On the Feasibility of Malware Unpacking via Hardware-assisted Loop Profiling

Binlin Cheng, *Shandong University;* Erika A Leal, *Tulane University;*
Haotian Zhang, *The University of Texas at Arlington;* Jiang Ming, *Tulane University*

https://www.usenix.org/conference/usenixsecurity23/presentation/cheng-binlin

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

On the Feasibility of Malware Unpacking via Hardware-assisted Loop Profiling

Binlin Cheng*
Shandong University, China
binlincheng@163.com

Erika A Leal
Tulane University, USA
eleal@tulane.edu

Haotian Zhang
The University of Texas at Arlington, USA
haotian.zhang@mavs.uta.edu

Jiang Ming†
Tulane University, USA
jming@tulane.edu

Abstract

Hardware Performance Counters (HPCs) are built-in registers of modern processors to count the occurrences of various micro-architectural events. Measuring HPCs values is a cost-effective way to characterize dynamic program behaviors. Because of the ease of use and tamper-resistant advantages, using HPCs coupled with machine learning models to address security problems is on the rise in recent years. However, lately the suitability of HPCs for security has been questioned in light of the non-determinism concerns: measurement errors caused by interrupt skid and time-division multiplexing can undermine the effectiveness of using HPCs in security applications.

With these cautions in mind, we explore ways to tame hardware event's non-determinism nature for malware unpacking, which is a long-standing challenge in malware analysis. Our research is motivated by two key observations. **First**, the unpacking process, which involves expensive iterations of decryption or decompression, can incur identifiable deviations in hardware events. **Second**, loop-centric HPCs profiling can minimize the imprecisions caused by interrupt skid and time-division multiplexing. Therefore, we utilize two mechanisms offered by Intel CPUs (i.e., Precise Event-Based Sampling (PEBS) and Last Branch Record) to develop a generic, hardware-assisted unpacking technique, called *LoopHPCs*. It offers a new, obfuscation-resilient solution to identify the original code from multiple "written-then-executed" layers. Our controlled experiments demonstrate that LoopHPCs can obtain precise and consistent HPCs values across different Intel CPU architectures and OSs.

*(1) Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, China; (2) School of Cyber Science and Technology, Shandong University, China; (3) Quan Cheng Laboratory, China.

†Corresponding author.

1 Introduction

Hardware Performance Counters (HPCs) are a set of special-purpose registers embedded in modern CPUs to record the counts of different micro-architectural events at runtime. Without the need for source code modifications, HPCs enable low-overhead access to a multitude of hardware-related activities, such as instruction counts, number of branches taken, hits/misses for L1∼L3 caches and translation lookaside buffer, and branch misprediction. HPCs are originally designed for computer professionals to perform low-level performance analysis or tuning [1–3]. For example, profiling a program with HPCs is a cost-effective way to attribute run-time costs to inefficient code bottlenecks (i.e., hotspots) [4, 5].

The intensive cyber arms race pushes security researchers to take advantage of hardware features [6, 7]. In addition to the low overhead, HPCs are also transparent and tamper-resistant: user-space programs cannot manipulate the values of hardware counters because they are running at different privilege levels. Recently, we have witnessed a surge of applying HPCs to security applications, such as exploit prevention [8–11], malware defense [12–16], and side-channel attack detection [17–20]. However, one aspect that is vital for the use of HPCs but has largely been overlooked by these security applications is HPCs' non-determinism—many hardware events show run-to-run variation and overcounting [21].

The latest, in-depth study [22] paints a cautionary tale for using HPCs in security: without accommodating the measurement imprecisions caused by HPCs' non-determinism, the claimed effectiveness against cyber threats (e.g., ROP attack and malware) can be compromised. Besides, the authors [22] provide guidelines to compensate for the noise and overcounting issues associated with using HPCs, such as per-process filtering and adjusting HPCs data when context switches or page faults happen. However, the interrupt skid, first proposed by Weaver & Dongarra [23], remains unsolved. The interrupt skid represents the slipping phenomenon between the instruction that actually triggers a Performance Monitoring Interrupt (PMI) and the instruction indicated by the CPU. The authors [22] use the return miss counter as an example to demonstrate that the skid occurs at every sampling rate.

The second source of non-determinism is attributed to the imperfect time-division multiplexing. An embarrassing situation is that the small number of HPC registers stands in stark contrast to hundreds of hardware events. For example, recent Intel processors have eight programmable counters per physical core (four per virtual core if hyper-threading is enabled) [24], while the Skylake model defines more than 600 events [25]. However, in many cases, users need to measure a large number of events that are far beyond the available hardware counters. To improve the measurement efficiency, time-division multiplexing allows several events to timeshare a single hardware counter and schedules different events to be sampled during the time series of execution [26]. However, Lv et al. [27] observed an average 28.3% error rate, including outliers and missing values, and multiplexing more events simultaneously would further exacerbate the errors.

In view of HPCs' non-determinism nature, in this work, we approach the problem of using low-level hardware events in malware unpacking, a veritable challenge to large-scale malware analysis over the past two decades [28, 29]. We capitalize on common unpacking behaviors to minimize the non-determinism's impact, in hopes of reinforcing the confidence of using HPCs for security applications. Binary packing, which encodes executable code via encryption/compression and recovers the original code at runtime, has become the most common obfuscation method adopted by malware authors to stay under the detection radar [30–32]. Existing unpacking tools rely on dynamic binary instrumentation/translation [33–36] to identify newly generated code, or capturing the lookup to the newly rebuilt import address table to detect the original code's execution [37]. However, these software-level solutions are insufficient to defeat sophisticated packers with anti-unpacking techniques [38–40]. Our hardware-assisted unpacking idea is motivated by two key observations.

First, we hypothesize and empirically verify the causation between the performance bottleneck of binary unpacking and low-level HPCs abnormals. The process of generating original code from the packed data involves iterations of fixed decryption or decompression operations, which can be treated as a code hotspot. Our study shows that the hot loop of an unpacking algorithm can dominate up to 93%$\sim$99% of CPU cycles. **Second**, loop-level HPCs measurement can minimize the imprecisions caused by HPCs' non-determinism. We observe that interrupt skid effects go across instructions or basic blocks, but most of them are still within the loop body's scope. Besides, iteration-division multiplexing for the unpacking's hot loop reveals much smaller measurement errors than time-division multiplexing.

These two observations inspire us to develop *LoopHPCs*, an obfuscation-resilient unpacking technique using hardware features only. The core of LoopHPCs is loop-centric HPCs profiling: we first utilize a hardware tracing mechanism of Intel CPUs, Last Branch Record (LBR), to dynamically detect loop structures for a running program; then, we associate the HPCs values that are collected by Precise Event-Based Sampling (PEBS) to the detected loop. Advanced binary packers have evolved from a single "written-then-executed" layer to multiple layers [37], and the primary challenge of unpacking is to determine which layer contains the original code. We customize LoopHPCs to identify each unpacking layer at the hardware level and associate loop-centric HPCs values to the related layer. After that, we apply machine learning models to determine which layer contains the original code. Compared with software-level unpacking solutions [33–37], LoopHPCs is transparent and tamper-resistant on multiple platforms.

We have conducted a set of experiments to evaluate LoopHPCs from four dimensions: resilience to HPCs' non-determinism, consistency across multiple platforms, effectiveness against various binary packers, and performance. We first measure the non-determinism's impact across different Intel CPU architectures, including Kaby Lake, Coffee Lake, and Comet Lake. Our loop-centric HPCs profiling only leads to negligible interrupt skid errors, and iteration-division multiplexing on off-the-shelf packers reduces the average HPCs errors by as much as 22.0x (i.e., from 54.9% to 2.5%). Besides, we demonstrate that our collected HPCs values are consistent on both Windows and Linux OSs. Our comparative evaluation with software-level counterparts [35, 36] shows that LoopHPCs reveals better resistance to the packers equipped with a variety of evasion tricks, and LoopHPCs' unpacking results achieve the optimal malware detection rate. At last, we apply LoopHPCs to 74,938 in-the-wild packed malware samples, which cover Windows, Linux, and low-entropy packers [41]. We take two heuristics to evaluate whether the unpacking is successful, and our results are encouraging. In a nutshell, our paper makes the following contributions.

- Harnessing HPCs' non-determinism nature is challenging in the general case. Our study shows that, by taking advantage of common malware behaviors, we can find a practical solution (e.g., loop-centric profiling) to circumvent this challenge and amplify HPCs' benefits. Our study advances the proper use of HPCs in security.
- Our proposed unpacking technique exploits modern CPU features and represents a promising direction towards hardware-assisted malware analysis. LoopHPCs exhibits strong resistance to anti-unpacking methods that can impede its software-level counterparts.
- Our large-scale evaluation with packed malware in the wild demonstrates LoopHPCs' effectiveness across multiple platforms. We have released LoopHPCs' source code to facilitate reproduction and reuse at https://github.com/binlinc/LoopHPCs.

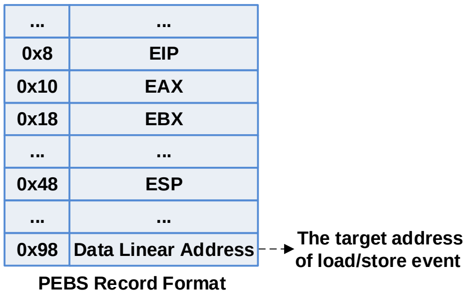


Figure 1: PEBS record logs a snapshot of CPU state.

2 Background and Related Work

In this section, we provide the background information needed to understand our work's motivation. We first present two hardware features utilized in this project (i.e., PEBS & LBR). Next, we discuss HPCs' non-determinism problem and existing efforts to remedy measurement errors. Then, we introduce the new trends of binary packing techniques and their impact on existing generic unpacking approaches. At last, we show performance data to demonstrate that the hot loop structure is prevalent in packers, which motivates us to apply HPCs to binary unpacking.

2.1 PEBS and LBR

HPCs support event-based sampling based on the occurrence of certain events, and a Performance Monitoring Interrupt (PMI) will be triggered if the monitored hardware event exceeds the pre-configured threshold. For example, a user configures the PMI using the `STORE` event with a threshold set at 50. Once the number of `STORE` events exceeds 50, a PMI will be triggered, and all hardware counters can be read in the user-defined PMI handlers.

Precise Event-Based Sampling (PEBS) Intel Core-based processors also support an advanced event-based sampling, namely Precise Event-Based Sampling (PEBS) [24]. When the monitored event overflow occurs, PEBS saves a snapshot of processor state, such as register values and load/store addresses, into a designated memory region (i.e., *PEBS buffer*). We configure the PMI to be raised as long as the PEBS buffer receives one record of CPU state. At this moment, we read all HPCs values, save PEBS buffer's record, reset it, and then resume PEBS's monitoring. The benefit of doing so is that we can associate HPCs to the CPU state at the time of the event, and the design of LoopHPCs relies on such detailed contextual information. As shown in Figure 1, the PEBS buffer contains the state of the general-purpose registers and data linear address (i.e., the target address of load/store event).

Last Branch Record (LBR) LBR is a hardware tracing feature offered by modern Intel processors [24]. According to different CPU models, LBR can record 16 or 32 most recent branch pairs (source address *vs.* target address) into a register. LBR mechanism is transparent without code injection, and also efficient due to the direct access to CPU registers. In our work, we overcome LBR's size limit by counting the `BRANCHES` event to detect a loop structure.

LBR vs. BTS Intel also provides another branch monitor mechanism, called Branch Trace Store (BTS), to record branch records into a memory buffer. LBR differs from BTS in three aspects: (1) LBR overwrites the records when the LBR stack is full, while BTS can halt the application when the memory buffer is full [42]; (2) LBR supports filtering branch types (e.g., branch, call, and ret), but BTS cannot [43]; (3) LBR has lower overhead than BTS because LBR accesses CPU register directly.

Table 1: The non-determinism sources from the paper [22].

Source	Affected Event	Solution
Multi-Cores	ALL Events	Single-core Configuration
Multi-Processes	ALL Events	Per-process Filtering
Context Switches	ALL Events	CS-PMI[1]
Page Faults	INST_RETIRED BRANCHES LOAD	Adjusted-HPCs[2]
Interrupt Skid	ALL Events	N/A

[1] CS-PMI: save and restore HPCs data during context switches.
[2] Adjusted-HPCs: adjust the affected events by deducting the number of page faults that occurred.

2.2 Non-determinism Pitfalls

The initial study by Weaver et al. [21, 23] raises doubts about whether HPCs can deliver expected, deterministic results, because many hardware events exhibit run-to-run variation even in a strictly controlled environment. The recent SoK paper [22] extends Weaver et al.'s work to question the suitability of using HPCs for security. The authors concluded that HPCs imprecisions related to non-determinism still persist in modern CPUs and thus impair the claimed security gains, such as preventing ROP attacks and detecting malware. Sadly, 37 out of 41 security papers that used HPCs overlooked the non-determinism issues, and *none of them tried to address the measurement errors due to non-determinism* [22].

As we summarized in Table 1, Das et al. [22] discuss several sources of non-determinism and propose possible solutions to mitigate the measurement errors.

(1) Multi-cores: the tested program is configured to run on a single core only, avoiding noise events from other cores.
(2) Multi-processes: only the hardware events from the target process are sampled (i.e., per-process filtering).

(3) Context switches: HPCs data must be saved and stored during context switches to avoid contamination from other processes.
(4) Page faults: certain events are approximately one over-count per each page fault. Users should deduct the number of page faults from the affected events.

However, Das et al. [22] did not give any recommendation regarding how to remedy the adverse impact of interrupt skid.

Interrupt Skid This non-determinism source happens when the event-based sampling is enabled. Due to hardware limits, when a PMI is triggered, there is an inevitable time delay to finally stop the processor, resulting in a discrepancy between the instruction indicated by the CPU versus the instruction that actually triggers the PMI. Even Intel's Precise Event Based Sampling (PEBS) is susceptible to interrupt skid [44]. Figure 2 shows an example of the skid phenomenon. Line 5 involves two *load* events because it reads value from locations of $matrix_a[i][k]$ and $matrix_b[k][j]$. When the matrix multiplication terminates ($1,000$ iterations in total), the expected number of load events at Line 5 should be $2,000$. However, when we configure Intel's PEBS to find out the specific instructions that raise the interrupt, we only observe 152 load events at Line 5. The remaining load events spread over Line 4 (764 times), Line 6 ($1,083$ times), and Line 7 (1 time), respectively.

In contrast, if we zoom out to focus on the innermost loop (Line 4∼Line 6), the observed load events within this loop are $1,999$, which is very close to the expected result. This leads to one of our key insights: the instruction discrepancy caused by an interrupt skid may go beyond instructions or basic blocks, but it is mostly within the loop body's scope.

Time-division Multiplexing Another major source of non-determination is due to time-division multiplexing, which allows multiple events to timeshare a single hardware counter. One counter can be dedicated to a single event during the whole profiling time; this sampling style is *accurate* but *not efficient* because the number of events that can be simultaneously measured is strictly limited to the number of HPC counters. The fundamental tension between less than ten available HPC counters and hundreds of hardware events pushes the rise of multiplexing to improve efficiency. However, the popular time-division multiplexing can incur large measurement errors in terms of both outliers and missing values, and it has become a big concern to the high-performance computing community [21, 26, 27, 45]. For example, an expected event could be lost because it only happens during an un-sampled time interval.

In our project, we need to empirically select the most significant hardware events from a set of candidates during the offline training phase; our purpose is to use the one-count-one-event style to measure these significant events for accuracy during the online unpacking phase. Even so, a less error-prone multiplexing approach is still necessary for us to reliably measure all candidate hardware events. Counter-

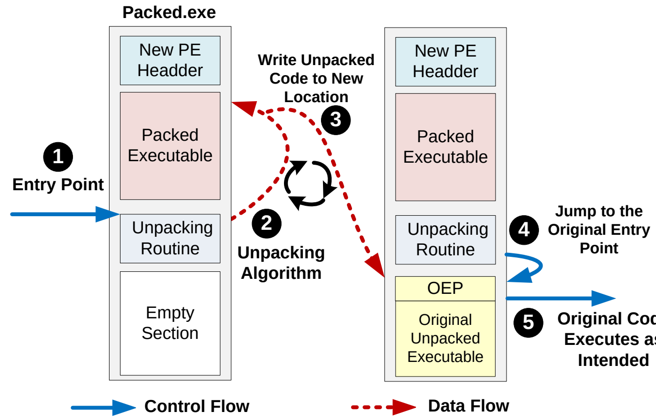


Figure 3: A typical workflow of the unpacking process.

Miner [27] adopts data mining and machine learning techniques to replace outliers and fill in missing values, at the cost of large latency overhead of reading counters. BayesPerf [45] presents a Bayesian model to infer true HPCs from noisy HPCs, and it also has an FPGA accelerated version to reduce the huge latency overhead. Instead of performing heavy computations like CounterMiner [27] and BayesPerf [45], another key insight is that the hot loop feature of unpacking (see §2.4) provides another option to correct for HPCs errors, that is iteration-division multiplexing—scheduling different events to be sampled during the iterations of the hot loop execution.

2.3 Multi-layer Unpacking & Evasions

Cyber-criminals are highly motivated to obfuscate malware code to evade security analysis. Among various obfuscation schemes, binary packing is believed to be a panacea to impede static code analysis [30–32]. A recent study on malware daily dataset shows that most new variants are just repackages of the previous version [28]. Binary packers first encode the executable through encryption or compression and attach an unpacking routine to a packed version. As shown in Figure 3, when the packed version starts running (❶), the unpacking routine first decodes the packed code (❷) and writes it to memory pages (❸). After that, the execution flow will jump to the original entry point (OEP) (❹) to resume malware payload execution (❺). In this way, the actual malicious code stays unrecognizable until at runtime, making it immune to security analyses that rely on static code features.

The evolution of packers reveals two notable trends towards frustrating reverse engineering: 1) the unpacking process passes through layers of self-modifying code (i.e., "written-then-executed" layers); 2) various anti-analysis tricks are embedded to deter unpacking attempts. The emerging low-entropy packers conceal packed malware behind non-packed programs [41], and the last "written-then-executed" layer does not necessarily contain the unpacked executable file [37].

Especially, the embedded anti-analysis tricks cover detection heuristics against different dynamic analysis components,

```
void multiply_matrices()
{
  1:  for (int i = 0; i < 10; i++) {
  2:     for (int j = 0; j < 10; j++) {
  3:        float sum = 0.0;
  4:        for (int k = 0; k < 10; k++) {
  5:            temp =matrix_a[i][k] * matrix_b[k][j];
  6:             sum = sum + temp;      }
  7:        matrix_r[i][j] = sum;  } } }
```

	Interrupt Skid Impact		Loop-centric HPCs Profiling	
	Expected Value	Observed Value	Expected Value	Observed Value
Line 4		764		
Line 5	2000	152	2000	1999
Line 6		1083		
Line 7		1		1

Figure 2: An example of interrupt skid *vs.* the effect of loop-centric profiling.

UPX

```
loop:
  dec   esi
  mov  eax, ebx
  mov  ecx, esi
  shr   eax, cl
  and   eax, 0x1
  ...
  test   esi, esi
  jne   loop
```

(a) Compression Packer

tElock

```
loop:
  ...
  rol   byte  ptr  [ebx+ecx],  0x5
  add  byte  ptr  [ebx+ecx],  cl
  xor  byte  ptr  [ebx+ecx],  0x67
  inc  byte  ptr  [ebx+ecx]
  dec  ecx
  jnle  loop
```

(b) Encryption Packer

Figure 4: The hot loop examples in the unpacking process.

Table 2: The performance data of hot loops in the unpacking routine of various off-the-shelf packers, and the malware payload is WannaCry [46]. The complete table is shown in Appendix A.

Sample	# of Max Iterations	Inst (%)	Cycle (%)
UPX	3,359,627	99%	99%
Enigma	432,110	95%	98%
Yoda's Protector	920,837	97%	98%
Obsidium	698,330	90%	94%
SoftwarePassport	3,145,470	91%	93%
Pelock	3,451,282	97%	98%
Telock	945,821	99%	98%
Pespin	418,183	92%	97%
Armadillo	3,361,391	93%	97%
ACProtect	918,139	92%	99%

such as debuggers, virtual machines, binary instrumentation, and hooking [38, 39]. As a result, *existing generic unpacking approaches [33–37] struggle to remain transparent from packed malware*.

2.4 Hot Loop in the Unpacking Process

The particular unpacking algorithms range from lossless data compression, XOR cipher, to symmetric encryption. The unpacking process overwrites a new memory area with either the decrypted or decompressed binary code of the original program, involving iterations of costly runtime operations. Therefore, the loop is such an essential structure that unpacking algorithms simply cannot be expressed without it. In this paper, we name the loop accounting for algorithmic bottlenecks as the *hot loop*. Figure 4 shows two hot loop examples in a compression packer (UPX) and encryption packer (tElock), respectively. We also collect three kinds of performance data for hot loops in different unpacking processes: the number of max iterations, the percentages of instructions retired and CPU cycles occupied. As indicated by Table 2, hot loops are indeed performance bottlenecks.

Justification Another criticism for using HPCs in malware detection is the so-called semantic gap [47]: generally, there is no causation between low-level hardware events and high-level malicious activities. We agree with this viewpoint. In contrast, the prevalence of hot loops in unpacking executions, the causation between hotspots and HPCs abnormals, and more opportunities to tame HPCs' non-determinism nature at the loop level well justify the feasibility of our research.

3 LoopHPCs Overview

We measure HPCs values at the loop level to characterize the unpacking layers that recover the original code. We illustrate the overview of LoopHPCs' online unpacking phase in Figure 5. The input to LoopHPCs is a packed malware sample. LoopHPCs works as a kernel driver to interact with hardware features (PEBS and LBR) and monitor the layers of self-modifying code. A transparent unpacking is impossible if the packed malware runs at the same privilege level as the unpacking tool [48]. Our design of using low-level hardware features offers a transparent environment to identify the layers containing the unpacked program, making user-level malware difficult to fingerprint the presence of LoopHPCs.

When the packed malware starts running, we first configure PEBS and LBR to capture the entry of each "written-then-execute" layer. Then, we enable loop-centric HPCs profiling for each layer. Typically, only the layers in charge of recovering the original program exhibit the hot loop characteristic and thus incur identifiable deviations in HPCs samples. Therefore, we further apply pre-trained machine learning classifiers to identify the original code in the memory. At last, we perform a memory dump to deliver the unpacked malware.

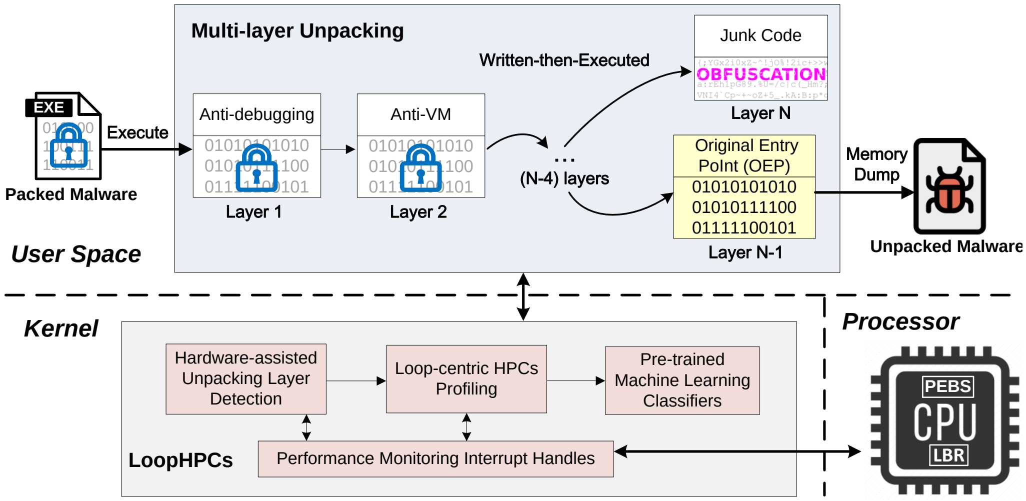


Figure 5: The overview of LoopHPCs' online unpacking phase.

Among eight programmable hardware counters, two of them are occupied for two events (`BRANCHES` and `STORE`) as control variables to assist in loop detection and unpacking layer detection, respectively. For the remaining six counters, we adopt the one-count-one-event sampling style for better accuracy during the online unpacking phase. Therefore, one important task during the offline training phase is to select six significant events with higher discriminative power. We achieve this goal using iteration-division multiplexing coupled with Principal Component Analysis (PCA) [49]. We have already integrated the recommendations of Das et al. [22] into LoopHPCs to compensate for related measurement errors. The next two sections elaborate on the details of LoopHPCs.

4 Loop-centric HPCs Profiling

In this section, we present loop-centric HPCs profiling, which attempts to further minimize the negative effects of non-determinism. Loop-centric HPCs profiling is also a key step for our hardware-assisted unpacking solution.

4.1 Loop Detection via Hardware Features

Dynamically detecting a loop structure has been well studied by the previous work on top of dynamic binary instrumentation platforms [50–53]. As a loop is typically an *intra-procedural structure*, a common heuristic to recognize a loop is the code region "between a branch with a negative offset and its target [52]." Because it is trivial for a program to detect whether it is running in a dynamic binary instrumentation environment [54], we explore using hardware features only to detect a loop by interacting with both PEBS and LBR. Our rule of loop detection is defined as follows:

Definition 1 *Assume (Source, Target) is a branch pair logged by LBR. "Source" is the source address of a branch pair in LBR, and "Target" is the corresponding target address. A **loop** is detected when Target<Source, and all instructions in the range of addresses [Target,Source] form a **loop body**.*

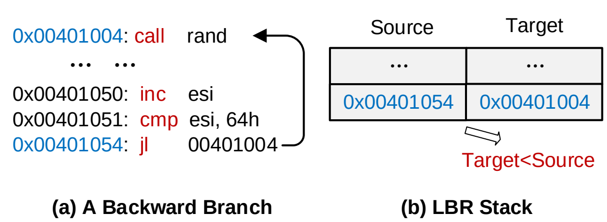


Figure 6: An example of loop detection using LBR.

Figure 6 shows an example of how to detect a loop from LBR's record. In Figure 6(a), there is a backward branch from 0x401054 to 0x401004, and the LBR stack in Figure 6(b) contains a corresponding branch pair (0x401054, 0x401004). For this branch pair, the target address (0x401004) is less than the source address (0x401054), so we can find a loop from 0x401004 to 0x401054 according to Definition 1.

Recursive Functions Please note that recursive functions also satisfy Definition 1. Figure 7 shows an example of the recursive function: when the upper-level function calls the lower-level function (e.g., from 0x401065 to 0x401041) or the lower-level function returns to upper-level one, they all result in backward branches. In this project, to prevent a possible evasion that transforms loops to semantically equivalent recursive functions, we do not differentiate these two cases.

Loop Termination We stop HPCs profiling or multiplexing when a loop is terminated. According to the loop detection

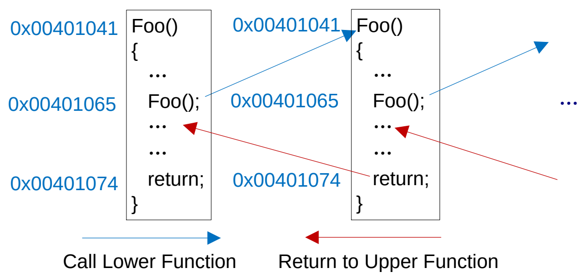


Figure 7: An example of the recursive function.

work of Tubella & Gonzalez [50], a loop is **active** if it is not terminated by any of the following three instructions.

(1) A not taken branch at the address *Source*;
(2) A taken branch or a jump from an address within the loop body to a target address outside the loop body;
(3) A return instruction at an address within the loop body.

We translate these three loop termination conditions into a new rule by just checking LBR's record.

Definition 2 *Assume addresses [Target,Source] is the loop body detected by Definition 1. For a new LBR record (S, T), if either S or T is not in the range of addresses [Target,Source], the loop [Target,Source] is terminated.*

Nested Loop The unpacking algorithm may be expressed as a composition of loops, namely a nested loop—it has an inner loop within the body of an outer one. We detect nested loops and enable HPCs profiling in the range of the outermost loop. Figure 8 shows an example of the nested loop.

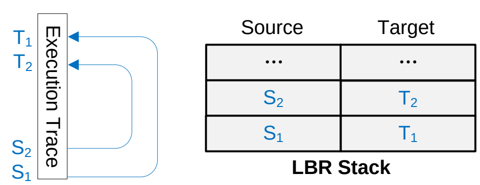


Figure 8: A nested loop example.

Definition 3 *Assume there are two loops $[S_1,T_1]$, $[S_2,T_2]$ traced by LBR. A **nested loop** is detected if $S_2 < S_1$ and $T_2 > T_1$. $[S_1,T_1]$ forms the outer loop, and $[S_2,T_2]$ constitutes the inner loop. For the nested loop, if the inner loop is active, the outer loop is active as well.*

Definition 1∼3 regulate how to find an active loop at runtime via hardware features. Given a transparent loop detection method, another practical challenge rears its head.

LBR's Limited Size Unlike BTS, LBR can only record 16 or 32 most recent branch pairs. When the number of records exceeds LBR's maximum value, the new branch pairs will overwrite the old ones [24]. Therefore, LBR is vulnerable to the so-called "history-flushing attacks" [55]. To bypass this limitation, we leverage one hardware counter as a control variable to virtually extend the LBR stack to a user-defined memory space. In particular, we configure one hardware counter to measure the `BRANCHES` event with a threshold set at **1**. In this way, a PMI will be raised whenever the LBR buffer receives a new branch pair, and then we detect whether current LRB records form an active loop or indicate a loop is terminated. Different actions will be taken according to the loop status, such as enabling/disabling HPCs profiling or iteration-division multiplexing. We save LBR's record to the user-defined memory space at the interval of every 32 LBR records before they are overwritten.

As LBR's buffer size is 32, another option is to customize the `BRANCHES` event with a threshold set at **32**. As a result, we detect an active loop and its termination only when the LBR buffer is full. This tradeoff lowers the PMI frequency for better runtime performance, but it is very likely to miss the exact loop starting and termination points. Under this threshold setting, we may miss the first few iterations (at most 32) of the loop, and the last group of HPCs values may contain noise data occurring out of the loop. In this paper, we stick to the first strategy for the accuracy concern.

4.2 Hardware Events Profiling

We record the HPCs values at the period of loop execution. When an active loop is detected, we enable available HPC registers to count the occurrences of preselected hardware events (§4.3 will discuss how to select them). When the loop is terminated, we stop the profiling, save HPCs values, and rest them. After collecting *n* hardware events for a loop, we represent them as an n-dimensional feature vector, and its definition is shown as follows.

Definition 4 *Profile(loop)=$[e_1,e_2,...,e_n]$, where e_i represents the counted value of i-th hardware event.*

In the case of a nested loop, since the outermost loop is always active during the execution of the nested loop, we take the feature vector of the outermost loop to represent the entire nested loop. During the offline training phase, we use the collected feature vectors to train multiple machine learning classifiers using the *scikit-learn* package [56], including Decision Trees, Random Forests, Nearest Neighbors, K-Nearest Neighbor, and Naive Bayes. The trained classifiers are used to predict the recovery of the original unpacked program at the online unpacking phase.

Regarding the non-determinism caused by interrupt skid, measuring HPCs at loop level can tolerate the instruction slipping phenomenon: the discrepancy between the instruction pointed out by the CPU versus the instruction that actually triggers the PMI can go across several instructions or even

Table 3: Significant events related to unpacking.

Selected Events	Description
STORE	All store instructions retired.
LOAD	All load instructions retired.
L1_HIT	Retired load instructions that hit L1 cache.
L3_HIT	Retired load instructions that hit L3 cache.
L1_MISS	Retired load instructions that missed L1 cache.
BRANCHES	All branch instructions retired.

basic blocks, but it does not break through the loop body's scope in most cases. We empirically verify this benefit.

4.3 Selection of Events

Modern Intel processors provide eight HPC registers at most [24]. We have already occupied one of them to control LBR's buffer size; another HPC register is for the exclusive use of detecting "written-then-executed" layers (see §5.2). Our loop-centric HPCs profiling can configure six hardware counters, and each one is dedicated to a single event during the whole profiling time. This subsection discusses how to determine these six *significant* events to be measured.

Not all hardware events are equally significant in characterizing the unpacking process. Based on our examination of the existing literature using HPCs for security and our understanding of binary packers, we shortlist 37 candidate events in Table A2. For example, SAP [57] observes that some packers generate the `SMC` event at run time, and therefore we consider this event as one of our candidate events.

Due to the limited number of HPCs, we run the ground-truth dataset and measure all candidate events by *iteration-division multiplexing*. That is, we divide the events into multiple batches of six events and run each batch at one iteration of the packer's hot loop. All of the batches will be run in a round-robin manner. The intuition behind our design is that the HPCs values are relatively stable during the packer's hot loop execution, which consists of iterations of fixed decryption or decompression operations.

Zhou et al.'s work [47] provides a quantitative analysis via Principal Component Analysis (PCA) [49] to select the six most significant ones from hundreds of hardware events. Among 37 candidates, we use the same methodology to select six events that are more valuable for modeling packers' hot loops. The selected six events are shown in Table 3.

Different from most malware detection work using HPCs, we can easily infer the reasons why these selected events can be mapped to malware unpacking behaviors. First, the unpacking process *reads* the packed payload, unpacks it, and *writes* it to a new memory region, resulting in a set of `LOAD` and `STORE` events. Second, as the unpacking unit is typically a fixed-size data block, many small memory segments would be loaded into CPU caches frequently, leading to remarkable cache hit/miss events. At last, the `BRANCHES` event is included because the hot loop typically involves a large number of backward branches. For example, the hot loop's iteration numbers in Table 2 are at the scale of $10^5 \sim 10^6$.

5 Hardware-assisted Binary Unpacking

5.1 Problem Scope and Research Questions

Multi-layers Complicated binary packers have evolved from the single-layer packer to the multi-layer packer. The so-called "written-then-executed" layers represent the iterations of dynamically generating new code in memory and then executing it. As shown in Figure 5, many layers serve only to frustrate unpacking attempts—they are not performance bottlenecks of the unpacking process. Instead, only one or several layers exhibit hot loops to recover the original program.

Multi-frames In addition to multiple layers, another evolutionary direction is multiple frames [39], which represents a completely different challenge. Multi-frame packer is out of our scope. According to packer classification in S&P'15 [39], single-frame packers include packers of Type-I, Type-II, Type-III and Type-IV, multi-frame packers include packers of Type-V and Type-VI. LoopHPCs aims to deal with packers from Type-I to Type-IV.

Usage Scenario LoopHPCs is intended to be used in offline, and it runs on the bare-metal machines.

Research Questions Specially, our hardware-assisted binary unpacking addresses two research questions:

(1) **Q1**: How to use hardware-level mechanisms to identify each "written-then-executed" layer?
(2) **Q2**: How to determine which layer contains the original code via loop-centric HPCs profiling?

5.2 Answer to Q1: Detect Unpacking Layers

The key to Q1 is exploring ways to map the instruction-level feature of "written-then-executed" to the corresponding hardware features. Although we have several optional hardware mechanisms, the choice of them also needs to meet two requirements to ensure accuracy: 1) they are not affected by interrupt skid; 2) they can capture the entry point (e.g., the first basic block) of each layer. For example, configuring the event `INST_RETIRED` with a threshold set at **1** is supposed to trigger a PMI at each executed instruction, but this configuration is susceptible to interrupt skid. Also, tracking all executed instructions will incur a large overhead. Using `NX bit` or "Page Faults" [58] can find a "written-then-executed" memory page, but multiple unpacking layers may locate in the same memory page. Therefore, our choice is to interact with PEBS and LBR.

Figure 9 illustrates our design. **First**, the `STORE` event is corresponding to the "write" operation. Recall the PEBS buffer shown in Figure 1, the "Data Linear Address" contains

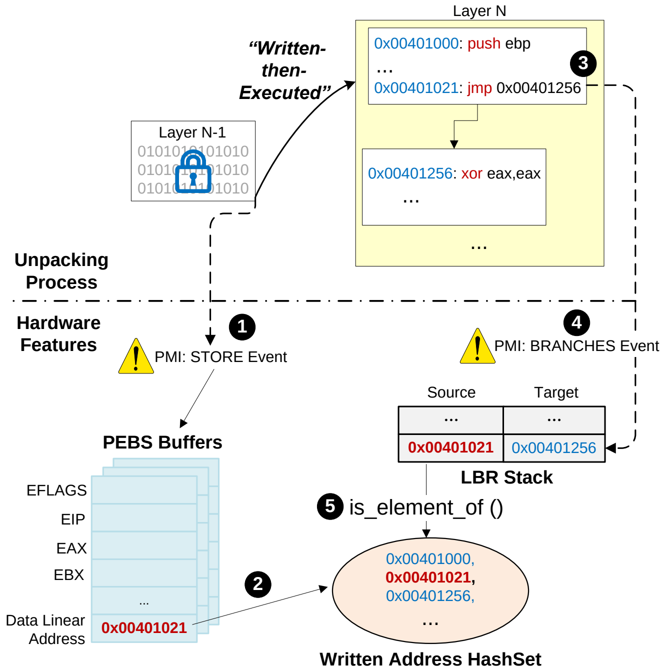


Figure 9: Capture unpacking layers via PEBS and LBR.

the target address of the store event. Besides, the "Data Linear Address" is not affected by interrupt skid. Therefore, we configure the `STORE` event with a threshold set at **1**; when this PMI is triggered (❶ in Figure 9), we will store each PEBS Buffer. Especially, we will maintain a `HashSet` data structure to store written addresses for future quick lookup (❷).

Second, to detect the "then-executed" behavior, we look up the end address of the first executed basic block in the written address `HashSet`. Similar to our solution to bypass LBR's limited size, we configure the event `BRANCHES` with a threshold set at **1**. When the first written basic block gets executed (❸ in Figure 9), a PMI will be raised (❹). As the last instruction of ❸ is a branch, now LBR's last record is this branch's source and target addresses. If the source address is an element of the written address `HashSet`(❺), we can determine that a "written-then-executed" layer is detected.

5.3 Answer to Q2: Measure Layer Cost

Given different unpacking layers, the next step is to detect which layer contains the original code. Our observation is that the sum of loop-centric HPCs values that occurred in each layer can be a prominent feature.

When a "written-then-executed" layer is detected, we enable loop-centric HPCs profiling for each active loop until the next layer starts. Suppose $loop_1$, $loop_2$,..., and $loop_m$ co-generate the $Layer_i$. We define "layer cost" as the sum of all feature vectors of $loop_1$ to $loop_m$. Note that a layer may be generated without any loop. For this layer, its layer cost is a zero vector.

Definition 5 *$Cost(Layer_i) = \sum_{i=1}^{m} Profile(loop_i)$, where $loop_1$, $loop_2$,..., and $loop_m$ co-generate the $Layer_i$.*

Once we get the layer cost of each layer, we tag it as "the original code" or not. Then, we use the tagged layer cost dataset of training set to train machine learning classifiers. After that, the trained classifiers can determine which layer contains the original code for the testing set.

6 Evaluation

We conducted a set of experiments to evaluate LoopHPCs' effectiveness from four aspects. (1) Our proposed loop-centric HPCs profiling can minimize the measurement errors due to interrupt skid and time-division multiplexing. (2) Our selected hardware events are consistent across different CPU architectures and OSs. (3) LoopHPCs outperforms the peer tools based on Pin [59] in anti-evasion effects, unpacking success rate, and performance. (4) We report the results of applying LoopHPCs to large-scale packed malware in the wild. Our experimental platforms have three recent Intel CPU architectures released in 2018∼2020: Kaby Lake, Coffee Lake, and Comet Lake.

6.1 Resilience to Non-determinism

The development of LoopHPCs has adopted Das et al.'s solutions [22] to remedy many measurement errors, such as per-process filtering and adjusting HPCs data when context switches or page faults occur. Besides, we only enable HPCs on a single core to avoid contamination from other cores. We focus on evaluating another two non-determinism sources that are not covered by Das et al.'s work.

Interrupt Skid Experiments We compare our loop-centric profiling with traditional event-based sampling using the loop instruction benchmarks from the SoK paper's dataset [22]. Each loop benchmark is a small code snippet to execute different string operations (e.g., lodsb, stosb, and movsb) for one million times. Table 4 shows interrupt skid evaluation results on top of the Kaby Lake architecture. Column 2 and Column 3 list benchmark names and hardware events affected by interrupt skid, respectively. For each affected event, we report its expected values, observed values, and skid ratio under two different profiling methods. It is clear that the skid ratio data of traditional event-based sampling are consistently high (with a peak value at 85.2%), while in all cases, our loop-centric profiling only introduces a skid ratio at 0.01%, which means only a tiny number of events skidding out of the loop body.

Multiplexing Experiments Next, we evaluate the measurement errors caused by our proposed iteration-division multiplexing (*Iter-MLPX*) versus traditional time-division multi-

Table 4: Interrupt skid evaluation results on top of the CPU architecture Kaby Lake.

Architecture	Benchmarks	Event	Event-based Sampling			Loop-centric Profiling		
			Expected Values	Observed Values	Skid Ratio[1]	Expected Value	Observed Value	Skid Ratio
Kaby Lake	loop stosb	store	1,000,000	147,700	85.2%	1,000,000	999,900	0.01%
	loop lodsb	load	1,000,000	389,836	61.0%	1,000,000	999,897	0.01%
	loop movsb	load	1,000,000	220,904	77.9%	1,000,000	999,902	0.01%
		store	1,000,000	358,791	64.1%	1,000,000	999,904	0.01%
	loop stosw	store	1,000,000	161,538	83.8%	1,000,000	999,894	0.01%
	loop lodsw	load	1,000,000	391,492	60.9%	1,000,000	999,905	0.01%
	loop movsw	load	1,000,000	259,595	74.0%	1,000,000	999,910	0.01%
		store	1,000,000	358,787	64.1%	1,000,000	999,909	0.01%

[1] Skid-Ratio=(Expected Values-Observed Values)/Expected Values

plexing (*Time-MLPX*). However, it is a fundamental challenge to measure multiplexing errors quantitatively due to inherent variations. Lv et al. [27] propose an error rate calculation method to roughly quantify how close the HPCs data collected by multiplexing are to the ideal data, which are obtained by the one-counter-one-event sampling style. Lv et al. compute the difference between two series of HPCs data via the dynamic time warping algorithm [60]. We use the same error rate calculation method in our experiments. Please refer to Appendix B for detailed calculation steps. Regarding testing samples, we use WannaCry [46] as the malware payload and pack it using 29 off-the-shelf binary packers from the CCS'18 paper [37]. We enable the multiplexing of 37 candidate hardware events listed in Appendix C when the packed WannaCry starts running, and then we terminate the multiplexing when the packed malware payload is recovered.

Figure 10 presents the error rate caused by multiplexing on Kaby Lake. Compared to Time-MLPX (blue bars), our Iter-MLPX (red bars) reduces the measurement errors significantly in all cases. Iter-MLPX reduces the average HPCs errors by as much as 17x (i.e., from 46.2% to 2.6%).

6.2 Consistency Across Multiple Platforms

In §4.3, we discussed how to select six significant events (shown in Table 3) using iteration-division multiplexing coupled with Principal Component Analysis. In this subsection, we evaluate whether they are consistent across different CPU architectures and OSs.

Across Architectures We test 29 off-the-shelf packers [37] on three Intel CPU architectures, and the malware payload is WannaCry. The running OS is fixed on Windows 10. We take the Kaby Lake architecture as a baseline. For both Coffee Lake and Comet Lake, we report their relative deviations compared to Kaby Lake. The results are shown in Table 5. As we have 29 packed samples to be reported, we only report minimum value and maximum value for simplicity purpose. Taking the store event as an example, the relative deviations of Coffee Lake compared to Kaby Lake range from -1.1% to 0.8%. We find that all relative deviations reveal a small fluctuation, indicating that the six selected hardware events remain relatively consistent across architectures.

Table 5: HPCs features across architectures. We use Kaby Lake as a baseline. For both Coffee Lake and Comet Lake, we report their relative deviations compared to Kaby Lake. The minimum value is -1.3%, and the maximum value is 4.8%.

Event	Relative Deviations	
	Coffee Lake	Comet Lake
STORE	-1.1%∼0.8%	-1.2%∼0.8%
LOAD	-1.1%∼1.0%	-1.3%∼1.1%
L1_HIT	-1.1%∼1.0%	-1.4%∼1.0%
L3_HIT	-1.2%∼3.4%	-0.1%∼**4.8%**
L1_MISS	**-1.3%**∼4.0%	-2.0%∼0.2%
BRANCHES	-0.8%∼1.7%	-0.6%∼0.4%

Across OSs Since UPX packer supports both Windows and Linux OS [61], we use UPX to pack six same-size executables on both Windows and Linux. These file sizes are 40KB, 80KB, 160KB, 320KB, 640KB, and 1028KB respectively. The underlying CPU architecture is fixed on Kaby Lake. We take the Windows OS as a baseline and report their relative deviations of Linux compared to Windows. We also report the minimum value and maximum value for packed samples. The results are shown in Table 6. The small deviation data demonstrate that the six selected hardware events are also consistent across OSs.

Table 6: HPCs features across OSs. We report the relative deviations of Linux OS compared to Windows OS. The minimum value is -2.3%, and the maximum value is 1.3%.

Event	Relative Deviations	Event	Relative Deviations
STORE	-0.1%∼0.1%	LOAD	-0.1%∼0.1%
L1_HIT	-0.1%∼0.8%	L3_HIT	-1.0%∼**1.3%**
L1_MISS	**-2.3%**∼1.1%	BRANCHES	-0.3%∼0.2%

6.3 Effectiveness against Binary Packers

In this subsection, we focus on evaluating our hardware-assisted unpacking technique from multiple dimensions with

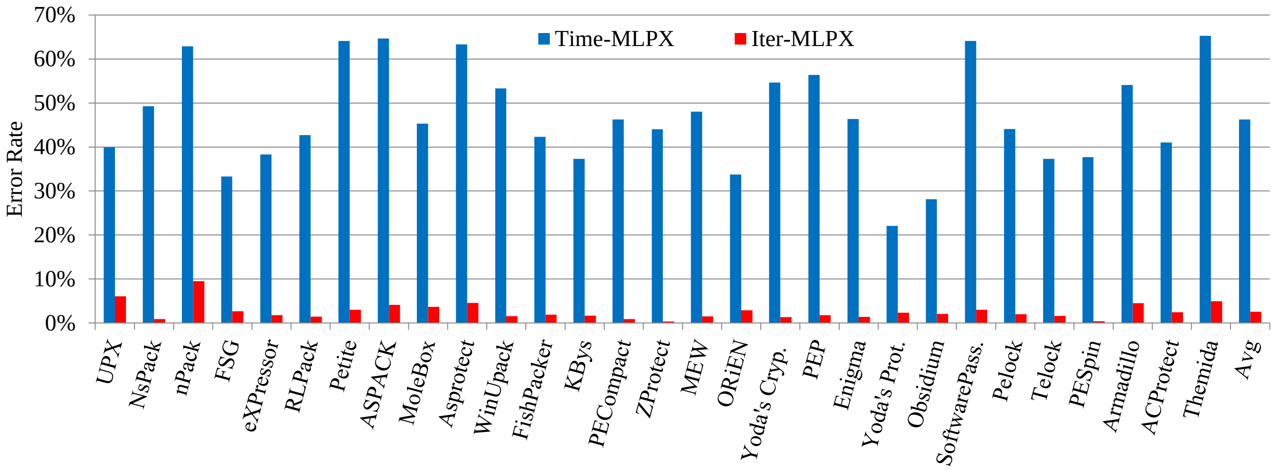


Figure 10: The measurement errors caused by multiplexing on top of the CPU architecture Kaby Lake.

Table 7: Selection of packers. A low entropy value means it is less than 7.0 [41].

ID	OS	Entropy	Source	#Number
(1)	Windows	High	CCS'18 [37]	29 packers
(2)	Linux	High	Stack Exchange [62]	12 packers[1]
(3)	Windows	Low	NDSS'20 [41]	6 packers

[1] UPX, Burneye, midgetpack, Shiva, cryptelf, ELFcrypt, ELF-Packer, pocrypt, oplzkwp, ps2-packer, elfuck, ELF-Encrypter.

Table 8: Distribution of packed samples.

Dataset ID	OS	Entropy	#Number
(1)	Windows	High	12,683
(2)	Linux	High	2,169
(3)	Windows	Low	2,433

ground-truth datasets, so that we can accurately verify the unpacking results. We first present how to generate representative ground-truth datasets, which cover Windows, Linux, and low-entropy packers.

6.3.1 Datasets

As shown in Table 7, our selection of packers covers three sources: (1) 29 off-the-shelf Windows packers collected by the CCS'18 paper [37]; (2) 12 Linux packers listed by a Stack Exchange [62] post; (3) 6 custom low-entropy packers on Windows. Regarding the last group of low-entropy packers, although Mantovani et al. [41] publish hash values for many low-entropy packed malware samples, we do not take them as the ground-truth dataset because their original unpacked executables are not available. Instead, we customize three open-source Windows packers (UPX, Yoda's Crypter, and Yoda's Protector) using two low-entropy packing schemes discussed by this work [41] ("Byte Padding" and "Encoding"). Therefore, we obtain six low-entropy packers on Windows.

We randomly selected malware samples among those submitted to VirusTotal [63] from 2019 to 2021. In addition, we only downloaded malware samples labeled as malicious by more than 30 antivirus engines. To collect packed malware, we use three popular packer detection tools together (PEiD [64], Exeinfo PE [65], and Detect It Easy [66]): we detect a packed sample if any tool labels it as packed. After that, our non-packed ones include 586 Windows and 234 Linux malware samples. And then, we pack these non-packed samples with Windows/Linux packers shown in Table 7. Besides, we remove non-executable samples after packing. Eventually, we generate three packed malware datasets as our ground truth datasets (shown in Table 8). Since we have non-packed versions for these datasets, we can accurately evaluate the unpacking results in follow-up experiments.

6.3.2 Results of Machine Learning Classifiers

In §5.3, the layer cost data are passed to machine learning (ML) classifiers to detect the layer containing the original code. We conduct three experiments to evaluate five ML classifiers: Decision Tree (DT), Random Forest (RF), K-Nearest Neighbors (KNN), AdaBoost (AB), and Naive Bayes (NB). As shown in Table 9, for each experiment, we report original code detection rates using four metrics.

First, we use Dataset (1) in Table 8 to evaluate the unpacking results of LoopHPCs for Windows high-entropy packed samples. We perform ten-fold cross-validations 1,000 times on these samples. To this end, the original samples are randomly divided into ten equally-sized subsets; nine subsets are used as the training set, and the remaining one is used as the testing set. The detection rates of the original code are shown in Column 2∼5 of Table 9. The detection rates of DT, RF, and KNN are higher than AB and NN. This is because DT, RF, and KNN are designed to classify outliers, while AB and NN are designed to classify clusters of examples. Overall, LoopHPCs achieves *zero* false positives, but it reveals small

Table 9: Detection rates of the original code. The classifiers' names are Decision Tree (DT), Random Forest (RF), K-Nearest Neighbors (KNN), AdaBoost (AB), and Naive Bayes (NB).

Classifiers	Experiment (1): Cross-Validations in Windows				Experiment (2): Windows–>Linux				Experiment (3): High-Entropy->Low-Entropy			
	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score
DT	98.8%	100.0%	97.2%	98.6%	99.2%	100.0%	98.0%	99.0%	98.9%	100.0%	97.5%	98.7%
RF	99.2%	100.0%	98.1%	99.0%	99.4%	100.0%	98.7%	99.3%	99.3%	100.0%	98.3%	99.1%
KNN	98.5%	100.0%	96.4%	98.2%	98.9%	100.0%	97.5%	98.7%	98.6%	100.0%	96.8%	98.4%
AB	94.3%	100.0%	87.4%	93.3%	95.9%	100.0%	90.9%	95.2%	94.8%	100.0%	88.5%	93.9%
NB	96.1%	100.0%	91.1%	95.3%	97.2%	100.0%	93.6%	96.7%	96.4%	100.0%	91.9%	95.8%

false negatives; that is, it fails to detect the original code for about 1.7% of packed samples. Upon further investigation, we find out that each of them has a relatively small original executable file size (less than 11.6KB), resulting in indistinctive hot loop features.

In the second experiment, we use Dataset (1) in Table 8 as training set, and Dataset (2) in Table 8 as testing set. The detection rates in Column 6∼9 of Table 9 indicate that the classifiers trained from Windows packers are also effective for Linux packers. As we have demonstrated in §6.2, unpacking-related hardware events are consistent across OSs.

In the third experiment, we use Dataset (1) in Table 8 as training set, and Dataset (3) in Table 8 as testing set. The detection rates in Column 10∼13 of Table 9 are also encouraging, indicating that the ML classifiers trained from high-entropy packers can also work on low-entropy packers. Although low-entropy packers have become an emerging threat to static packer detection [41], they still reveal hot loop features at runtime.

As the trained ML classifier using Random Forest achieves the best result in Table 9, we will use it in our comparative and large-scale experiments (§6.3.3 & §6.4).

6.3.3 Comparative Evaluation of Binary Unpacking

We conduct a separate experiment to compare LoopHPCs with existing generic unpacking methods. We use CAPE sandbox [67] and a typical debugger (OllyDbg) to represent two "wait-and-dump" approaches. For CAPE sandbox, we configure it to *wait* the sample calls the API "ExitProcess", and then *dump* the sample. For OllyDbg, we collect unpacking scripts from multiple sources [68–72]. We *wait* for the unpacking script to finish executing, and then *dump* the sample. In addition to these two "wait-and-dump" approaches, we also select another DBI-based unpacking tool: Arancino [36]. Arancino relies on Intel Pin [59] to trace "written-then-executed" layers.

Table 10 shows comparative evaluation results with 29 off-the-shelf Windows packers, and the malware payload is WannaCry. Since we have the ground truth of these packed samples, we compare the first basic block of unpacked results with the original WannaCry. The data in Column 2∼5 indicate that only LoopHPCs can succeed to unpack in all cases. In contrast, sandbox failed in 9 cases, debugger failed in 14 cases, and DBI failed in 8 cases. We investigated these failed

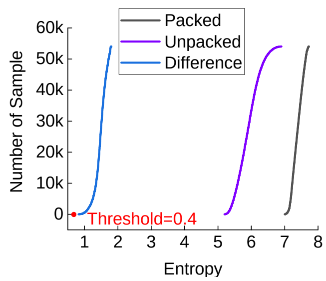
Table 10: Comparative evaluation with ground truth dataset.

Packed Samples	Wait-and-Dump		DBI	LoopHPCs
	Sandbox	Debugger		
UPX	✓	✓	✓	✓
NsPack	✓	✓	✓	✓
nPack	✓	✓	✓	✓
FSG	✓	✓	✓	✓
eXPressor	✓	✓	✓	✓
RLPack			✓	✓
Petite	✓	✓	✓	✓
Aspack	✓	✓	✓	✓
MoleBox	✓	✓	✓	✓
Asprotect			✓	✓
FishPacker	✓	✓	✓	✓
KBys	✓	✓	✓	✓
PECompact	✓	✓	✓	✓
Yoda's Crypter	✓		✓	✓
Yoda's Protector			✓	✓
MEW	✓	✓	✓	✓
ORiEN	✓	✓		✓
PEP	✓		✓	✓
Pelock	✓		✓	✓
Telock	✓			✓
Pespin	✓			✓
Armadillo				✓
ACProtect	✓			✓
Enigma				✓
ZProtect		✓	✓	✓
Obsidium			✓	✓
SoftwareProtect				✓
Themida				✓

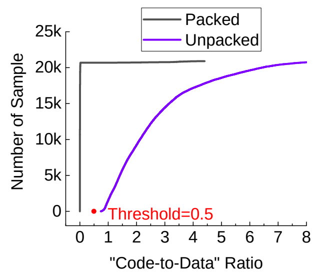
cases and summarized three reasons: (1) the sample crashed at execution time; (2) the sample fingerprinted the unpacking environment, and then it terminated the process of unpacking the original code; (3) the unpacking tool mistook the layer of unpacking routine as the layer of original code.

6.4 Packed Malware In the Wild

We randomly selected malware samples among those submitted to VirusTotal [63] between 2019 to 2021. We only downloaded malware samples labeled as malicious by more than 30 antivirus engines. In addition, we use three popular packer detection tools together (PEiD [64], Exeinfo PE [65], and Detect It Easy [66]) to detect packed samples. Eventually, we collected 74,938 packed malware samples. 83.1% of them are Windows packed malware, and the remaining ones are Linux packed malware. We also calculate their entropy distribution—27.9% of packed samples reveal low-entropy values.



(a) High-entropy packers



(b) Low-entropy packers

Figure 11: The unpacking result of packers in the wild (CDF).

For 96.5% of packed malware, LoopHPCs can deliver the original payload within 300 seconds. The failure cases are caused by two reasons: (1) 2.6% of packed malware samples are not executable; (2) 0.9% of them do not reveal significant hot-loop features due to the small size of the original payload. A challenge of this experiment is that we do not have non-packed versions. Therefore, we adopt two statistic heuristics from the previous work [37,73,74] to roughly verify LoopHPCs's outputs: entropy difference and "code-to-data" ratio. The entropy difference means the difference value of entropy before/after unpacking, and the "code-to-data" ratio means the ratio of code size to data size in the binary. Empirically, an entropy difference value ≥ 0.4 or a code-to-data ratio ≥ 0.5 is the threshold to determine whether the unpacking is successful [37,74]. For high-entropy packed samples, we measure their entropy differences; for low-entropy packers, we switch to the code-to-data ratio. Figure 11 shows that all entropy differences are beyond the threshold of 0.4, and all "code-to-data" ratios of unpacked samples are also above the threshold of 0.5.

7 Discussion

This section discusses possible attacks to LoopHPCs, and LoopHPCs's limitations.

7.1 Possible Attacks and Countermeasures

Detection Attack One possible detection attack is to hack into the OS kernel to detect the presence of LoopHPCs. Countering privilege escalation is out of the scope of the proposed solution. Another possible attack is to fingerprint the driver name of LoopHPCs at the user level. Our countermeasure is to randomize the driver name of LoopHPCs.

Time-based Attack As LoopHPCs introduces runtime overhead to the unpacking process, another indirect way is to detect timing discrepancies. User-level programs have multiple options to obtain time information [75], either via API calls (e.g., "GetSystemTime") or specific instructions (e.g., `rdtsc`). Our countermeasure is to add kernel modules to intercept these querying methods and return those queries with expected time values. However, a skilled attacker can inquire time from an external resource through the network [76]. How to prevent time-based attack with external resources is still an open problem [76].

Mimicry Attack Like the classical mimicry attack [77], whether malware authors can craft instruction sequences to exert some influence over hardware events? Tang et al. [10] study various mimicry attacks (e.g., padding, substitution, and grafting), and their impact on their HPCs measurement. As proposed by Tang et al. [10], we can increase the number of selected hardware events and randomize them to increase the difficulty of mimicry-attack.

Hiding Hot Loop Is it possible to evade our loop-centric profiling by hiding hot-loop structures? In §4.1, we have dealt with a possible evasion by implementing loops via recursion. Next, we discuss another two common loop transformations. The **first** strategy, loop unrolling, is typically performed by the compiler optimization to minimize branch penalties at the expense of bloating the program's size [78]. Considering the large scale of hot-loop iterations in most packed samples, if attackers impose loop unrolling on unpacking algorithms, the repeated loop body will expand the code size of packed malware with several orders of magnitude. The **second** strategy, loop splitting, simplifies a loop by breaking it into multiple smaller loops. However, loop splitting is not a trivial task because it needs to resolve complex memory dependencies in a loop [79, 80]. This attack can scatter the significant HPCs values of a hot loop into multiple small loops. Actually, we have considered such a case in LoopHPCs' design. Recall that if multiple loops co-generate a new layer, we sum up all of the related loop-centric HPCs values as the layer cost (see Definition 5). This loop splitting transformation can reduce the HPCs values of a single loop, but it does not affect the layer cost. Therefore, our design is immune to this attack. The **third** strategy: the attackers use OS-provided crypto APIs or CPU-provided crypto instructions to hide the loop structure. For the OS-provided crypto APIs, the crypto instructions are implemented in the Windows DLL. For this case, LoopHPCs can profile loops in the DLL to resist this attack. However, LoopHPCs cannot handle CPU-provided crypto instructions (e.g., Intel Advanced Encryption Standard New Instructions).

Fake Hot Loop Another attack is to intentionally generate additional hardware event in the not-so-hot loop to make it "hot". We call this attack as "fake hot loop." We note that another unpacking tool, BinUnpack [37], suffers from a similar attack (called fake API calls) as well. Inspired by BinUnpack's countermeasure, we can use OEP search heuristic to rule out the fake hot loop because the OEP does not appear after the execution of the fake hot loop.

7.2 Limitations

First, LoopHPCs does not support AMD processors because they do not have an equivalent hardware tracing mechanism like LBR [81]. As LoopHPCs is designed to assist security analysts in malware analysis, running it only on Intel processors is not a fundamental limitation. **Second**, LoopHPCs is sensitive to the file size of the original program. Our evaluation shows that when the size of the original program is less than 11.6KB, LoopHPCs may miss the unpacking layer containing the original code. One possible solution is to perform rank-preserving power transform [10] to positively scale the collected HPCs values. We leave it as our future work. **Third**, LoopHPCs runs in bare-metal machines, and therefore LoopHPCs can't be integrated into VM-based analysis. **Forth**, given the limited size of ground-truth datasets, the ML engines may be subject to overfitting.

8 Conclusion & Future Work

Using HPCs for security is recently a controversial topic. Without taming the non-determinism nature of hardware events, the claimed security gains can be compromised. In this paper, we study this challenge and propose a hardware-assisted, loop-centric profiling technique to minimize HPCs measurement imprecisions. We demonstrate its benefits (e.g., transparent and across-platforms) in malware unpacking.

As a frequently-adopted approach to achieve better performance [82], we will study how to implement LoopHPCs in a dedicated circuit (e.g., FPGA) in the future.

Acknowledgments

We sincerely thank Usenix Security 2023 anonymous reviewers for their insightful and helpful comments. This work was supported by the National Science Foundation (NSF) under grant CNS-1850434 and CNS-2128703. This work was also supported in part by the National Natural Science Foundation of China grants (62172144, 61972297, 61976085, and 61902190), Natural Science Foundation of Hubei Province of China grant (2022CFB510), Education Department of Hubei Province in China (D20212501, B2021578).

References

[1] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC'96)*, 1996.

[2] Shirley Victoria Browne, Jack Dongarra, N Garner, Kevin S London, and Philip J Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC'00)*, 2000.

[3] James Reinders. *VTune Performance Analyzer Essentials—Measurement and Tuning Techniques for Software Developers*. Intel Press, 2005.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*, 1997.

[5] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is More. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, 2000.

[6] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[7] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. Happer: Unpacking Android Apps via a Hardware-Assisted Approach. In *Proceedings of the 42nd IEEE Symposium on Security & Privacy (S&P'21)*, 2021.

[8] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security Breaches As PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11)*, 2011.

[9] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Proceedings of the 42th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[10] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, 2014.

[11] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the 18th International Symposium on Recent Advances in Intrusion Detection (RAID'15)*, 2015.

[12] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, 2013.

[13] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17)*, 2017.

[14] Hossein Sayadi, Nisarg Patel, Sai Manoj PD, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification. In *Proceedings of 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[15] Kanad Basu, Prashanth Krishnamurthy, Farshad Khorrami, and Ramesh Karri. A Theoretical Study of Hardware Performance Counters-Based Malware Detection. *IEEE Transactions on Information Forensics and Security*, 15, 2020.

[16] Sai Praveen Kadiyala, Pranav Jadhav, Siew-Kei Lam, and Thambipillai Srikanthan. Hardware Performance Counter-Based Fine-Grained Malware Detection. *ACM Transactions on Embedded Computing Systems*, 19(5), 2020.

[17] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Recent Advances in Intrusion Detection (RAID'15)*, 2015.

[18] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding Contention-Based Channels and Using Them for Defense. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA'15)*, 2015.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'16)*, 2016.

[20] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'16)*, 2016.

[21] Vincent M Weaver and Sally A McKee. Can Hardware Performance Counters be Trusted? In *Proceedings of 4th IEEE International Symposium on Workload Characterization*, 2008.

[22] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[23] Vincent M Weaver and Jack Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? In *Proceedings of 3rd Workshop on Functionality of Hardware Performance Monitoring*, 2010.

[24] Intel. Intel64 and IA-32 Architectures Software Developer's Manual Volume 3. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, 2021.

[25] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So Many Performance Events, So Little Time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'16)*, 2016.

[26] Todd Mytkowicz, Peter F Sweeney, Matthias Hauswirth, and Amer Diwan. Time Interpolation: So Many Metrics, So Few Registers. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*, 2007.

[27] Yirong Lv, Bin Sun, Qingyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. CounterMiner: Mining Big Performance Data From Hardware Counters. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, 2018.

[28] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A Close Look at a Daily Dataset of Malware Samples. *ACM Transactions on Privacy and Security*, 22(1), 2019.

[29] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[30] Robert Lyda and James Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy*, 5(2), 2007.

[31] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing Packed Malware. *IEEE Security and Privacy*, 6(5), 2008.

[32] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5), 2011.

[33] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, 2008.

[34] Kevin A. Roundy and Barton P. Miller. Hybrid Analysis and Control of Malware. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*, 2010.

[35] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[36] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17)*, 2017.

[37] Binlin Cheng, Jiang Ming, Jianming Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[38] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[39] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P'15)*, 2015.

[40] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security*, 15, 2020.

[41] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[42] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*, 2021.

[43] Marcus Botacin, Paulo Lício De Geus, and André Grégio. Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):1–30, 2018.

[44] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the Precision of Precise Event Based Sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*, 2020.

[45] Subho S Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K Iyer. BayesPerf: Minimizing Performance Monitoring Errors Using Bayesian Statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[46] Matt Suiche. WannaCry—The Largest Ransomware Infection in History. https://www.raconteur.net/infographics/wannacry-the-biggest-ransomware-attack-in-history/, May 2017.

[47] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware Performance Counters Can Detect Malware: Myth or Fact? In *Proceedings of the 13th ACM Symposium on Asia Conference on Computer and Communications Security (ASIACCS'18)*, 2018.

[48] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proceedings of the 33th IEEE symposium on security and privacy (S&P'12)*, 2012.

[49] Hervé Abdi and Lynne J. Williams. Principal Component Analysis. *WIREs Computational Statistics*, 2(4):433–459, 2010.

[50] Jordi Tubella and Antonio Gonzalez. Control Speculation in Multithreaded Processors Through Dynamic Loop Detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*, 1998.

[51] Tipp Moseley, Dirk Grunwald, Daniel A Connors, Ram Ramanujam, Vasanth Tovinkere, and Ramesh Peri. LoopProf: Dynamic Techniques for Loop Detection and Profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA'06)*, 2006.

[52] Tipp Moseley, Daniel A Connors, Dirk Grunwald, and Ramesh Peri. Identifying Potential Parallelism via Loop-Centric Profiling. In *Proceedings of the 4th International Conference on Computing Frontiers (CF'07)*, 2007.

[53] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[54] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS'19)*, 2019.

[55] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[56] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85), 2011.

[57] Botacin, Marcus and Zanata, Marco and Grégio, André. The Self Modifying Code (SMC)-Aware Processor (SAP): A Security Look on Architectural Impact and Support. *Journal of Computer Virology and Hacking Techniques*, pages 1–12, 2020.

[58] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[59] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*. ACM, 2005.

[60] Donald J Berndt and James Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the AAAI-94 Workshop on Knowledge Discovery in Databases (KDD)*, 1994.

[61] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. UPX The Ultimate Packer for eXecutables). https://github.com/upx/upx, 2021.

[62] StackExchange. Packers/Protectors for Linux. https://reverseengineering.stackexchange.com/questions/3184/packers-protectors-for-linux, 2020.

[63] VirusTotal. VT Intelligence: Combine Google and Facebook and apply it to the field of Malware. https://www.virustotal.com/gui/intelligence-overview, [online].

[64] Aldeid. PEiD. https://www.aldeid.com/wiki/PEiD, 2022.

[65] A.S.L. EXEINFO PE. http://exeinfo.atwebpages.com/, 2022.

[66] Horsicq. Detect It Easy. https://github.com/horsicq/Detect-It-Easy, 2022.

[67] kevoreilly. CAPE: Malware Configuration And Payload Extraction. https://github.com/kevoreilly/CAPEv2/, 2022.

[68] ThomasThelen. OllyDbg Script. https://github.com/ThomasThelen/OllyDbg-Scripts, 2018.

[69] Zhengyan Gao. OllyDbg Unpacking Script. https://github.com/dubuqingfeng/ollydbg-script, 2016.

[70] OpenRCE. OllyDbg OllyScripts. http://www.openrce.org/downloads/, 2007.

[71] fen_blue. HA_OllyDBG1. https://download.csdn.net/download/cherishlive/5803815, 2013.

[72] Douglas Poerschke Rocha. scripts-ollydbg. https://github.com/poerschke/scripts-ollydbg, 2022.

[73] Sebastiano Mariani, Lorenzo Fontana, Fabio Gritti, and Stefano D'Alessio. PinDemonium: A DBI-Based Generic Unpacker for Windows Executable. Black Hat USA, 2016.

[74] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*, 2008.

[75] Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bülent Yener. AVLeak: Fingerprinting Antivirus Emulators Through Black-Box Testing. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT'16)*, 2016.

[76] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Computing Surveys (CSUR)*, 2019.

[77] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, 2002.

[78] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[79] Junyi Liu, John Wickerson, and George A. Constantinides. Loop Splitting for Efficient Pipelining in High-Level Synthesis. In *Proceedings of the 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*, 2016.

[80] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Using Loops for Malware Classification Resilient to Feature-Unaware Perturbations. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*, 2018.

[81] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'21)*, 2021.

[82] Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou, Ajay Joshi, and Manuel Egele. Efficient Context-Sensitive CFI Enforcement Through a Hardware Monitor. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'20)*, 2020.

[83] Intel Corparation. Intel ((R)) 64 and IA-32 Architectures Software Developer's Manual. *Combined Volumes, Dec*, 2016.

Appendix

A Hot Loop Performance Data

Table A1: The performance data of hot loops in unpacking routine of various off-the-shelf packers. The malware payload is WannaCry.

Sample	# of Max Iterations	Inst (%)	Cycle (%)
UPX	3,359,627	99%	99%
NsPack	3,362,465	91%	94%
nPack	3,308,595	88%	99%
FSG	3,342,276	99%	99%
eXPressor	3,360,712	88%	94%
RLPack	3,359,695	91%	99%
Petite	3,359,263	95%	99%
Aspack	3,359,692	97%	98%
MoleBox	3,348,356	92%	96%
Asprotect	3,351,325	92%	96%
WinUpack	3,358,472	98%	99%
FishPacker	3,358,372	93%	94%
KBys	3,359,171	96%	98%
PECompact	3,359,627	97%	99%
ZProtect	2,006,224	95%	96%
MEW	3,361,243	95%	97%
ORiEN	983,804	92%	95%
Yoda's Crypter	3,360,253	96%	99%
PEP	993,346	91%	95%
Enigma	432,110	95%	98%
Yoda's Protector	920,837	97%	98%
Obsidium	698,330	90%	94%
SoftwarePassport	3,145,470	91%	93%
Pelock	3,451,282	97%	98%
Telock	945,821	99%	98%
Pespin	418,183	92%	97%
Armadillo	3,361,391	93%	97%
ACProtect	918,139	92%	99%

B Multiplexing Error Rate Calculation

CounterMiner [27] calculates the measurement errors of multiplexing in three steps:

(1) It runs each sample two times to measure HPCs values via the one-counter-one-event (OCOE) sampling style, which generates two series of HPCs values (S_{ocoe1}, S_{ocoe2}). Then, it calculates the dynamic time warping between S_{ocoe1} and S_{ocoe2}, denoted as $dist_{ref}$.
(2) It runs the samples in the third time to measure their HPCs values via multiplexing, generating a third series of HPCs values (S_{mlpx}). CounterMiner computes the dynamic time warping between one time series collected by MLPX (S_{mlpx}) and one by OCOE (S_{ocoe1} or S_{ocoe2}), represented by $dist_{mea}$.
(3) At last, CounterMiner defines the error rate caused by multiplexing as follows:

$$Error\ Rate = |1 - \frac{dist_{ref}}{dist_{mea}}| \times 100\% \qquad (1)$$

The value of error rate ranges from 0.0% to 100.0%. Theoretically, $dist_{mea}$ is larger than $dist_{ref}$ due to inherent variations of multiplexing. Ideally, we hope the $dist_{mea}$ is close to $dist_{ref}$, so the error rate is also close to 0.0%.

C Candidate Hardware Events

Table A2: Candidate hardware events

	Hardware Event	Description
1	INST_RETIRED	All instructions retired.
2	CYCLES	Number of CPU cycles.
3	LOAD	All load instructions retired.
4	STORE	All store instructions retired.
5	BRANCH	All branch instructions retired.
6	BR_C	Conditional branch instructions retired.
7	BR_NOT_TAKEN	Not taken branch instructions retired.
8	BR_NEAR_TAKEN	All near taken branch instructions.
9	FAR_BRACHES	All far branch instructions retired.
10	NEAR_CALL	All near call instructions retired.
11	CALL_D	Direct near call instructions retired.
12	CALL_ID	Indirect near call instructions retired.
13	NEAR_RET	All near ret instructions retired.
14	MISP_ BR	Mispredicted branch instructions.
15	MISP_BR_C	Mispredicted conditional branch.
16	MISP_CALL	Mispredicted near call instructions.
17	MISP_RET	Mispredicted near return instructions.
18	MISP_NEAR_TAKEN	Mispredicted near taken branch instructions.
19	L1_HIT	Retired load instructions that hit L1 cache.
20	L2_HIT	Retired load instructions that hit L2 cache.
21	L3_HIT	Retired load instructions that hit L3 cache.
22	L1_MISS	Retired load instructions that missed L1 cache.
23	L2_MISS	Retired load instructions that missed L2 cache.
24	L3_MISS	Retired load instructions that missed L3 cache.
25	LLC_MISS	Last level cache misses.
26	LLC_HIT	Last level cache hits.
27	ICACHE_MISS	Instruction cache misses.
28	MIS_ITLB	I-TLB misses.
29	MIS_STLB	STLB (2nd level TLB) misses.
30	DTLBL	D-TLB load hits.
31	DTLBS	D-TLB store hits.
32	MIS_DTLB_LOAD	D-TLB load misses.
33	MIS_DTLB_STORE	D-TLB store misses.
34	STLB_HIT	Shared-TLB hits after i-TLB misses.
35	MIS_STLB_LOAD	STLB load misses.
36	MIS_STLB_STORE	STLB store misses.
37	SMC.MACHINE_CLEAR	Self-modifying code, causing the entire pipeline of the machine and the trace cache to be cleared.

D Collecting HPCs in Different OSs

The key task of LoopHPCs is to collect HPCs values in different OSs. This task includes two steps: registering PMI handle and implementing PMI handle.

Registering PMI Handle Like Das et al.'s work [22], we collect HPCs values using performance monitoring interrupt (PMI). The implementation of registering PMI handle differs in different OSs: (1) For Linux OSs, we register PMI handle via Interrupt Description Table (IDT) (see Figure A1). (2) For Windows OSs, we register PMI handle using `HalSetSystemInformation` function (see Figure A2).

```
//Create a gate descriptor for our ''PMI_Handle''.
pack_gate (&desc, GATE_INTERRUPT,PMI_Handle,0, 0, 0);
//Write the gate descriptor into the IDT.
write_idt_entry(idt, pebs_vector,&desc);
```

Figure A1: Registering PMI handle in Linux OSs.

```
//Register our ''PMI_Handle'' as the PMI handle.
HalSetSystemInformation (HalProfileSourceInterruptHandler,
                         sizeof(PVOID*),
                         &PMI_Handle);
```

Figure A2: Registering PMI handle in Windows OSs.

Implementing PMI Handle The implementation of PMI Handle is shown in Algorithm 1. This algorithm follows Intel manual's recommendation [83], which is independent of OSs.

Algorithm 1 Algorithm of PMI Handle

```
function PMI_HANDLE
    Disabled counters.
    Disabled PEBS.
    Check overflow conditions.
    Read the HPCs values from the register.
    Reset the DS area.
    Enable PEBS.
    Enable counters.
end function
```