Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

A Machine Learning Methodology based on Performance Monitoring Counters for Process Classification

Advisor:
Prof. Cristina Silvano

Co-Advisor:
Prof. Alessandro Barenghi

Master thesis of:
Andrea Brunato

Student ID:
851413

Academic Year 2016–2017

These Riots Are Just The Beginning...

Abstract

Most modern processors include hardware performance counters: architectural registers that allow engineers to monitor a process execution with high accuracy.
Data collected adopting these tools allow an in depth analysis of a process architectural performance while keeping into account the complexity of modern CPUs, that use hierarchical caches, simultaneous multi-threading and out-of-order execution.
By analyzing counter-based information we can inspect a process execution's impact over the whole CPU architecture as well as over the single functional unit.
However performance monitoring counters can be used for purposes beyond the simple performance analysis of a process. The main goal of this thesis is to classify different processes according to their runtime behaviour measured by Performance Monitoring Counters.
In this thesis we present a new approach for creating a process signature, which takes into account the whole execution of the target task. This is different from state-of-the-art proposals which leverage those performance bound to the execution of a specific piece of code. In this thesis, we provide a broader solution for characterizing a generic process execution, without looking into its code and assuming the task as a black box.
In order to achieve this goal, we have modified the Linux kernel to provide an automatic mechanism for gathering performance data of a target process. The collected statistics are then elaborated to build up a process performance signature, which is used to train a support vector machine for classification purposes. The effectiveness of our approach is demonstrated by the high accuracy of the trained classification algorithm when tested on new performance data. The support vector machine is then inserted in the operating system, to provide in-kernel classification of generic processes running on the system.

Sommario

La maggior parte dei processori moderni include contatori hardware di performance (PMC): speciali registri che permettono di monitorare l'esecuzione di un processo con alta accuratezza. I dati raccolti utilizzando questi strumenti consentono di fare un'analisi approfondita delle prestazioni architetturali di un processo, tenendo conto della compessità delle moderne CPU, che usano gerarchie di memoria, multithreading simultaneo ed esecuzione fuori ordine. Analizzando queste informazioni possiamo ispezionare l'impatto dell'esecuzione di un processo sia sull'intera CPU sia sulla singola unità funzionale.
Tuttavia questi contatori hardware possono essere usati per scopi ben più complessi della semplice analisi di prestazioni. L'obbiettivo di questa tesi è quello di classificare diversi processi in base al loro comportamento dinamico, monitorato utilizzado i PMC. In questa tesi introduciamo un nuovo modo di generare una 'firma' di un processo attraverso le sue prestazioni, utilizzando come informazione l'intera esecuzione del processo. Questa idea è diversa rispetto allo stato dell'arte, che invece prende in considerazione solo quelle prestazioni legate all'esecuzione di una parte specifica del codice. In questa tesi forniamo una soluzione più generale per classificare l'esecuzione di un generico processo, senza studiare il suo codice e assumendo il processo come una 'scatola nera'.
Per raggiungere questo obbiettivo abbiamo modificato il kernel di Linux perchè possa fornire un meccanismo automatico per collezionare le prestazioni di un processo. Le statistiche così raccolte sono quindi elaborate per creare la firma del processo, che viene poi utilizzata per allenare una Support Vector Machine con obbiettivo la classificazione. La validità di quanto proposto trova riscontro nell'alta accuratezza del nostro algoritmo di classificazione quando viene testato con nuovi dati. La SVM viene infine inserita nel sistema operativo, perchè possa eseguire la classificazione all'interno del kernel per generici processi eseguiti nel sistema.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

BOB Branch Order Buffer

CISC Complex Instruction Set

ELF Extensible File Format

MSR Model Specific Register

MSROM Microdoce Store ROM

OS Operating System

PCA Principal Component Analysis

PMC Performance Monitoring Counter

RAT Register Alias Table

RCU Read-Copy-Update

RISC Reduced Instruction Set

ROB Reorder Buffer

ROP Return Oriented Programming

RS Reservation Station

SVD Singular Value Decomposition

SVM Support Vector Machine

VMM Virtual Machine Monitor

Chapter 1

Introduction

1.1 Introduction

Modern applications have to meet high quality standards, like being scalable, responsive and efficient. Any system which is not meeting these requirements is not competitive: for this reason, a special attention has to be paid to the performance of software products.
However measuring programs performance using traditional profiling tools in modern architectures is becoming more difficult. Since the introduction of hierarchical caches, simultaneous multi-threading and out-of-order execution, the processor's architecture complexity has meaningfully increased. Furthermore, the common performance bottleneck is moving from disks to the memory subsystem, caches, memory bus, and CPU interconnects: all architectural components which cannot be well described with common profilers. In order to provide a tool dealing with this complexity, modern processors have been equipped with special hardware counters whose solely purpose is to give the developers an insight on how well the application is performing on the computer architecture. These counters, known as *Performance Monitoring Counters* (PMC), allow specialized engineers to monitor applications and tune them accordingly. Providing a real world example, PMCs are used everyday by performance engineers working at Netflix, an entertainment company representing one of the largest source of Internet streaming traffic nowadays [6] [1]. While the practice of using PMC for performance analysis is well consolidated, several questions rise when thinking about their usage for other purposes:

- Do they leak too many details about a process execution?
- Can a malicious attacker use them to get secrets elaborated by the monitored task?
- Is it possible to use them as a tool to capture the runtime 'signature' of a generic process?

1.2 Problem and Contribution

Performance Monitoring Counters allow engineers to perform an in-depth performance analysis of a given application. However such information can be used for purposes beyond the simple performance analysis of a process. The state-of-the-art proposals leverage PMCs as an offensive or defensive tool. In the first case, PMCs extend the attack range of side channel attacks, allowing the attacker to gather low-level architectural statistics which can be analyzed to deduce a secret being elaborated by a target process. In the second scenario instead, PMC statistics are used to recognize the execution of a particular pattern of instructions, which is known to belong to a malicious behaviour. In this case, the state-of-the-art projects build up a performance signature of the target malware, which is used by the operating system to detect and consequently kill its execution at runtime.
The main goal of this thesis is to classify different processes according to their runtime behaviour, measured by using PMCs. In this thesis work, we present a novel approach for creating a process fingerprint, which takes into account the whole execution of the target task. This is different from state-of-the-art proposals which leverage those performance bound to the execution of a specific piece of code. In this thesis, we provide a broader solution for characterizing a generic process execution, without accessing its code and assuming the task as a black box.
In order to achieve this goal, we have modified the Linux kernel to provide an automatic mechanism for gathering performance data of a target process. The collected statistics are then elaborated to build up a performance signature, which is used to train a support vector machine for classification purposes. The effectiveness of our approach is proved by the high accuracy of the trained classification algorithm when tested on new performance data. The support vector machine is then inserted in the operating system, to provide in-kernel classification for generic processes running on the system.

1.3 Organization of the Thesis

This thesis is structured as follows: Chapter 2 provides the background about the main concepts used in this thesis. Chapter 3 introduces the state-of-the-art, describing how PMCs have been used so far for purposes beyond the simple performance analysis by researches. Chapter 4 describes the main goal of this thesis, discussing the advantages and limitations of our solution and gives a detailed description on how we implemented our work. Chapter 5 presents and discusses the experimental results of this thesis. Chapter 6 concludes this thesis, summarizing our solution and providing a list of possible future works.

Chapter 2

Background

2.1 Introduction

This Chapter describes all those fundamental concepts that have been used in this thesis. In Section 2.2, we provide a definition of the Intel Skylake architecture, with special focus on microcode decomposition and assists in Section 2.3. Performance Monitoring Counters are introduced in Section 2.4, with an overview of the functionalities they provide to gather performance data. We describe in detail the observable micro-architectural performance (called performance events) in Section 2.5, while in Section 2.6 we report a description on how PMC are implemented in hardware. In Section 2.7, we describe how a programmer can control PMCs with user-space programs, while in the Section 2.8 we describe its corresponding kernel implementation, with special focus on those functions we have modified. Some details about fixed point arithmetic, used to re-implement the support vector machine in-kernel, are then given in Section 2.9. In Section 2.10, we provide an overview about the ELF binary format, information that we have used in order to include a performance signature as a special section inside the program binary file.

2.2 Intel Skylake Micro-architecture

Motivation In this Section, we provide a description about Skylake, our target architecture. We discuss each functional unit in detail because PMCs monitor the performance of such micro-architectural components.

The Skylake architecture implements dynamic scheduling with an out-of-order, speculative execution featuring Hyper-threading, putting inside each physical core two different logical processors.
The pipeline can be broken down into 3 main areas: the front-end, the back-end or execution engine and the memory subsystem.

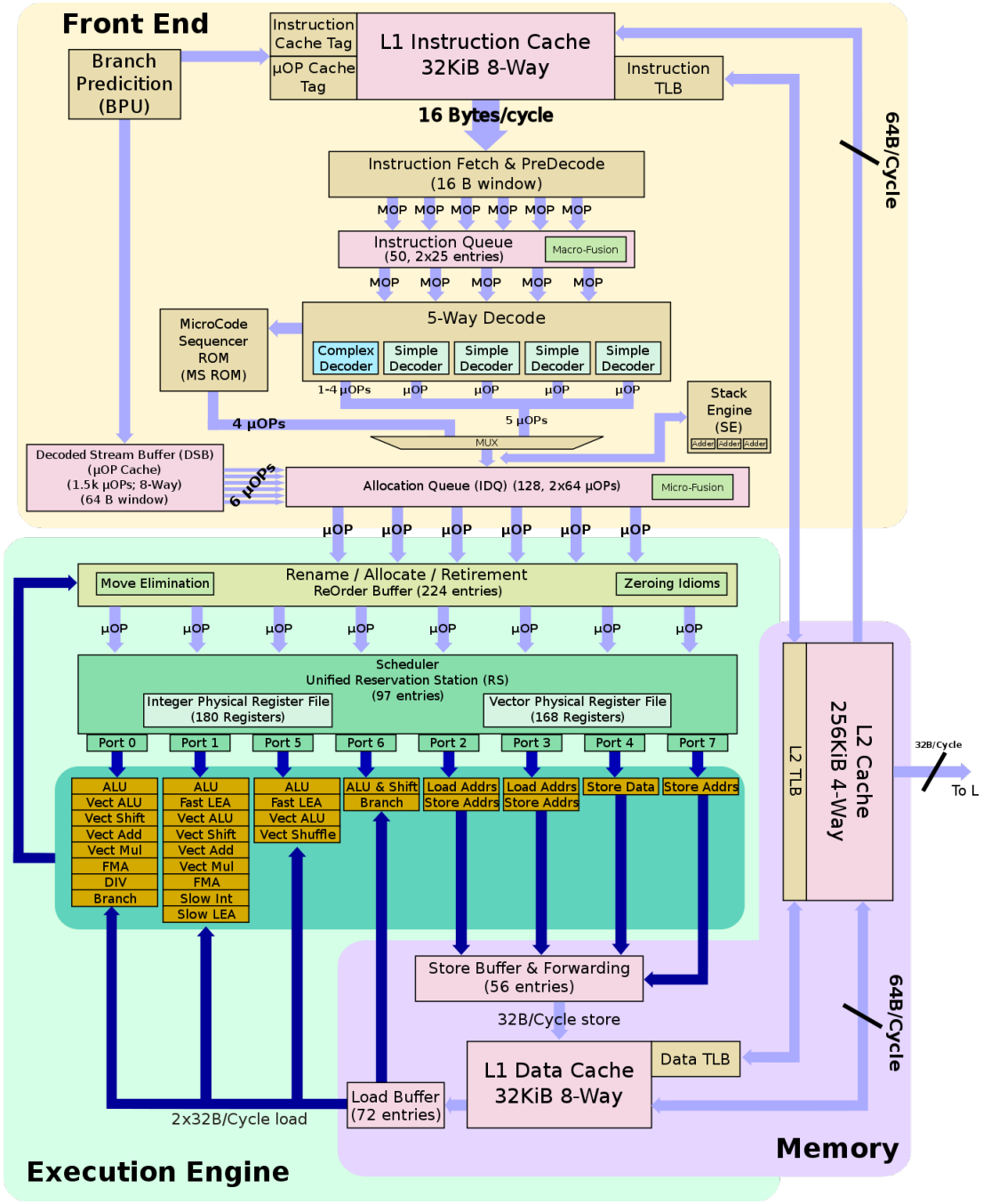

Figure 2.1: Skylake's Architecture

Front End

The goal of the front end is to supply the back-end with a sufficient stream of operations which it gets by decoding instructions from memory.
An efficient functioning of the front-end is essential to keep the whole core performance high.

Fetch & Pre-decoding The front-end has to fulfill the task of fetching x86 instructions from memory, decoding them, and delivering them to the execution units. Skylake fetching is done on a 16-byte fetch window.
x86 instructions are complex, variable length, have inconsistent encoding: at the pre-decode buffer the instruction boundaries get detected and marked.
The pre-decoder has a throughput of 6 macro-ops or until 16 bytes are consumed.

Instruction Queue & MOP-Fusion The pre-decoded instructions are delivered to the Instruction Queue (IQ).
Inside the instruction queue, the instructions are analyzed and if they result compatible, they will be converted into a unique, fused instruction. Fused instructions remain the same thought the entire pipeline and get executed as a unique operation, helping to improve the core performance.

Decoding The instruction decode stage breaks down each instruction into *micro-op*erations: the other stages of the pipeline work exclusively with micro-operations. Up to five pre-decoded instructions are sent to the decoders each cycle. Decoders read the macro-operations and emit regular, fixed length micro-operations. Instructions are elaborated by different decoders according to their complexity. If the macro instruction is so simple that can be translated into a single µOP, it will be sent to one of the four simple decoders. If the macro-ops is instead more complex, it will be sent to a complex decoder which breaks it down up to four µOPs. Those instructions that transform into more than four µOPs are detoured through the *Microcode Sequencer* (MS)ROM. When that happens, up to 4 µOPs/cycle are emitted until the microcode sequencer is done. During that time, the decoders are disabled.

µOP cache & x86 tax Decoding the variable-length, inconsistent, and complex x86 instructions is an expensive task in terms of performance and power consumption. For this reason an helper mechanism to avoid this onerous task has been implemented: a µOP cache, also called *Decode Stream Buffer* (DSB) has been added to the architecture. This cache is organized into 32 sets of 8 cache lines, with each line holding up to 6 µ-op. An hit in the cache allows to skip the decode stage and to deliver directly to the *Instruction Decode Queue* (IDQ) up to 6 µ-ops. The µ-op cache has an average hit rate of 80%.

Allocation Queue The emitted µOPs from the decoders are sent directly to the *Allocation Queue* (AQ) or Instruction Decode Queue (IDQ). The Allocation Queue acts as the interface between the front-end (in-order) and the back-end (out-of-order). The IDQ it is partitioned between two active threads. The queue's purpose is effectively help absorb bubbles which may be introduced in the front-end, ensuring that

a steady stream of 6 µOPs are delivered each cycle.

µOP-Fusion & LSD The IDQ does a number of additional optimizations as it queues instructions. The *Loop Stream Detector* (LSD) is a mechanism inside the IDQ capable of detecting loops that fit in the IDQ and lock them down. That is, the LSD can stream the same sequence of µOPs directly from the IDQ continuously without any additional fetching, decoding, or utilizing additional caches or resources. Streaming continues indefinitely until reaching a branch mis-prediction. The LSD in Skylake is capable of detecting loops up to 64 µOPs per thread.

Back End

Skylake's execution engine deals with the actual processing of out-of-order operations. Micro-ops received from the decode queue are written into a *reorder buffer* (ROB) while they are on the flight in the execution unit. The instructions are sent from the allocation queue to the ROB at the rate of 6 µOPs each cycle.

Renaming & Allocation The Reorder Buffer has a capacity of 224 entries. It is at this stage that architectural registers are mapped onto the underlying physical registers. Other additional bookkeeping tasks are also done at this point such as allocating resources for stores, loads, and determining all possible scheduler ports. The ROB is tailored for discovering register dependencies between micro-ops. However, micro-ops that execute out-of-order can also have memory dependencies. For this reason, out of order engines have a *load buffer* and a *store buffer*, keeping track of in-flight memory operations and are used to resolve dependencies. Register renaming is also controlled by the *Register Alias Table* (RAT) which is used to mark where the data we depend on is coming from (after that value, too, came from an instruction that has previously been renamed). The *Register Allocation Table* (RAT) matches each register with the last reorder buffer entry that updates it. The Renamer uses the RAT to rewrite the source and destination fields of micro-ops when they are written in the ROB. Since Skylake performs speculative execution, it can speculate incorrectly. When this happens, the architectural state is invalidated and as such needs to be rolled back to the last known valid state. Skylake has a 48-entry *Branch Order Buffer* (BOB) that keeps tracks of those states for this very purpose.

Scheduler The scheduler holds the µOPs while they wait to be executed. The scheduler decides which micro-ops in the ROB gets executed and places them in the *Reservation Station* (RS). The scheduler has a capacity of 97 entries being competitively shared between the two threads. The scheduler includes the two register files for integers and vectors. It is in those register files that output operand data is stored. In Skylake, the integer register file was also slightly increased from 160

entries to 180. At this point µOPs are not longer fused and will be dispatched to the execution units independently. A µOP could be waiting on an operand that has not arrived (e.g., fetched from memory or currently being calculated from another µOPs) or because the execution unit it needs is busy. Once the µOP is ready, they are dispatched through their designated port. The scheduler will send the oldest ready µOP to be executed on each of the eight ports each cycle. The scheduler had its ports rearranged to better balance various instructions. The reservation station has one port for each functional unit that can execute micro-ops independently. Each reservation station port holds one micro-op from the ROB. The reservation station port waits until the micro-op's dependencies are satisfied and forwards the micro-op to the functional unit.

Retirement When the functional unit completes executing the micro-op, its result is written back to the ROB and forwarded to any other reservation station port that depends on it. Once a µOP executes, or in the case of fused µOPs both µOPs have executed, they can be retired. Retirement happens in-order and releases any used resources such as those used to keep track in the reorder buffer. The ROB stores the results of completed micro-ops until they are retired, meaning that the results are committed to the register file and the micro-ops are removed from the ROB. Although micro-ops can be executed out of order, they must be retired in program's specification order such that the program's behaviour and exceptions are correct. When a micro-operation causes an hardware exception, all the following micro-ops in the ROB are squashed, and their result is discarded.

2.3 Microcode

In this section we describe how microcode works in modern architectures. This is useful to understand why we have chosen microcode correlated events.

2.3.1 Definition

The Intel architecture defines a complex instruction set (CISC), but processors are actually designed following reduced instruction set (RISC) principles. As shown in Figure 2.2, this is accomplished by having the instruction decode stages break down each instruction into micro-operations, which resemble RISC instructions. The Intel architecture features a large instruction set, whereas some instructions are used infrequently, and some instructions are very complex, which makes it impractical for an execution core to handle all the instructions in hardware. While the frequently used instructions are handled by the core's fast path, Intel CPUs use a microcode

table to break down rare and complex instructions into sequences of simpler instructions. Infrequently used instructions that require more than 4 micro-ops use a slower decoding path that relies on a sequencer to read micro-ops from a *microcode store ROM* (MSROM). As described in the previous section, the other stages of the execution pipeline work exclusively with micro-ops. The microcode is completely invisible to software developers, and its design is mostly undocumented. Intel, as described in [18], does not want it to be public knowledge: special cryptographic protection and hardware lock-down have been implemented when microcode is updated.

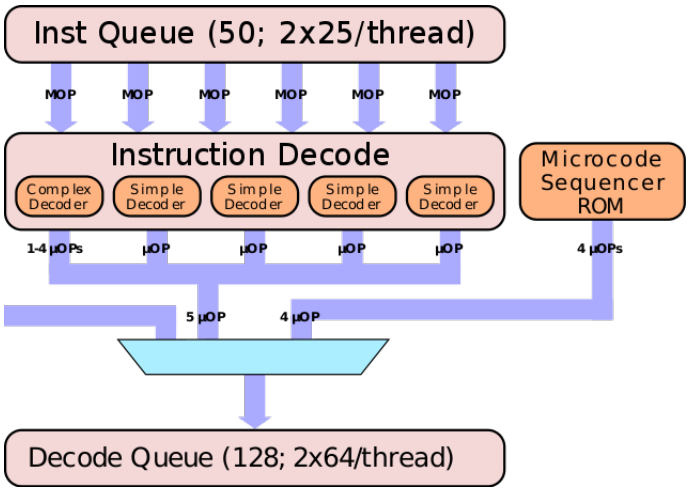

Figure 2.2: Decoding on the Skylake architecture

2.3.2 Exception Handling

The core's functional units handle common cases in fast paths implemented in hardware. As shown in Figure 2.3, when an input cannot be handled by the fast paths, the execution unit issues a microcode assist, which points the MSROM to a routine in microcode that handles the edge cases. This routine is then sent to the corresponding functional unit, which supports the execution of microcode procedures via dedicated microcode call and return micro-ops. The Intel paper in [18] assumes that all execution units in the core can issue microcode assists, which are performed at micro-op retirement.

2.4 Performance Monitoring Counters

2.4.1 Definition

Performance Monitoring Counters (PMCs) are a set of special purpose *Model Specific Registers* (MSR) built into modern processors to store the counts of hardware related activities. PMCs provide a fundamental tool for monitoring applications, tracing dynamic control flow and identifying performance hotspots. Performance

Monitoring Counters provide access to a wealth of detailed informations related to CPU's functional units, caches and main memory accesses. These counters permit selection of processor performance parameters to be monitored and measured: these parameters are also known as performance events. Each performance event is associated to a functional unit inside the processor's architecture. PMCs can be grouped under two different categories:

1. *Fixed Counters* which, if set, can measure only one, fixed event.
2. *Generic Counters* which can be programmed to measure personalized events, which vary from one architecture to another.

Thanks to generic counters, the performance analysis can be personalized according to the developer needs: each PMC can be set with the corresponding hexadecimal representation of the event to be monitored. Programmable counters allow specialized performance engineers to precisely measure the performance of computation-bound application and tune them accordingly.

2.4.2 PMC Functionalities

Performance monitoring counters have been designed at hardware level to provide different functionalities:

- *Event counting* — A PMC is configured to count one or more types of events, specified in the corresponding bit field. During the monitoring, the stored performance value is incremented on each event occurrence. The programmer can then read the counter value.
- *Interrupt-based event sampling* — A PMC is preset to a modulus value that will cause the counter to overflow after a specified number of events has been counted.

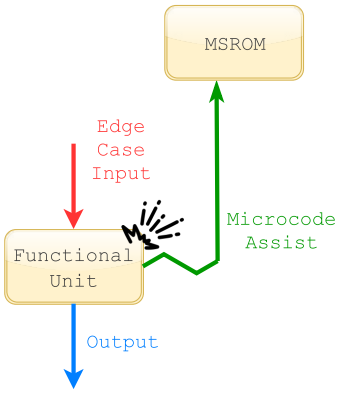

Figure 2.3: Microcode Assist

Instead of stopping the program at regular time intervals, the performance-monitoring hardware interrupts the application after a specific number of performance events has occurred. To support event-based sampling (EBS), performance-monitoring hardware generates a *Performance Monitoring Interrupt* (PMI) when a performance event counter overflows. A *Performance Monitor Interrupt* (PMI) is a special interrupt service routine for handling PMCs overflow. The PMI saves sample data and re-initializes the performance event counter to overflow again after the specified number of events have been counted. When the monitoring session terminates, the user can plot the data samples saved by the PMI to create an event-based analysis.

- *Processor event-based sampling (*PEBS*)* — PEBS captures more precise processor state information compared to interrupt based event sampling. While using the same means of an interrupt-based sampling, the main difference is that using PEBS it's possible to access to very precise execution details of the monitored program execution, such as register values and the last sixteen branch addresses taken by the program. In PEBS, the processor writes a record of its architectural state to a memory buffer after the counter overflows at the completion of the instruction that caused the event. This mechanism enables to locate the eventing condition in the instruction space and to reconstruct the function arguments from the saved register content. In order to use PEBS it is needed to hook the PMI and setup a specific *Interrupt Service Runtime* (ISR). PEBS limitation is that it can be enabled only for a subsets of events, typically related to *at-retirement* instructions. Another issue to take into account is the *skid*, representing the number of instructions executed between an event of interest happening and the kernel being able to stop and record the event. While it is possible to instruct the processor to enforce constant or zero skid, in practice it may not be able to satisfy such requirement.

To the interested reader, we suggest to consult [21] and [5], which provide more details about less commonly used PMC functionalities. In [20] it is also documented how to use PMC to analyze an application performance.

2.5 Performance Events

Performance Events are those micro-architectural performance that can be monitored by PMCs. Each event represents a specific hardware event occurrence. There are two classes of events:

1. *Non Architectural Events*: The visible behaviour of architectural performance monitoring of this kind of events is consistent across processors implementation.

This class supports counting and interrupt-based event usage, with a smaller set of available events.

2. *Architectural Events*: events for monitoring performance using counting or interrupt-based event sampling usage. They vary from one processor to another. For a given processor it's possible to inspect its available events in [21].

Performance events can be semantically grouped in different ways. A smart approach to group them is described in [27], which identifies five different categories:

- *Program Characterization Events* — Typically independent of the processor implementation, they describe the attributes of a program. The most common examples of these events are the number and type of instructions (for example, loads, stores, floating point, branches, and so on) completed by the program.
- *Memory Access Events* — Give an insight of the processor's memory hierarchy, counting references and misses to various caches and transactions on the processor memory bus.
- *Pipeline stall events* — Allowing the analysis of how well is the program's instruction flow is going through the pipeline.
- *Branch prediction events* — Describe the performance of branch prediction hardware.
- *Resource Utilization events* — Let the user to estimate the usage of a certain resources.

As reference for the Intel architectures, all available architecture-dependent events are listed in [21].

2.6 PMC Hardware Implementation

In this Section we describe how PMCs are implemented at hardware level to give the reader a comprehensive overview of such functionalities. This is the hardware-level interface that the kernel has to manipulate in order to control the counters.

2.6.1 Definition

The *Performance Monitoring Unit* (PMU) permits the selection of processor performance parameters to be monitored and measured. An architecture's PMU is implemented in hardware by several MSR. A PMU is composed of MSRs that keep track of an event occurrence (properly called Performance Monitoring Counters), and other ones devoted to the configuration of parameters. The PMU implementation is architecture dependent and historically there have been four different versions.

Skylake, the target architecture, implements the fourth and latest version which is discussed in this Section.

2.6.2 Architectural Performance Monitoring Version 4

The fourth architectural performance monitoring version implements and supports all the previous versions. Configuring an architectural performance monitoring event involves programming *Performance Event Select,* `IA32_PERFEVTSELx` model specific registers. Performance Monitoring Counter (`IA32_PMCX`) registers keep track of a performance monitoring session's values. `PERFSEL_X` registers specify the event that PMC registers will monitor and for this reason they are coupled together. This is the core implementation. Other model specific registers have been introduced to handle the most common operations in a faster way. These MSRs are:

- `IA32_PERF_GLOABAL_CTRL` — for enabling/disabling events.
- `IA32_PERF_GLOBAL_STATUS` — for checking the status of event overflow.

A complete and detailed description of how the event selection MSRs works is given in [21]. In this Section we report the most important registers implementing the core functionality.

General Purpose Monitoring Counter

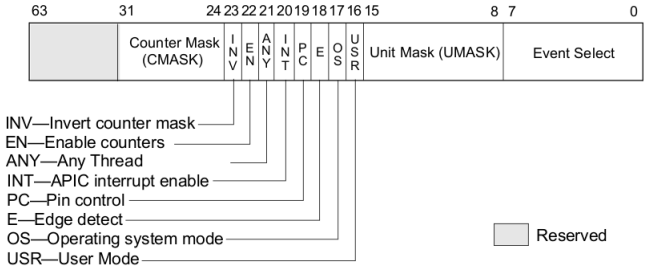

Figure 2.4: IA32_PERFEVTSELx

Register Fields:

- Event select field (bits 0 through 7) — Selects the event logic unit used to detect micro-architectural conditions. The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event.
- Unit mask (UMASK) field (bits 8 through 15) — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance

event, its corresponding UMASK value defines a specific micro-architectural condition.

- USR (user mode) flag (bit 16) — Specifies that the selected micro-architectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.

- OS (operating system mode) flag (bit 17) — Specifies that the selected micro-architectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.

- E (edge detect) flag (bit 18) — Enables, when set, edge detection of the selected micro-architectural condition. The logical processor counts the number of de-asserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state.

- PC (pin control) flag (bit 19) — When set, the logical processor toggles the PMi pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PMi pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

- INT (APIC interrupt enable) flag (bit 20) — When set, the logical processor generates an exception through its local APIC on counter overflow.

- AnyThread (bit 21) — for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core.
  When bit 21 is 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

- EN (Enable Counters) Flag (bit 22) — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled.

- INV (Invert) flag (bit 23) — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). If counter-mask is programmed to zero, INV flag is ignored.

- Counter mask (CMASK) field (bits 24 through 31) — When this field is not zero, a logical processor compares this mask to the events count of the detected micro-architectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one, otherwise the counter is not incremented. This mask is intended for software to characterize micro-architectural conditions that can count multiple occurrences per cycle. If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

Fixed Function Performance Counters

Some architectural performance events are counted using fixed-function MSRs (`IA32_FIXED_CTR_X`)
Each of the fixed-function PMC can count only one architectural performance event. Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (`IA32_FIXED_CTR_CTRL`).
Since the monitoring functionality is fixed, there is no need to specify any UMASK.

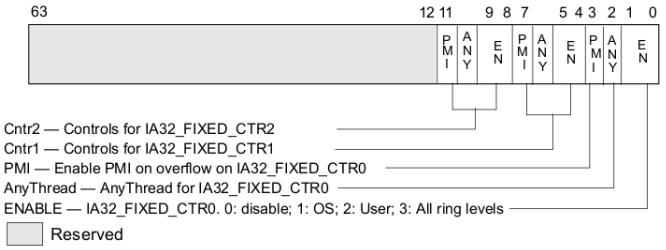

Figure 2.5: IA32_FIXED_CTR_CTRL MSR

Fields:

- Enable field (lowest 2 bits within each 4-bit control) — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of $(11B)_1$ enables the counter to increment irrespective of privilege levels.

- PMI field (the fourth bit within each 4-bit control) — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

- AnyThread field (bit position $2 + 4n$, for $n \in \{0, 1, \ldots\}$) — When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When an AnyThread bit is 0 in `IA32_PERF_FIXED_CTR_CTRL`, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the `IA32_PERF_FIXED_CTR_CTRL` MSR.

2.7 User-space PMC Usage

In this Section we discuss how PMCs can be accessed with user-space programs. This description allows to understand the PMCs provided functionalities that are used in this thesis.

2.7.1 Assembly

The most trivial and low-level approach is to access and set PMC values via assembly instructions. It is possible to read and modify the counter MSR via the following commands:

- `WRMSR` — Writes the contents of registers EDX:EAX into the 64-bit model specific register specified in the ECX register. In order to correctly execute this instruction, administrative privileges are required. Section 2.6 describes the actual monitoring counters hardware implementation.

- `RDPMC` — Read performance-monitoring counter specified by `ECX` into `EDX:EAX`. This is the dedicated assembly instruction to directly read a target performance monitoring counter value. The advantage of using such instruction is the possibility to get low-latency reads without having the kernel to trap such access. The `RDPMC` instruction allows to access from ring 3 privileges mode the values of performance counters. This is useful because through eliminating the kernel as a middleman the performance data can be retrieved more efficiently. `RDPMC` support is allowed only if an event is currently enabled in a process's context. By issuing this command, the programmer gets the raw hardware count of a PMC, by accessing directly its 48-bit value. Since the kernel doesn't intercept

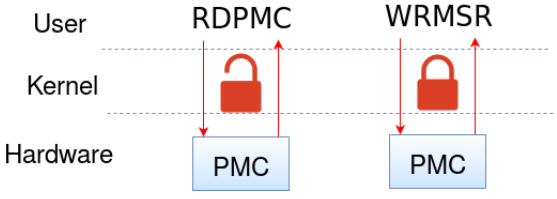

Figure 2.6: Accessing PMC via assembly

this assembly instruction, no virtualization layer is offered to the user. The retrieved count value is as raw as the underlying hardware: the user has to take into account overflows (which may happen more or less frequently according to the selected performance event), preemption and concurrent monitoring of several processes.

2.7.2 PERF_EVENT_OPEN System Call

The Linux kernel provides a dedicated system call to control PMCs. The exposed system call has no **glibc** (GNU C Library) wrapper for this function. It is convenient for the program using such functionality to define an hand-made wrapper, by calling the `SYSCALL` function with the `__NR_PERF_EVENT_OPEN` constant parameter, as shown below:

```
sys_perf_event_open(){
        int fd;
        fd = syscall(__NR_perf_event_open,...);
        return fd;
    }
```

Algorithm 1: `SYS_PERF_EVENT_OPEN`

`PERF_EVENT_OPEN` is a complex system call, exposing a huge amount of functionalities and parameters to be set. The Linux documentation [5] provides a comprehensive description reference. Performing a call to `PERF_EVENT_OPEN` creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured; these can be grouped together to measure multiple events simultaneously. The advantage of using `PERF_EVENT` against the `RDPMC` instruction is that the kernel provides a virtualization layer on top of the raw, bare-metal hardware counters. Thanks to this virtualization layer, the user doesn't have to worry about setting up all of low level bit fields in the MRSs described in Section 2.6, handling count overflows, and keeping the correct count when different processes are executed. When using `PERF_EVENT`, another important detail to take into account to perform

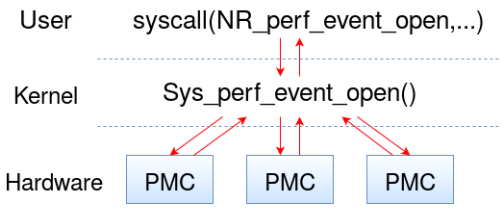

Figure 2.7: `PERF_EVENT_OPEN` Syscall Abstraction Layers

accurate analysis is to check the number of actual hardware counters the underlying architecture can offer to the user. The Intel documentation [21] describes for each architecture the number of PMCs implemented in hardware. The sampling/counting activity using more events than the real hardware implemented PMCs won't be accurate. The kernel allows the user to do this, but the events will be multiplexed per each hardware counter. `PERF_EVENT_OPEN` provides the functionalities described in Section 2.4, taking as input the following parameters:

- The `pid` and `cpu` integer, which specify which process and CPU to monitor:
  - `pid` $= 0$ and `cpu` $= -1$ — This measures the calling process/thread on any CPU.
  - `pid` $= 0$ and `cpu` ≥ 0 — This measures the calling process/thread only when running on the specified CPU.
  - `pid` > 0 and `cpu` $= -1$ — This measures the specified process/thread on any CPU.
  - `pid` > 0 and `cpu` ≥ 0 — This measures the specified process/thread only when running on the specified CPU.
  - `pid` $= -1$ and `cpu` ≥ 0 — This measures all processes/threads on the specified CPU.
    This requires `CAP_SYS_ADMIN` capability or a `/PROC/SYS/KERNEL/PERF_EVENT_PARANOID` value less than 1.
- The STRUCT `PERF_EVENT_ATTR`, which provides detailed configuration information about the events being created.
  This is a complex structure and has a huge amount of fields: a detailed description on each parameter is reported in [5].
  The commonly used and most important fields are:
  - the `TYPE` — under which a generic event is grouped into according to its nature: it can be `HARDWARE`, `SOFTWARE`, `TRACEPOINT`, `HW_CACHE`, `RAW` and `BREAKPOINT`.
  - the `CONFIG` — parameter for identifying events. Events of raw type are identified by setting this parameter with an unique hexadecimal number obtained by concatenating the *Event Number* and *Umask* values. A complete mapping of all events with description can be found inside [21].
  - `SAMPLE_PERIOD` — defines the number of occurrences of an event, not the number of timer ticks. An event is a sampling one if `PERF_EVENT_OPEN` is called with this parameter set to a value greater than zero. A sample is recorded when the sampling counter overflows, i.e., wraps from 2^{48} back

to 0. The hardware counter's initial value of the selected event will be set to 2^{48} - `SAMPLE_PERIOD`. What kind of data has to be recorded on each overflow is specified by the user with the `SAMPLE_TYPE` field. Even is the sampling leader is set to count time related events, it is not forbidden to sample under different dimensions. For instance it is possible to get the number of retired not taken branch instructions each `SAMPLE_PERIOD` L1 cache misses, or vice-versa, or any other combination of events. It is important to remark that, as reported in [8], in order to sample at constant rate, this parameter has to be used. Specify the sampling period with the `SAMPLE_FREQ` allows the kernel to dynamically change the sampling period in order to achieve the desired frequency.

  - `DISABLED` — specifies whether the counter starts as disabled or enabled.
  - `EXCLUDE_USER`, `EXCLUDE_KERNEL`, `EXCLUDE_HV` — If set, the count excludes events that respectively happen in user space, kernel and hyper-visor space.
  - `SAMPLE_TYPE` — The several boolean values merged in this field specify what piece of information has to be collected in the samples. The user can ask for a huge variety of meta-parameters to be recorded. For a complete and comprehensive list, [5] fully describes it.

- The `GROUP_FD` argument allows event groups to be created. An *event group* always has one event which is the group leader. The leader is created first, setting this parameter with the default value of -1. The remaining group members are created with subsequent `PERF_EVENT_OPEN` calls with this parameter being set to the file descriptor of the group leader. An event group is scheduled into the CPU as a unit: all the event members are enabled and disabled at the same moment in the same processor. This allows to meaningfully compare, add, divide their values with each other. Grouping events together is in practice really useful for performance sampling: the event leader is initialized as sampling event and upon overflow all event values belonging to the group are collected.

Correlated System Calls `PERF_EVENT_OPEN` fulfills the task of setting up the events, returning to its caller the file descriptor associated with the initialized event counter. Several other useful system calls are provided to interact with performance counters: when invoking them them, it is required to specify the file descriptor associated to the opened performance file. A comprehensive list of all the system calls interacting with PMCs is provided in [5]. In this work we cover the most commonly used and important ones:

- `IOCTL` — enables/disables the counting, refreshes, resets the count value and dynamically modifies the sample period with the respectively parameters `PERF_EVENT_IOC_ENABLE/DISABLE/REFRESH/RESET/PERIOD`.
- `READ` — reads the current value of a performance event. Further meta-informations such as the event id, running time and enabled time may be provided according to the values specified in the `READ_FORMAT` field. It is not advisable to perform this system call while the counters are enabled: the retrieved value won't be accurate since this system call has latency for accessing PMCs value.
- `MMAP` — the kernel stores the samples into a ring buffer, whose memory is intended to be shared with user-space. `MMAP` allows to access this ring-buffer and read the samples. The sample format that will be found is the one the user specifies using `SAMPLE_TYPE` when instantiating the events.

Important Parameters In the `/SYS` folder there are several `PERF_EVENT` related configuration files. Here we cover those dealing with the user capabilities for making the system call and the ones that tune the sampling machanism:

- **Permissions**
  - `/PROC/SYS/KERNEL/PERF_EVENT_PARANOID` — The existence of the `PERF_EVENT_PARANOID` file is the official method for determining if a kernel supports `PERF_EVENT_OPEN`.
    The `PERF_EVENT_PARANOID` file can be set to restrict access to the performance counters.
    - 2 allow only user-space measurements.
    - 1 allow both kernel and user measurements (default).
    - 0 allow access to CPU-specific data but not raw tracepoint samples.
    - -1 no restrictions.
  - `/SYS/BUS/EVENT_SOURCE/DEVICES/CPU/RDPMC` — If this file is 1, then direct user-space access to the performance counter registers is allowed via the `RDPMC` instruction. This file value is by default 1: it is possible to read by user-space the performance counters registers.
  - `SYSCTL PERF_CPU_TIME_MAX_PERCENT` — Specifies how much CPU time the kernel should be allowed to use to handle sampling events. If this limit is exceeded, the system will drop the sampling frequency in the attempt of reducing its CPU usage. This parameter can be set with the following values:
    - 0: disable the mechanism. Do not correct the sampling rate no matter how CPU time it takes.

    * 1-100: attempt to throttle the sample rate to this percentage of CPU.

- **Sampling**

  - `PROC/SYS/KERNEL/PERF_EVENT_MAX_SAMPLE_RATE` holds the maximum sample rate at which the kernel is bound to trigger PMI. This limit has been introduced to bound the users sample rate such that the monitoring session doesn't impact the overall machine performance and potentially locks up the machine. In the scenario when too much work is submitted to the CPU, the kernel will block some samples to be recorded and the user will get a token value representing that the monitoring has been throttled.

2.7.3 High-Level Tools

On top of the `PERF_EVENT` interface provided by the kernel, several different libraries and tools have been implemented. For instance Figure 2.8 represents the high-level tools abstraction layers.

- `PERF` — Linux performance monitoring tool
- `PAPI` — Performance API
- `JEVENTS` — C library utility for event selection and and ready-to-use functions dealing with the circular buffer
- `OPROFILE` — Open source project that includes a statistical profiler for Linux systems
- `PERFMON2`— Performance Library providing support for the `PERF_EVENTS` interface

Higher level interfaces are adding other virtualization layers on top of the `PERF_EVENT` functionalities.
This is good because a uniform interface is provided to the user.
On the other hand however, this may add more noise to the monitoring and for this

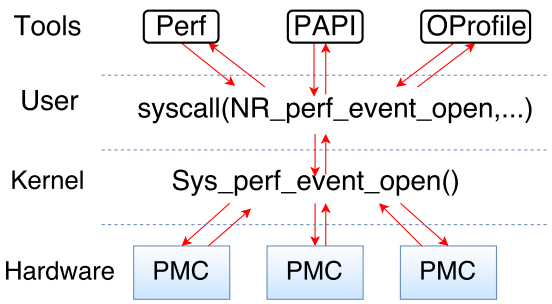

Figure 2.8: High-level Tools Abstraction Layers

reason many engineers have chosen to directly deal with the system call, in order to get as much precision as possible in their measurements.

2.8 Perf Event Kernel Implementation

In this Section we provide an in depth description about the `PERF_EVENT` subsystem kernel implementation. This is fundamental to understand how we have modified the Linux kernel in order to reach our goal. The mentioned parameters and function have been directly used and modified in our thesis.

While the kernel developers take special effort to implement new features and fixing the already existing ones without breaking the user API, the inner structure of the kernel can be changed anytime. `PERF_EVENT` is a well consolidated subsystem and it is unlikely to change drastically from a kernel version to another.
However, when reading this Chapter the reader has to take into account that this in-depth analysis has been made studying the version 4.9.1 and, while the later versions follow the same ideas hereby described, there is no guarantee that something will be changed in the future releases. The whole `PERF_EVENT` subsystem actually becomes part of the kernel only if it is compiled with `CONFIG_PERF_EVENTS` set.

2.8.1 PMC Virtualization

`PERF_EVENT` system adds an abstraction layer on top of the physical monitoring counters building up virtual counters. While the physical ones monitor the event occurrences, the virtual counters store the actual count at software level. This is done for several reasons:

- Avoiding overflow — A physical PMC monitors an event with a 48 bit-width register, but the kernel backs up the correct value inside an unsigned 64 bit integer variable. Thanks to this approach the count is saved inside a bigger dimension variable that stores the correct value, even on physical counter overflow.

- Allowing multi-threading and preemption — Since the virtual counters are variables held inside the processor's context, they are saved and restored on context switches. The virtual counters will always store the correct value: the user has the freedom to run different threads performing heterogeneous profiling tasks without worrying about the hardware current count.

- PMC selection — The user cannot specify on which physical counter wants his events to be counted. This task is completely managed by the kernel that chooses the most appropriate counter with its own criteria. This forbids the user to directly control hardware configuration details.

- Multiplexing — If a process is assigned to monitor more events than the actual number of hardware PMCs, the kernel will multiplex them just by adding more virtual counters. In such a way it's possible to monitor a huge number of events with constrained physical resources, but this approach comes with an high cost. In this scenario the kernel won't be able to return the exact number of the occurred events, but an estimate of the count. This is achieved by scaling the total count based on the total time enabled against the total time of the application running, using the following formula:

$$FinalCount = RawCount * \frac{TimeEnabled}{TimeRunning}$$

  Where:

  - Time Enabled represents the time the counter has been active during the measurements.
  - Time Running is the time spent by the whole program to run.

  The final count value provides an estimate of what the counts would have been if each event had been counted for the duration of the process. If a precise event counting is required, asking the kernel to monitor more events than the number of hardware available counters isn't advisable.

2.8.2 Virtual Counters Definition

In practice, the kernel represents each virtual counter using a `STRUCT PERF_EVENT` instance. `PERF_EVENT` is the core structure inside the perf_event subsystem, featuring a huge amount of fields: here we provide a description of the most important ones. For the interested reader, there is no reference documentation that can be consulted. The only way to understand how the perf_event subsystem works kernel-side is to directly read the code, or the content of this Section. In `[SRC]/INCLUDE/LINUX/PERF_EVENT.H` are defined the principal structures: `PERF_EVENT`, but also `HW_PERF_EVENT` representing performance event hardware details and PMU providing a uniform interface for generic performance monitoring unit. The most important fields of the `PERF_EVENT` structure are the following:

- `COUNT` — stores the actual event count. In order to avoid overflow, the kernel defines this variable as an unsigned integer 64-bit long. The actual hardware-implemented counters are 48-bit width and for this reason the kernel reads them frequently, adding the deltas to this field. In details, the last observed hardware counter value is saved inside the `PREV_COUNT` field in the `HW_PERF_EVENT` structure. This value is then accessed to compute the delta, which is added into the `COUNT` field inside the event structure. The count field is updated using the `CMPXCHG` instruction and is designed in order to be always readable by `RDPMC`.

- STATE — keeps track of the state of the event: a special ENUM is defined to represent all possible states of an event. The main states are PERF_EVENT_STATE_ACTIVE/INACTIVE/ERROR, respectively representing when a counter is enabled, disabled or in error state.
- TOTAL_TIME_ENABLED, TOTAL_TIME_RUNNING — respectively represent the total number the counter have been enabled and running in nanoseconds. This is a useful for checking whether counter multiplexing has taken place.
- ATTR — PERF_EVENT_ATTR structure. This is where the user-defined preferences for the performance monitoring are saved.
- ID — integer identifier assigned to each virtual counter
- PMU — represents an high level interface of the whole performance monitoring unit.
- HW_PERF_EVENT — stores performance event hardware details.
- GROUP_LEADER — pointer to the PERF_EVENT leader.
- SIBLING_LIST — linked list of all the virtual counters in the same group. The group leader does not belong to this linked list, and can only be accessed through the GROUP_LEADER field.
- OWNER — pointer to the TASK_STRUCT owning this virtual counter
- OVERFLOW_HANDLER — of type PERF_OVERFLOW_HANDLER_T, in practice this is the function pointer for the PMI calls.

Another important component strictly correlated to the virtual counters is the performance monitoring unit, PMU, which represents from an high abstraction layer all the hardware counters. This structure is implemented as a uniform interface for controlling the underlying hardware system. This is a smart design choice because it allows the user to just call the exposed functions, without caring about all the architecture-dependent details. When a new architecture support for performance counting is implemented, the kernel developers have to only worry about implementing the low level functions that set up the right MSR bits for monitoring, starting, stopping the PMC. Using this scalable design approach, the kernel will support performance monitoring in a new processor without completely change its code every time, but by simply having the PMU function pointers redirecting to the correct architecture-dependent functions. The most important fields are the following ones:

- PMU_ENABLE / PMU_DISABLE — fully enable/disable this PMU
- ADD — adds a counter to the PMU

- DEL — deletes a counter from the PMU
- START / STOP — starts/stops a counter present on the PMU
- READ — updates the counter value of the event
- SCHED_TASK — callback function used for context-switches
- COUNT — access and return the counter values of the specified virtual counter

The HW_PERF_EVENT structure stores performance event hardware details. Its most important fields are the following ones:

- STATE — integer value keeping track of whether the hardware counter has stopped or the count value stored inside the virtual counter is up-to-date, with the corresponding values PERF_HES_STOPPED/UPTODATE.
- PREV_COUNT — The last observed hardware counter value, updated with a CMPXCHG instruction such that the READ function exposed by the performance monitoring unit can be called nested.
- SAMPLE_PERIOD — The period to start the next sample with.
- LAST_PERIOD — The period the current sample is started with.
- PERIOD_LEFT — How much is left of the current period.

2.8.3 Initialization

In kernel space, the system call is redirected from the system call table to the following function, located in /KERNEL/EVENT/CORE.C, defined in this way:

```
SYSCALL_DEFINE5 (perf_event_open,
                 struct perf_event_attr __user * attr_uptr,
                 pid_t pid,
                 int cpu,
                 int group_fd,
                 unsigned long flags)
```

Algorithm 2: PERF_EVENT system call

The kernel has to provide the promised functionalities to the user, initializing a new virtual counter with the following steps:

1. It checks for the consistency of the user-defined PERF_EVENT_ATTR and it makes a copy.
2. It verifies that the user actually has the capabilities for the functionalities she's asking for.

3. Calls an internal function, PERF_EVENT_ALLOC, which actually takes care of instantiating the requested PERF_EVENT struct. In case the event belongs to a group as a simple member of it, it is attached to the event leader.

4. It attaches the virtual counters to the corresponding process record. This is implemented adding PERF_EVENT to the TASK_STRUCT, which is the kernel representation of a process. Each TASK_STRUCT has indeed the member STRUCT LIST_HEAD PERF_EVENT_LIST which is the kernel representation to a linked list of PERF_EVENTs.

5. If the specified SAMPLE_PERIOD parameter is greater than zero, sampling will be performed. As consequence the kernel takes care of setting up everything needed for this special kind of monitoring: it calls the function PERF_EVENT_SET_OUTPUT, which provide initialization of the event's ring buffer. What a ring buffer is, how sampling deals with it and other details are described in Section 2.8.6.

6. The kernel calls ANON_INODE_GETFILE, setting up a virtual counter to be returned to the user inside a file.

The PERF_EVENT subsystem also takes care of providing the correct operations permitted on the performance file returned to the user by the PERF_EVENT_OPEN system call. The structure file_operations providing such functions user-side, is 'overridden':

```
static const struct file_operations perf_ops={

    .llseek = no_llseek,
    .release = perf_release,
    .read = perf_read,
    .poll = perf_poll,
    .unlocked_ioct = perf_ioctl,
    .compat_ioctl = perf_compat_ioctl,
    .mmap = perf_mmap,
    .fasync = perf_fasync,
    }
```

Algorithm 3: PERF_EVENT File Operations

These functions are the only way the user can control the kernel allocated virtual counters.
When these file operations are called, the PERF_EVENT subsystem takes care of the request by changing its internal state.
In the following sections, we give an insight about the most important file operations and briefly discuss how they are implemented.

2.8.4 Context Switch

In-kernel performance monitoring has been implemented in such a way that the virtualized counters are local to the single process they are monitoring. This design allows the kernel to allow multi-threading and preemption.
We have studied the actual implementation of the scheduler, whose source code can be found at `[SOURCE]/KERNEL/SCHED/CORE.C` to inspect its behaviour w.r.t. virtual counters.
When `__SCHEDULE`, the main scheduler function, is invoked, if the next `TASK_STRUCT` to be executed is different than the current one, the `CONTEX_SWITCH` function call is done. `CONTEXT_SWITCH`, in order to change the running tasks, calls two paired functions: `PREPARE_TASK_SWITCH` and subsequently `FINISH_TASK_SWITCH`.
`PREPARE_TASK_SWITCH` takes as inputs the runqueue preparing to switch, the current task that is being switched out and the task it is going to switch to. This function's job is to disable and make the current process running being ready to be removed. The function invokes `PERF_EVENT_TASK_SCHED_OUT` which removes the events of the current task, with interrupts disabled. Each event is stopped and its count field is updated subsequently with the invocation. In practice, this is performed invoking `PERF_PMU_SCHED_TASK`, which disables and subsequently enable the performance monitoring unit by calling `PERF_PMU_ENABLE/DISABLE`. For the next task to be executed, `PERF_EVENT_TASK_SCHED_IN` takes care of restoring the event value and then enable it.

2.8.5 Enabling and Disabling

IOCTL controls the `PERF_EVENT` device. From user-space, this function is called specifying the file descriptor to be modified, the command to be executed and optional argument flags. In kernel-space this function retrieves the `PERF_EVENT` structure stored inside the file, passing it to an internal handler function `_PERF_IOCTL`. This dispatch different functions according to the input command. The most important commands, fundamental for using `PERF_EVENT`, are the following:

- `PERF_EVENT_IOC_ENABLE` — `_PERF_EVENT_ENABLE` function is called. This verifies the `PERF_EVENT` subsystem to be available for changing their state (it spinlocks waiting for it otherwise) and when everything is ready the inner call to `__PERF_EVENT_ENABLE` is performed, which actually takes care of making a cross CPU call to enable the performance event. The actual change of the STATE field is performed by an inner function called `__PERF_EVENT_MARK_ENABLED`, which puts a event into inactive state and update time fields. It is important to remark that enabling the leader of a group effectively enables all the group members that aren't explicitly disabled.

- PERF_EVENT_IOC_DISABLE — _PERF_EVENT_DISABLE function is called. This verifies all pointers to PERF_EVENTS to be available for changing their state (it spinlocks waiting for them otherwise) and when everything is ready the inner call to __PERF_EVENT_DISABLE is performed, which actually takes care of making a cross CPU call to enable the performance event. The actual change of the STATE field is done by setting the event state as PERF_EVENT_STATE_OFF. It is important to remark that disabling the leader of a group effectively disables all the group members.

- PERF_EVENT_IOC_RESET — _PERF_EVENT_RESET function is called, directly setting up the PERF_EVENT COUNT field to zero.

2.8.6 Simple Counting and Sampling

The user can retrieve performance with different approaches:

- Simple Counting — When simple counting, the event count is directly stored in the virtual counter structure, namely PERF_EVENT. This is shared to the user via the returned file when invoking PERF_EVENT_OPEN. In order to retrieve the value, the user has to perform the READ system call, specifying the performance file, the buffer to write the informations in, and the size. On kernel-space, PERF_READ is invoked, accessing the file stored PERF_EVENT structure. A check is performed to see whether the event belongs to a group or not, and the functions PERF_READ_GROUP or PERF_READ_ONE are called accordingly. In an optional way, if the user has requested some meta-informations about the counters upon PERF_EVENT_OPEN call, these are retrieved here. PERF_EVENT_READ_VALUE is called, which via PERF_EVENT_COUNT retrieves the actual virtual counter value. Then the data is copied to user-space.

- Sampling — When a sampling event is specified, the kernel attach to its corresponding virtual counter a ring buffer. Circular buffering makes a good implementation strategy for a queue that has fixed maximum size. This is used in the kernel to store the samples. The ring buffer is a kernel structure used where data has to be temporarily shared between a reader and a writer. The reader is the user space program performing the monitoring, which is supposed to read the samples and copy them as soon as they are available. The writer instead is the PERF_EVENT overflow handler, which is called upon every counter's overflow. This is defined upon virtual counter allocation, assigning to the PERF_EVENT STRUCT's pointer OVERFLOW_HANDLER to a function. As default, this pointer is assigned to PERF_EVENT_OUTPUT_FORWARD function. This takes care of two important different tasks: retrieve the actual events count and attach it to the ring buffer. The functionality of retrieving the events

count is implemented by an indirect call to `PERF_OUTPUT_READ_GROUP`, which by invoking `PERF_EVENT_COUNT` access the counter values and copies it. Other functions such `PERF_OUTPUT_PUT` take care of attaching the samples to the ring buffer. The ring buffer is memory mapped to user-space: this is the chosen implementation since it's the fastest way to share memory between kernel and user spaces. The user can access the shared memory area performing the `MMAP` function call, which is implemented by the `PERF_EVENT` subsystem via `PERF_MMAP`. The `JEVENT` library [3] is really useful for retrieving the samples using a user-space program, without caring about handling the raw ring buffer structure.

2.9 Fixed Point Arithmetic

In this Section we provide an overview about fixed point arithmetic, which it is necessary to re-implement the SVM inside the kernel.

Linux allows user-space processes to use floating point instructions. The Linux kernel however, as described in [23], entirely works without using floating point operations. This design choice follows one of the main purposes of the kernel project: to provide an operating system that works on top of any device. Not every device is equipped with hardware implemented floating point units: for this reason it is forbidden to kernel developers to use such operations. In practice, however, it is useful to use numbers with fractional part: under these circumstances fixed point arithmetic is used.

Fixed point numbers are represented as integers where a 'fixed point' is defined, separating the fractional from the integer one. Obviously, the fixed point cannot be moved. Given a fixed binary point position, shifting the bit pattern of a number to the right by 1 bit always divide the number by 2. Similarly, shifting a number to the left by 1 bit multiplies the number by 2. By reusing all integer arithmetic circuits of a computer, fixed point arithmetic is orders of magnitude faster than floating point arithmetic. This is the reason why it is being used in many games and digital signal processing applications. On the other hand, it lacks the range and precision that floating point number representation offers.

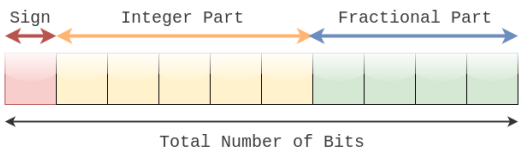

Figure 2.9: Bit Layout for Fixed Point Numbers

2.10 Elf File Format

Executable and Linkable Format (ELF) is the main binary format in use on modern Linux systems to store object files. In this Section we provide an high level overview about the ELF file format: understanding its structure is fundamental to inject a performance signature inside the target process binary, as described in Chapter 4.

Object files are binary representations of programs intended to execute directly on a processor. This format encodes different binary files having several purposes:

- A *relocatable objects file* — holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An executable objects file — holds a program that are in a form that an operating system can launch in a process.
- A dynamically loadable objects file — holds code and data suitable that can be loaded by an executable after it has started executing.
  Dynamically loadable shared libraries are examples of such objects.

Object files can be created by:

- The assembler — translates assembly instructions into the object file.
- The linker — forming executables from collections of relocatable objects.

As shown in Figure 2.10, a generic `ELF` object file can be seen under two different aspects:

- Linking View: The linking view it's used by the linker to retrieve all the required informations within the sections.

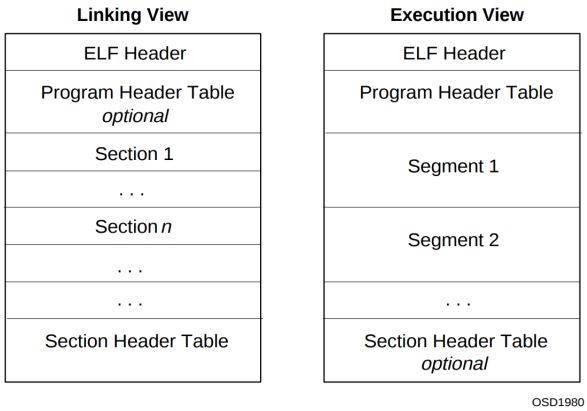

Figure 2.10: ELF Object File Format

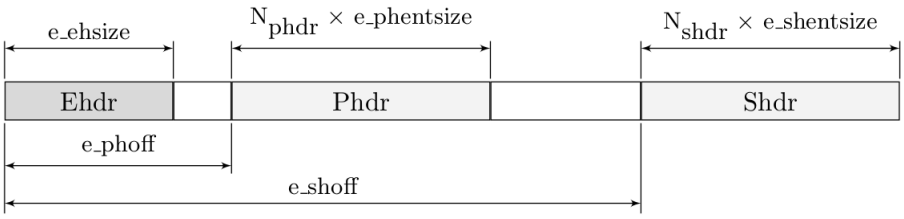

Figure 2.11: The ELF executable header describes the layout of the object

- Execution View: The execution view it's used by the loader in order to access the piece of informations

The Linux kernel provides ELF format support in the file `/FS/BINFMT_ELF.C`, whereas the function `LOAD_ELF_BINARY` takes care of correctly loading the binary file such that the program will be properly executed on top of the operating system. The `LOAD_ELF_BINARY` function starts by examining the `ELF` executable header to check that the file in question is indeed consistent to the `ELF` binary format. In order to do so, the whole `ELF` executable header structure, as shown in Figure 2.11, specified at the beginning of the file, is completely scanned.
It is important to the loader to read the `ELF` header as first thing because it is the only structure inside the binary which is guaranteed to be found at a fixed position. The `ELF` header provides indeed a complete description about the whole file organization.
All other components, as shown in Figure 2.11, can be found separated by different offsets which are specified in the `ELF` header table. These offsets can change from a binary file to another.

The Executable Header is mainly used to check meta-informations about the elf file itself and then to access the Program Header Table or the Section Header Table according to the reader's goal.
These two tables, even if they both share information about the binary, have different purposes:

- A program header table, if present, tells the system how to create a process image. The program header table is a contiguous array of program header table entries, one entry per segment.
  Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. This table is accessed by the `LOAD_ELF_BINARY` kernel function to load the required process informations and then start building up the `TASK_STRUCT`, representing the process record in Linux.

- A section header table contains information describing the file's sections. The

section header table is an array of section header table entries. For linking, data in an ELF object is grouped into sections.
Files used during linking must have a section header table; other object files may or may not have one. Sections hold most of the object file information for the linking view: instructions, data, symbol table, relocation information, and so on.

When the kernel loader finishes scanning all the required program segments, the kernel representation of a process, which is a structure TASK_STRUCT, is ready to be executed.

Chapter 3

State-of-the-Art

This Chapter describes the state-of-the-art regarding all main contexts where PMC are used with purposes different from the ones they have been designed for. Section 3.1 explains how PMCs can be used in an offensive way to enforce side channel attacks, while Section 3.2 and 3.3 describe two well know practical examples. The work presented in Section 3.4 uses PMCs in a different offensive scenario, featuring their sample functionality to implement rootkits. Section 3.5 explores instead, with a defensive approach, the use of performance monitoring counters to identify and mitigate ROP attacks. Section 3.6 leverages PMC to detect control-flow modifying kernel rootkits. Section 3.7 analyzes the presented state-of-the-art, discussing its advantages and limitations.

3.1 PMC as a tool for Side Channel Attack

Side channels are a powerful class of attacks that circumvents traditional security protections and access controls. In the typical side channel attack scenario there is a victim thread which is elaborating secret information and an attacker exploiting architectural hints to deduce the secret being processed by the victim program. Side Channel Attacks success relies on the fact that most of algorithm security's design doesn't take into account the process execution at micro-architectural level. Unlike traditional attacks that exploit vulnerabilities in what the system does, side channel attacks allow information to be obtained by observing how the system performs. Every algorithm is indeed executed using architectural components that may leak some hint about the processed information: input data tend to change the execution characteristics of applications.
Side channel attacks are implemented, from a high-level point of view, through three phases:

1. Observation of the architectural leak — Observation of a relationship between an architectural component execution and its input.

Thanks to this relationship it's possible to deduce the input of the leaky component or at least to make an educated guess.
In this phase the attacker collects several measurements in order to completely understand and characterize the leaky behavior.

2. Monitoring as the system operates with some actual data — The attacker collects statistics and other useful informations of the leaky component while the victim process is executing.

3. Retrieving to the secret — The attacker is able to deduce the secret by matching the victim process's statistics against the leaky component's behaviour.

In order to successfully build up a side channel attack, the following issues need to be addressed:

- Program Activity — The target program has to deal with secret data. The more its execution trace is data-dependent, the better it is for the attacker. If the program activity has low dependence with the sensible data, the side channel attack will result more difficult to implement.

- Side Channel Signal — the more information gathered by the attacker is related to the program execution trace the better it is. If the collected information has high values of noise it will be harder for the attacker to distinguish the useful data.

To summarize, side channel attacks, from an high-level point of view, are hard to be mitigated because an algorithm dealing with secret information needs to process it in a real world micro-architecture. A program that uses some data as its input will generate data-dependent activity in the processor and other system components. This data-dependent activity will create signals in various side channels. Data-dependent activity in the system cannot be avoided: even if the program's control flow does not depend on the value of the input there will be at least some wire or architectural component whose activity is input-dependent. It is possible however to design systems, programs and micro-architectures in order to reduce the magnitude of data dependent execution traces, but it is not possible to completely eliminate them.
In order to perform a side channel attack, architectural statistics while the victim thread is executing needs to be collected and analyzed. For this reason, side channel attacks usually meet the physical attack assumption: the attacker needs to physically connect her measuring tools to the target machine. PMCs are used to log detailed information about how the microprocessor components are used at runtime: this make them an extremely dangerous tool for performing side channel attacks. Performance events describe fine-grained, hardware statistics: they allow side channel attack to be performed under the software attack assumption: the attacker only

needs to place her malicious program inside the target machine. The malicious program will read performance counter values related to the underlying architecture, being able to access piece of information traditionally available only by deploying physical measurements. PMCs represent a valuable, easily accessible tool for performing a side channel attack against any kind of micro-architecture. In order to do so, the following inspection about the victim thread needs to be done:

- How is the sensible piece of information elaborated by the victim thread?
- Which are the architectural components involved in the computation?
- What kind of events are supported for those architectural components?
- Is it possible to use these events as hint for rebuilding the secret piece of information being elaborated?

If it's possible to answer these questions, it'll be easy to find a pattern inside the execution trace of the victim process. Finding such kind of pattern is the key for performing a successful side channel attack.

3.2 Branch Prediction Attack

Branch prediction attacks [10, 11, 24] allow exploit branch prediction information leakage of any victim program running on the same core of an attacker snooping thread. The shared micro-architectural component exploited in this attack is the Branch Prediction Unit: the victim application and the attacker compete for the shared *Branch Target Buffer* (BTB) by executing concurrently. The attacker aims to learn about the victim execution flow in terms of `TAKEN / NOT-TAKEN` branches. If the target task takes different branches according to the secret, the attacker can use this piece of information to deduce it. A Branch Prediction Attack relies on the fact that the BTB and the predictor, in modern CPUs, are coupled: only branches that hit in the BTB will be predicted, while a static prediction algorithm will be used on a BTB miss. The vulnerability is the deterministic behaviour of the branch predictor: a branch instruction pointing towards an address missing in the BTB will be predicted as NOT-TAKEN.
The spy process continuously executes a certain number of branches and then measure the overall time execution of these branches.
The number of branches is chosen such that, when it starts executing, the attacker will fill the whole BTB with its own target addresses.
The BTB is a cache structure, where a part of the branch addresses is used as cache index and the cache data is the last target address of that branch.
It is possible to understand the BTB organization, its number of ways and the address bits used as index. As shown in the work in [24], this can be achieved by

running some benchmarks on the target machine, by changing the address distance between branch instructions. In such a way the attacker can understand the underlying architectural features: it fills the whole BTB with its own branch addresses. Afterwards, the victim is scheduled: its execution starts and sooner or later a branch instruction will come. At this point, since the attacker has completely filled the BTB with its own addresses, the victim branch will be deterministically predicted as NOT-TAKEN. This prediction will bring into two different possible scenarios:

- Mis-prediction — The prediction was wrong and the branch is taken: one of the attacker target addresses will be evicted from the BTB such that the new target address of the victim process can be stored in.
  When the attacker thread will try to re-execute its branch, it will find a mis-prediction on the branch whose target address has just been evicted.
  This mis-prediction will require more cycles to be resolved.
  The spy thread is able to detect this situation and understand if the victim has taken a branch because it continuously keeps track of the elapsed time of its branches execution.

- Correct prediction — The prediction was right and the branch is not taken: the attacker's measurements won't change.

Knowing whether each branch was taken or not allows the attacker to draw conclusions about the secret information processed by the victim.
The work in [10] proposes a variant of this attack. It is said that choosing a number of branches implemented inside the attacker thread higher than the actual number of the BTB's ways it'll increase the accuracy of the measurements, for mainly two reasons:

- It guarantees the eviction of the target entry from all the possible places that can store it. This is a nice condition to work with, since the attacker is sure to find its branches mis-predicted when needed.

- It increases the differential time between the two scenarios where the victim branch is mis-predicted or correctly predicted.
  If the victim branch turns out to be taken, the mis-prediction will trigger many further mis-predictions since the entry of the evicted spy branch needs to be re-stored and another not yet re-executed spy branch entry has to be evicted.

3.3 Cache Side Channel Attack

Cache side channel attacks [13, 19] allow to exploit cache-based information leakage of any victim program, without requiring knowledge about its implementation. The shared micro-architectural component exploited in this attack is the cache: the

victim application and the attacker compete for the shared cache, by executing concurrently. The attacker's goal is to learn about the victim's cache usage by observing effects of the cache availability in its own program. The access pattern to different memory locations allows the attacker to draw conclusions about the secret information processed by the victim. This attack consists in two phases:

1. Profiling Phase — The victim program behaviour is studied: its activity and memory access patterns in response to any input are recorded inside a cache template matrix.
   This matrix stores cache-hit traces which are correlated to the victim's secret.

2. Exploitation Phase — The snooping thread runs concurrently with the victim. The attacker requests cache pages and measures how much time the system has taken to provide them: if it's passed too/too few much time this is implies that some of its cache pages have been used by the victim thread.
   The attacker can deduce the input by matching memory access patterns of the victim thread against its recorded behaviour analyzed in the profiling phase.

PMCs offer lots of core and uncore events related to caches access patterns. For a given process it's possible to monitor cache hits and misses and other kind of statistics. In [29] there are practical implementation on how to exploit performance monitoring counters to implement such attack.

3.4 PMC as a tool for Rootkits

The work in [26] uses performance monitoring counters to implement a new version of rootkits. Traditional rootkits overwrite the system call addresses redirecting them to malicious code such that when a user-space program invokes that system call, malicious instructions are executed. PMC rootkits achieve the same result without modifying the kernel image. This can be accomplished by moving the malicious code to the performance monitoring interrupt (PMI) handler, and using performance events to detect when a system call is performed. A preparation step in order to implement such attack is required: performance counters are set to sample user-space processes. Their count value is set such that, by counting the next event occurrence, the counter will overflow and trigger the PMI. To conclude this preparation step, the attacker is required to get on the target machine the privileges for hooking malicious code as PMI handler.
The malicious code takes care of resetting the counter values to their initial ones, such that another PMI will be invoked upon the next system call invocation. The authors of [26] have found a way to deterministically identify system call on Intel architectures using performance events. The selected event is `BR_INSTR_RETIRED.FAR_BRANCH`, counting the number of 'far' branches being used: a far branch indicates when a

call/return is using a far pointer, which is the case of the `SYSCALL` instruction, taking a far jump towards ring 0 kernel code.
In order to filter out noise regarding all the other user space instructions and retain only those corresponding to the system calls, the EXCLUDE_USER parameter is set. Furthermore, a second event code was identified to be capable of counting the same far branches in those processors supporting Intel's Last Branch Recording (LBR). The event code `ROB_MISC_EVENTS.LBR_INSERTED` counts for every branch that is inserted into the LBR. Combining this with the LBR filter that only logs far branches occurring in ring 0, the obtained result is the same as the one using the far branch event. The advantages of this attack is that no modification of the kernel image is required. On the other hand, there is no form of persistence: the above mentioned preparation step must be done upon each boot on the target machine. While this project is unlikely to become a serious threat, this is a clear example of the PMC's native expressibility: it is possible to trap the single system calls.

3.5 PMC as a tool for ROP mitigation

Return-Oriented Programming (ROP) is a software exploit for compromising the system, leveraging malicious code execution already residing in the process's memory space. The adversary combines short instruction sequences (Gadgets) from different locations in memory, where each sequence ends with a return instruction that enables the chained execution of multiple gadget sequences.
Each gadget performs an atomic operation of the malicious payload. This can be accomplished when the attacker finds a software vulnerability such as a stack-based buffer overflow to control and corrupt the software program stack. The adversary then writes the addresses of the gadgets on top of the program stack in the order that reflects the execution of the malicious payload.
[30] proposes Sigdrop, a new approach for ROP detection leveraging performance monitoring counters. In this work, performance statistics bound to ROP execution are identified and actively used to detect such attack at runtime. The used functional unit is the *Return Address Stack* (RAS), a last-in-first-out hardware stack that stores predicted target addresses of return instructions. The processor manages the RAS based on the assumption that each call instruction has an associated return instruction: when a call instruction is fetched, the processor pushes the address of the next instruction on the RAS. When a return instruction is fetched, the processor pops the RAS to predict the target address of the return. The RAS unlikely mis-predicts the target address. [30] observes that when a return oriented programming attack is happening, there will be a lot of mis-predicted returns: exactly one for each gadget. This happens because ROP breaks the above mentioned normal RAS functioning and the target address will not be on top of the return address stack. Sigdrop implements

a kernel subsystem that extracts runtime signatures of the monitored program and, by analyzing them, detects whether a ROP attack is happening. In detail, Sigdrop monitors the number of total instructions and return instructions in counting mode, and monitors the number of mis-predicted return instructions in sampling mode. After having collected all the performance data, the ROP classification is performed by checking the gathered performance against a certain threshold.

3.6 PMC as a tool for Rootkits mitigation

The work in [31] presents NumChecker, a Virtual Machine Monitor (VMM) based framework to detect control-flow modifying kernel rootkits in a guest Virtual Machine (VM). NumChecker leverages PMCs for the system security purpose: it detects malicious modifications to a system call in the guest VM, by checking the number of certain hardware events that occur during the system call's execution. To automatically count these events, NumChecker leverages the PMCs using the `PERF_EVENT` subsystem.
In this work, three different performance events are used to detect a malicious modification of the kernel flow: the total number of retired instruction, retired returns, retired branches.
The rootkit detection is "execution-oriented" and has two phases:

- Offline profiling — The system calls of the trusted guest OS are measured. To improve the accuracy of the measurement, the execution of a test program is repeated several times. When the measurement is complete, the system call interception is disabled. The results are stored as the "clean copy" to be used at runtime.

- Online Checking — The system calls of a running monitored guest OS are measured and compared with that of the corresponding trusted OS. To detect control-flow modifying kernel rootkits, NumChecker focuses on validating the execution of system calls.
  If the control-flow of a kernel function is maliciously modified, the number of the hardware events that occur during the execution will be different: Numchecker checks, in percentage, the total number of PMCs statistic deviations from uninfected executions.

Because the events are automatically counted by the PMCs, the work in [31] provides a solution for rootkit detection with low overhead.

3.7 Beyond the State-of-the-Art

By analyzing the state-of-the-art presented in this Section, several considerations can be expressed. PMCs can be used as an offensive tool leveraging the side channel attack approach, or as defensive tool to prevent malware attacks. In the first case, PMC extend the attack range of side channel attacks, allowing the attacker to gather low-level architectural statistics which can be analyzed to deduce a secret being elaborated by a target process.In the second scenario instead, PMC statistics are used to recognize the execution of a particular pattern of instructions, which is known belonging to a malicious behavior. In this case, the state-of-the-art projects build up a performance signature of the target malware, which is used by the operating system to detect and consequently terminate its execution at runtime. However this approach leverages those performance bound to the execution of a specific piece of code, without taking into account the whole target process execution. Malware detection at runtime is performed by simply checking the gathered performance against an expected threshold.
We propose a new approach to this methodology. In this thesis we present a new way of creating a process signature, which takes into account the whole execution of the target task. This novel approach provides the following advantages w.r.t. the state-of-the-art:

- Generality — the performance signature is built on top of the whole process execution. This can be done with any generic target process. We adopt machine learning techniques to classify different processes, which is a more general solution than checking expected performance statistics against a given threshold.

- No access to the code — By monitoring the whole process execution, there is no possibility to access the single instructions to detect some micro-architectural specific pattern to be monitored. We build up a process signature considering the target process as a 'black box'. The state-of-the-art projects instead build up the target process signature with an in-depth analysis of its code, to find a pattern to build the signature.

The next Chapter will introduce the proposed methodology.

Chapter 4

Proposed Methodology

In this Chapter we propose a new methodology to classify different programs execution using their micro-architectural performance. In order to reach our goal, we develop a process signature using PMCs statistics and we inject it in the program binary as a section. We have modified the Linux kernel loader to recognize this special section and to provide an automatic way of performing classification according to the read signature. The classification is done by a SVM algorithm, which we have re-implemented with fixed-point arithmetic in the Linux kernel. In Section 4.1 we define the goal of this thesis, showing its advantages w.r.t. the state-of-the-art. Section 4.2 describes the core implementation of our work, discussing how we have modified the kernel and the classification part. Section 4.3 gives an overview about the performance signature representation, both in kernel and in user-space.
Eventually, in order to develop an efficient performance signature, we have taken into account all the events advantages and disadvantages as discussed in Section 4.4.

4.1 Thesis Goal

The main goal of this thesis is to classify different processes according to their runtime behaviour, measured using performance monitoring counters. In this thesis, we present a novel approach of creating a process fingerprint, which takes into account the whole execution of the target task. This new way of building up a process signature provides several advantages w.r.t. the state-of-the-art works, that, as presented in Chapter 3, build up a task fingerprint using ad-hoc architectural features, recognized by an in-depth code analysis.
In this thesis we provide instead a broader solution for characterizing a generic process execution, without any knowledge about the code and assuming the task as a black box. As result, we obtain a signature which is intrinsically bound to the whole target process. We prove the effectiveness of our approach by training a machine learning algorithm to classify a process against the other ones, using as solely in-

formation its signature. The trained model is able to correctly classify the target tasks with high accuracy (as shown in section 5.3, above the 99%). We train it with different tasks obtaining high quality results. This demonstrates the efficiency of our solution, which achieves our goal.

4.1.1 Motivations

In order to achieve the goal of this thesis we have chosen to use performance monitoring counters for the following reasons:

- Generality — We would like to exploit performance monitoring counters that are present in almost all modern CPUs. As described in Chapter 2, four different version of PMUs have already been released, and the next process generations are really likely to be equipped with such components.
- Non-Intrusiveness — Performance monitoring counters directly monitor the performance the functional units without dealing with the processor complexity, which involves hierarchical caches, simultaneous multi-threading and out-of-order execution.
- Low Overhead — Performance monitoring counters represent the least invasive way of monitoring a process execution. As already demonstrated in [32], the monitoring introduces low overhead.

Our work improves the state-of-the-art under the following aspects:

- Generic Process Monitoring — the process is assumed to be a black box, for which we measure the performance, without any insight of the process source code functionality and microcode architecture. We do not classify it by simply comparing its performance statistics against a given threshold, but we use machine learning algorithms to solve the classification problem in a general way.
- Fine-grained monitoring accuracy — Even under the assumption of monitoring a generic process, we monitor its performance with an high level of accuracy. The result is a signature intrinsically bound to its target task.

4.2 In-Kernel Performance Monitoring

4.2.1 Motivation

The goal of the proposed methodology is to develop an automatic framework categorizing different programs according to their runtime behaviour. We have chosen to develop our solution in-kernel, using some of the `PERF_EVENT` provided functionalities. The reasons behind this choice are the following:

- System-level control — The kernel is the most correct place to develop a framework providing monitoring for a generic process running in the system.
- Virtual Counters — We use the kernel level virtual counters such that we can develop our work without assuming any knowledge of hardware-related issues like preemption and physical registers overflow.

The goal is to classify a process according to its runtime behavior in the kernel. As presented in Chapter 2, there are different ways to control PMCs, but only the sampling functionality is the one useful to our purposes because:

- The samples describe the process runtime behaviour.
- We can perform performance analysis while the target task it's running.

This cannot be achieved if we collect only the final counter values.

4.2.2 Proposed Approach

The goal is to implement an in-kernel classification mechanism for a target process running on the system. In order to achieve it, we have re-implemented all the fundamental system calls dealing with the `PERF_EVENT` subsystem because it is not possible to make a system call from the kernel itself because this is against the Linux kernel principles. We have modified the `PERF_EVENT` subsystem to provide special monitoring from the kernel. In particular, as discussed in the above subsection, we have chosen to sample the target task using a group of events.
We set the leader event as the sampling one to count on `PERF_COUNT_HW_CPU_CYCLES`

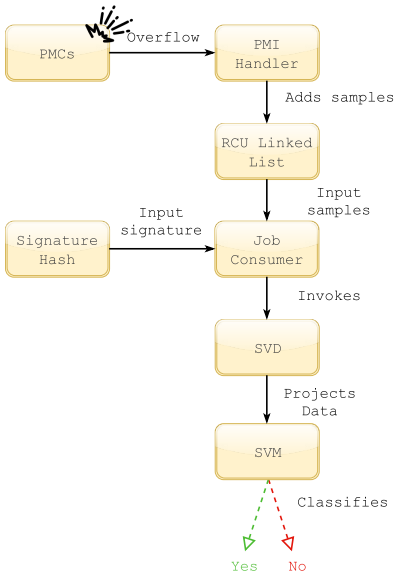

Figure 4.1: The Proposed In-kernel Classification Mechanism

occurrences and the slaves to the hexadecimal numbers specified in the process's performance signature. We have also changed the default sampling functionality in the kernel: it is too much oriented for user-space usage. It collects the samples into a circular buffer which is intended to be shared with the user. For in-kernel sampling this is useless: we have modified the overflow handler to support a different way of gathering samples. The overflow handler's main functionality is implemented in the function `PERF_EVENT_ GROUP_READ`. Inside this function, we have added some code to save the event count inside the `PERF_SAMPLES` vector, which is defined in the `TASK_STRUCT`. Upon event overflow, another couple of variables, `SAMPLE_POSITION` and `SAMPLE_COUNT` are updated: the first stores the position index corresponding to the next free place available in the array, the latter keeps track of how many samples are being gathered. When SAMPLE_COUNT reaches the threshold limit, the function `ADD_JOB` it's called, copying the entire array in a job list featuring Read-Copy Update (RCU) mechanism, as can be seen in the Figure 4.1.
The added jobs are periodically taken by the `JOB_CONSUMER` function which acts as a consumer, by retrieving the performance signature from an hash structure and performing classification.
The RCU mechanism in this scenario is a powerful solution: it allows the overflow handler to insert a new element in the job list without waits, while the whole structure is always synchronized with the consumer of such elements. As shown in Figure 4.1, the `JOB_CONSUMER` projects the performance data into the new SVD dimensional space and gives it as input to the kernel-implemented SVM algorithm, which we have developed taking as reference the [15] C implementation. The classification part of the algorithm has been re-implemented using fixed-point arithmetic, following the Linux kernel principles described in Section 2.9, and then injected in the kernel.

4.2.3 Performance Monitoring Initialization

The core system implementation described in the subsection above it's initialized when a process with signature invokes the `FORK` system call. We have decided to monitor forking threads because the idea, as explained in Chapter 5, is to detect whether a remotely logged user in the machine using `SSH` is instructing malicious code: in that case the system using our solution is able to detect its malicious intent. Since remotely logged users use the `SSH` utility, we plan to attach a performance signature to the `SSHD` daemon.
When the user issues an order, the daemon forks a new thread to execute it: we start monitoring that command behavior.
In order to monitor the forked processes of the target task, we have modified the `FORK` system call, which is defined in the file `[SRC]/KERNEL/FORK.C`. As shown in Figure 4.2, whenever a process makes the FORK system call, the `_COPY_PROCESS` function is invoked, which creates a new process as a copy of the old one, but without actually

starting it yet. We have identified the execution of this function as the correct moment to check for the presence of a signature in the forking current process: if that is the case the kernel has to provide monitoring support for the thread to be executed. The control is achieved by checking the integer value `HAS_SIGNATURE` inside the current `TAST_STRUCT`. If that's the case, the process signature of the current task is copied in the forked son. Furthermore,as shown in Figure 4.2, we have created a special kernel thread called `PERF_MONITOR` which is started when the `_COPY_PROCESS` function terminates. This invokes all the required functions to correctly initialize the in-kernel performance monitoring.
We have implemented a special synchronization mechanism to be sure of monitoring the target task from the very beginning of its execution. In order to do so, the function `PERF_MONITOR` awaits for a special completion defined in `_COPY_PROCESS`, which is released when the task is ready to be put in the runqueue. At that point its `PID` is valid and `PERF_EVENT` can properly accept it for the monitoring purpose. `PERF_MONITOR` interacts with the `PERF_EVENT` subsystem to monitor the target task in sampling mode. As sampling event we have chosen a time event, whose hexadecimal is specified by the constant `PERF_COUNT_HW_CPU_CYCLES`. Then, we open up all the needed virtual counters, whose hexadecimal value is taken directly from the performance signature. `_COPY_PROCESS` meanwhile awaits this initialization phase to be completed. When `PERF_MONITOR` finishes, it commits its completion and allows `_COPY_PROCESS` to conclude its execution, by kicking the target process in the runqueue. Thanks to this design we are able to start the monitoring before the target task is executed in the system, and we are sure to gather performance data from the very beginning of its execution.

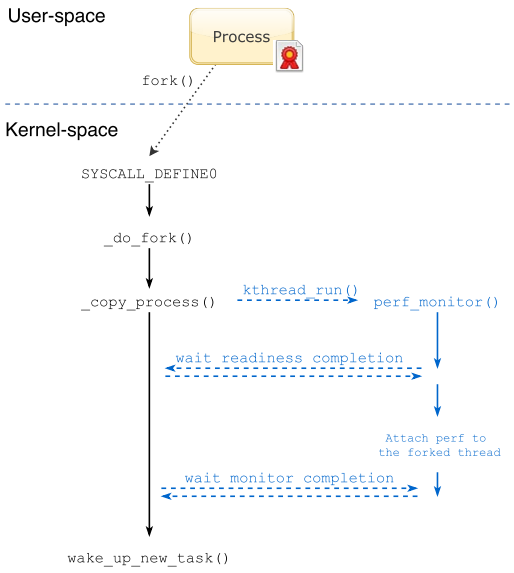

Figure 4.2: Fork in the modified kernel

4.2.4 Singular Value Decomposition

Singular Value Decomposition (SVD) [33] is a Principal Component Analysis (PCA) technique that seeks a dimensional basis that best captures the variance in the data. We have adopted SVD to preprocess the performance data of the target task because by performing feature selection we can analyze the data in a lower dimensional space.

Overview Given a data matrix, the problem solved by SVD is to find orthogonal vectors (also called principal components) that can be best used to represent data. PCA is a commonly used technique to preprocess data allowing the user to perform feature selection. As result, the data to be analyzed will contain less features that preserve the majority of information. The assumption that the user has to take into account before applying SVD is that the variance in the dataset represents information. This technique can be used over numerical data only. The advantages of SVD is that it allows to convert an high dimensional dataset into another one with less features, also reducing the noise in the data. On the other hand, the drawback regards the selected dimensional basis: the new features are a linear combination of the previous ones and it will be no longer possible to give semantic meaning to them. Performing feature selection using the singular value decomposition technique involves the following steps:

1. Normalize input data

2. Select the direction with the largest projected variance: this is the first principal component and data will be projected along it.
   After this, another orthogonal direction that captures the second largest projected variance is selected and called as second principal component, and so on. The input data is projected along the principal components, which results a linear combination of the input features.

3. The principal components are sorted in order of decreasing strength.

4. The input matrix A is decomposed into the multiplication of three different matrices: $A = U\Lambda V^T$, where:

   - A stores the original data
   - U represents the data projected using the new features
   - Λ is a diagonal matrix whose elements represent the strength of each feature. Each of its element is called singular value.
   - V^T represents the projection in the new feature space.

- U, Λ ,V are unique.
  U, V are column orthonormal, i.e., columns are unit vectors, orthogonal to each other.

5. A new dataset with less features can be obtained by eliminating the weak components, i.e. those singular value with low variance.

4.2.5 Support Vector Machine

Support Vector Machine (SVM) [33] is a supervised machine learning algorithm formally defined by the optimal hyperplane separating the labeled data taken into account. We have chosen to use the SVM algorithm to perform in-kernel classification of the running process, because the classification part of this algorithm is fast: it simply has to understand where the data point to be classified lies w.r.t. the separating hyperplane.

Overview As shown in Figure 4.3, the goal of an SVM is to select a hyperplane $w.x + b = 0$ that maximizes the distance γ between the hyperplane and any point of the training set. In the perfect scenario all the training points are as far from the hyperplane as possible, while being on the correct side of that hyperplane. An added advantage of choosing the separating hyperplane to have as large a margin as possible is that there may be points closer to the hyperplane in the full data set but not in the training set. If that's the case, there is a better chance that these points will be classified properly. In this thesis we train the SVM and analyze it by partitioning our data in the following sets:

- Training Set — dataset used to train the model.
- Validation Set — dataset used to estimate how well your model has been trained.

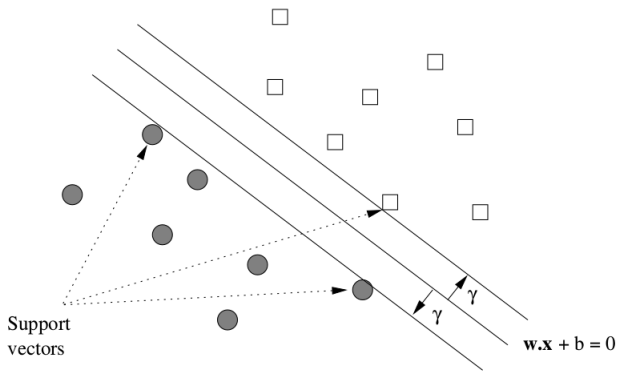

Figure 4.3: Support Vector Machine

- Test Set — dataset used to estimate the model accuracy.

The data given as input to the SVM may be linearly separable or not by its own characteristics.
If the data is not linearly separable, the following precautions can be adopted:

- Kernel Function — a kernel function projects data points in a high dimensional space where they become linearly separable by the hyperplane. When projecting data points, the 'Curse of Dimensionality' issue needs to be addressed: as the number of dimension increases, the number of chosen support vectors increments too. It may be possible that the model identifies too many support vectors, over-fitting on the training data.

- Slack variable — the SVM algorithm is allowed to find an hyperplane which doesn't divide all the data points exactly. In this scenario, a slack variable is introduced, which penalizes those hyperplanes which divide data points located in the wrong side of the hyperplane.

4.3 Process Signature Structure

In this Section we describe two different kinds of signature, which are used by the kernel to fulfill different operations. We then show how the signature is injected as a section in the binary of the target process, and how we have tweaked the kernel loader to recognize this special section. Eventually we give an insight on how we represent such signature in the Linux kernel.

4.3.1 Data signature

The data signature contains the information about the hexadecimal representation of the events we want to sample our process with. This is implemented by defining the signature structure the following way:

- `EVENT_HEX` — integer array of `PERF_COUNTER_NUMBER` of elements, representing how many counters will have to be configured to enable the process monitoring. In each array cell is specified the hexadecimal number that has to be assigned to PMC.

We tweak the kernel to save performance data of those processes equipped with this data signature. When the kernel finishes sampling the process, it creates a special file with `DEBUGFS`. `DEBUGFS` is a special file system available in the Linux kernel designed for sharing kernel information to the user-space. Using this mechanism, we are able to analyze the retrieved performance with user-space tools.

4.3.2 Performance Signature

The proper performance signature contains the following informations:

- `HEX` — Array of the event hexadecimals to be monitored.
- `SUPPORT VECTOR MODEL` — model of our trained and tuned support vector machine. This will be used to perform in-kernel classification.
- `V_REDUCED` — Before training the support vector machine algorithm, we perform feature selection.
  Performing principal component analysis with SVD has many advantages but constrains us to insert inside the signature the V matrix, representing the linear transformation of the input features, as presented in Chapter 2.

4.3.3 Signature Injection

The data and performance signatures defined in the previous subsection share the same injection mechanism, shown in Figure 4.4:

1. As first step we store all useful informations in a user-space `C` structure and we save the signature content in a binary file
2. The `OBJCOPY` utility is invoked to inject the binary-translated signature inside the program `ELF` file as a section: the elf binary file now contains a new section and it's ready to be read by the kernel loader when the process will be launched.

4.3.4 Kernel Signature Representation

As described in Chapter 2 the function `LOAD_ELF_BINARY` loads the elf binary in the kernel, eventually building up a runnable `TASK_STRUCT`.
We have modified this function to detect our injected performance signature and, if that's the case, to insert in the `TASK_STRUCT` all the required information for our

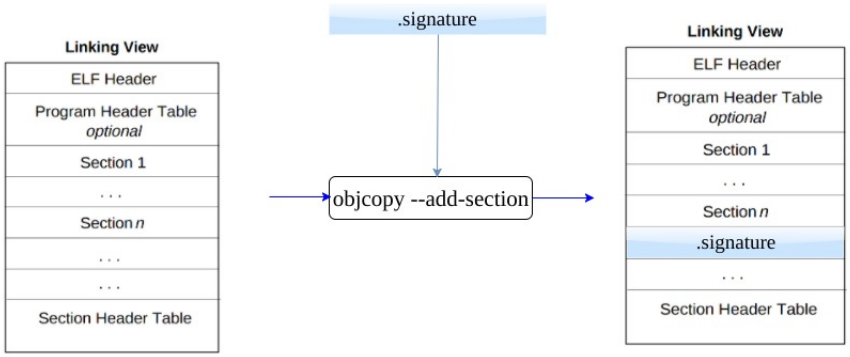

Figure 4.4: Signature Section Injection

special-purpose performance monitoring.
We have added the following fields to the task structure to keep track of the performance signature information:

- `HAS_SIGNATURE` — points out whether a process has a performance signature or not.
- `NEEDS_MONITOR` — indicates whether the process needs monitoring. In our work only forked thread are monitored: we use this flag to distinguish between the father and the sons.
- `EVENT_HEX` — integer array of `PERF_COUNTER_NUMBER` of elements, representing how many counters will have to be configured to enable the process monitoring. In each array cell is specified the hexadecimal number that has to be assigned to PMC.
- `VT` — V matrix transposed obtained using singular value decomposition when training the support vector machine.
  This is useful to project the new retrieved data in the same feature space the support vector machine has been trained in.
- `STRUCT SVM_MODEL SVM` — This represents the whole support vector machine model. It contains the support vectors, their labels and all the required parameters.
- `PERF_SAMPLES` — unsigned integer vector 64 bit width of dimensions `PERF_COUNTER_NUMBER * PERF_SIGN_SAMPLES_NR`
- `SAMPLE_COUNT` — this integer value stores the actual count of retrieved samples for the considered task.
- `SAMPLE_POSITION` — this integer value stores the position in which the next sample value has to be positioned inside the PERF_SAMPLES vector

When the kernel loads the first process with a signature, an hash structure using as key the process ID containing the signature is initialized.
Then, every time a process with signature is loaded in the kernel memory, its signature it's added in the hash structure.
This will be retrieved during the actual process execution in order to classify the current process performance against the expected ones.

4.4 Event Selection Criteria

Our goal is to develop a signature from the piece of information gathered through sampling the target process.

We decide to use the `PERF_EVENT` sampling functionality because it allows to gather runtime statistics.
In order to build up a process signature, all event's limitations and advantages have to be analyzed: we want to select the ones providing us as much information as possible.
The ideal event we are looking for is the one allowing to detect a unique pattern in the execution of the target task.
When building up a process signature, no matter what kind event is chosen, the result will always be dependent on the following:

- Instruction flow — The instruction flow influences the number and the kind of retired micro-operations: if a branch is taken or not, different instructions are executed.
  This factor influences the most of the performance events: when different instructions are issued, different performance for the total execution time, memory subsystem and pipeline related events are strongly influenced.

- Input data — A process execution depends upon its input data. This influences the total execution time, as well as the instruction flow described above. When monitoring a process whose execution heavily depends upon its input, this issue needs to be taken into account.
  On the other hand, however, this opens up the possibility to understand a process input data by using the solely information of its performance. While this doesn't cover the generic scenario, a practical example is given in Section 6.2.1.

- Noise — Since the performance data is collected using hardware-implemented real counters, these are inevitably subject to noise.

Approaching the problem from a generic point of view, building up a signature for a generic process is not trivial. The above mentioned characteristics change according to the selected process: this is the main reason why a unique signature for a complex program providing different functionalities may not be enough. In practice, however, processes happen to implement certain patterns in their execution: it that's the case, building up a process signature becomes feasible. In order to develop a process signature capturing a micro-architectural performance pattern, the semantic of each event and all its limitations have to be taken into account. Section 2.4 describes how performance events can be grouped under different criteria according to the functional unit they monitor. From that list, the most useful events for the purpose of this work are those belonging to the 'Program Characterization group. These events are powerful because:

- Related to the retirement phase — a downside of performance monitoring regards speculative execution. If performance events regarding the issuing and execution of microinstructions are selected, the measurements will also consider speculative µOPs. Statistics not reflecting the actual process execution will be collected, which adds up noise. This cannot be avoided because at hardware level the counter cannot distinguish between a speculative instruction which was intended to be executed or not.

- Related to the assembly — assembly instructions of a target process are traditionally used to develop a static signature for that program. When we select events describing the kind of micro-operations that have retired, we are providing the dynamic counterpart to such analysis. This is powerful because we can build up a signature based upon the target process internal code without actually inspecting it.

- Less vulnerable to noise — the total execution time, cache hit/miss and many other events are related to the CPU condition before launching the monitored program. This events instead are bound to the task itself and don't rely on the processor preconditions.

- Architectural independent — these events, even if listed as architecturally dependent in [21], in practice are available in most of the processors. The results of this work can be easily generalized with other architectures, which are really likely to provide the same events.

All the above mentioned considerations are valid for the purpose of this work, which is to build up a process's signature. If performance monitoring counters are used for other objectives, it may be better to select different events. Other practical projects have chosen different events for achieving different goals:

- Energy Consumption: [25] shows that is possible to estimate the dynamic power consumption of a processor based on performance counters, within an error of 5%, using a subset of counters common in almost all architectures: fetched instructions, L1_Hit, dispatch stalls.

- Uops retired: [26] shows that is possible to implement a rootkit performance monitoring counter attack relying on the possibility to distinguish a system call (relying either on the `BR_INST_RETIRED.FAR_BRANCH` or on the Last Branch Retired (LBR) facility) performed by a function running in user space.

- Cache miss/hit: [29] makes use of cache events to develop cache side channel attacks.

- Branch Instructions: [10] and [30] leverage branch related events to respectively implement a side channel attack and to detect malware being executed in the system.

Here we list those events that have marked down as good candidates for developing a process signature, following the above presented considerations:

- `UOPS_ISSUED.ANY` : The number of uops issued by the Resource Allocation Table (RAT) to the Reservation Station.
- `ARITH.FPU_DIVIDER_ACTIVE`: Cycles when the divider is busy executing divide or square root operations.
  It accounts for floating point operations.
- `UOPS_DISPATCHED_PORT.PORT_[X]`: Counts the number of cycles in which a µOP is dispatched in the selected port.
  Each port provides access to different functional units. At each port is associated a different event.
  These events provide an insight on the set of operations being executed by the target task.
- `UOPS_EXECUTED.THREAD` — Counts the number of µOP that begin execution across all ports.
- `UOPS_EXECUTED_X87` — Counts the number of x87 µOP that begin execution.
- `INSTRUCTION_RETIRED.ANY` — Number of instruction at retirement.
- `BR_INST_RETIRED.ALL_BRANCHES` — Branch Instructions that retired.
- `BR_INST_RETIRED.CONDITION` — Counts the number of conditional branch instructions retired.
- `BR_INST_RETIRED.NEAR_CALL` — Direct and Indirect near call instructions retired.
- `BR_INST_RETIRED.NOT_TAKEN` — Counts the number of not taken conditional branch instructions retired.
- `BR_INST_RETIRED.NEAR_TAKEN` — Number of near taken branches retired.
- `BR_INST_RETIRED.FAR_BRANCH` — Number of far branches retired.
- `BR_MISP_RETIRED.ALL_BRANCHES` — Mis-predicted branch instructions at retirement.
- `BR_MISP_RETIRED.CONDITIONAL` — Mis-predicted conditional branch instructions retired.

- `BR_MISP_RETIRED.NEAR_TAKEN` — Number of near branch instructions retired that were mis-predicted and taken.
- `UOPS_DISPATCHED_PORT_X` — µOP dispatched in the selected port.
- `MEM_INST_RETIRED.ALL_LOAD` — All retired load instructions.
- `MEM_INST_RETIRED.ALL_STORE` — All retired store instructions
- `FP_ASSIST.ANY` — Cycles with any input/output floating point assist.
- `OTHER_ASSIST.ANY` — Number of times a microcode assist is invoked by the hardware other than FP-assist.

The next Chapter reports some experimental results by applying the proposed methodology.

Chapter 5

Experimental Results

This Chapter shows the experimental results of the proposed methodology, providing real-world case studies where we test the efficiency of our work. Section 5.1 describes the experimental setup required to reproduce the obtained results. In Section 5.2 we show and discuss how we have analyzed the experimental results, while in Section 5.3 we present practical application scenarios.

5.1 Experimental Setup

The target architecture is the 6th generation Intel processor's codename Skylake Intel(R) Core(TM) i5-6500 CPU. While the Skylake target machine's operating system is a Gentoo Linux distribution, we have run our modified kernel in an emulated virtual machine. The tool used for the virtual machine emulation was the `QEMU` utility, version 2.5.0. The emulation is performed with the Debian operating system version 9. In the Debian OS, we have installed the vanilla kernel version number 4.9.6 to run our experiments on (available at [4]). The Linux kernel in order to include the `PERF_EVENT` subsystem has been compiled with `CONFIG_PERF_EVENTS` on.

5.2 Experimental Results

Support vector machine is an offline machine learning algorithm, that we train with the retrieved performance data with user-space tools. Performance data is stored as a matrix whose columns represent the features and whose rows represent the different samples. By representing our data in this way we set the number of

Figure 5.1: Data Preprocessing

event occurrences registered within `SAMPLE_PERIOD` for a certain counter to be the features in the dataset. As shown in Figure 5.1, when analyzing the performance statistics of a running process, we prepare the data to be classified as follows:

- Noise — By collecting some samples, we notice that there is noise present in the form of outliers in the measurements. This is due to process preemption. We cannot use such raw data to train the SVM algorithm without first cleaning it. The noise is order of magnitude bigger than the sample value we want to monitor; for this reason we cannot choose to compute the standard deviation or mean values because they are sensible to such huge outliers. In order to have a real estimate of the true mean value we have chosen to compute the median value, taking into account the fact that the majority of samples isn't affected by noise. We then process the data in the following way:

  1. We compute per each feature its corresponding median value
  2. We parse all the sample data and check whether its distance w.r.t. the median is higher than several orders of magnitude. If that's the case, we substitute that noisy value with the median value itself.

  This filters out the noise in our data, resolving this issue.

- Numerical Data — All gathered data are numerical, since they represent the number of event occurrences registered in SAMPLE_PERIOD time. As preprocessing we perform normalization, such that all the features will have values on the same scale.

- Redundant features — PMC events monitor functional units performance in the very same architecture: the collected data is by its own nature correlated. In order to reduce noise and capture the redundancy in data, we perform PCA using SVD. It is important to remark that we assume the variance of our data to be information. This step only makes sense if noise removal is performed on raw data, otherwise the noise would be inadvertently misinterpreted as information.

5.3 Application Scenario

5.3.1 Application Scenario analysis

The solution we have proposed in this thesis can be adopted in all those scenarios where being able to distinguish between two programs at runtime is useful.
While this can be applied in a broader variety of contexts, we have declined our work for providing support in the following real-world scenario:

- Anomalous program behavior detection — A system modified with our solution is able to detect whether a program has been compromised or not while it's executing. When an anomalous application behavior is detected, the system can adopt any kind of management policy for those processes representing a threat. A trivial and efficient policy would be to suppress that process before it compromises the system.

- Semantic malware detection — Taking into account the common, real world scenario of a user remotely logged in a machine, our work allows to detect the execution of harmful process, and to adopt a customizable management policy when malicious code is issued. An example of this is shown in the case study 2, where we compare the execution of two different programs implementing completely different different tasks: the compression of a file (which is always unharmful) and the encryption of a file, which may be in practice used by a ransomware.

Our solution has the limitation of being deployable only with those electronic devices equipped with performance monitoring counters. This is not a problem, since as described in Chapter 4, almost all modern processors are equipped with PMU. Our application has instead the advantage of working under the following generic scenarios:

- Operating System — We have modified the Linux kernel, which is the core of every Linux distribution.
  Our solution can be used in all the available Linux distributions.

- Multitasking — The monitoring isn't affected by the execution of other processes: multitasking doesn't affect our measurements, thanks to the virtual counter implementation described in Chapter 2.

- Servers — Our solution is extensible to any server running with a Linux distribution.

- SO running on high end embedded devices — The widespread operating system Android in high-end embedded devices is built on top of the Linux kernel. For this reason our solution is extensible to such devices.

5.3.2 Case Study: Matrix Multiplication and Prime Numbers

In this case study we take into account two programs performing similar operations to demonstrate how much detailed and execution-bound the performance signature can be.
The matrix multiplication process multiplies two matrices in a non-optimized way, meanwhile the prime number program computes how many prime numbers there are

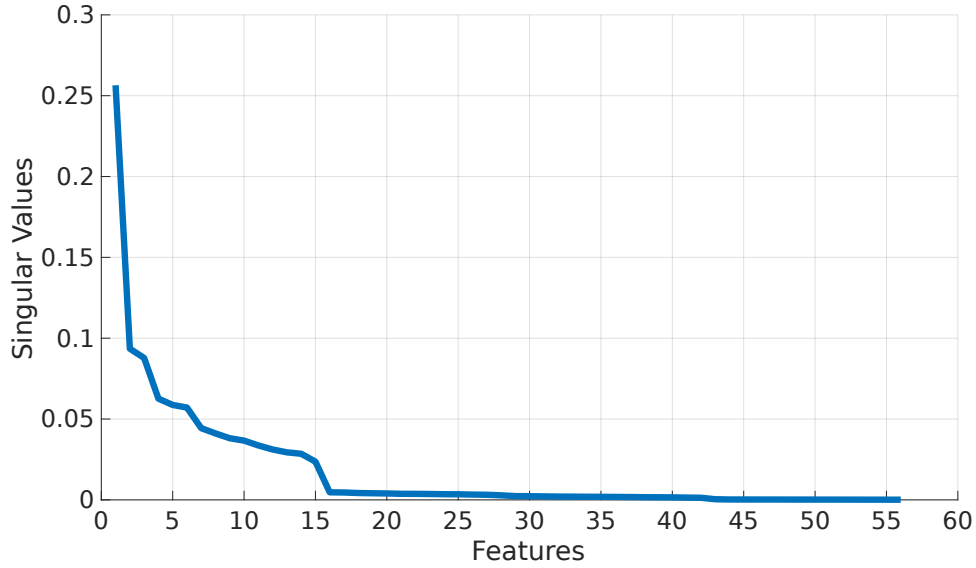

Figure 5.2: Singular Value Decomposition of Prime Numbers and Matrix Multiplication

under a certain specified threshold.
The selected performance events, chosen accordingly to the principles explained in Chapter 4, are:

- `BR_INST_RETIRED.ALL_BRANCHES` — Branch Instructions that retired.
- `UOPS_DISPATCHED_PORT_0` — Counts the number of cycles in which a µOP is dispatched on port 0.
- `INST_RETIRED.ANY_P` — Number of instruction at retirement.
- `MEM_INST_RETIRED.ALL_LOAD` — All retired load instructions.

We analyze the performance statistics and clean them from noise as explained in Section 5.2.
Afterwards we project our data using the singular value decomposition algorithm, obtaining the chart shown in Figure 5.2.

We plot on the x-axis the number of feature belonging to the dataset, while on the y-axis the value of the singular values. To each feature is associated the value representing how much information it holds. We use this chart to choose how many features to keep, taking into account the retained information. The more sloping there is in the graph between a feature and another the more it is convenient to include the next feature. By selecting 15 features we can keep more than the 99,95% of information: this is more than enough for our classification purposes. We select that value as threshold and discard all the remaining features, losing less than 0,05%

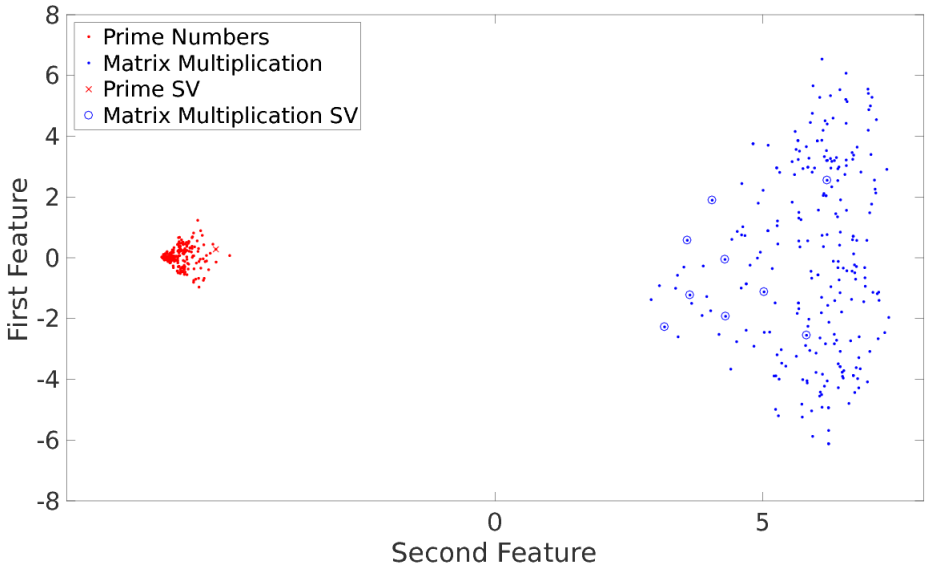

Figure 5.3: Clusters of Prime Number and Matrix Multiplication

of the information. We then train the SVM algorithm and inspect whether our projected data is linearly separable or not. The training data detains 250 positive and 250 negative instances, and results linearly separable using the linear kernel. Figure 5.3 shows the the support vectors identified by the SVM and the data points: they are grouped in distant clusters, which result linearly separable.

We report the confusion matrix in table 5.1, which describes the results obtained when testing the SVM with an independent test set (we can generate it by simply launching the target process and retrieving the provided samples):

Table 5.1: Prime and Matrix multiplication Confusion Matrix

	Predicted Yes	**Predicted No**
Actual Yes	9921	79
Actual No	20	9980

The cross-validation error, obtained by performing 10-fold cross-validation and providing an estimate on the test error, is 0.02.
As can be seen in table 5.1, our trained SVM performs on novel performance data with an accuracy of 0.99, a precision of 0.99 and a recall of 0.99.
These values are high enough to safely conclude that our classification algorithm achieves our goal.

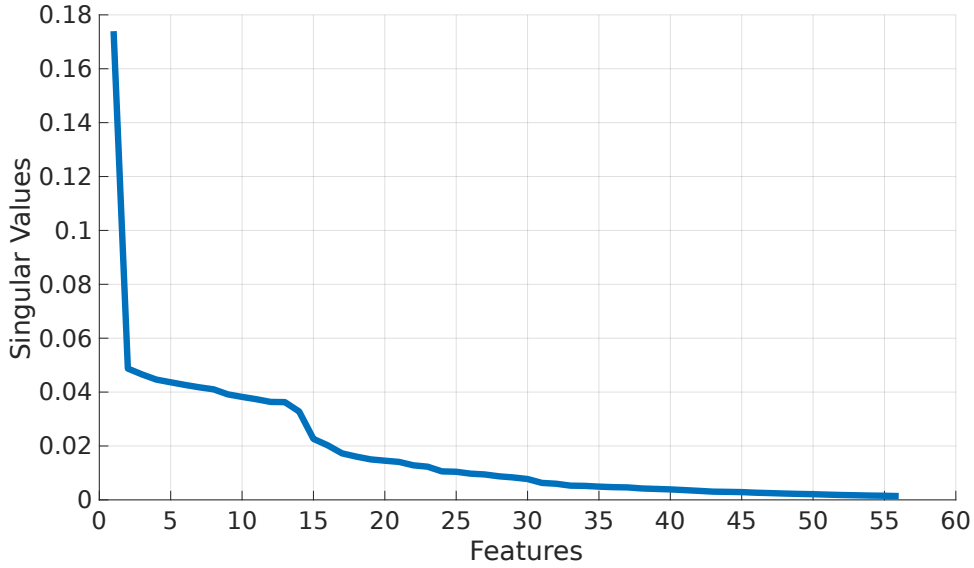

Figure 5.4: Singular Value Decomposition of TAR and RSA

5.3.3 Case Study: Tar vs Rsa

This case study takes into account a couple of more practical and commonly used processes: TAR and RSA (`OPENSSL` implementation [7]). We have chosen to monitor these programs because they are common utilities: we use the first algorithm to compress a file, the second one to encrypt it. This example, as explained in Section 5.4, provides a meaningful result for the proposed methodology in this thesis. The selected performance events, chosen accordingly to the principles explained in Chapter 4, are:

- `BR_INST_RETIRED.ALL_BRANCHES` — Branch Instructions that retired.
- `UOPS_DISPATCHED_PORT_0` — Counts the number of cycles in which a µOP is dispatched on port 0.
- `INST_RETIRED.ANY_P` — Number of instruction at retirement.
- `MEM_INST_RETIRED.ALL_LOAD` — All retired load instructions.

As described in Section 5.2, we find noise in the measurements and we clean our data before using it.
As shown in the Figure 5.4, we plot on the x-axis the number of features belonging to the dataset, while on the y-axis the value of the singular values.

To each feature is associated the value representing how much information it holds. We use this chart to choose how many features to keep, taking into account the retained information. The more sloping there is in the graph between a feature and another the more it is convenient to include the next feature. By selecting 20

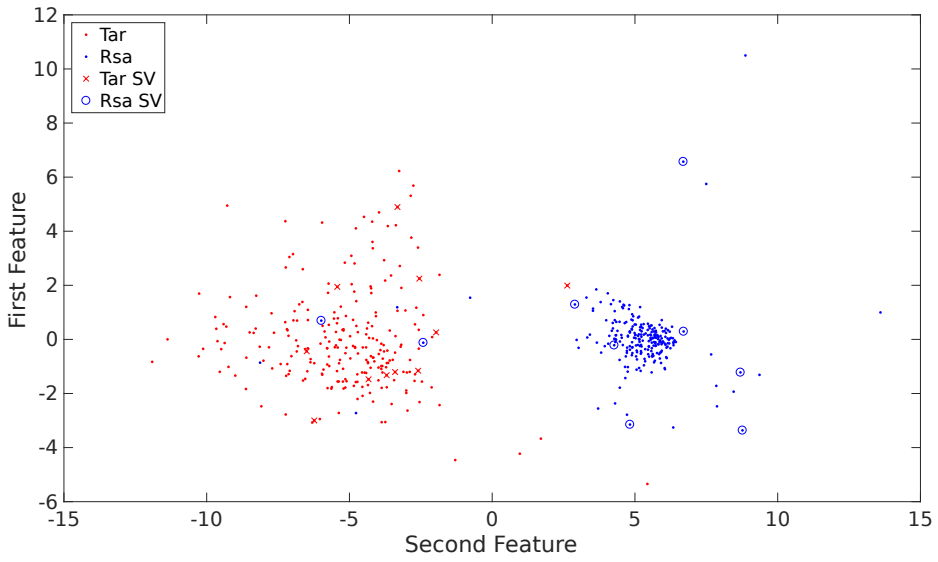

Figure 5.5: Clusters of TAR and RSA

features we can keep more than the 99,95% of information, which is convenient. We select that as threshold and discard all the remaining features, discarding less than 0,05% of the information. Figure 5.5 shows the support vectors identified by the SVM and the data points: they are grouped in different clusters. We train the SVM and we inspect the (10-fold) cross-validation error, which provides an estimate on the test error, which results of 0.02.

Table 5.2: Tar and Rsa Confusion Matrix

	Predicted Yes	**Predicted No**
Actual Yes	9918	82
Actual No	123	9877

From table 5.2, we can conclude that the trained SVM has an accuracy of 0.98 on 20000 samples extracted from an independent test sets, with a recall of 0.98. We can safely conclude that the solution we have proposed in this thesis can distinguish with an high level of accuracy the execution of Tar against Rsa.

5.3.4 Case Study: Detecting RSA

The goal of this last case scenario is to inspect whether our implemented solution is able to detect malicious behavior being issued by a remotely logged user via SSH. We monitor the execution of RSA being used for encryption (`OPENSSL` implementation [7]) against other generic process execution. The goal is to distinguish between the encryption of a file, which is potentially harmful, and other generic program ex-

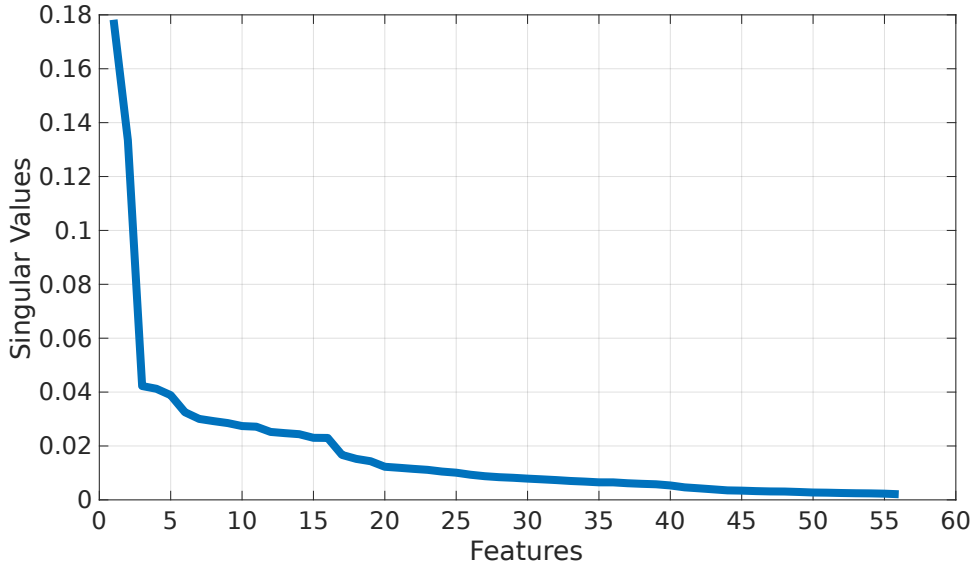

Figure 5.6: Singular Value Decomposition of RSA, Matrix, Prime and Tar

ecution, which does not represent a threat. The selected performance events, chosen accordingly to the principles explained in Chapter 4, are:

- `BR_INST_RETIRED.ALL_BRANCHES` — Branch Instructions that retired.
- `UOPS_DISPATCHED_PORT_0` — Counts the number of cycles in which a µOP is dispatched on port 0.
- `INST_RETIRED.ANY_P` — Number of instruction at retirement.
- `MEM_INST_RETIRED.ALL_LOAD` — All retired load instructions.

As described in Section 5.2, we find noise in the measurements and we clean our data before using it. We label the RSA algorithm as belonging to the positive class, while we assign the other general purpose programs with the negative label. The other programs we monitor as an example of utilities implementing a non-malicious behavior are Tar when compressing a file, the matrix multiplication and prime number finder we have described in the other case studies. We perform principal component analysis through SVD and we find a similar chart of the one presented in the other two case studies, as shown in Figure 5.6. We inspect the data to see whether is result linearly separable or not: in Figure 5.7 we report the cluster shapes of all the monitored utilities.

As can be observed from table 5.3, the trained SVM has an accuracy of 0.97 on 20000 samples extracted from an independent test sets, with a recall of 0.97. The (10-fold) cross-validation error, which provides an estimate on the test error, which results of 0.02. Taking into account that RSA presents an encryption behavior, operation performed also by well-known ransomware, we can conclude that the solution

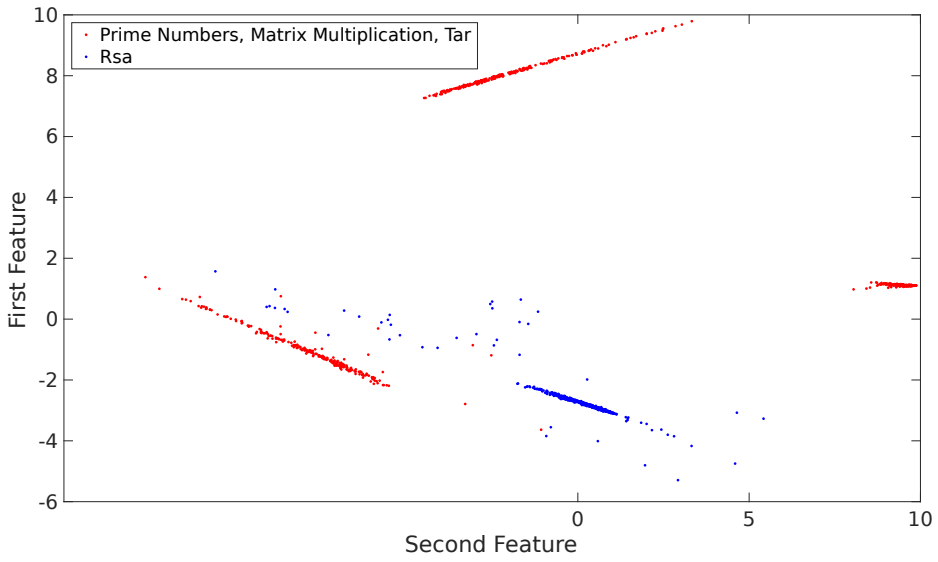

Figure 5.7: Cluster of RSA, Prime Number, Matrix Multiplication, TAR

Table 5.3: Confusion Matrix of RSA and Prime Number, Matrix Multiplication, TAR

	Predicted Yes	**Predicted No**
Actual Yes	9946	54
Actual No	627	9373

we have proposed in this thesis can detect malicious software being executed in the system.

5.4 Conclusion

In this Chapter we have presented the results of our work, taking into account real-world case studies to provide practical application scenarios.
In the first case study we have shown how much the performance signature is bound to its corresponding program, by monitoring similar applications. This demonstrates the advantages of selecting microcode related events, as we have discussed in Chapter 4.
The second case study takes into account a security scenario, considering a remotely logged user in the target machine with SSH that dispatches two different commands for compressing or encrypting a file.
This is an important example because as consequence of being able to classify such programs, we are able to tell whether a user is encrypting a file or compressing it. Taking into account that RSA performance represent an encryption behavior, oper-

ation performed also by well-known ransomware to extort money from the victim, being able to detect at runtime whether a program is running with a malware-like execution is an important and interesting result.
The third case study extends what we have have demonstrated in the second example. We still assume a remotely logged user in the target machine, but in this case he dispatches several different commands. We show that we are able to distinguish between the execution of a generic program non representing a threat for the system against RSA. This shows a practical application of our work for detecting malware executing in the system by using its micro-architectural performance.

Chapter 6

Conclusions and Future Work

This Chapter provides an overall analysis of the work proposed in this thesis. Section 6.1 gives a summary of the presented work, by analyzing the problems and the advantages of the proposed approach. Section 6.2 concludes the thesis providing suggestions on future directions of this work.

6.1 Summary, Benefits and Limitations

We have presented a novel approach to categorize different processes according to their runtime behaviour, measured using performance monitoring counters. We have analyzed the state-of-the-art, which uses PMCs as an offensive tool leveraging the side channel attack approach, or as defensive tool to prevent malware attacks. In the first case, PMC extend the attack range of side channel attacks, allowing the attacker to gather low-level architectural statistics which can be analyzed to deduce a secret being elaborated by a target process.
In the second scenario instead, PMC statistics are used to recognize the execution of a particular pattern of instructions, which is known belonging to a malicious behaviour.
In this case, we have improved the state-of-the-art by developing a performance signature for the whole execution of a generic process, without leveraging those performance bound to the execution of a specific piece of code.
Our work provides several advantages w.r.t. the state-of-the-art, allowing to build up a signature assuming the task as a black box and without analyzing its code. As result, we obtain a process signature which is intrinsically bound to the whole target program due to the fine-grained statistics offered by PMCs.
We have implemented the SVM algorithm inside the Linux kernel to classify a process runtime performance against the expected ones, which are injected in its signature and copied by the kernel upon execution of such program. As shown in Chapter 5, the trained model is able to correctly classify the target tasks with high accuracy,

demonstrating the efficiency of our solution. Our work's limitations regard the availability of performance monitoring counters on the target device, and also pertain the nature of the target task: if complex and heterogeneous functionalities are provided by the very same binary, one performance signature may not be enough.

6.2 Future Works

The solution proposed in this thesis allows to classify generic tasks by measuring their runtime performance usign PMCs.
This is a very general solution and it can be deployed for a great variety of improvements:

- Implementation of direct monitoring — We have tweaked the Linux kernel to provide an automatic performance monitoring mechanism for the forked processes of the target task. As described in 5, we have covered the most complex case meeting real world scenario whereas a remote user perform operations on the target machine. A further work in this direction is about implementing an automatic monitoring approach for those processes which aren't supposed to fork.

- Online machine learning algorithms — We have trained the SVM algorithm in order to perform classification of target processes, which has demonstrated extremely good results. In this direction it may be interesting to train other machine learning classification models to inspect their efficiency. Since the work proposed in this thesis provides samples of a task while it's running, it may be interesting to use online machine learning techniques too.

- Computational Resources Management — The fine-grained performance monitoring functionality provided in this thesis can be used to control a process's resources usage.
  In particular the user activity logged into the remote machine may be monitored, such that the issued jobs don't exceed a certain resource utilization threshold.

While the above mentioned projects extend our idea of monitoring a generic process without analyzing its internal code, another interesting approach would be to leverage those performance events counting microcode assists.

6.2.1 Reverse Engineering the Input using Microcode Assists

In this Section we explore the functional units exceptions related to the FPU to see how microcode assist can leak information about the considered functional unit execution. As described in Section 2.3, the execution units inside a core can receive

different input values. If this input assumes common values, it is elaborated using fast paths implemented in hardware. Any other given input which cannot be handled by fast paths is elaborated with the help of microcode assist. This special assist is issued from the functional unit and points the MSROM to a routine in microcode that will handle the special case. The use of microcode assist can be measured using performance counters. Some examples of special conditions for which µOP assist is requested are listed here:

- Floating Point instructions issue assist to handle denormalized inputs, as shown in Figure 6.1.
- `REP MOVS`: Rep operation use the `CX` register to determine how many times to loop the `MOVS` assembly instruction. The `MOVS` family assembly instructions copy variable sized arrays from a source to another. These instructions can handle small arrays in hardware, but issue microcode assists for larger arrays.
- `VMASKMOV` assembly instruction, which reads a series of data elements from memory into a vector register. While doing this, a mask register decides whether elements are moved or ignored.

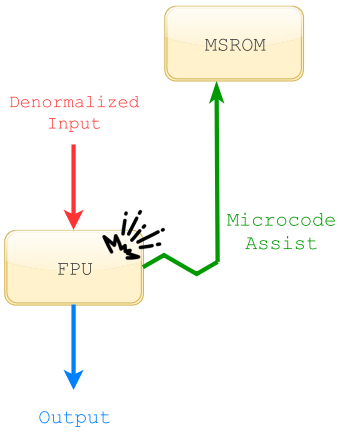

Figure 6.1: FPU exception handling

Floating Point Unit The floating point unit is an interesting functional unit inside the processor's architecture to be studied for its relation with microcode assist. The floating point unit's exceptions are the followings:

- `INVALID OPERATION` — this is signaled if there is no usefully definable result, such as zero divided by zero or infinity minus infinity.
- `DIVISION BY ZERO` — this is signaled when dividing a non-zero number by zero. The result is a correctly signed infinity.

- OVERFLOW — this is signaled when the rounded result won't fit. The default result is a correctly signed infinity.
- UNDERFLOW — this is signaled when the result is non-zero and between -FLT_MIN (the minimum representable number in the normal way) and FLT_MIN. The default result is the rounded result. which relies upon dealing with under-normalized input
- INEXACT — this is signaled any time the result of an operation is not exact. The default result is the rounded result.

The Skylake architecture offers the following events to monitor microcode assists:

- FP_ASSIST.ANY — Counts the number of floating point operations executed that required micro-code assist intervention.
- OTHER_ASSIST.ANY — All other microcode assists related to the remaining functional units in the architecture.
- X87_UOPS_ISSUED — Counts the number of x87 µOP dispatched.

We have built up some self-monitoring programs using the PERF_EVENT subsystem in user-space, and we have observed that the number of microcode assists that are issued to the FPU is different according to the input and according to the assembly instruction being used. This consideration may be used in future to understand what kind of input has taken a functional unit, by simply observing its performance.

Implementation We have developed two programs in the same way: they initialize the monitoring, reset the FPU status words (which keeps track of the exceptions) and enable the counter with the above mentioned events. In the first program we monitor the execution of x87 instruction, in particular the FDIVP operation, obtaining the following results:

- INVALID OPERATION, DIVISION BY ZERO, OVERFLOW, INEXACT — As documented, the number of issued micro-operations in such edge cases is greater than the normal. When these exceptions are thrown, exactly two x87 µOP are issued to handle the situation. Exactly one floating point assist is issued.
- UNDERFLOW — While, as expected, the issued micro-operations are greater than the normal, in the underflow scenario the number of retired x87 micro-ops is exactly 6.

Thanks to the above considerations it's possible to deduct some information about the input given to the floating point unit: we can tell whether the input triggered no exception, the underflow exception or all the remaining ones. In the second program we monitor a more modern assembly instruction: DIVSS

- `INVALID OPERATION, DIVISION BY ZERO, OVERFLOW, INEXACT` — As documented, the number of issued micro-operations in such edge cases is greater than the normal. No floating point assist is issued.
- `UNDERFLOW` — The issued micro-operations are greater than the normal. One floating point assist is issued.

This observation may be used in a future work to:

- Reconstruct data being elaborated in a target process.
- Find other, undocumented, microcode assists released by other functional units.

Bibliography

[1] Brendan Gregg's Blog. *http://www.brendangregg.com/*.

[2] How programs get run: ELF binaries. *lwn*.

[3] JEvents. Available at `https://github.com/andikleen/pmu-tools/tree/master/jevents`.

[4] Linux Kernel. Available at `https://kernel.org`.

[5] Linux Kernel Documentation. Available at `https://kernel.org`.

[6] Netflix Accounts for More Than a Third of All Internet Traffic. Available at `http://time.com/3901378/netflix-internet-traffic/`.

[7] Openssl.org. Available at `https://www.openssl.org/`.

[8] Perf Wikia. Available at `https://perf.wiki.kernel.org/index.php/Main_Page`.

[9] wikichip.org. Available at `https://en.wikichip.org/`.

[10] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *IACR Cryptology ePrint Archive*, 2006:351, 2006.

[11] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, pages 225–242, 2007.

[12] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.*, 2017.

[14] Robert Locke Callan, Alenka G. Zajic, and Milos Prvulovic. A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 242–254, 2014.

[15] Chih-Chung Chang and Chih-Jen Lin. LIBSVM – A Library for Support Vector Machines.

[16] Chih-Chung Chang Chih-Wei Hsu and Chih-Jen Lin. A Practical Guide to Support Vector Classification. page 16, 2016.

[17] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. 1995.

[18] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 897–912, 2015.

[20] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. (248966-018), March 2009.

[21] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. (248966-018), March 2009.

[22] Joseph Koshy. *Libelf by example*. 2012.

[23] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[24] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey H. Kulick. Microbenchmarks for determining branch predictor organization. *Softw., Pract. Exper.*, 34(5):465–487, 2004.

[25] Rance Rodrigues, Arunachalam Annamalai, and Israel Koren an Sandip Kundu. A Study on the Use of Performance Counters to Estimate Power in Microprocessors. *IEEE Trans. on Circuits and Systems*, 60-II(12):882–886, 2013.

[26] Matt Spisak. Hardware-assisted rootkits: Abusing performance counters on the ARM and x86 architectures. In *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, August 8-9, 2016.*, 2016.

[27] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[28] Brinkley Sprunt. The overhead of profiling using PMU hardware counters. 2014.

[29] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting Hardware Performance Counters. In *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, DC, USA, 10 August 2008*, pages 59–67, 2008.

[30] Xueyang Wang and Jerry Backer. SIGDROP: Signature-based ROP Detection using Hardware Performance Counters. *CoRR*, abs/1609.02667, 2016.

[31] Xueyang Wang and Ramesh Karri. Numchecker: detecting kernel control-flow modifying rootkits by using hardware performance counters. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 79:1–79:7, 2013.

[32] Vincent M. Weaver. Self-monitoring overhead of the Linux perf_ event performance counter interface. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 102–111, 2015.

[33] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, New York, NY, USA, 2014.