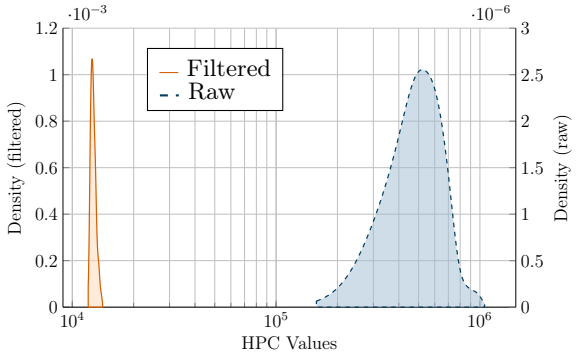

Fig. 4. Distribution of raw and filtered HPC values for the AES benchmark on Branch Mispredicts (Event 10) with $n = 10$ and $r = 100$.

predictor is a component that continuously updates its prediction tables based on the history of recently executed branches, regardless of which process generated them. Consequently, even in a perfectly controlled, noise-free environment, G_2 HPCs events exhibit a non-zero variability that reflects the nature of the predictor's training process. A successful filter should eliminate the former while preserving the latter, since the intrinsic variability carries genuine information about the monitored process' interaction with the underlying hardware.

Figure 4 presents both raw and filtered HPC values for Event 10 (branch mispredictions), obtained during the execution of the AES benchmark with $n = 10$ and $r = 100$. The distribution of raw HPC exhibits a broad, asymmetric profile. This considerable spread reflects the superposition of two distinct noise sources. The first is the same environmental contamination observed for G_1 events: the cumulative HPC values of concurrent processes, hardware interrupts, and context switches inflate and disperse the measurements. The second source is specific to G_2 events: the branch predictor's internal state is a shared microarchitec-

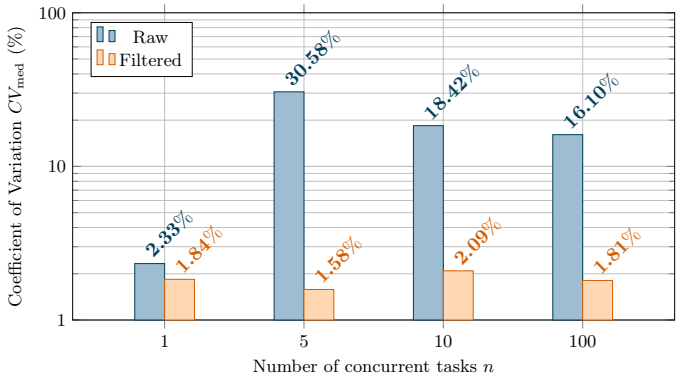

Fig. 5. CV_{med} for raw and filtered HPC for the AES benchmark on Branch Mispredicts (Event 10) with ($n \in \{1, 5, 10, 100\}$) and $r = 100$.

Table 4. CV_{med} for raw and filtered HPC for the AES benchmark across G_2 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value.

Event	**Metric**	**Number of concurrent processes n**			
		1	**5**	**10**	**100**
10	CV_{med} Raw	2.33%	30.58%	18.42%	16.10%
	med Raw	$\sim 10^4$	$\sim 10^5$	$\sim 10^5$	$\sim 10^6$
	CV_{med} Filtered	1.84%	1.58%	2.09%	1.81%
	med Filtered	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$

tural resource whose training history is influenced by all concurrent processes. When concurrent processes execute branch patterns that differ from those of the monitored process, they effectively pollute the predictor's tables, leading to an increased misprediction rate for the monitored process. This cross-process interference on the predictor state introduces a source of variability that is mechanistically distinct from the simple counter accumulation effect seen in G_1. The distribution of filtered HPCs, shown on the left axis of Figure 4, collapses into a narrow peak. This narrowing demonstrates that the filter successfully isolates the misprediction behavior of the monitored process, neutralizing the vast majority of the noise introduced by concurrent processes. However, a careful comparison with the G_1 results reveals that the filtered peak for Event 10 is perceptibly wider than the spike observed for Event 5 (Load instructions), consistent with the expected non-zero intrinsic variability of branch prediction outcomes.

Figure 5 and Table 4 present the CV_{med} for Event 10 across the four noise levels ($n \in \{1, 5, 10, 100\}$), with $r = 100$. In the noise-free baseline ($n = 1$), the CV_{med} Raw of Event 10 is 2.33%. Filtering reduces this value to 1.84%. This gain is consistent with expectations: even at $n = 1$, the OS kernel periodically executes during timer interrupts, context switches, and housekeeping routines. By suspending HPCs counting during these intervals, the filter removes this residual OS noise, yielding a cleaner measurement of the monitored process' misprediction behavior. The introduction of concurrent processes dramatically degrades

Table 5. CV_{med} for raw and filtered HPC for the matrix multiplication benchmark across G_2 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value.

Event	Metric	Number of concurrent processes n			
		1	**5**	**10**	**100**
10	CV_{med} Raw	0.51%	28.17%	19.26%	11.75%
	med Raw	$\sim 10^4$	$\sim 10^5$	$\sim 10^5$	$\sim 10^6$
	CV_{med} Filtered	0.39%	1.19%	0.97%	0.87%
	med Filtered	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$

the raw HPCs quality, for example at $n = 5$, the CV_{med} Raw surges to 30.58%. This sensitivity can be attributed to the compound effect of two mechanisms: the HPCs counting accumulation from co-running processes (as in G_1), and the cross-process pollution of the branch predictor's internal tables, which amplifies the misprediction count of the monitored process. Filtering proves effective in this scenario, reducing the CV_{med} to only 1.58%. As the noise increases further, CV_{med} Raw follows the same pattern already observed for G_1: it decreases from its peak at $n = 5$ to 18.42% at $n = 10$ and 16.10% at $n = 100$. The corresponding increase in the median order of magnitude of raw HPCs values confirms that the raw HPCs are increasingly contaminated by extraneous activity. CV_{med} Filtered, by contrast, remains confined to a narrow band between 1.58% and 2.09% across all noise levels, with the median order of magnitude holding steady at 10^4. This stability demonstrates that the filter effectively decouples the misprediction measurement from the system's overall activity level, recovering the true per-process event value regardless of the noise.

To ensure that these findings generalize beyond the AES workload, we repeat the analysis using the matrix multiplication benchmark. Table 5 reports the corresponding results.

The matrix multiplication benchmark offers a different execution profile from AES benchmark: its highly regular loop-based control flow, as reflected by its low CV_{med} of only 0.51% (compared to 2.33% for AES). This difference is expected: the loop structure of matrix multiplication produces predictable branch patterns that the predictor can learn effectively, whereas the AES benchmark includes a key preparation phase whose implementation relies on data-dependent control flow, generating less predictable branch sequences.

Despite this structural difference, the impact of concurrent process on raw HPCs measurements is comparable in magnitude: CV_{med} Raw reaches 28.17% at $n = 5$ (against 30.58% for AES benchmark), confirming that the environmental noise dominates the workload characteristics at moderate noise levels. CV_{med} Filtered remains below 1.2% across all noise levels, and drops below 1% for $n \geq 10$. This is systematically lower than CV_{med} Filtered for AES benchmark (1.58% to 2.09%), which is coherent with the lower branch prediction variability. The fact that the filter preserves this workload-specific difference provides evidence that the filtering mechanism retains the microarchitectural signature of each process while removing only the environmental noise.

4.4 Analysis of Group 3 Events

G_3 encompasses HPC events that are directly and strongly dependent on the state of shared microarchitectural resources. The principal representative is the number of L1 instruction cache misses (Event 1). The fundamental difficulty posed by G_3 events stems from the nature of cache-related HPC. Unlike the isolated HPC of G_1 or the moderately sensitive HPC of G_2, cache miss counts are the product of a complex, multi-level interaction: every concurrent process competes for the same finite cache capacity, and each context switch or concurrent memory access can evict cache lines belonging to the monitored process, directly altering its miss count.

A closer examination of the events in G_3 reveals that they naturally divide into two subgroups based on the nature of the microarchitectural resource they monitor: instruction related events, comprising L1 instruction cache misses (Event 1) and ITLB misses (Event 3), and data related events, comprising L1 data cache misses (Event 2) and DTLB misses (Event 4). As we shall see, these two subgroups exhibit markedly different behaviors under noise, reflecting the distinct access patterns that instruction fetches and data accesses impose on their respective caches and TLBs.

Figure 6 presents raw and filtered HPC values for Event 1 (L1 instruction cache misses), obtained during the execution of the AES benchmark with $n = 10$ and $r = 100$. The distribution of raw HPC reveals a qualitatively different structure compared to the previous groups. Rather than the unimodal bell shape observed for G_1 and the broad but single-peaked profile of G_2, the distribution of raw HPC for Event 1 exhibits a multimodal structure with at least three peaks, spanning three orders of magnitude from approximately 10^3 to 10^6. This atypical shape reflects the existence of several distinct cache operating regimes that the system transitions between during execution. The distribution of filtered HPCs, collapses the multimodal raw structure into a single, narrower unimodal peak. The disappearance of the multimodal structure confirms that the filter successfully removes the counter accumulation component of the noise. However, in marked contrast with the filtered peaks observed for G_1 events and the narrow peaks of G_2, the distribution of filtered HPCs for Event 1 retains a non-negligible width. This residual spread reflects an irreducible source of variability specific to cache-dependent events: even though the filter accurately isolates the counter increments attributable to the monitored process, the number of cache misses incurred by the monitored process itself varies from run to run because the cache state at the start of each measurement interval is not deterministic. In other words, while the filter removes the counting contamination, it cannot undo the physical contamination (cache state perturbation) that alters the monitored process' actual execution behavior.

Figure 7 and Table 6 present the CV_{med} for all G_3 events across the four noise levels ($n \in \{1, 5, 10, 100\}$), with $r = 100$. We begin with a detailed analysis of Event 1, which displays the most extreme behavior. The CV_{med} results reveal two distinct trends that align with the instruction/data subgroup decomposition. Instruction-related events (Events 1 and 3) exhibit the most extreme

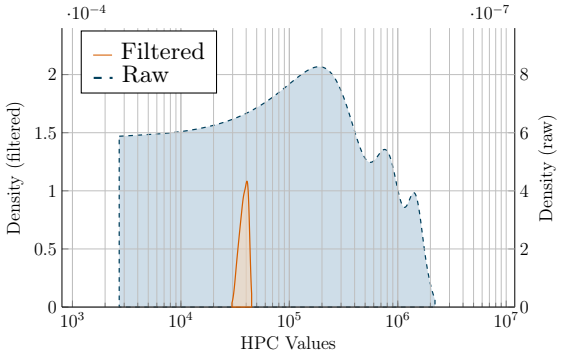

Fig. 6. Distribution of raw and filtered HPC values for the AES benchmark on L1 I-Cache Misses (Event 1) with $n = 10$ and $r = 100$.

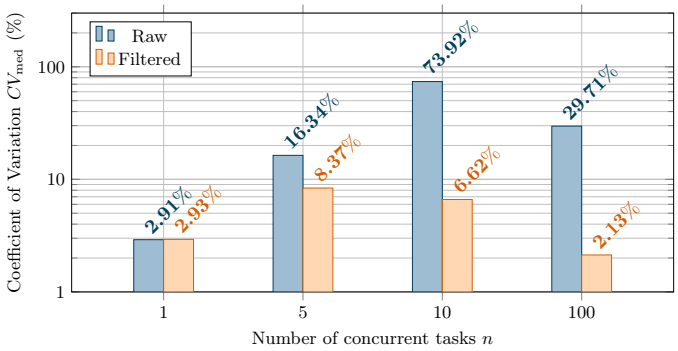

Fig. 7. CV_{med} for raw and filtered HPC for the AES benchmark on L1 I-Cache Misses (Event 1) with ($n \in \{1, 5, 10, 100\}$) and $r = 100$.

and non-monotonic behavior under noise. Event 1 starts at 2.91% with $n = 1$, surges to a peak of 73.92% at $n = 10$, then partially restabilizes at 29.71% for $n = 100$. Event 3 follows a qualitatively similar trajectory, peaking at 11.92% at $n = 5$ before declining to 3.37% at $n = 100$. This non-monotonic pattern can be understood through the lens of cache and TLB pressure dynamics on the instruction side. At intermediate noise levels, the aggregate instruction working set of the concurrent processes approaches but does not fully exceed the capacity of the shared resource. In this transitional regime, the system oscillates unpredictably between states where the monitored process' instruction footprint is partially cached and states where it is largely evicted, producing the multimodal distributions and the high CV_{med} values. At extreme contention ($n = 100$), the instruction cache and ITLB are systematically flushed at almost every context switch, and the monitored process consistently experiences near-maximal miss counts. This quasi-deterministic worst-case regime paradoxically reduces variability, as the system converges toward a single, uniformly degraded operating mode rather than oscillating between multiple regimes. Filtering substantially reduces the CV_{med} for both instruction-related events, but a non-negligible residual variability persists. For Event 1, CV_{med} Filtered ranges from 2.93% to 8.37%,

Table 6. CV_{med} for raw and filtered HPC for the AES benchmark across G_3 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value

Event	Metric	Number of concurrent processes n			
		1	**5**	**10**	**100**
1	CV_{med} Raw	2.91%	16.34%	73.92%	29.71%
	med Raw	$\sim 10^4$	$\sim 10^5$	$\sim 10^5$	$\sim 10^6$
	CV_{med} Filtered	2.93%	8.37%	6.62%	2.13%
	med Filtered	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$
2	CV_{med} Raw	0.91%	11.51%	7.32%	6.52%
	med Raw	$\sim 10^4$	$\sim 10^5$	$\sim 10^5$	$\sim 10^5$
	CV_{med} Filtered	0.87%	1.43%	1.83%	1.27%
	med Filtered	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$
3	CV_{med} Raw	6.14%	11.92%	6.66%	3.37%
	med Raw	$\sim 10^2$	$\sim 10^3$	$\sim 10^3$	$\sim 10^4$
	CV_{med} Filtered	5.80%	5.21%	5.36%	4.61%
	med Filtered	$\sim 10^2$	$\sim 10^2$	$\sim 10^2$	$\sim 10^2$
4	CV_{med} Raw	1.14%	1.98%	1.37%	1.27%
	med Raw	$\sim 10^3$	$\sim 10^3$	$\sim 10^4$	$\sim 10^4$
	CV_{med} Filtered	1.09%	0.96%	1.13%	1.27%
	med Filtered	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$

for Event 3, it remains between 4.61% and 5.80%. This residual spread reflects the irreducible physical contamination specific to instruction-related resources: while the filter removes the counting contamination introduced by concurrent processes, it cannot restore the instruction cache or ITLB state to what it would have been in the absence of contention. Since the instruction stream of a given program is largely deterministic, even small perturbations in the initial cache state propagate throughout execution, producing run-to-run variability that the filter cannot eliminate. Data-related events (Events 2 and 4) display a markedly more moderate and stable response to noise. Event 2 reaches a maximum CV_{med} Raw of 11.51% at $n = 5$, while Event 4 never exceeds 1.98%. This difference could be explained by how instruction-related and data-related accesses interact with their respective caches under contention. Instruction-related resources appear to be more sensitive to transitions, as suggested by the spikes observed for events 1 and 3. Data-related resources, on the other hand, exhibit a more gradual response to increasing noise, with lower overall variability. The filtering response of data-related events is correspondingly more effective. Event 2 CV_{med} Filtered remains between 0.87% and 1.83% across all noise levels, representing a consistent reduction from its raw counterpart. Event 4 shows an even more striking pattern: the filtered and raw CV_{med} values are nearly identical at all noise levels (1.09% against 1.14% at $n = 1$, 1.27% against 1.27% at $n = 100$), suggesting that the data TLB is inherently less susceptible to inter-process contamination, and the residual variability is almost entirely intrinsic to the workload itself.

The median order of magnitude of the counter values further reinforces this observation. For both subgroups, the raw median shifts upward with increasing noise—reflecting the progressive inflation by extraneous activity—while the filtered median remains stable at its baseline value, confirming the filter's abil-

Table 7. CV_{med} for raw and filtered HPC for the matrix multiplication benchmark across G_3 events with ($n \in \{1, 5, 10, 100\}$) and $r = 100$. The symbol $\sim$ denotes the order of magnitude of the median counter value

Event	Metric	Number of concurrent processes n			
		1	**5**	**10**	**100**
1	CV_{med} Raw	6.04%	31.83%	80.89%	36.47%
	med Raw	$\sim 10^3$	$\sim 10^5$	$\sim 10^5$	$\sim 10^6$
	CV_{med} Filtered	5.47%	7.10%	5.32%	5.35%
	med Filtered	$\sim 10^3$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$
2	CV_{med} Raw	0.90%	12.36%	7.37%	5.06%
	med Raw	$\sim 10^4$	$\sim 10^5$	$\sim 10^5$	$\sim 10^6$
	CV_{med} Filtered	0.63%	0.88%	1.07%	0.99%
	med Filtered	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$	$\sim 10^4$
3	CV_{med} Raw	1.84%	14.75%	9.45%	4.93%
	med Raw	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$	$\sim 10^4$
	CV_{med} Filtered	1.58%	3.35%	3.04%	2.73%
	med Filtered	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$
4	CV_{med} Raw	1.21%	3.05%	1.76%	1.84%
	med Raw	$\sim 10^3$	$\sim 10^3$	$\sim 10^4$	$\sim 10^4$
	CV_{med} Filtered	1.20%	1.38%	1.20%	1.87%
	med Filtered	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$	$\sim 10^3$

ity to recover the process-intrinsic signal. However, the magnitude of the raw inflation is consistently larger for instruction-related events (e.g., 10^4 to 10^6 for Event 1) than for data-side events (e.g., 10^4 to 10^5 for Event 2), reflecting the greater susceptibility of instruction resources to shared contention. Table 7 presents the results for the matrix multiplication benchmark under the same experimental conditions. The analysis of the matrix multiplication benchmark in Table 7 corroborates the instruction/data dichotomy. On the instruction side, Event 1 reaches an even more extreme CV_{med} Raw peak of 80.89% at $n = 10$, which the filter reduces to 5.32%, Event 3 peaks at 14.75% and is filtered down to 3.35%. On the data side, Events 2 and 4 follow the same moderate pattern observed for AES, with CV_{med} Filtered remaining consistently below 2%.

5 Malicious Behaviour Detection

The purpose of this section is to demonstrate the value of filtering mechanism by analyzing how it affects the detection performance of several algorithms for each HPC individually. Using the same platform, we study the exploitation of a buffer overflow vulnerability that leads to the execution of arbitrary code. The victim process computes the hash of a given file. The exploit leads to the leak of */etc/passwd*, which is never called during normal execution.

To ensure a rigorous comparison and evaluate the exclusive contribution of filtering, the programs were designed so that benign and malicious executions have similar execution times. This ensures that any observed differences in the HPC metrics come from the nature of the execution rather than its duration. Each execution type, benign and malicious, was executed 100 times, and the corresponding HPCs were recorded and labeled accordingly. To assess the relevance

of filtering under varying levels of system activity, we consider two noise configurations: a baseline scenario with $n = 1$ and a high-contention scenario with $n = 100$. For each noise level and each HPC, we thus obtain four distinct groups of 100 observations: Raw Benign, Raw Malicious, Filtered Benign, and Filtered Malicious, yielding a comprehensive dataset that allows us to assess the impact of both noise and filtering on the separability between benign and malicious execution profiles. Each dataset is partitioned into a training set (70%) and a test set (30%) using scikit-learn's [16] functions with a fixed seed, which performs a shuffle of the indices before splitting, ensuring both reproducibility and a randomized distribution of benign and malicious samples across both partitions. A standard scaler is subsequently fitted on the training partition only, computing the empirical mean and standard deviation of the training features; both partitions are then centered and normalized using these statistics, thereby standardizing the feature space without introducing data leakage from the test set. To carry out this comparison, we employ six widely used machine learning classifiers: AdaBoost, K-Nearest Neighbors (KNN), Logistic Regression (LogReg), Naive Bayes (NB), Random Forest (RF), and Support Vector Machine (SVM). Detection performance is assessed using three complementary metrics: the *F1-score*, the *False Positive Rate (FPR)*, and the *False Negative Rate (FNR)*. Finally, we present results for a carefully selected subset of events. This subset includes both events that are capable of capturing the microarchitectural fingerprint of the attack and events that are inherently insensitive to it. This deliberate choice allows us to observe and contrast the impact of filtering across both categories.

A value $n = 1$, whose results are presented in Table 8, represents the ideal operating condition in which the HPC signal is as little affected as possible by noise. In the absence of concurrent processes, the OS generates little interference. Under these conditions, the raw HPC values already allow for strong discrimination for most classifier-event combinations. Events 4, 5 and 6 consistently yield *F1-scores* above 0.85, and the best overall configurations reach 0.929 for AdaBoost and RF on Event 4, as well as 0.921 for AdaBoost and KNN on Event 5. These results confirm that exploiting buffer overflows leaves a detectable microarchitectural signature across multiple HPCs, even without any filtering. This is clearly visible in Table 8, where Events 4 and 5 consistently achieve high *F1-scores* across all classifiers, while Event 1 systematically underperforms, indicating that this particular counter carries little discriminating information for this type of attack. Filtering yields limited but consistent improvements in this low-noise scenario. The most notable gain is for Event 4, where KNN, LogReg, NB and SVM all achieve an *F1-score* of 0.966 after filtering, compared to a maximum of 0.929 for AdaBoost and RF with raw HPC on the same event. While the improvement in the *F1-score* is moderate, the reduction in the *False Positive Rate (FPR)* is significant. SVM on Event 4, for example, achieves an *False Positive Rate (FPR)* of 80.6% with raw HPC, meaning that more than four out of five benign executions are incorrectly classified as malicious. Filtered HPC reduces this rate to 3.2%. This discrepancy between an acceptable *F1-score* and a high *False Positive Rate (FPR)* reveals a fundamental limitation of evaluat-

Table 8. Performance across 4 HPC events, for the binary classification of benign and malicious executions of a buffer overflow with $n = 1$.

		Raw			Filtered		
Classifier	**Event**	**F1**	**FPR**	**FNR**	**F1**	**FPR**	**FNR**
AdaBoost	1	0.712	0.581	0.103	0.677	0.387	0.276
	4	**0.929**	0.032	0.103	0.929	0.032	0.103
	5	**0.921**	0.161	0.000	0.918	0.129	0.035
	6	0.839	0.226	0.103	0.849	0.290	0.035
KNN	1	0.712	0.290	0.276	0.625	0.484	0.310
	4	0.909	0.032	0.138	**0.966**	0.032	0.035
	5	**0.921**	0.161	0.000	0.918	0.129	0.035
	6	0.875	0.226	0.035	0.814	0.194	0.172
LogReg	1	0.690	0.290	0.310	0.700	0.323	0.276
	4	0.862	0.258	0.035	**0.966**	0.032	0.035
	5	0.867	0.161	0.103	0.918	0.129	0.035
	6	0.800	0.226	0.172	0.857	0.226	0.069
NB	1	0.625	0.484	0.310	0.690	0.290	0.310
	4	0.853	0.323	0.000	**0.966**	0.032	0.035
	5	0.754	0.452	0.103	0.918	0.129	0.035
	6	0.743	0.484	0.103	0.857	0.226	0.069
RF	1	0.702	0.258	0.310	0.525	0.516	0.448
	4	**0.929**	0.032	0.103	0.929	0.032	0.103
	5	0.807	0.161	0.207	0.897	0.097	0.103
	6	0.788	0.355	0.103	0.833	0.194	0.138
SVM	1	0.687	0.484	0.207	0.677	0.387	0.276
	4	0.699	**0.806**	0.000	**0.966**	**0.032**	0.035
	5	**0.921**	0.161	0.000	0.918	0.129	0.035
	6	0.771	0.452	0.069	0.862	0.258	0.035

ing detection solely through *F1-score*. Our results highlight that filtering allows improving detection performance, but also tends to ensure that the classifier's decision boundary is based on the application's actual microarchitectural fingerprint. Event 1 consistently stands out as the weakest event among all classifiers, with overall *F1-scores* ranging from 0.625 to 0.712. This trend persists after filtering, which illustrates that not all events can benefit from the filtering mechanism: when an event is inherently unable to capture the behavioral differences induced by the exploitation of the buffer overflow, removing environmental noise cannot compensate for this lack of discriminative power.

The introduction of 100 concurrent processes, as shown in Table 9, reveals that detection based on raw HPC measurements is significantly impacted by the increase of noise, leading to a notable degradation in classification quality. The average *F1-score* for raw HPC drops compared to the first scenario. Furthermore, the inter-event variation increases, reflecting unequal sensitivity of the counters to noise. As Table 9 shows, raw HPC yield a consistently poor detection landscape: *F1-scores* range from 0.31 to0.77, with AdaBoost achieving an *F1-score* of only 0.311 on Event 1. However, filtering reveals the true discrim-

Table 9. Performance across 4 HPC events, for the binary classification of benign and malicious executions of a buffer overflow with $n = 100$.

		Raw			Filtered		
Classifier	**Event**	**F1**	**FPR**	**FNR**	**F1**	**FPR**	**FNR**
AdaBoost	1	0.311	0.290	0.759	0.588	0.226	0.483
	4	0.769	**0.355**	0.138	**0.929**	**0.032**	0.103
	5	0.444	0.419	0.586	0.628	0.194	0.448
	6	0.583	0.710	0.276	0.716	0.452	0.172
KNN	1	0.462	0.355	0.586	0.600	0.419	0.379
	4	0.632	0.323	0.379	**0.929**	**0.032**	0.103
	5	0.462	0.677	0.483	0.714	0.226	0.310
	6	0.516	0.548	0.448	0.689	0.355	0.276
LogReg	1	0.548	0.516	0.414	0.737	0.226	0.276
	4	0.481	0.387	0.552	0.897	0.097	0.103
	5	0.597	0.581	0.310	0.646	0.484	0.276
	6	0.652	1.000	0.000	0.656	0.387	0.310
NB	1	0.567	0.613	0.345	0.712	0.290	0.276
	4	0.481	0.387	0.552	0.897	0.097	0.103
	5	0.615	0.516	0.310	0.632	0.323	0.379
	6	0.632	0.742	0.172	0.597	0.355	0.414
RF	1	0.444	0.419	0.586	0.571	0.516	0.379
	4	0.655	0.323	0.345	0.912	0.065	0.103
	5	0.492	0.645	0.448	0.604	0.258	0.448
	6	0.509	0.484	0.483	0.618	0.290	0.414
SVM	1	0.500	0.419	0.517	0.667	0.226	0.379
	4	0.480	0.290	0.586	0.897	0.097	0.103
	5	0.618	0.581	0.276	0.690	0.290	0.310
	6	0.508	0.581	0.448	0.691	0.226	0.345

inative potential of the HPC by removing the noise that previously obscured the microarchitectural signature of the attack. Event 4 illustrates this clearly: once filtered, its ability to characterize the presence of the exploitation emerges distinctly. AdaBoost and KNN both achieve an *F1-score* of 0.929, RF reaches 0.912, and LogReg, NB, and SVM converge to 0.897, all with *False Positive Rate (FPR)* values for event 4 reduced to a range from 3.2% to 9.6%. This represents a significant reduction in *False Positive Rate (FPR)* compared to the best result from the raw HPC. The contrast in Table 9 is striking: Event 4's filtered columns stand out with uniformly high *F1-scores* and low error rates, whereas the remaining events show more modest improvements. It should be noted that Events 1 and 6 remain the most challenging to use for detection under both conditions, as confirmed by their consistently lower scores across both tables. Unlike Event 4, whose discriminative potential is revealed once environmental noise is removed, these events are inherently unable to capture the microarchitectural signature of the attack: filtering cannot enhance what the underlying HPC does not measure.

6 Discussion

Our results demonstrate that filtering effectiveness depends heavily on the event's nature, confirming that HPCs cannot be treated as interchangeable features. For isolated events (G_1), filtering eliminates external noise to yield highly stable measurements. For speculative events (G_2), the residual variation represents structured information about the speculative environment, which can be exploited to detect branch predictor attacks [17]. For cache and TLB events (G_3), while filtering cannot prevent physical resource sharing or cold-start effects, it removes the most chaotic noise, restoring a monotonic and interpretable relationship between system load and process execution.

From a security perspective, this mechanism acts as a signal conditioner that isolates the process's true microarchitectural footprint and makes environmental impacts explicit. Our buffer overflow case study validates this claim. Under low system load, raw HPCs achieve acceptable *F1-scores* but suffer from a high *False Positive Rate (FPR)*. Filtering mitigates these false alerts by exposing the actual process fingerprint without altering the detection logic. Under realistic contention, raw counters become entirely unusable, whereas filtered HPCs maintain robust detection capabilities for a subset of events. This underscores that aggregate accuracy metrics like the *F1-score* can be misleading if the underlying balance between *False Positive Rate (FPR)* and *False Negative Rate (FNR)* is ignored.

7 Conclusion

This work addresses the issue of measurement reliability for HPCs in multitasking environments and its direct implications for security applications. Through an experimental evaluation combining statistical and classification analysis, we demonstrated that process-based HPC filtering is an essential prerequisite for the effective use of microarchitectural telemetry. Future works target the evaluation of filtered HPC fusion to create more robust and reliable security metrics.

Acknowledgements

This work was funded by the French Directorate General of Armament (DGA) and the Brittany Region, France.

References

[1] J. Demme et al., "On the feasibility of online malware detection with performance counters," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[2] B. Singh, D. Evtyushkin, J. Elwell, R. D. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[3] K. Arıkan et al., "Processor security: Detecting microarchitectural attacks via count-min sketches," *IEEE Trans. Very Large Scale Integr. Syst.*, 2022.

[4] S. Carnà, S. Ferracci, F. Quaglia, and A. Pellegrini, "Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters," *Digital Threats: Research and Practice*, 2022.

[5] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[6] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," *IEEE Symposium on Security and Privacy*, 2019.

[7] P. Bhade, J. Paturel, O. Sentieys, and S. Sinha, "Lightweight hardware-based cache side-channel attack detection for edge devices (edge-cascade)," *ACM Transactions on Embedded Computing Systems*, 2024.

[8] S. Carnà, S. Ferracci, F. Quaglia, and A. Pellegrini, "Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters," *ACM Digital Threats*, 2023.

[9] OpenHW Group, *Cva6 user manual: Csr performance counters*, `https://docs.openhwgroup.org/projects/cva6-user-manual/01_cva6_user/CSR_Performance_Counters.html`, Accessed: 2026-03-27, 2026.

[10] S. Kadiyala, P. Jadhav, S. Lam, and T. Srikanthan, "Hardware performance counter-based fine-grained malware detection," *ACM Transactions on Embedded Computing Systems (TECS)*, 2020.

[11] M. Mushtaq et al., "Whisper: A tool for run-time detection of side-channel attacks," *IEEE Access*, 2020.

[12] C. Li and J. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 2019.

[13] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Conference on Security Symposium*, 2014.

[14] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: The case of AES*, Cryptology ePrint Archive, Paper 2005/271, 2005. [Online]. Available: `https://eprint.iacr.org/2005/271`.

[15] E. Contributors, *Embench-iot: A benchmark suite for embedded iot systems*, `https://github.com/embench/embench-iot`, Accessed: 2025-03-17, 2019.

[16] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.

[17] C. Li and J. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," *30th International Symposium on Computer Architecture and High Performance Computing*, 2018.