SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security

Sanjeev Das*, Jan Werner*, Manos Antonakakis†, Michalis Polychronakis‡, Fabian Monrose*
*University of North Carolina at Chapel Hill, †Georgia Institute of Technology, ‡Stony Brook University
*{sdas, jjwerner, fabian}@cs.unc.edu, †manos@gatech.edu, ‡mikepo@cs.stonybrook.edu

***Abstract*—Hardware Performance Counters (HPCs) have been available in processors for more than a decade. These counters can be used to monitor and measure events that occur at the CPU level. Modern processors provide hundreds of hardware events that can be monitored, and with each new processor architecture more are added. Yet, there has been little in the way of systematic studies on how performance counters can best be utilized to accurately monitor events in real-world settings. Especially when it comes to the use of HPCs for security applications, measurement imprecisions or incorrect assumptions regarding the measured values can undermine the offered protection.**

To shed light on this issue, we embarked on a year-long effort to (*i*) study the best practices for obtaining accurate measurement of events using performance counters, (*ii*) understand the challenges and pitfalls of using HPCs in various settings, and (*iii*) explore ways to obtain consistent and accurate measurements across different settings and architectures. Additionally, we then empirically evaluated the way HPCs have been used throughout a wide variety of papers. Not wanting to stop there, we explored whether these widely used techniques are in fact obtaining performance counter data correctly. As part of that assessment, we (*iv*) extended the seminal work of Weaver and McKee from almost 10 years ago on non-determinism in HPCs, and applied our findings to 56 papers across various application domains.

In that follow-up study, we found the acceptance of HPCs in security applications is in *stark contrast* to other application areas — especially in the last five years. Given that, we studied an additional representative set of 41 works from the security literature that rely on HPCs, to better elucidate how the intricacies we discovered can impact the soundness and correctness of their approaches and conclusions. Toward that goal, we (*i*) empirically evaluated how failure to accommodate for various subtleties in the use of HPCs can undermine the effectiveness of security applications, specifically in the case of exploit prevention and malware detection. Lastly, we showed how (*ii*) an adversary can manipulate HPCs to bypass certain security defenses.

***Index Terms*—Hardware Performance Counters; Malware Detection; Exploit Defense; Non-determinism;**

I. Introduction

Modern processors (such as Intel, AMD, ARM) support a variety of hardware performance counters for monitoring and measuring events during process execution related to instructions (*e.g.,* instruction retired, cycles), memory accesses (*e.g.,* cache hits or misses, main memory hits or misses), and the execution behavior on the CPU pipeline, among others. In Intel processors, the functional unit that facilitates the use of HPCs is called the Performance Monitoring Unit (PMU).

The hardware events supported by performance counters can be classified as either architectural or non-architectural events (the latter are also known as micro-architectural events). Architectural events comprise events that remain consistent across different processor architectures, *e.g.,* instructions, branches, cycles. Non-architectural events consist of events that are specific to the micro-architecture of a given processor, *e.g.,* cache accesses, branch prediction, and TLB accesses. Unlike architectural events, non-architectural events vary among processor architectures and may also change with processor enhancements. Table I presents a list of commonly used architectural and non-architectural events in Intel processors. A more comprehensive list of events is provided by Intel [1].

Although performance counters were initially introduced for debugging purpose, they have been used in a myriad of applications, such as dynamic software profiling [2], CPU power modeling [3], vulnerability research and malware defenses [4][5][6]. Additionally, several profiling tools (*e.g.,* PAPI [7], *perf_event* [8], VTune [9]) have been designed using performance counters for different environment settings. These tools allow for performance counter data to be obtained using several methods, but that flexibility comes at a price: these techniques can yield different counter values for the same application. Consequently, it is imperative that performance counter implementations, and proposals that rely upon them, follow certain principals in order to obtain trustworthy values.

To that end, we studied a number of tools, techniques and papers to understand how issues related to non-determinism and overcounting in HPCs are commonly dealt with. Our painstaking analyses led to several intriguing revelations. First, we found mixed opinions on using HPCs: more than 45% of the papers in application areas that include performance analysis and optimizations, high performance computing and OS support *do not recommend* using HPCs. The main stated reasons for dismissing HPCs are the lack of determinism in performance counter values [10–23] and lack of portability of HPC events (*e.g.,* some events may be present in one architecture but not in another [24–27]). These factors limit the applicability of performance counters in different application domains.

Yet, we found that the use of HPCs for security purposes is in stark contrast to other areas, as evidenced by the increasing number of techniques that rely on HPC measurements for offensive and defensive research. Only 10% of these papers acknowledged the non-determinism issues in HPCs. None of them attempt to address the measurement error, and only a few of the papers [28–31] explicitly argue why their approach can tolerate non-determinism and overcounting effects.

As we show later, while non-determinism may not have dire consequences for certain applications (*e.g.,* power estimation or high performance computing), it can have significant impact on approaches whose *security* rests on having accurate and

consistent HPC measurements. For example, malware and exploit defenses [5, 6, 32] are susceptible to non-determinism effects and contamination of events. This is mainly because in security applications, the attack models rely on small variations in performance counter data to differentiate suspicious from benign behaviors [6, 32]. Even a minor variation of 1-5% in counter values [28], can lead to poor performance of those models. Therefore, it is particularly important that in security settings, these challenges are addressed.

In bringing these challenges and pitfalls to the forefront, we make several contributions, including:

- Summarizing proper techniques for instantiating and using hardware performance counters.
- Studying nearly 100 papers to synthesize how HPCs have been used and adopted in different application domains.
- Extending the seminal work of Weaver & McKee [33] in order to better examine what problems persist in modern processors 10 years after the original study.
- Contrasting HPC-based monitoring techniques and presenting ways to improve them.
- Empirically demonstrating how failure to cover key aspects in the collection of HPC events can undermine the correctness of approaches that rely on accurate HPC measurements for security purposes.
- Demonstrating how an adversary can manipulate HPCs to bypass certain security defenses.
- A set of recommendations regarding the use of HPCs.

II. Background and Challenges

For pedagogical purposes, we present a primer on hardware performance counters and then discuss the implementation challenges that are involved in accurately monitoring events.

To obtain performance counter information from the hardware, the counters must first be configured according to the events of interest. Afterwards, at run time, the counters can be read using two techniques, namely *polling* or *sampling*. We discuss each step in turn.

1) Configuring the counters: Performance counters must be configured in kernel mode, but they can be read in user mode. Moreover, although hundreds of events are available for monitoring, only a limited number of counters can be used simultaneously. Therefore, one must carefully pick and configure which events to monitor using the available counters. The number of available counters varies between processor architectures, *e.g.,* modern Intel CPUs support three fixed and four programmable counters per core [1]. Fixed counters monitor fixed events such as instruction retired, logical cycles, and reference cycles, while programmable counters can be configured to monitor architectural and non-architectural events. The configuration of performance counters is performed by writing into model specific registers (MSRs).

2) Reading Counter Values: Performance counters can be read by either sampling or polling.

- Polling: The counters can be read at any instant. Counters are read using the MSRs. For that purpose, Intel uses specific instructions (`rdmsr`, `wrmsr`) to read from and write to MSRs, respectively. Reading counters from user space can be done using the `rdpmc` instruction. Fig. 1 presents an example of how polling can be used to measure a user space process.
- Event-based sampling: HPCs also support sampling of various metrics based on the occurrence of events. This feature is enabled in most CPUs through a specific interrupt, called *Performance Monitoring Interrupt (PMI)*, which can be generated after the occurrence of a certain number of events. For instance, one can configure HPCs with a certain threshold, which will result in the generation of a PMI once a counter exceeds that threshold. Fig. 2 shows how an event is configured with a specific threshold, n, for the number of instructions retired that should generate a PMI. At each PMI (*i.e.,* after every n retired instructions), the numbers of cycles and arithmetic, call, direct call, and return instructions are read.

A. Challenges and Pitfalls

Unfortunately, the seemingly simple nature of using HPCs for performance measurements becomes complicated due to several sources of contamination, which may lead to discrepancies in the actual measurement of events[1]. Reasons that can lead to inaccurate measurement include:

- *External sources:* The runtime environment may vary across runs. For example, OS activity, scheduling of programs in multitasking environments, memory layout and pressure, and multi-processor interactions may change between different runs. Similarly, micro-architectural state changes may cause discrepancies in the events' count.
- *Non-determinism:* Weaver *et al.* [34] provide an overview of the impact of non-deterministic events (*e.g.,* hardware interrupts like periodic timers) in HPC settings. Many sources of non-determinism can be hard to predict and mostly depend on OS behavior and the other applications (besides the measured one) running on the system.
- *Overcounting:* Performance counters may overcount certain events on some processors. For instance, on Pentium D processors, the instruction retired event may be overcounted on several occasions, most notably, during x87/SSE exceptions, lazy floating point handling by OS, and execution of certain instructions such as `fldcw`, `fldenv`, and `emms` [34].
- *Variations in tool implementations:* Many tools have been developed to help obtaining performance counter measurements. Oftentimes, these tools yield different results even in a strictly controlled environment for the same application. The variation of measurements may result from the techniques involved in acquiring them, *e.g.,* the point at which they start the counters, the reading technique (polling or sampling), the measurement level (thread, process, core, multiple cores), and the noise-filtering approach used.

[1] Although this paper is mainly focussed on the Intel x86 architecture, non-determinism due to HPCs has also been observed on AMD x86 processors [34], and thus our findings may be relevant for those as well.

TABLE I: Hardware Events

Architectural Events	Description	Non-architectural Events	Description
1. Ins	Instruction retired	9. Uops_Retired	All micro-operations that retired
2. Clk	Unhalted core cycles	10. Mem_Load_Uops_Retired	Retired load uops
3. Br	Branch instructions	11. Mem_Store_Uops_Retired	Retired store uops
4. Arith_Ins	Arithmetic instructions	12. Br_Miss_Pred_Retired	Mispredicted branches that retired
5. Call	Near call instructions	13. Ret_Miss	Mispredicted return instructions
6. Call_D	Direct near call instructions	14. Call_D_Miss	Mispredicted direct call instructions
7. Call_ID	Indirect near call instructions	15. Br_Far	Far branches retired
8. Ret	Near return instructions	16. Br_Inst_Exec	Branch instructions executed
		17. ITLB_Miss	Misses in ITLB
		18. DTLB_Store_Miss	Store uops with DTLB miss
		19. DTLB_Load_Miss	Load uops with DTLB miss
		20. LLC_Miss	Longest latency cache miss

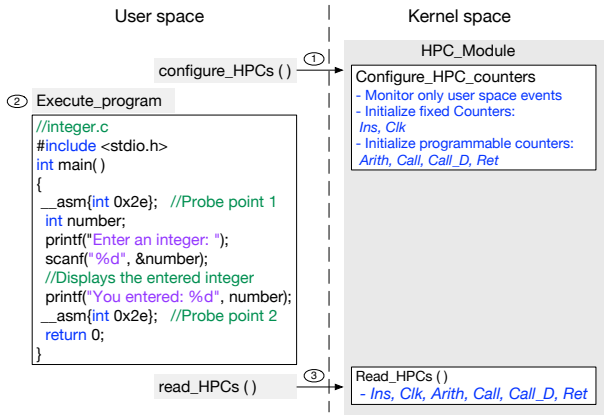


Fig. 1: Polling: *First, the counters are configured in kernel space with the events of interest. Later, they can be read either in user or kernel space.*

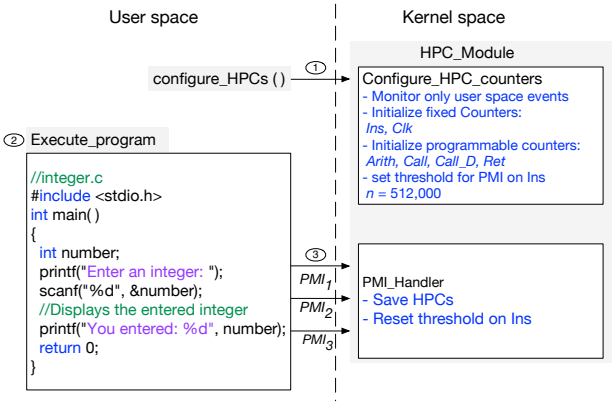


Fig. 2: Event-based Sampling: *PMI is configured using instruction retired event in kernel space, with a threshold set at n. Once instructions retired exceed n, a PMI is triggered; then all the counters can be read in kernel space.*

III. Proper Instantiation and Usage

Acquiring accurate measurements using performance counters requires careful consideration of operational aspects.

Context switch monitoring: Performance counters measure system-wide events and are agnostic to which process is running on the processor. Most processors support separation of kernel and user space events, but per-process filtering of counter values is not available. In order to profile the runtime behavior of a process, performance counter values must be saved during context switches to avoid any contamination due to events from other processes. Monitoring context switches involves patching the kernel, which is a non-trivial task, especially in closed-source OSes.

Interrupt handling: Recall that performance counters are typically used in conjunction with performance monitoring interrupts (PMI). This feature is is not essential when reading events in sampling mode; it can also profile events at a finer granularity. PMIs can be handled in different ways, such as by writing a callback routine, API hooking, or hooking the PMI handler [35]. Note that proper PMI handling may be challenging due to, for example, the lack of documentation or inability to patch the operating system.

Process filtering upon PMI: Since the performance monitoring unit operates at the hardware level, it is OS-process agnostic. Therefore, when PMI is configured, the PMU can generate hardware interrupts for all processes running on a given processor core. Consequently, to accurately profile an application, it is necessary to implement a technique for filtering performance counter data relevant solely to the process of interest. Otherwise, counter data will be contaminated by the events of other processes.

Minimizing the impact of non-deterministic events: As discussed in §II-A, performance counters suffer from non-determinism and event overcounting. Thus, for several applications, it is important to consider only deterministic events. A deterministic event is defined as an event whose value does not vary between identical runs and matches the expected values that would be obtained through alternative means, e.g., through code inspection. Unfortunately, this is a daunting challenge because some events may be deterministic in some processor architectures, but not in others [34]. We return to this later in §VI.

IV. Contemporary Approaches and Tools

Based on our study, we have identified two main strategies in the literature for recording HPC measurements at runtime. The first is via source code instrumentation. In this approach, the source code is instrumented with probing points before and after the main code. The probing points instruct the kernel module when to initialize and when to terminate the monitoring of the performance counters. For example in Fig. 1, the sample code is instrumented using the probe points `__asm{int 0x2e}`. The major challenge of this approach is that it requires source code modification, which may not always be possible, especially for closed-source software.

Alternatively, a binary-compatible approach can be taken. One solution is to create the target process in suspended mode, and then enable the performance counters. Once the process terminates, performance counter values for the process are extracted. The drawback of this method is that monitoring starts immediately after the process is created, which is much earlier than the actual program begins. This is because a process is a container for a set of resources used and follows several steps before the actual execution of the program begins [36]. An alternative is to monitor a running process by attaching to and detaching from it after a particular condition is met. This approach is followed by popular monitoring tools, *e.g.,* Perf, and Oprofile [37]. In this technique, a new thread is created in suspended mode in the address space of the existing process, and then the performance counters are initialized. The counters are terminated once the process halts or after a predefined state is reached. One can use this method to monitor events at a finer granularity, *e.g.,* to monitor a function.

V. Dos and Don'ts

It should be clear by now that obtaining accurate and reliable performance counter values can be tricky, if not frustrating. To lessen the burden on programmers, a number of utilities and tools have been developed to obtain performance counter information on different platforms. To study how these tools and techniques have been used, we selected a set of papers that mentioned the issues of non-determinism in performance counters that were initially highlighted by Weaver & McKee [33]. The papers were normalized based on the citation count and the ranking of the venue they appeared in, after which we choose the most cited of these papers in the past 10 years. The result was a set of 56 papers, listed in Table II.

Performance counter tools facilitate the measurement of events at three levels of granularity:

Coarse Measurement: logs the aggregate occurrence of the configured events for the entire execution. The most popular tool that supports this functionality is *Perf_event*, which uses a polling mechanism to read the counters.

Snippet Measurement: analyzes events related to a section of the program instead of the entire program. To support this capability, a high-level interface is provided to obtain the performance counter values. *PAPI* is one such cross-platform tool that employs source code instrumentation to measure events for a section of the program [7]. A polling mechanism is used to read the counters.

Fine-grained Measurement: samples events using an interrupt-based mechanism. HPCs are preconfigured to generate an interrupt every n events. *Perf_event* and *Oprofile* support monitoring of an application based on the number of occurrences of specific events. The use of sampling can allow for the most fine-grained measurement, *e.g.,* when $n = 1$.

Table II lists a variety of works that rely on common tools (*e.g.,* Perf_event, Pfmon, perfctr, PAPI, Intel VTune) or custom implementations for profiling HPC information. As described earlier, a potpourri of mechanisms are used, including polling [69–71, 75–77] or event-based sampling [4–6, 28, 32, 68, 78, 79] to measure events at different granularities. Some proposals [4–6, 28, 32, 78, 79] apply per-process filtering in order to sample only the events of a specific process. Tools such as Perf_event and Intel VTune support that feature.

A. Trends and Concerns

Given the major issues of non-determinism and overcounting in HPCs, we decided to look more closely at the 56 papers to assess *i*) whether the authors acknowledged the impact of these issues in their analysis, *ii*) whether they addressed the challenges that arise due to non-determinism, and *iii*) the extent to which measurement error (due to non-determinism) was explicitly accounted for in their empirical analysis.

As noted in §I, we found mixed opinions on using HPC in empirical analyses. The lack of determinism in performance counter values [10–23] and the lack of portability of HPC events [7, 25, 26, 99] were two of the main reasons for not using HPCs in certain domains. Despite the acknowledgment of these issues by works in all application domains, there has been virtually no attempt towards resolving the measurement errors resulting from them. Alas, a few approaches [70–72] even go as far as arguing that their techniques are not affected, claiming that measurement variations are insignificant.

That dismissal is juxtaposed by two extreme views on HPCs. On one hand, Lundberg et al. [61] noted that non-determinism is a commonly observed behavior in multiple CPU environments because of OS operations and multiple applications running on the system. Two different runs of the same program with exactly same inputs may not be identical. Because of this fact, the performance counter may record different event counts for two different runs of the same program, and thus should not be relied upon. On the other extreme, relying entirely on the non-deterministic nature of performance counters, Suciu *et al.* [73] leverage that "feature" to generate random numbers.

After noticing the rise of HPC usage in security for offensive (*e.g.,* [109]) and defensive techniques (*e.g.,* [5, 6, 32]), we decided to extend our analysis to cover 41 security papers (shown in Table III) that used HPCs and appeared in the past 10 years[2]. In Table III, we display whether HPCs were recommended or not. In order to make that determination, we used the following criteria for 'yes': the authors either explicitly advocate using HPCs (irrespective of whether they acknowledge issues related to non-determinism) or HPCs are used for the main application domain of their approach, even if the paper disregards HPCs for a specific purpose. For instance, Gruss *et al.* [100] showed that HPCs cannot be used for detecting "flush+flush" attacks, but explicitly state that HPCs can be used for their main application domain, such as for detecting cache attacks, rowhammer attacks, and also "flush+reload" attacks. Only 4 (10%) out of these papers acknowledged the non-determinism issues associated with using HPCs. While all of them recommend using HPCs for security purposes, none attempt to address the measurement error due to non-determinism, and only a few [28–31] argue why their approaches are not affected (*e.g.,* because they observed only minor variations in the measurement of events). While it may be true that some approaches (e.g., [31]) may not be adversely

[2]We normalized the papers based on the citation count and the ranking of the venue they appeared in.

TABLE II: Hardware Performance Counters Usage in Selected Papers

Application	Authors	Non-determinism acknowledged	Non-determinism challenges addressed	Measurement error addressed	Recommend using HPCs
Auto-tuning/Profiling	Zaparanuks *et al.* [38]	●	○	○	○
	Weaver [8]	●	●	○	○
	Weaver *et al.* [34]	●	●	○	○
	Weaver & Dongarra [39]	●	●	○	○
	Rohou [40]	●	○	○	●
	Nowak *et al.* [41]	●	○	○	●
	Lim *et al.* [42]	●	○	○	●
	Moseley *et al.* [43]	●	○	○	○
Debugging	O'Callahan *et al.* [44]	●	●	○	●
Performance Analysis and Optimization	Chen *et al.* [45]	●	○	○	●
	Chen *et al.* [46]	●	○	○	●
	Mushtaq *et al.* [27]	●	○	○	○
	Tuduce *et al.* [47]	●	○	○	●
	Segulja & Abdelrahman [48]	●	●	○	●
	Wang *et al.* [49]	●	○	○	●
	Bird [10]	●	○	○	○
	Michalska *et al.* [11]	●	○	○	○
	Flater & Guthrie [12]	●	○	○	○
	Röhl *et al.* [50]	●	○	○	●
	Hoske [51]	●	●	○	●
	Born de Oliveira [52]	●	●	○	●
	Lim *et al.* [13]	●	○	○	○
	Michalska *et al.* [14]	●	○	○	○
	Akiyama & Hirofuchi [53]	●	○	○	●
	Chang *et al.* [15]	●	○	○	○
	Doyle *et al.* [16]	●	○	○	○
	Stevens *et al.* [54]	●	○	○	○
	Melhus & Jensen [55]	●	●	○	●
	Wicht *et al.* [56]	○	◑	◑	●
High Performance Computing	Zhou *et al.* [17]	●	○	○	○
	Bock & Challacombe [57]	●	○	○	●
	Mushtaq *et al.* [18]	●	○	○	○
	Hussein *et al.* [58]	○	◑	◑	●
	Merrifield *et al.* [59]	●	●	○	●
	Teabe *et al.* [60]	○	×	×	×
	Pruitt & Freudenthal [19]	●	◑	◑	●
	Lundberg [61]	○	×	×	×
	Molnár & Végh [20]	●	○	○	○
	Ozdal *et al.* [62]	●	○	○	○
	Peraza *et al.* [63]	●	○	○	○
	Torres & Liu [64]	○	×	×	●
	Al-hayanni *et al.* [65]	○	×	×	●
	Torres & Liu [66]	○	×	×	●
OS Support	Bergan *et al.* [67]	●	○	○	●
	Lu *et al.* [21]	●	○	○	○
	Stefan *et al.* [68]	●	○	○	●
	Lu *et al.* [22]	●	○	○	○
Power Analysis	Singh *et al.* [69]	○	×	×	●
	Davis *et al.* [25]	●	○	○	○
	Davis *et al.* [26]	●	○	○	○
	Goel [70]	●	●	○	●
	Singh [71]	●	●	○	●
	Hussein *et al.* [24]	●	○	○	○
	Da Costa *et al.* [72]	●	●	○	●
Random No. Generation	Suciu *et al.* [73]	●	○	○	●
	Marton *et al.* [74]	●	○	○	●

● Yes ○ No × Not Applicable based on column 3 ◑ Respondent's answer inconsistent with description provided in the paper

affected by minor variations in measurements, an adversary can deliberately skew measurements to have higher variations, to defeat security defense mechanisms. We show precisely that later on in §VIII.

As noted earlier, for an accurate profiling of a given application, its counter data *must not* be contaminated by events from other processes. Since security applications will have significant impact even due to minor variations in the measurement of events (as discussed in §I), we mainly focused our attention on security approaches. Our painstaking analysis of these works revealed several common mistakes. For the sake of brevity, Table IV highlights some observations for the security research area. We selected these security papers because they represent the state of art techniques and their methodologies are clearly stated.

In short, we found evidence of:

- *No per-process filtering:* As discussed in §IV, performance counters measure events across the same CPU core. Any implementation that does not apply per-process filtering will capture events from other processes (*e.g.,* [77]).
- *PMI-based filtering only:* Many approaches [5, 6, 79] filter performance counter data relevant to a process while

TABLE III: Analysis of security papers using HPCs

Application	Authors	Non-determinism acknowledged	Non-determinism challenges addressed	Measurement error addressed	Recommend using HPCs
Exploit	Xia *et al.* [80]	○	×	×	●
	Yuan *et al.* [81]	○	×	×	●
	Aweke *et al.* [82]	○	×	×	●
	Zhou *et al.* [77]	○	×	×	●
	Pfaff *et al.* [32]	○	×	×	●
	Torres & Liu [83]	○	×	×	●
	Wang & Backer [78]	○	×	×	●
	Das *et al.* [79] ∗	○	×	×	●
	Herath & Fogh [84]	○	×	×	●
Malware	Demme *et al.* [5] †	○	×	×	●
	Tang *et al.* [6] ∗	○	×	×	●
	Wang & Karri [4]	○	×	×	●
	Bahador *et al.* [85]	○	◑	◑	●
	Wang & Karri [86]	○	×	×	●
	Kazdagli *et al.* [87] †	○	◑	◑	●
	Wang *et al.* [88]	○	×	×	●
	Garcia-Serrano [89]	○	×	×	●
	Zhang *et al.* [90]	○	×	×	●
	Singh *et al.* [76] ∗	○	×	×	●
	Jyothi *et al.* [91]	○	×	×	●
	Patel *et al.* [92]	○	×	×	●
	Peng *et al.* [93] ∗	○	×	×	●
Side-channel Attack	Martin *et al.* [94]	○	×	×	●
	Uhsadel *et al.* [95]	○	×	×	●
	Bhattacharya & Mukhopadhyay [96]	○	×	×	●
	Chiappetta *et al.* [97]	○	×	×	●
	Maurice *et al.* [98]	○	×	×	●
	Hunger *et al.* [99]	○	×	×	●
	Gruss *et al.* [100]	○	×	×	●
	Payer [101]	○	×	×	●
	Zhang *et al.* [102]	○	×	×	●
	Nomani & Szefer [30]	●	●	○	●
	Gulmezoglu *et al.* [103]	○	◑	◑	●
	Irazoqui [29]	●	●	○	●
	Allaf *et al.* [104]	○	×	×	●
Firmware Verification	Wang *et al.* [105]	○	×	×	●
	Wang *et al.* [106]	○	×	×	●
Integrity Checking	Malone *et al.* [28]	●	●	○	●
	Bruska *et al.* [107]	○	×	×	●
Virtual Machine Introspection	Vogl & Eckert [108]	○	×	×	●
Vulnerability Analysis	Copos & Murthy [31]	●	●	◑	●

●Yes ○ No × Not Applicable based on column 3 ◑ Respondent's answer inconsistent with description provided in the paper ∗ Windows † Android *Others: Linux*

TABLE IV: Subset of Papers in the Security Domain

Authors	Per-process filtering	PMI-based filtering
Malone *et al.* [28]	●	○
Wang & Karri [4]	●	○
Tang *et al.* [6]	●	●
Demme *et al.* [5]	●	◑
Singh *et al.* [76]	●	○
Pfaff *et al.* [32]	●	○
Zhou *et al.* [77]	○	○
Wang & Backer [78]	●	○
Das *et al.* [79]	●	●

● Yes ○ No ◑ Respondent's answer inconsistent with description provided in the paper. Notes: [77] uses polling but with no per-process filtering of HPCs. [5, 6, 79] filter process at PMI but do not monitor context switches. [6, 79] explicitly mention the issue of data contamination due to context switches.

handling PMIs. However, they do not save and restore the counter data during context switches, thereby suffering from contamination of counter data from other processes (§IV). A common solution is to obtain performance counter data by applying process filtering only at PMI, but doing so still does *not* monitor context switching. Not preserving the performance counter information at context switches will lead to errors in the HPC measurements. In §VII, we evaluate the extent to which counter values using this approach differ from the more accurate measurement approach that takes context switches into account.

- *Lack of compensation for non-determinism and overcounting:* As explained in §V-A (also shown in Table III), the non-determinism and overcounting issues are significant oversights. We further highlight this problem in §VI by performing a deeper analysis of the non-determinism and overcounting issues that can undermine many security approaches that rely on HPCs.

It is prudent to note that we supplemented our understanding of these works with responses we received to questions listed in Appendix A. We sent our questionnaire to the lead authors of papers listed in Table II and III. Even after repeated attempts, the response rate was only 28%. Given the recent adoption of HPCs in security applications, it is not surprising that the majority of the responses came from the security domain (>21%). One factor for the lower number of responses from other areas may be because those works are older.

VI. Non-determinism and HPCs — A Re-examination 10 years later

In the newer generations of processors, hardware performance counters are expected to have matured and provide error-free measurements. Given our observations in §V, we decided to revisit the HPC accuracy experiments originally performed by Weaver & McKee [33] to investigate what of the authors' findings still hold in modern processors, and whether new issues have arisen.

A. Experimental Setup

Our experimental platform was based on Linux (Ubuntu-16.04, x64) running on Intel processors. The experiments were conducted across different processor architectures, including Sandy Bridge, Haswell, and Skylake, all configured to use only a single core. We use only one core to ensure that the performance counter information obtained is from the same core that we configured, and to avoid any noise (events) from other cores. We used the Perf tool v4.4.90 to obtain performance counter measurements. Perf is incorporated in the Linux kernel and has an accompanying user-space utility.

To validate the measurements, we also implemented two additional tools for comparison. The first is based on Intel's Pin [110], and the second is a custom instruction debugger denoted `INS-Dbg`. Pin is a popular dynamic binary instrumentation tool commonly used for profiling the dynamic behavior of programs. Our Pin tool uses instruction-level instrumentation to step at each instruction and extract the necessary information about the program execution. In our experiments, we intercept each instruction to measure the number of instructions, branches, and memory operations (store and load instructions). `INS-Dbg` traps on each instruction executed by the program. The trap flag of the processor is set to single-step on each instruction.

B. Benchmarks

Our set of benchmark programs consists of the original 21 handcrafted assembly programs provided by Weaver & McKee [33]. A brief description of each program is given below:

- *x86_64*: executes 1 $million$ instructions.
- *brs_predictable*: executes 1 $million$ branches.
- *brs_random*: executes 2 $million$ branches.
- *rep_stosb*: uses the string instruction with the `rep` prefix to repetitively execute 1 $million$ store operations.
- *floating_point*: reports the instruction retired count when there is an execution of a floating point operation.
- *ins_type*: A set of benchmarks that execute specific types of instructions, as indicated by the benchmark's name, *e.g.,* arithmetic, logic, string, SSE, system call.

C. Findings

Our first objective was to investigate the non-determinism and overcounting issues of performance counters across different platforms. Our experiments consisted of measurements of the most commonly used events, *i.e.,* instructions and branches retired. Since our benchmark programs consist of a small amount of handcrafted assembly code, we were able to easily inspect them and calculate the expected measurement value in each case. To minimize the reported error, we executed these programs 100 times and computed the mean value of the events for each tool. Overall, we found that:

- In all three processor architectures, the reported number of events is the same, for both instruction and branch retired events, across all the tools.
- Each benchmark program has a minimum of one overcount for both instruction and branch retired events using performance counters, while both Pin and `INS-Dbg` yield the exact number of instructions and branches retired.
- Unlike the findings by Weaver & McKee [33], we do not see an overcount of instructions retired due to floating point initialization. Instead, there is an overcount in instruction and branch retired events, which is seen regardless of the presence of floating point operations. Therefore, this overcount cannot be accounted for by floating point initialization, so that issue may have been resolved in modern processors.
- The `rep stosb` instruction shows a large undercount of instructions retired. Although the store operation `stosb` was executed one million times, the performance counter shows a count of only 253 instructions. More importantly, despite the absence of branch instructions in the code, the HPC tool reports 246 branch instructions. While Weaver & McKee [33] noted the inconsistency of performance counters in handling the `rep` instruction prefix, they did not investigate the root cause of the problem. We return to this issue later in §VI-D.
- The *ins_string* program, which solely executes string instructions, exhibits an undercount of instructions and an overcount of branch retired events, similar to *rep_stosb*. Similarly, *ins_branch* and *ins_systemcall* report a significant overcount of branch instructions retired.

Our findings related to instructions retired undercount and branches retired overcount are a cause for concern, as other events (besides instruction and branch retired events) may exhibit similar inconsistencies. To better understand why these discrepancies occur, we further investigated the cases involving string instructions to uncover their root cause.

D. Analysis of String Instruction Benchmarks

To further investigate the behavior of HPCs with string-related instructions, we wrote 20 variants of the original *rep_stosb* program. These benchmarks comprise all 10 string operations, which include one-byte instructions (`lodsb`, `stosb`, `movsb`, `scasb`, `cmpsb`) and two-byte instructions (`lodsw`, `stosw`, `movsw`, `scasw`, `cmpsw`). These instructions perform load, store, copy, scan, and comparison operations. Each of these variants executes one string operation 1 million times. We also investigated the implementation of equivalent repeated string operations using the `loop` instruction, as an alternative to the `rep` prefix. Table V presents the results of this experiment for instruction count and branch retired events. We can make the following observations:

1) There is an undercount of instructions and an overcount of branch retired events when the `rep` prefix is used.

TABLE V: An analysis of `rep` and `loop` instructions for instruction and branch retired

	Expected values		HPC		Pin		`INS-Dbg`	
Benchmark	Ins	Br	Ins	Br	Ins	Br	Ins	Br
rep_lodsb	1000007	0	254	247	1000007	0	1000007	0
rep_stosb	1000007	0	253	246	1000007	0	1000007	0
rep_scasb	1000007	0	254	247	1000007	0	1000007	0
rep_movsb	1000008	0	499	491	1000008	0	1000008	0
rep_cmpsb	1000008	0	498	490	1000008	0	1000008	0
rep_lodsw	1000007	0	497	490	1000007	0	1000007	0
rep_stosw	1000007	0	497	490	1000007	0	1000007	0
rep_scasw	1000007	0	497	490	1000007	0	1000007	0
rep_movsw	1000008	0	987	979	1000008	0	1000008	0
rep_cmpsw	1000008	0	986	978	1000008	0	1000008	0
loop_lodsb	2000007	1000000	2000253	1000247	2000007	1000000	2000007	1000000
loop_stosb	2000007	1000000	2000252	1000246	2000007	1000000	2000007	1000000
loop_scasb	2000007	1000000	2000253	1000247	2000007	1000000	2000007	1000000
loop_movsb	2000008	1000000	2000501	1000494	2000008	1000000	2000008	1000000
loop_cmpsb	2000008	1000000	2000500	1000493	2000008	1000000	2000008	1000000
loop_lodsw	2000007	1000000	2000496	2000490	2000007	1000000	2000007	1000000
loop_stosw	2000007	1000000	2000497	1000491	2000007	1000000	2000007	1000000
loop_scasw	2000007	1000000	2000496	1000490	2000007	1000000	2000007	1000000
loop_movsw	2000008	1000000	2000988	1000981	2000008	1000000	2000008	1000000
loop_cmpsw	2000008	1000000	2000985	1000978	2000008	1000000	2000008	1000000

2) Using the `rep` prefix, string instructions that involve *two* memory accesses (*e.g.,* `movsb`) show nearly twice the number of instructions and branches than string instructions with a *single* memory access (*e.g.,* `lodsb`). Similarly, *word* (2-byte) instructions (*e.g.,* `movsw`) have roughly twice the number of events than their counterpart *byte* instructions (*e.g.,* `movsb`).
3) When the `loop` instruction is used, HPCs, Pin, and `INS-Dbg` have small measurement variations, which seem to follow a certain pattern. For example, the *loop_lodsb* benchmark shows an overcount of 246 instructions and 247 branches, compared to the expected values. Those inconsistencies follow the same pattern that was observed for the `rep` cases. Also, the overcounts double in the case of string instructions with two memory accesses. Upon further investigation, we noticed that the amount of overcount in instruction and branch events is equal to the number of page faults, which explains the correlation with the number of memory accesses. We present our detailed analysis to verify the effect of page faults on event count in Appendix B.
4) To dig deeper, we also investigated the memory load and store instructions for `rep` and `loop` cases. The measurement for the store instruction event is unaffected by page faults in both the `rep` and `loop` cases, but the load instruction event is directly affected by page faults (discussed later in Appendix B).

E. Summary and Impact of Findings

Overall, our analysis sheds light on challenges and pitfalls stemming from the use of HPCs that were not reported before. Our findings show that almost 10 years after the initial study by Weaver & McKee [33], numerous issues related to non-determinism with HPC measurements still persist in modern processors. We find that the most commonly used events (including instruction, branch, and memory load and store events) are affected by non-determinism and overcounting; the root cause of the measurement errors in instruction, branch and load events can be minimized; but the `rep` string instruction shows how the performance counter measurements significantly differ from other non-HPC approaches. We note that our study is not exhaustive, that there may be more cases of inconsistency not covered here, and that other hardware events may be affected, further exacerbating the measurement errors.

Taken as a whole, our investigation indicates that the issues of non-determinism and overcounting could severely impact the correctness of any techniques that solely rely on performance counter measurements. As a matter of fact, for critical use-cases such as security-related applications, these issues can have catastrophic consequences. For instance, most of the HPC-based security defenses we studied [4–6, 32, 77–79, 84, 88, 93, 101, 106], at their core, depend on threshold values of a particular event. An adversary can induce page faults to significantly affect the accuracy of the measured events, as we discuss further in Appendix B. Using such techniques, an adversary can manipulate the count of events to undermine a HPC-based defense. Under such an adversarial model, most of the security defenses listed in Table III can be undermined. We present a case study on the actual implementation of one such evasive technique in §VIII. Before doing so, we first provide guidance to researchers on how to properly use HPCs, with a key focus on illustrating how the recording methodology used in a set of representative works can drastically impact the ability to build robust models.

VII. Case Study I: On Weak Security Foundations

Two of the most prominent areas of security research that have used performance counters include exploit protection [32, 77–79, 84, 101] and malware detection [4–6, 88, 93, 106] (see Table III for others). Accordingly, for our case study, we selected representative approaches [5, 6, 78] from both areas.

In §V we highlighted three common mistakes that are encountered in security applications involving HPCs. The first two are related to the incorrect retrieval of HPC information for a given process. These mistakes arise due to improper handling of performance counter information during context switches. While a PMI-based filtering approach is better than

no per-process filtering at all, it still does not preserve the event counts during context switches.

For our case study, we implemented a custom module by augmenting the PMI approach with context switch monitoring. We refer to our implementation as *CS-PMI*. An overview of the CS-PMI implementation is given in Alg. 1 in Appendix C. In addition to sampling HPCs at performance monitoring interrupts, our approach monitors context switches in order to save and restore the counter values of the process of interest.

A. Using HPCs for ROP detection

For applications such as Return-oriented programming (ROP) attack detection, a PMI-only approach would not have significant impact on accuracy due to the relatively short instruction sequences that make up a ROP payload. For instance, the ROP chain used by the Metasploit exploit for *CVE-2012-4792* consists of only 27 instructions. The odds of encountering context switches during such a small code segment is very low. Hence, it is feasible that by setting a low PMI one could attempt to detect ROP execution, as was done by Wang & Backer [78]. To substantiate this conjecture, we performed a case study on a ROP defense model using the PMI approach.

Many ROP defenses [32, 77–79] use heuristics which leverage branch misprediction events that occur at return instructions. A ROP chain consists of several gadgets (i.e., instruction sequences) that end in return instructions. These return instructions do not have their corresponding call instructions, and therefore the CPU encounters branch mispredictions during their execution.

We evaluated the effectiveness of the CS-PMI and PMI approaches for detecting a ROP chain using the recently proposed model of Wang & Backer [78]. We performed the experiment on a bare-metal environment running Windows 7 (32-bit) on top of a Sandy Bridge processor (Intel i7-2600). To replicate a realistic real-world environment, we ensured that CPU utilization is above 60% [3]. We used the Metasploit exploits for CVE-2012-4792, CVE-2013-1347 and CVE-2012-1535, which are commonly used to test exploit protections [6, 32, 77–79]. They represent typical examples of ROP attacks, which in most cases have a small footprint.

Similar to Tang *et al.* [6], we instrument the start and end of the ROP chain using *int3* instructions (0xCC) in order to specifically model the ROP stage of an exploit. We configured the performance counter to generate a PMI for every 6 mispredicted return instructions as done by Wang & Backer [78]. At each PMI, we monitor two additional events — instruction retired and return instructions, using programmable counters. A typical ROP chain in Metasploit consists of 12–13 return instructions, generating the same number of return mispredictions. Therefore, a ROP execution will likely have 2 PMIs (each of 6 return mispredictions corresponding to 6 gadgets), which will reflect the signature of a ROP attack.

To detect a ROP attack, we set a threshold using the instruction retired event Wang & Backer [78]. The number of instructions retired I = *number of gadgets * average number of instructions in a gadget*. For our samples, the average number of instructions in a gadget is 3. Therefore, for a PMI, if *number of instructions retired* $<= 6*3$, it is considered as a ROP attack. For true positives, we measured how many of the PMIs between the start and end of the ROP chain have the signature of a ROP attack. For false positives, we measured the number of PMIs that do not occur as a result of the execution of the ROP chain, but still match the signature of a ROP attack.

Results: We ran each exploit 20 times and averaged the results. The CS-PMI approach correctly raises an alert 79 times, whereas the PMI approach does so 77 times. Hence both approaches have ample opportunities to detect the prescribed signature for a ROP attack. On the other hand, the numbers of false positives using the two approaches are 2019 and 2078, respectively. We note that although the heuristics that are used in these works are too poor to be used in practice, for the study in this paper, it appears that a PMI-only approach is not significantly worse than its CS-PMI counterpart. This is primarily due to the fact that there are not many opportunities for a context switch to occur during the small number of instructions that are executed in the ROP chain.

B. Using HPCs for Malware Detection

In contrast to ROP prevention, the PMI approach directly impacts the accuracy of malware classification techniques due to the longer execution trace induced by malware. Yet, several works [5, 6] have employed PMI-based filtering to model malicious behavior. To highlight the pitfalls of using the PMI approach in this setting, we performed a case study on malware classification at a fine-grained level. Similarly to previous works [5, 6], we use a machine learning (ML) approach for the classification of malware and benign programs.

Experimental setup: We used a machine with a Sandy Bridge processor (Intel i7-2600). Performance counter traces were collected on VMware virtual machines (VMs) running Windows 7 OS (32-bit), with each VM pinned to a separate physical core. Performance counters were configured to monitor events of the guest VM only (hypervisor events are excluded). Empirically, we verified that there is no contamination of events between the processor cores.

Dataset: We used 313 malware samples obtained from Vxheaven [111]. These malware samples were first seen during 2006–2011, as reported by *VirusTotal*. We labelled the samples into 137 malware families using the state of the art malware labeling tool, AVClass [112]. Our benign dataset contains real world applications, comprising Internet Explorer, Firefox, VLC player, WMplayer and Adobe Reader. The Alexa top-20 websites were loaded on Internet Explorer and Firefox, 10 media files were played on VLC and WMplayer, and 10 PDF files were opened using Adobe Reader.

Data collection: To compare the approaches in a real-world setting, we ensured that the CPU utilization was above 60% during the experiments. We profiled each malware sample in a fresh VM to avoid any affect of other malware executions. Each sample was allowed to run for one minute, during which the HPC traces were collected.

In this case study, we choose the heuristics proposed by Tang *et al.* [6] for malware classification, as they are representative

[3] To ensure that CPU load is above 60% during the experiments, we executed a few Chrome processes, running 4K and live videos on YouTube.

TABLE VI: Performance evaluation of K-way classification: Decision Tree (J48)

	TPR		FPR		Precision		Recall		F-Measure		ROC Area	
Class	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ
poshkill	0.363	-0.015	0.050	-0.002	0.342	0.002	0.363	-0.015	0.351	-0.006	0.763	0.001
bonk	0.345	0.001	0.045	0.001	0.353	-0.001	0.345	0.001	0.349	0.000	0.772	-0.007
alcaul	0.358	-0.016	0.044	0.004	0.369	-0.030	0.358	-0.016	0.363	-0.022	0.765	-0.002
thorin	0.976	0.022	0.002	-0.002	0.970	0.024	0.976	0.022	0.973	0.023	0.988	0.011
toffus	0.720	0.213	0.023	-0.018	0.693	0.236	0.720	0.213	0.706	0.225	0.878	0.098
werly	0.955	0.009	0.003	-0.001	0.959	0.010	0.955	0.009	0.957	0.010	0.980	0.004
afgan	0.925	0.046	0.005	-0.003	0.929	0.044	0.925	0.046	0.927	0.045	0.964	0.023
smee	0.892	0.043	0.008	-0.004	0.884	0.056	0.892	0.043	0.888	0.050	0.948	0.023
bonding	0.727	0.221	0.019	-0.014	0.731	0.200	0.727	0.221	0.729	0.210	0.875	0.099
delf	0.624	0.325	0.028	-0.023	0.619	0.314	0.624	0.325	0.621	0.320	0.836	0.142
cisum	0.889	0.041	0.008	-0.002	0.894	0.029	0.889	0.041	0.891	0.035	0.948	0.019
tiraz	0.931	0.021	0.005	0.000	0.934	0.009	0.931	0.021	0.933	0.015	0.968	0.010
bube	0.819	0.079	0.012	-0.004	0.836	0.065	0.819	0.079	0.827	0.072	0.914	0.041
leniog	0.873	0.060	0.008	-0.004	0.890	0.057	0.873	0.060	0.881	0.059	0.943	0.030
IE	0.794	0.050	0.014	-0.005	0.806	0.068	0.794	0.050	0.800	0.059	0.906	0.023
Weighted Avg.	0.746	0.078	0.018	0.006	0.747	0.076	0.746	0.078	0.746	0.077	0.897	0.036

Δ = Difference between the approaches (*i.e.,* CS-PMI - PMI)

of the state of the art. Accordingly, the PMI is set at $512,000$ instructions retired and 4 additional events are monitored using programmable counters — store micro-operations, indirect call instructions, mispredicted return instructions and return instructions.[4] Thus, each PMI consists of a tuple of five events.

Feature construction: To build a feature vector, we use the temporal model proposed by Tang *et al.* [6]. We choose temporal model over non-temporal because Tang *et al.* [6] show that malicious behavior cannot be sufficiently represented by performance counter events of one PMI. A feature vector consists of N (= 4) consecutive tuples, where each tuple contains 5 events collected at a PMI, totaling 20 features.

We selected the unique feature vectors in each family. Some malware families may generate substantially lower numbers of feature vectors than others. To avoid biasing [113] the classification, we selected the top 14 malware families that generated the highest number of feature vectors. To further negate bias by an individual family, we used an equal number of feature vectors, randomly selected from each of the 14 families. Similarly, in the case of benign applications, an equal number of feature vectors was chosen randomly from each application. The total feature vectors obtained from benign applications and malware samples were balanced.

1) Classification Results: We use the Weka toolkit [114] for ML classifier training and testing purposes. Similar to other works [5, 6], we performed our evaluation using three standard ML classifiers — Decision Tree, K-Nearest Neighbors (KNN), and Support Vector Machine. Based on the standard practices, we selected the *J48, IBk, and SMO* algorithms corresponding to the aforementioned classifiers [115] (with the default configuration parameters of Weka). It is possible that by fine-tuning the advanced ML classifiers the results can be improved, but nonetheless, we believe that the same difference will still persist between the CS-PMI and PMI approaches. According to widely held practices [116], training and testing were performed using a 10-fold cross-validation model.

a) Binary classification: First, we perform a coarse-grained binary classification, where feature vectors are divided into two classes, malware and benign. An equal number of feature vectors are selected from the benign and malware samples for both the CS-PMI and PMI approaches. Our results show that the average classification accuracy (F-Measure) using the three algorithms is 89% with the CS-PMI approach versus 87% with the PMI approach. However, the difference is far greater under the more fined-grained K-way classification.

b) K-way classification: A K-way classification across 15 classes, including 14 malware families and one benign application, Internet Explorer (IE), was performed. The results for decision tree and KNN classifiers are shown in Table VI and VIII (Appendix D). The results for the KNN classifier is also given in Appendix D (*e.g.,* Fig. 5), but we omit the figures for the support vector machine classifier due to space limitations. The difference between the two approaches is also evident from the confusion matrices (*i.e.,* Fig. 4 and 5 in Appendix D). The overall difference using a decision tree classifier is roughly 8%, but for the individual classes, there is a substantial improvement: *toffus (23%), bonding (21%), delf (32%), bube (7%), leniog (6%) and IE (6%)*. Similarly, with a KNN classifier, we observe a notable improvement: *toffus (10%), bonding (36%), delf (24%), cisum (7%) and leniog (9%)*. Support vector machine also shows similar improvement for the individual classes: *smee (17%), bonding (26%), delf (18%), leniog (16%) and IE (6%)*. Overall, 30% of the families have roughly 17% classification improvement for all 3 classifiers.

To see why there is such a large difference between the two approaches, it is sufficient to note that for *Poshkill.1445*, for example, we observed $11,735$ context switches during the one minute period it ran. For the PMI approach, each time this sample is restored for execution at the context switch, there are contaminated events from previous process. Also, at the time of context switch from the malware to other process, the events related to this sample are lost because they are not saved. Therefore, on $2 * 11,735$ occasions, the PMI data may be either contaminated or incomplete. If we assume that the context switches occurred at a regular interval, for $150,476$ feature vectors obtained from this sample, approximately 15% of them are either contaminated or have incomplete events. Table VI and VIII also show that 12 out of 15 classes have

[4]The corresponding performance events in the Intel manual are mem_uops_retired.all_stores, br_inst_exec.taken_indirct_near_call, br_misp_exec.return_near, br_inst_exec.all_indirect_near_return.

better accuracy using the CS-PMI approach for decision tree and KNN classifiers, while 3 of the classes (i.e., *poshkill*, *bonk* and *alcaul*) have marginally lower F-measures as compared to the PMI approach. As depicted by the confusion matrices (Fig. 4 and 5), these 3 classes also exhibit a higher confusion rate than others. This is because they use a similar technique to attach their code to files on the host system or network[5] and may not contain enough distinctive behaviors [113].

c) Summary of Findings: Our experiments show that differences in the way the data is recorded and the approaches taken to do so, not only affect the accuracy of the resulting techniques, but also impact the reproducibility of the research findings. For that reason, we do not recommend the use of PMI-only approaches, which unfortunately, is a common practice when using HPCs for security applications (see Table IV).

The observant reader will note that irrespective of which classifier was used, the confusion between the benign case (IE) and the malicious samples is beyond what one would expect in any practical deployment of such detectors. This, to us, indicates that the use of HPCs for malware detection is inadequate, since the features commonly built from such measurements fail to correctly distinguish between benign and malicious activities in real-world settings. The situation may improve once we have better support for low-skid interrupts and sampling [117]. In particular, a known issue with measurements involving interrupts is "skid," wherein it takes some amount of time to stop the processor to pinpoint exactly which instruction was active at time of the interrupt, resulting in a discrepancy between the instruction indicated versus the one that actually caused the interrupt. This is particularly common in modern, out-of-order, CPU designs. Until then, we suggest that HPCs are too fragile to be used as the sole source of features for the studied types of security applications.

VIII. Case Study II:There Be Dragons

Earlier in §V, we highlighted that the effects of non-determinism have been overlooked in most security papers. To shed light on the severity of this oversight, we present two analysis efforts. The first directly shows how non-determinism impacts the accuracy of a ROP defense model, while the second presents a generic approach to evade a ROP defense by inducing noise during the measurement process.

A. How Non-determinism Affects Accuracy

To concretely demonstrate the impact of non-determinism, we revisit the ROP detection technique discussed in §VII-A. Recall that factors that influence the accuracy of such defenses include skid during PMI delivery, overcounting due to hardware interrupts, where and when the PMI occurs during the ROP chain, and the strength of the heuristic itself. The sources of non-determinism are skid and overcounting of instructions due to PMI events. The frequency of the PMI will impact the level of overcounting, leading to our conjecture that as the frequency of PMIs increases, so does the overcounting of events.

Showing this is challenging since non-determinism is a deep-rooted issue which cannot be easily compartmentalized because there is no noise-free platform that can be used to give an exact measurement. However, based on our understanding of return-oriented programming, we can control some factors to show the impact of non-determinism on the detection heuristic. In what follows, we assume that:

- Similar to Wang & Backer [78], a window of 6 gadgets (correspondingly, 6 return misses) is used as the trigger for detecting a ROP chain. Thus, for a chain of 12 gadgets, there are two windows of opportunity for detection.
- For a given ROP chain, we can compute the threshold value for the number of instructions retired for an arbitrary set of gadgets. For example, in the case of the Metasploit exploit (CVE-2012-4792), for 6 gadgets, the number of instructions retired is $<= 15$, whereas, for 7 gadgets, the number of instructions retired is $<= 17$.

We used the same setup and exploit samples as in §VII-A. We obtained HPC measurements using PMI by varying the number of return misses, *i.e.,* ret_miss = α, for α = 6, 3, 2, 1. We use the CS-PMI approach for data collection. The exploits were executed 20 times for each PMI setting. We evaluated the true positive rate (TPR) and false positive rate (FPR) based on the criteria above. Each exploit run included the execution of ROP chain and non-ROP (*i.e.,* benign) code. For example, when the exploit is run in Internet Explorer, the run consists of both benign code and the ROP chain execution. A true positive is specific to the ROP chain execution, whereas a false positive is specific to the non-ROP part. To demarcate the ROP chain, we instrument the start and the end of ROP-chain using "int3" instruction (as also described in §VII-A). The TPR is evaluated as the number of true positives over all the total windows of opportunity to detect the ROP chain.

In our evaluation, we ensured that a window of 6 gadgets was considered, per the original heuristic. For example, in the case of PMI set to ret_miss = 1, we take 6 consecutive PMIs as one window of opportunity to detect the ROP exploit. Similarly, for a PMI set at ret_miss = 3, we take 2 consecutive PMIs as one window to detect the ROP chain. We do not consider ret_miss = 4 or 5, since the original heuristic is based on a window of ret_miss = 6, and 4, 5 are not divisible of 6.

Results: Empirical results show that as the frequency of PMI increases (smaller values of α), the true positive rate decreases: TPRs at α = 6, 3, 2, 1 are 19.17%, 11.67%, 9.17%, 0%, respectively. Furthermore, the false positive rate increases with the increased frequency of PMI; FPRs at α = 6, 3, 2, 1 are 0.007%, 0.007%, 0.011%, 0.009%, respectively. These results support the earlier conjecture that non-determinism will significantly impact the performance of a model.

Summary: Taken as a whole, these results show that the non-determinism of HPCs is indeed a serious factor, significant enough to undermine the soundness of techniques that rely on them as a foundation for security defenses. Adjusting the PMI has a direct impact on the level of noise, leading to a clear degradation in accuracy. The non-determinism with HPCs in these experiments arises because PMI is a hardware interrupt, and each PMI leads to overcounting of instructions. Additionally, the skid during PMI delivery (as mentioned in §VII) worsens the overcounting issue. To illustrate that further, we plot in Fig. 6 (in Appendix E) the observed skid when

[5]https://www.pandasecurity.com/homeusers/security-info/44002/information/PoshKill.B

PMI is set at different number of ret_miss events. To observe the skid for a PMI set at ret_miss = α, we collect the actual number of return misses recorded by HPC at each PMI during the execution of the exploits. The skid is the difference between the expected measurement and the measurement reported by the HPC. For example, if one sets the PMI to be delivered after six return misses, and the value reported by the measurement unit equals to eight, than the observed skid is two. The result shows that skid is inconsistent at every sampling rate.

B. Evading a ROP Defense using Non-determinism

Lastly, we show how non-determinism can be leveraged by an adversary to evade a ROP defense. To that end, we return to the heuristic proposed by Wang & Backer [78]. As described in §VIII-A, the ROP detection is based on the assumption that for a given number of return misses, α, the number of instructions retired is lower than some threshold, Δ.

Our attack simply triggers page faults to manipulate the number of instructions retired to defeat the ROP detection. As discussed in §VI, we observe that page faults directly impact the measurement of instruction retired event. In the attack, we assume 1) the availability of classes of gadgets — namely, arithmetic and load gadgets, and a gadget that triggers repeated demand paging page faults; 2) the availability of a function pointer to `malloc()`. Given that, the attack is straightforward: we deliberately insert points of manipulation within an existing ROP chain. The points of manipulation produce an overcount of one instruction for every page fault triggered. The amount of page faults triggered is controlled by a counter register, and that counter can be set arbitrarily to accommodate any parameter values l, α and Δ assumed by the heuristic. Here, l specifies the number of gadgets in a ROP chain.

Implementation: First, an adversary allocates a memory buffer of size m using a return-to-libc technique to call `malloc()`. This buffer is allocated but not mapped in memory, and is used to trigger demand page faults. Next, the existing ROP chain is modified to include manipulator blocks that produce an arbitrary overcount for every α gadgets, thus foiling the detection heuristic. An illustration is shown in Fig. 3.

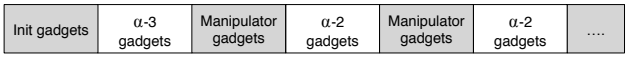


Fig. 3: Modified ROP chain with manipulator gadgets.

We distinguish between two types of so-called manipulator blocks: a manipulator *initialization* block that is executed before any other gadget in the ROP chain, and regular *manipulator* blocks that are inserted every α-2 gadgets.

1) The *initialization* block consists of three gadgets:
   - The first gadget (*e.g.,* `pop ecx; ret`) sets the value of ecx to m.
   - The second gadget (*e.g.,* `mov edi, eax; ret`) copies the value returned by `malloc()` stored in the register `eax` to the destination register `edi`.
   - The third gadget (*e.g.,* `rep stosd; ret`) repeatedly accesses the memory pointed by `edi` and triggers page faults. Memory is accessed m times, thereby triggering instruction overcount above the detection threshold Δ.
2) Regular manipulator blocks consist of two gadgets: one gadget (*e.g.,* `pop ecx; ret`) that sets a value of the counter register `ecx` to m, and other gadget (*e.g.,* `rep stosd; ret`) that repeatedly accesses the memory pointed by `edi` to trigger page faults.

Our approach increases the overall length of the gadget chain since we introduce two extra gadgets for every $\alpha - 2$ gadgets of the original chain (see Fig. 3). Given any combination of the parameters α and Δ used by the heuristic, an adversary can choose a value m such that the instruction retired count is always above Δ. The modified chain will be of length $l' = l + 3 + 2 * \lceil (l + 3 - \alpha)/(\alpha - 2) \rceil$. Thus, there will be $N = l'/\alpha$ opportunities for detecting the modified ROP chain. Consequently, to evade detection, each manipulator block must produce an overcount of $x > \Delta - 2 * \alpha$ instructions.

Results: The ROP detection mechanism [78] used a detection window based on the number of return misses $\alpha = 6$. To demonstrate the feasibility of hiding the ROP we evaluate the length of the expanded chain and instructions retired for different sizes of $\alpha <= 6$. Our results show that in the worst case, the manipulators need to generate an overcount that matches the average number of instructions. For example, for $\alpha = 6$, the average number of instructions is 11088, whereas for $\alpha = 3$, it is 5614. In the average case (as presented in [78]), an adversary needs to generate only $x > \Delta - 2 * \alpha$ instructions to defeat the detection mechanism. Hence, for $\alpha = 3, 4, 5, 6$ an overcount of $x = 20$ instructions would be sufficient to defeat the detection mechanism.

Summary: Our evasive technique demonstrates that regardless of the parameters used by the detection mechanism, an adversary can disguise the execution of a ROP chain by hiding within deliberately introduced noise.

IX. Other Related Work

In the past, several works have highlighted challenges with using hardware performance counters. The most germane is the work of Weaver & McKee [33] that showed that the instruction retired event had a variation of 1% for the CPU SPEC benchmarks across different x86 architectures. They noted several sources of variations and showed that by adjusting the measurements one could reduce the observed variation to less than 0.002%. Later, Weaver & Dongarra [39] showed that hardware interrupts (*e.g.,* periodic timers) can lead to significant overcounting of retired instructions and recommended that the recorded measurements be compensated by subtracting the number of hardware interrupts. Although they briefly mentioned the influence of page faults, they did not perform a detailed study of the effects of page faults on different events.

Furthermore, Weaver *et al.* [34] found that many events on x86_64 architectures are non-deterministic, *i.e.,* their measurements vary between different runs even under strictly controlled settings. They showed that only a small number of events are deterministic and have no overcounting issues, *e.g.,* branches retired on Sandy Bridge and Westmere architectures. It was noted that although hardware interrupts are one of the main sources of variation, the impact may be difficult to measure in many systems. More recently, Weaver [118]

summarized the available features and overhead of the most common performance counter tools.

Other researchers [38, 43, 119] have also noted the inaccuracy in HPC measurements, specifically for loads, stores, floating-point operations, and branch mispredictions. However, these works focused on micro-architectural events and attributed the inaccuracy to the interface of their performance counter monitoring tool (*i.e.,* PAPI), and the lack of processor architecture details. Zaparanuks *et al.* [38] studied the difference in counts reported for specific events across common HPC tools, including perfctr, perfmon2, and PAPI. In that work, it was observed that the measurement setup could lead to an inaccuracy of several thousand instructions. Moseley *et al.* [43] reported that HPCs have several practical issues that hinder their usage in real-world settings, for example, lack of verification of the events, lack of portability of events across architectures, sampling skid during the interrupt delivery, and the high overhead of reading the counters. Recently, O'Callahan *et al.* [44] tried to get around the skid issue by interrupting the processor some number of events earlier than the actual desired event. Unfortunately, while most of these works do touch on the non-determinism and inaccuracy of events, none empirically evaluated the drawback of incorrect data collection, such as using PMI-based filtering.

Concurrent to our work, Zhou *et al.* [120] showed that under realistic settings, HPC information cannot distinguish between benign software and malware. Interestingly, even after cross-validating their models multiple times by taking repeated measurements, the best F-measure obtained was 80.22% using a Decision Tree model, which is similar to the results we presented in Table VI. Furthermore, they also showed that benign application infused with ransomware cannot be detected by contemporary HPC-based malware detection. Taken together, these works describe a cautionary tale for using HPCs for purposes beyond their original intent.

X. Recommendations and Conclusion

Hardware performance counters have been available for more than a decade. Their inclusion has been a boon for many application areas, with the biggest benefit being the ability to identify and understand the behavior of code hotspots. But, understanding micro-architectural interactions — which is imperative for various types of security analysis — is tedious and the potential gains can quickly diminish. As noted by Moseley *et al.* [43], because features frequently have vague documentation, an expert level of understanding is required to collect and apply data from performance counters. Consequently, users are often measuring something different from what they expect. This, in our experience, is particularly true in that subsequent generations of processors may be deceptively diverse in what they measure (*e.g.,* [43, Section 3]). Hence, users are left to treat processors as black boxes.

While we disagree with the point of view of Moseley *et al.* [43] that, from an academic perspective, one motivation for the use of hardware performance counters is that it enables writing of more papers, we understand their frustration. Here, we take a different approach and suggest some tenable goals that could help minimize some of the headaches associated with using HPCs.

- First, (for both reviewers and authors alike) empirical results based on performance counters should not be compared with those from other profiling tools (*e.g.,* Pin). As discussed in §VI, the fact that the design philosophy varies widely between these tools, coupled with the issues of non-determinism and overcounting, inevitably leads to results that are not easily comparable.
- Modern processors are inherently complex, so it is important that authors verify HPC-based findings on different CPU architectures. While certain architectural events may remain consistent across architectures, many micro-architectural events vary widely. Moreover, some events may be present in one architecture, unavailable in another, or have an entirely different implementation.
- For profiling a specific application, it is imperative that per-process filtering of events is applied by saving and restoring the counter values at context switches. Failure to do so will impact accuracy, and can call into question the overall soundness of the approach.
- For critical applications such as security defenses, the issues related to non-determinism and overcounting cannot be overlooked. This is especially true when considering adversarial models in which the adversary can freely manipulate the counters to thwart defenses.
- Lastly, if security is to be taken more seriously as a science [121], it is important that authors take more precautions in documenting their work. It was rather difficult to get even the most basic information about HPC configuration and usage in many of the papers we surveyed. Respondents to our survey indicated that our questions were enlightening, and some agreed that the lack of detail affected the reproducibility of their work.

Overall, the fact that the number of events being added to processors is growing faster than the number of counters, is unsettling. It has come to the point where machine learning[6] is required to manage the abundance of information. Researchers should think carefully about the adoption of HPCs for security defenses, as this was not the designers' original intent for them.

XI. Acknowledgments

We thank Prof. Vince Weaver for his guidance on the proper usage of various benchmarks, including those in the original paper by Weaver & McKee [33]. We express our gratitude to the anonymous reviewers for their suggestions on how to improve the paper. We also thank Murray Anderegg for his assistance with deploying the infrastructure used in this study. This work was supported in part by the Department of Defense (DoD) under awards FA8750-16-C-0199 and FA8750-17-C-0016, the Office of Naval Research (ONR) under award N00014-15-1-2378, and the National Science Foundation (NSF) under awards CNS-1617902 and CNS-1749895. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the DoD, ONR, or NSF.

[6] See, for example, *Intel platform modeling tool with machine learning*, https://software.intel.com/en-us/articles/intel-platform-modeling-with-machine-learning

REFERENCES

1. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 : System Programming Guide* 2016.
2. Ammons, G., Ball, T. & Larus, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices* **32,** 85–96 (1997).
3. Zamani, R. & Afsahi, A. *A study of hardware performance monitoring counter selection in power modeling of computing systems* in *International Green Computing Conference* (2012), 1–10.
4. Wang, X. & Karri, R. *Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters* in *Design Automation Conference* (2013), 1–7.
5. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S. & Stolfo, S. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News* **41,** 559–570 (2013).
6. Tang, A., Sethumadhavan, S. & Stolfo, S. J. *Unsupervised anomaly-based malware detection using hardware features* in *Symposium on Recent Advances in Attacks and Defenses* (2014), 109–129.
7. Mucci, P. J., Browne, S., Deane, C. & Ho, G. *PAPI: A portable interface to hardware performance counters* in *Department of Defense HPCMP Users Group Conference* (1999), 7–10.
8. Weaver, V. M. *Linux perf_event features and overhead* in *International Workshop on Performance Analysis of Workload Optimized Systems* (2013).
9. Malladi, R. K. *Using Intel® VTune™ Performance Analyzer Events/Ratios & Optimizing Applications* 2009.
10. Bird, S. *Software knows best: A case for hardware transparency and measurability.* MA thesis (University of California, Berkeley, 2010).
11. Michalska, M., Boutellier, J. & Mattavelli, M. A methodology for profiling and partitioning stream programs on many-core architectures. *Procedia Computer Science* **51,** 2962–2966 (2015).
12. Flater, D. & Guthrie, W. F. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *Journal of Research of the National Institute of Standards and Technology* **118,** 260–279 (2013).
13. Lim, R., Malony, A., Norris, B. & Chaimov, N. *Identifying optimization opportunities within kernel execution in GPU codes* in *European Conference on Parallel Processing* (2015), 185–196.
14. Michalska, M., Zufferey, N. & Mattavelli, M. Tabu search for partitioning dynamic dataflow programs. *Procedia Computer Science* **80,** 1577–1588 (2016).
15. Chang, J., Nakshatrala, K. B., Knepley, M. G. & Johnsson, L. A performance spectrum for parallel computational frameworks that solve PDEs. *Concurrency and Computation: Practice and Experience* **30** (2018).
16. Doyle, N. C., Matthews, E., Holland, G., Fedorova, A. & Shannon, L. *Performance impacts and limitations of hardware memory access trace collection* in *Design, Automation & Test in Europe* (2017), 506–511.
17. Zhou, X., Lu, K., Wang, X. & Li, X. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing* **72,** 716–727 (2012).
18. Mushtaq, H., Al-Ars, Z. & Bertels, K. *DetLock: portable and efficient deterministic execution for shared memory multicore systems* in *High Performance Computing, Networking, Storage and Analysis* (2012), 721–730.
19. Pruitt, D. D. & Freudenthal, E. A. *Preliminary investigation of mobile system features potentially relevant to HPC* in *International Workshop on Energy Efficient Supercomputing* (2016), 54–60.
20. Molnár, P. & Végh, J. *Measuring performance of processor instructions and operating system services in soft processor based systems* in *International Carpathian Control Conference* (2017), 381–387.
21. Lu, K., Zhou, X., Bergan, T. & Wang, X. Efficient deterministic multithreading without global barriers. *ACM SIGPLAN Notices* **49,** 287–300 (2014).
22. Lu, K., Zhou, X., Wang, X.-P., Bergan, T. & Chen, C. An efficient and flexible deterministic framework for multithreaded programs. *Journal of Computer Science and Technology* **30,** 42–56 (2015).
23. Zankl, A., Seuschek, H., Irazoqui, G. & Gulmezoglu, B. in *Solutions for Cyber-Physical Systems Ubiquity* 325–357 (2018).
24. Hussein, A., Payer, M., Hosking, A. & Vick, C. A. *Impact of GC design on power and performance for Android* in *ACM International Systems and Storage Conference* (2015), 13:1–13:12.
25. Davis, J. D., Rivoire, S., Goldszmidt, M. & Ardestani, E. K. No hardware required: building and validating composable highly accurate OS-based power models. *Microsoft Technical Report* (2011).
26. Davis, J. D., Rivoire, S., Goldszmidt, M. & Ardestani, E. K. *Chaos: Composable highly accurate os-based power models* in *International Symposium on Workload Characterization* (2012), 153–163.
27. Mushtaq, H., Al-Ars, Z. & Bertels, K. *Efficient software-based fault tolerance approach on multicore platforms* in *Design, Automation & Test in Europe* (2013), 921–926.
28. Malone, C., Zahran, M. & Karri, R. *Are hardware performance counters a cost effective way for integrity checking of programs* in *ACM workshop on Scalable trusted computing* (2011), 71–76.
29. Irazoqui, G. *Cross-core Microarchitectural Side Channel Attacks and Countermeasures.* PhD thesis (Northeastern University, 2017).
30. Nomani, J. & Szefer, J. *Predicting program phases and defending against side-channel attacks using hardware performance counters* in *Hardware and Architectural Support for Security and Privacy* (2015), 9:1–9:4.
31. Copos, B. & Murthy, P. *Inputfinder: Reverse engineering closed binaries using hardware performance counters* in *Program Protection and Reverse Engineering Workshop* (2015).
32. Pfaff, D., Hack, S. & Hammer, C. *Learning how to prevent return-oriented programming efficiently* in *International Symposium on Engineering Secure Software and Systems* (2015), 68–85.
33. Weaver, V. M. & McKee, S. A. *Can hardware performance counters be trusted?* in *International Symposium on Workload Characterization* (2008), 141–150.
34. Weaver, V. M., Terpstra, D. & Moore, S. *Non-determinism and overcount on modern hardware performance counter implementations* in *IEEE International Symposium on Performance Analysis of Systems and Software* (2013), 215–224.
35. Li, X. & Crouse, M. *Transparent ROP Detection using CPU Performance Counters* in *THREADS* (2014).
36. Russinovich, M. E., Solomon, D. A. & Ionescu, A. *Windows internals* (2012).
37. Levon, J. & Elie, P. *Oprofile: A system profiler for linux* 2004.
38. Zaparanuks, D., Jovic, M. & Hauswirth, M. *Accuracy of performance counter measurements* in *IEEE International Symposium on Performance Analysis of Systems and Software* (2009), 23–32.
39. Weaver, V. & Dongarra, J. *Can hardware performance counters produce expected, deterministic results?* in *Workshop on Functionality of Hardware Performance Monitoring* (2010).
40. Rohou, E. *Tiptop: Hardware performance counters for the masses* in *International Conference on Parallel Processing Workshops* (2012), 404–413.
41. Nowak, A., Yasin, A., Mendelson, A. & Zwaenepoel, W. *Establishing a Base of Trust with Performance Counters for Enterprise Workloads* in *USENIX Annual Technical Conference* (2015), 541–548.
42. Lim, R. V., Carrillo-Cisneros, D., Alkowaileet, W & Scherson, I. *Computationally efficient multiplexing of events on hardware counters* in *Linux Symposium* (2014), 101–110.
43. Moseley, T., Vachharajani, N. & Jalby, W. *Hardware performance monitoring for the rest of us: a position and survey* in *IFIP International Conference on Network and Parallel Computing* (2011), 293–312.
44. O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A. & Partush, N. *Engineering record and replay for deployability* in *USENIX Annual Technical Conference* (2017), 377–389.
45. Chen, D., Vachharajani, N., Hundt, R., Liao, S.-w., Ramasamy, V., Yuan, P., Chen, W. & Zheng, W. *Taming hardware event samples for FDO compilation* in *International Symposium on Code Generation and Optimization* (2010), 42–52.
46. Chen, D., Vachharajani, N., Hundt, R., Li, X., Eranian, S., Chen, W. & Zheng, W. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers* **62,** 376–389 (2013).
47. Tuduce, I., Majo, Z., Gauch, A., Chen, B. & Gross, T. R. *Asymmetries in multi-core systems–or why we need better performance measurement units* in *Exascale Evaluation and Research Techniques Workshop* (2010).
48. Segulja, C. & Abdelrahman, T. S. *What is the cost of weak determinism?* in *International Conference on Parallel architectures and Compilation Techniques* (2014), 99–112.
49. Wang, W., Davidson, J. W. & Soffa, M. L. *Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines* in *IEEE International Symposium on High-Performance Computer Architecture* (2016), 419–431.
50. Röhl, T., Eitzinger, J., Hager, G. & Wellein, G. in *Tools for High Performance Computing* 17–28 (2016).

51. Hoske, D. *An experimental study of a nearly-linear time Laplacian solver.* MA thesis (Karlsruhe Institute of Technology, 2014).
52. Born de Oliveira, A. *Measuring and predicting computer software performance: Tools and approaches.* PhD thesis (University of Waterloo, 2015).
53. Akiyama, S. & Hirofuchi, T. *Quantitative evaluation of intel pebs overhead for online system-noise analysis* in *International Workshop on Runtime and Operating Systems for Supercomputers* (2017), 3:1–3:8.
54. Stevens, J., Tschirhart, P. & Jacob, B. *The Semantic Gap Between Software and the Memory System* in *International Symposium on Memory Systems* (2015), 43–46.
55. Melhus, L. K. & Jensen, R. E. *Measurement Bias from Address Aliasing* in *IEEE International Parallel & Distributed Processing Symposium* (2016), 1506–1515.
56. Wicht, B., Vitillo, R. A., Chen, D. & Levinthal, D. Hardware counted profile-guided optimization. *arXiv preprint arXiv:1411.6361* (2014).
57. Bock, N. & Challacombe, M. An optimized sparse approximate matrix multiply for matrices with decay. *SIAM Journal on Scientific Computing* **35,** C72–C98 (2013).
58. Hussein, A., Hosking, A. L., Payer, M. & Vick, C. A. Don't race the memory bus: taming the GC leadfoot. *ACM SIGPLAN Notices* **50,** 15–27 (2015).
59. Merrifield, T., Devietti, J. & Eriksson, J. *High-performance determinism with total store order consistency* in *European Conference on Computer Systems* (2015), 31:1–31:13.
60. Teabe, B., Tchana, A. & Hagimont, D. Enforcing CPU allocation in a heterogeneous IaaS. *Future Generation Computer Systems* **53,** 1–12 (2015).
61. Lundberg, M. *Implementation of Fast Real-Time Control of Unstable Modes in Fusion Plasma Devices* 2017.
62. Ozdal, M. M., Jaleel, A., Narvaez, P., Burns, S. M. & Srinivasa, G. Wavelet-Based Trace Alignment Algorithms for Heterogeneous Architectures. *IEEE Transactions on CAD of Integrated Circuits and Systems* **34,** 381–394 (2015).
63. Peraza, J., Tiwari, A., Ward, W. A., Campbell, R. & Carrington, L. *VecMeter: measuring vectorization on the Xeon Phi* in *International Conf. on Cluster Computing* (2015), 478–481.
64. Torres, G. & Liu, C. *The Impact on the Performance of Co-running Virtual Machines in a Virtualized Environment* in *International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing* (2016), 21–27.
65. Al-hayanni, M. A., Shafik, R., Rafiev, A., Xia, F. & Yakovlev, A. *Speedup and Parallelization Models for Energy-Efficient Many-Core Systems Using Performance Counters* in *International Conference on High Performance Computing & Simulation* (2017), 410–417.
66. Torres, G. & Liu, C. Adaptive virtual machine management in the cloud: A performance-counter-driven approach. *International Journal of Systems and Service-Oriented Engineering* **4,** 28–43 (2014).
67. Bergan, T., Hunt, N., Ceze, L. & Gribble, S. D. *Deterministic Process Groups in dOS* in *OSDI* (2010), 177–192.
68. Stefan, D., Buiras, P., Yang, E. Z., Levy, A., Terei, D., Russo, A. & Mazières, D. *Eliminating cache-based timing attacks with instruction-based scheduling* in *European Symposium on Research in Computer Security* (2013), 718–735.
69. Singh, K., Bhadauria, M. & McKee, S. A. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News* **37,** 46–55 (2009).
70. Goel, B. *Per-core power estimation and power aware scheduling strategies for CMPs.* MA thesis (Chalmers University of Technology, 2011).
71. Singh, K. *Prediction strategies for power-aware computing on multicore processors.* PhD thesis (Cornell University, 2009).
72. Da Costa, G., Pierson, J.-M. & Fontoura-Cupertino, L. Mastering system and power measures for servers in datacenter. *Sustainable Computing: Informatics and Systems* **15,** 28–38 (2017).
73. Suciu, A., Banescu, S. & Marton, K. Unpredictable random number generator based on hardware performance counters. **189,** 123–137 (2011).
74. Marton, K., Tóth, P. & Suciu, A. *Unpredictable random number generator based on the performance data helper interface* in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012), 335–340.
75. Treibig, J., Hager, G. & Wellein, G. *Likwid: A lightweight performance-oriented tool suite for x86 multicore environments* in *International Conference on Parallel Processing Workshops* (2010), 207–216.
76. Singh, B., Evtyushkin, D., Elwell, J., Riley, R. & Cervesato, I. *On the detection of kernel-level rootkits using hardware performance counters* in *ACM Asia Conference on Computer and Communications Security* (2017), 483–493.
77. Zhou, H., Wu, X., Shi, W., Yuan, J. & Liang, B. *HDROP: Detecting ROP attacks using performance monitoring counters* in *International Conference on Information Security Practice and Experience* (2014), 172–186.
78. Wang, X. & Backer, J. SIGDROP: Signature-based ROP Detection using Hardware Performance Counters. *arXiv preprint arXiv:1609.02667* (2016).
79. Das, S., Chen, B., Chandramohan, M., Liu, Y. & Zhang, W. ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security* **73,** 374–388 (2018).
80. Xia, Y., Liu, Y., Chen, H. & Zang, B. *CFIMon: Detecting violation of control flow integrity using performance counters* in *IEEE/IFIP International Conference on Dependable Systems and Networks* (2012), 1–12.
81. Yuan, L., Xing, W., Chen, H. & Zang, B. *Security breaches as PMU deviation: detecting and identifying security attacks using performance counters* in *Asia-Pacific Workshop on Systems* (2011), 6:1–6:5.
82. Aweke, Z. B., Yitbarek, S. F., Qiao, R., Das, R., Hicks, M., Oren, Y. & Austin, T. ANVIL: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices* **51,** 743–755 (2016).
83. Torres, G. & Liu, C. *Can Data-Only Exploits Be Detected at Runtime Using Hardware Events?: A Case Study of the Heartbleed Vulnerability* in *Hardware and Architectural Support for Security and Privacy* (2016), 2:1–2:7.
84. Herath, N & Fogh, A. These are Not Your Grand Daddys CPU Performance Counters–CPU Hardware Performance Counters for Security. *Black Hat* (2015).
85. Bahador, M. B., Abadi, M. & Tajoddin, A. *Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition* in *International Conference on Computer and Knowledge Engineering* (2014), 703–708.
86. Wang, X. & Karri, R. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on CAD of Integrated Circuits and Systems* **35,** 485–498 (2016).
87. Kazdagli, M., Huang, L., Reddi, V. & Tiwari, M. *Morpheus: Benchmarking computational diversity in mobile malware* in *Hardware and Architectural Support for Security and Privacy* (2014), 3:1–3:8.
88. Wang, X., Chai, S., Isnardi, M., Lim, S. & Karri, R. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization* **13,** 3:1–3:23 (2016).
89. Garcia-Serrano, A. Anomaly detection for malware identification using hardware performance counters. *arXiv preprint arXiv:1508.07482* (2015).
90. Zhang, T., Zhang, Y. & Lee, R. B. *Dos attacks on your memory in cloud* in *ACM Asia Conference on Computer and Communications Security* (2017), 253–265.
91. Jyothi, V., Wang, X., Addepalli, S. K. & Karri, R. *Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks* in *IEEE International Conference on VLSI Design* (2016), 587–588.
92. Patel, N., Sasan, A. & Homayoun, H. *Analyzing hardware based malware detectors* in *Design Automation Conference* (2017), 25:1–25:6.
93. Peng, H., Wei, J. & Guo, W. *Micro-architectural Features for Malware Detection* in *Advanced Computer Architecture* (2016), 48–60.
94. Martin, R., Demme, J. & Sethumadhavan, S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News* **40,** 118–129 (2012).
95. Uhsadel, L., Georges, A. & Verbauwhede, I. *Exploiting hardware performance counters* in *Workshop on Fault Diagnosis and Tolerance in Cryptography* (2008), 59–67.
96. Bhattacharya, S. & Mukhopadhyay, D. *Who watches the watchmen? : Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms* in *Conference on Cryptographic Hardware and Embedded Systems* (2015), 248–266.
97. Chiappetta, M., Savas, E. & Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* **49,** 1162–1174 (2016).
98. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O. & Francillon, A. *Reverse engineering Intel last-level cache complex addressing using*

performance counters in *Symposium on Recent Advances in Attacks and Defenses* (2015), 48–65.
99. Hunger, C., Kazdagli, M., Rawat, A., Dimakis, A., Vishwanath, S. & Tiwari, M. *Understanding contention-based channels and using them for defense* in *IEEE International Symposium on High-Performance Computer Architecture* (2015), 639–650.
100. Gruss, D., Maurice, C., Wagner, K. & Mangard, S. *Flush+ Flush: a fast and stealthy cache attack* in *Detection of Intrusions, Malware and Vulnerability Assessment* (2016), 279–299.
101. Payer, M. *HexPADS: a platform to detect "stealth" attacks* in *International Symposium on Engineering Secure Software and Systems* (2016), 138–154.
102. Zhang, T., Zhang, Y. & Lee, R. B. *Cloudradar: A real-time side-channel attack detection system in clouds* in *Symposium on Recent Advances in Attacks and Defenses* (2016), 118–140.
103. Gulmezoglu, B., Zankl, A., Eisenbarth, T. & Sunar, B. *PerfWeb: How to violate web privacy with hardware performance events* in *European Symposium on Research in Computer Security* (2017), 80–97.
104. Allaf, Z., Adda, M. & Gegov, A. *A comparison study on Flush+ Reload and Prime+ Probe attacks on AES using machine learning approaches* in *UK Workshop on Computational Intelligence* (2017), 203–213.
105. Wang, X., Konstantinou, C., Maniatakos, M. & Karri, R. *Confirm: Detecting firmware modifications in embedded systems using hardware performance counters* in *International Conference on Computer-Aided Design* (2015), 544–551.
106. Wang, X., Konstantinou, C., Maniatakos, M., Karri, R., Lee, S., Robison, P., Stergiou, P. & Kim, S. Malicious firmware detection with hardware performance counters. *IEEE Transactions on Multi-Scale Computing Systems,* 160–173 (2016).
107. Bruska, J., Blasingame, Z. & Liu, C. *Verification of OpenSSL version via hardware performance counters* in *Disruptive Technologies in Sensors and Sensor Systems* (2017).
108. Vogl, S. & Eckert, C. *Using hardware performance events for instruction-level monitoring on the x86 architecture* in *European workshop on system security* (2012).
109. Spisak, M. *Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures* in *WOOT* (2016).
110. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. & Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. **40,** 190–200 (2005).
111. *VX Heaven, Computer virus collection* 2014.
112. Sebastián, M., Rivera, R., Kotzias, P. & Caballero, J. *Avclass: A tool for massive malware labeling* in *Symposium on Recent Advances in Attacks and Defenses* (2016), 230–253.
113. Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H. & Van Steen, M. *Prudent practices for designing malware experiments: Status quo and outlook* in *IEEE Symposium on Security & Privacy* (2012), 65–79.
114. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* **11,** 10–18 (2009).
115. Bouckaert, R. R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald, A. & Scuse, D. WEKA manual for version 3-7-3. *University of Waikato, New Zealand,* 588–595 (2010).
116. Sommer, R. & Paxson, V. *Outside the closed world: On using machine learning for network intrusion detection* in *IEEE Symposium on Security & Privacy* (2010), 305–316.
117. Weaver, V. M. *Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface* tech. rep. (2016).
118. Weaver, V. M. *Self-monitoring overhead of the Linux perf_event performance counter interface* in *IEEE International Symposium on Performance Analysis of Systems and Software* (2015), 102–111.
119. Maxwell, M, Teller, P, Salayandia, L. & Moore, S. *Accuracy of performance monitoring hardware* in *Los Alamos Computer Science Institute Symposium* (2002).
120. Zhou, B., Gupta, A., Jahanshahi, R., Egele, M. & Joshi, A. *Hardware Performance Counters Can Detect Malware: Myth or Fact?* in *ACM Asia Conference on Computer and Communications Security* (2018), 457–468.
121. Herley, C. & van Oorschot, P. C. *Sok: Science, security and the elusive goal of security as a scientific pursuit* in *IEEE Symposium on Security & Privacy* (2017), 99–120.

APPENDIX A
QUESTIONS FOR LEAD AUTHORS

For the reproducibility of results we suggest that answers to the following questions be specifically addressed.

1) *High-level understanding*
   a) For what purpose did you use HPC?
      ❑ Performance
      ❑ Power analysis
      ❑ Security
      ❑ Debugging
      ❑ Operating system support
      ❑ Other
   b) How many total counters were used?
      ❑ <=4
      ❑ >4 and <=7
      ❑ >7
   c) How many configurable/programmable counters were used simultaneously?
      ❑ <=4
      ❑ >4
   d) How were the measurements taken?
      ❑ Polling
      ❑ Event-based sampling
   e) Which of the following tools were used?
      ❑ PAPI
      ❑ Oprofile
      ❑ perfctr
      ❑ perfmon2
      ❑ perf_event
      ❑ LIKWID
      ❑ LiMiT
      ❑ Intel VTune
      ❑ Custom
   f) Was the performance counter module natively implemented in the OS kernel (e.g., Perf_event)?
      ❑ Yes
      ❑ No
      ❑ Unsure
2) *Measurement granularity*
   a) At what level were measurements collected?
      ❑ Process/Application
      ❑ Processor core
      ❑ System-wide
   b) Performance counters were configured for the events in:
      ❑ User space only
      ❑ Kernel space only
      ❑ Both
      ❑ Unsure
3) *Context-switch monitoring*
   a) Was context-switching monitored?
      ❑ Yes
      ❑ No
      ❑ Unsure
   b) If yes, were performance counter values saved/restored at context switches?
      ❑ Yes
      ❑ No
      ❑ Unsure
4) *Performance monitoring interrupt (PMI)*
   a) At each PMI, was process-based filtering performed?
      ❑ Yes
      ❑ No
      ❑ Unsure
      ❑ Not applicable
   b) If measurements were obtained by event-based sampling, how was the sampling counter configured?
      ❑ For user space only
      ❑ For kernel space only
      ❑ Both
      ❑ Unsure
      ❑ Not applicable
5) *Experimental setup*

a) How was the performance counter monitoring module implemented?
❑ On a bare-metal system
❑ In a virtual machine
b) If a virtual machine was used, which of the following virtualization software was used?
❑ VMware
❑ KVM
❑ Virtualbox
❑ Other
❑ Not applicable
c) How many cores were there in your virtual machine or bare-metal system?
❑ One
❑ More than one
❑ Unsure
d) If more than 1 core was present, were HPCs on all cores monitored?
❑ Yes
❑ No
❑ Unsure
❑ Not applicable

6) *Non-determinism and overcounting issues*
a) Was the measurement error, resulting from non-determinism and overcounting of HPCs, addressed in your approach?
❑ Yes
❑ No
❑ Unsure
❑ Not applicable
b) Would you recommend using HPCs for the main application domain of your paper?
❑ Yes
❑ No
❑ Unsure

APPENDIX B
TOWARD MINIMIZING PERFORMANCE COUNTER ERROR

The results in Table V indicated that the overcount for memory operations using the `loop` instruction was approximately 1 overcount per 4 KB memory access. Our conjecture was that every 4 KB memory access there was an overcount in the branch and instructions retired counts. To validate this, we examined the number of page faults during the program execution. Our analysis shows that the number of page faults directly impacts the instruction and branch count. We used the Perf tool for measuring page faults. To see if we can then minimize the reported error, we adjusted the number of instructions and branches by deducting the number of page faults that occurred.

Listing 1: Source code for page fault experiment

```
# Aim: Trigger page fault on memory access

 .globl _start
_start :
 xor %rcx,%rcx
 mov $512,%rcx   # Setup a counter
 mov $buffer1,%rsi
mem:
 mov (%rsi),%rax # Triggers page fault
 add $4096,%rsi  # Advance to next page
 loop mem
exit :
 xor %rdi,%rdi # we return 0
 mov $60,%rax  # SYSCALL_EXIT
 nop
 syscall      # exit
.bss
.lcomm buffer1 ,8388608
```

To further comprehend the effect of page faults on instruction and branch events, we specifically wrote a test program (see Listing 1) that executes a loop, where a new memory page is accessed to trigger a page fault in each iteration. After varying the number of memory accesses and forcing the page faults, we observed a direct effect of page faults on the overcount of the instruction and branch events (shown in Table VII).

TABLE VII: Impact of page faults

	Expected values		HPC			Adjusted-HPC	
Benchmark	Ins	Br	Ins	Br	Pf	Ins - Pf	Br - Pf
page_fault_test	10	1	11	3	1	10	2
	13	2	15	5	2	13	3
	19	4	23	9	4	19	5
	31	8	39	17	8	31	9
	55	16	71	33	16	55	17
	103	32	135	65	32	103	33
	199	64	263	129	64	199	65
	391	128	519	257	128	391	129
	775	256	1031	513	256	775	257
	1543	512	2055	1025	512	1543	513

Pf: page faults

That analysis led to other findings. First, the adjusted measurement matched the expected values with minor variations, except `rep` string instructions. Second, for the `rep` string instructions, the overcount in branch events was minimized by adjusting the page faults, but the instruction retired event showed a large undercount. We found that HPC considers `rep` string instruction as one complex instruction, which performs similar memory operations repetitively. Therefore, it increments the instruction retired counter only when `rep` string instruction is retired. Pin and `INS-Dbg` tools, on the other hand, consider each memory operation as one instruction and increment the instruction counter at each repetition.

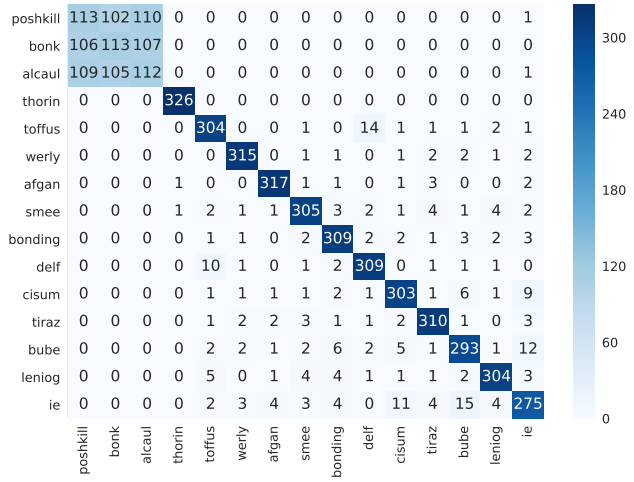
Digging deeper, we examined the memory load and store instructions for `rep` and `loop` string instructions. Since the load and store instruction events are only available in Skylake processors, we conducted this experiment only on Skylake (Haswell and Sandy Bridge architectures monitor load and store micro-op events). We found that the store instruction event is consistent and unaffected by page faults both for `rep` and `loop` cases (*e.g.*, `stosb`, `movsb`, `stosw`, `movsw`). Also, the store instruction event is incremented only once for `rep stosb/movsb/stosw/movsw`. Conversely, the load instruction event is directly affected by the page faults, but the effect is not consistent, such as in *rep_stosb/movsb/stosw/movsw*. Therefore, the number of page faults *cannot* be directly deducted in the load instruction measurement to adjust the values and minimize the error.

APPENDIX C
ALGORITHM FOR CS-PMI APPROACH

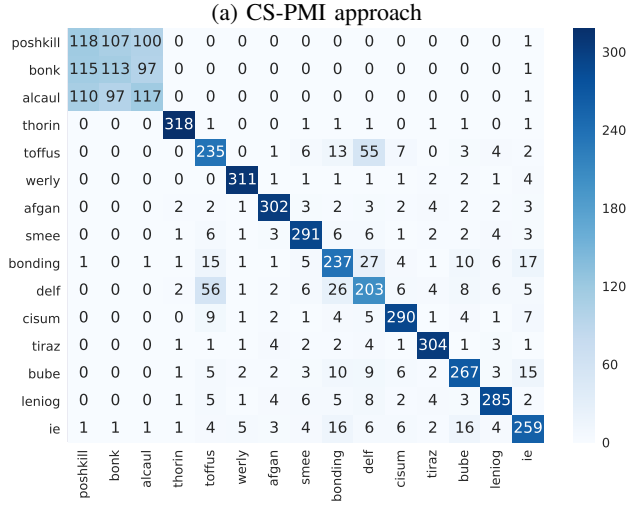
In Sec. VII, we presented our case study using CS-PMI approach, which augments the PMI approach with context switch monitoring. The CS-PMI approach is shown in Algorithm 1.

APPENDIX D
EXTENDED K-WAY CLASSIFICATION RESULTS

In Sec. VII, we presented our case study on K-way classification of malware using two approaches, CS-PMI and

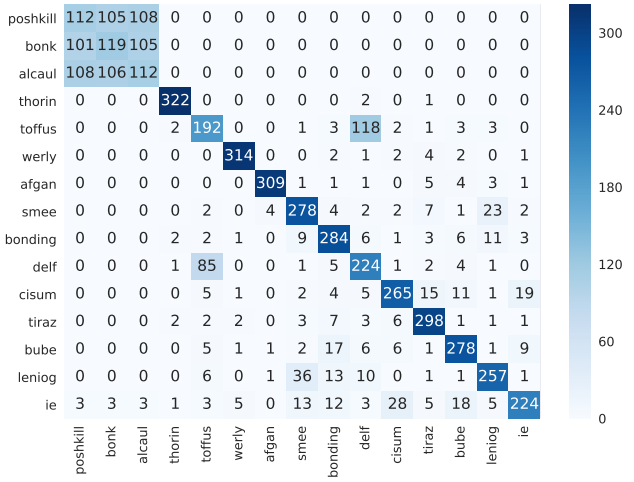


(a) CS-PMI approach

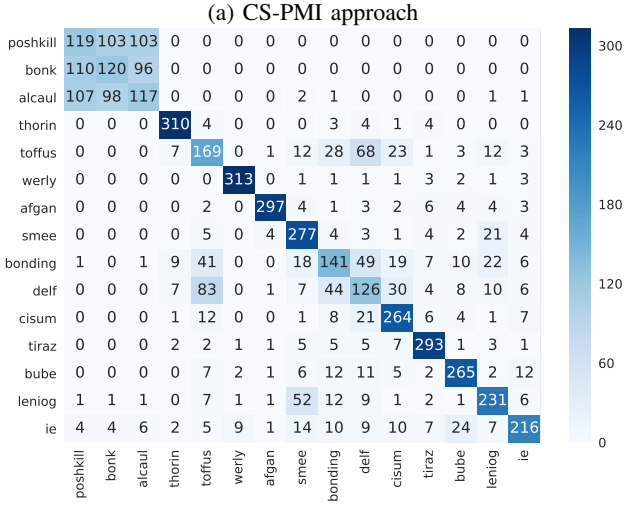


(b) PMI approach

Fig. 4: K-way classification: Decision Tree (J48).



(a) CS-PMI approach



(b) PMI approach

Fig. 5: K-way classification: KNN (IBk).

Algorithm 1 CS-PMI approach

```
1: function MODULE_INITIALIZATION
2:     Hook context switch
3:     Hook PMI interrupt
4: end function
5: function CONTEXT_SWITCH_HOOK
6:     if Next_process == Test_app then
7:         Test_app_flag = TRUE
8:         if Test_app runs for the first time then
9:             Initialize HPC
10:        else
11:            Restore HPC values
12:        end if
13:    else if Previous_process == Test_app then
14:        Save HPC values
15:    end if
16: end function
17: function PMI_HOOK
18:    if Test_app_flag == TRUE then
19:        Record HPC values
20:        Reset HPC
21:        Reset threshold value for triggering PMI
22:    end if
23: end function
24: function MODULE_UNLOAD
25:    Record HPC values saved during the last context switch
26:    Remove context switch and PMI hooks
27: end function
```

PMI. Table VIII presents the results for K-way classification using KNN algorithm. Fig. 4 and 5 show the corresponding confusion matrices for decision tree and KNN classifiers using two approaches.

APPENDIX E
SKID DURING PMI

We discussed the skid issue during the PMI delivery in §VIII-A. Fig. 6 shows the observed skid when PMI is set at different number of ret_miss events.

APPENDIX F
CODE AND DATA ARTIFACTS

We provide a stepwise tutorial at https://github.com/UNCSecLab/hpc.git on how to best use performance counters under different settings. Our kernel module implementing the CS-PMI approach is available on GitHub. To promote reproducibility of our experiments, our extensions to the benchmark suite by Weaver & McKee [33] are also available.

TABLE VIII: Performance evaluation of K-way classification: KNN (IBk)

	TPR		FPR		Precision		Recall		F-Measure		ROC Area	
Class	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ	PMI	Δ
poshkill	0.366	-0.022	0.049	-0.003	0.348	-0.003	0.366	-0.022	0.357	-0.012	0.658	-0.010
bonk	0.368	-0.002	0.045	0.002	0.369	-0.010	0.368	-0.002	0.368	-0.006	0.661	-0.002
alcaul	0.359	-0.015	0.046	0.002	0.360	-0.019	0.359	-0.015	0.359	-0.017	0.657	-0.009
thorin	0.951	0.038	0.006	-0.004	0.918	0.057	0.951	0.038	0.934	0.048	0.973	0.021
toffus	0.517	0.073	0.037	-0.013	0.501	0.134	0.517	0.073	0.509	0.103	0.740	0.043
werly	0.961	0.003	0.003	-0.001	0.958	0.009	0.961	0.003	0.960	0.006	0.979	0.002
afgan	0.910	0.038	0.002	-0.001	0.964	0.014	0.910	0.038	0.936	0.027	0.954	0.020
smee	0.850	0.003	0.027	-0.011	0.695	0.105	0.850	0.003	0.765	0.061	0.912	0.007
bonding	0.434	0.437	0.028	-0.013	0.520	0.283	0.434	0.437	0.472	0.363	0.703	0.225
delf	0.386	0.300	0.040	-0.007	0.406	0.187	0.386	0.300	0.395	0.241	0.673	0.153
cisum	0.811	0.001	0.022	-0.012	0.727	0.126	0.811	0.001	0.767	0.065	0.895	0.006
tiraz	0.897	0.015	0.010	0.000	0.869	0.000	0.897	0.015	0.883	0.007	0.944	0.008
bube	0.813	0.040	0.013	-0.001	0.821	0.025	0.813	0.040	0.817	0.032	0.900	0.021
leniog	0.709	0.079	0.018	-0.008	0.736	0.106	0.709	0.079	0.722	0.092	0.846	0.044
IE	0.663	0.025	0.012	-0.003	0.804	0.050	0.663	0.025	0.726	0.036	0.826	0.014
Weighted Avg.	0.666	0.073	0.024	0.005	0.666	0.075	0.666	0.073	0.665	0.074	0.821	0.039

Δ = Difference between the approaches (*i.e.,* CS-PMI - PMI)

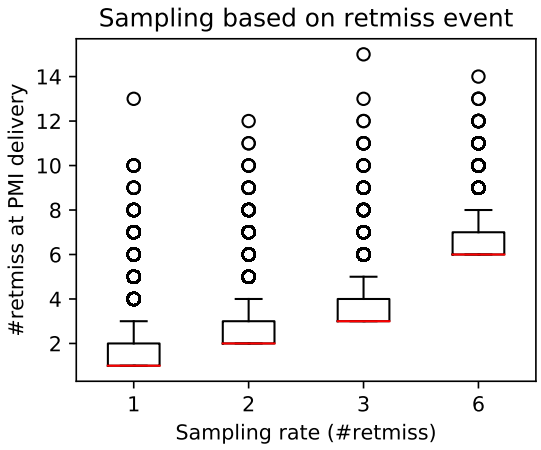


Fig. 6: Observed skid at different ret_miss based sampling: X-axis denotes the value of the return miss counter at which the interrupt is expected to happen (*i.e.,* α). Y-axis shows the value of return misses at the PMI delivery. The whiskers of the box plot, and the dots indicate the skid in the measurement.