Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Detecting ROP with Statistical Learning of Program Characteristics

Mohamed Elsabagh melsabag@gmu.edu
Daniel Barbará dbarbara@gmu.edu
Dan Fleck dfleck@gmu.edu
Angelos Stavrou astavrou@gmu.edu

Technical Report GMU-CS-TR-2016-5

Abstract

Return-Oriented Programming (ROP) has emerged as one of the most widely used techniques to exploit software vulnerabilities. Unfortunately, existing ROP protections suffer from a number of shortcomings: they require access to source code and compiler support, focus on specific types of gadgets, depend on accurate disassembly and construction of Control Flow Graphs, or use hardware-dependent (microarchitectural) characteristics. In this paper, we propose EigenROP, a novel system to detect ROP payloads based on unsupervised statistical learning of program characteristics. We study, for the first time, the feasibility and effectiveness of using *microarchitecture-independent* program characteristics — namely, memory locality, register traffic, and memory reuse distance — for detecting ROP. We propose a novel directional statistics based algorithm to identify deviations from the expected program characteristics during execution. EigenROP works transparently to the protected program, without requiring debug information, source code or disassembly. We implemented a dynamic instrumentation prototype of EigenROP using Intel Pin and measured it against in-the-wild ROP exploits and on payloads generated by the ROP compiler ROPC. Overall, EigenROP achieved significantly higher accuracy than prior anomaly-based solutions. It detected the execution of the ROP gadget chains with 81% accuracy, 80% true positive rate, only 0.8% false positive rate, and incurred comparable overhead to similar Pin-based solutions.

1 Introduction

Since its introduction by Shacham in 2007 [38], Return-Oriented Programming (ROP) has become an increasingly popular technique for bypassing Data Execution Prevention (DEP) defenses on modern operating systems. DEP ensures that all writable memory pages of a program are non-executable, which prevents the execution of any input data, effectively mitigating all classic code injection attacks. In a ROP attack, on the other hand, the attacker does not inject new code. Instead, existing sequences of instructions in the process executable memory, called *gadgets*, are chained together to perform the intended computation. While the traditional Address Space Layout Randomization (ASLR) randomizes the location of most libraries and executables, ROP attacks can still bypass ASLR by finding a few code segments in statically known locations, or through brute-forcing and de-randomization by exploiting memory disclosure vulnerabilities.

Over the past few years, research in ROP defenses has become an arms race, where emerging defenses are countered by new subtle variations of ROP attacks. Defenses can be categorized into two broad categories. The first category attempts to prevent ROP attacks at compile time, by eliminating gadgets from binaries [32] or enforcing Control-Flow Integrity (CFI) [9]. The second category aims at detecting ROP attacks at runtime, by monitoring the execution of programs [33, 13, 15, 30, 16, 41].

Defenses in the second category can further be classified based on the detection approach into signature-based and anomaly-based. Signature-based solutions detect ROP attacks by identifying static signatures (patterns) in the execution trace of programs. The most common method is to detect gadgets execution by enforcing predefined constraints over the program counter and the call stack, either through dynamic instrumentation [15, 23, 18] or by leveraging existing hardware branch tracing features [13]. These solutions incur very low overhead, but the employed signatures are often incomplete due to strong constraints on the ROP structure, allowing the defenses to be bypassed by attackers [14, 12, 19].

Anomaly-based detection, on the other hand, learns a baseline of normal (clean) behavior and detects attacks by measuring statistical deviations from the normal behavior. This approach has the significant advantage of being able to protect against a broad spectrum of attacks, including zero-day. Until recently, anomaly-based approaches have only leveraged software characteristics, e.g., network traffic and system call sequences [31, 24]. Meanwhile, attacks have increased in complexity, becoming stealthier and harder to detect. Therefore, researchers have explored the potential of using hardware characteristics, such as instruction mixes and branch prediction rate, to detect ROP attacks [30, 16, 33, 41, 34].

Using hardware characteristics has a major advantage over software characteristics: it is harder for attackers to gain sufficient control over the hardware in order to evade detection. For example, it is easy to craft ROP payloads that mimic the behavior of clean software execution by chaining gadgets that invoke benign sequences of system calls, while still executing the attack payload. On the other hand, it is hard to craft payloads that, while still attacking the system, maintain precise control of the branch prediction rate of the hardware. This is because attacks, by definition, have to go against the normal flow of the program, inevitably resulting in misprediction of branches and returns by the hardware branch predictor.

Prior work that utilized hardware characteristics used two classes of characteristics: 1) architectural characteristics, which are dependent on the instruction set architecture (ISA), such as the number of load and store instructions retired. And, 2) microarchitectural characteristics, meaning characteristics that depend on the underlying microarchitecture configurations, such as branches misprediction rate and cache misses. These characteristics were typically measured by reading the hardware performance counters (HPC) of the underlying processor. However, a common pitfall is that characteristics measured using HPC may actually *hide* the underlying program behavior, making the HPC-based metrics appear similar for inherently different behaviors [20, 45].

In this paper, we introduce EigenROP, a novel system for detecting ROP attacks. We study, for the first time, the feasibility and value of using microarchitecture-independent program characteristics for the detection of ROP attacks. We propose a new type of anomaly-based ROP detectors that leverages *microarchitecture-independent* program characteristics, including memory reuse distance [47], register traffic load [17], memory locality [27], among others, in addition to traditional hardware characteristics (see Section 4).

EigenROP employs a novel anomaly detection algorithm that builds on concepts from directional statistics. The fundamental idea is that strong relationships among the different program characteristics will appear as principal axes in some high-dimensional space. Since ROP executes against the control flow of the program, it is reasonable to assume that it causes some unexpected changes in the relationships between the program characteristics learned from benign runs. Such changes can be detected as statistically significant deviations in the directions of the axes in the high-dimensional space. We investigate if and to what extent ROP causes changes in program characteristics, and verify our hypothesis with extensive experiments using multiple in-the-wild ROP payloads and payloads generated by the ROPC ROP compiler.

EigenROP operates in two phases: a learning phase and a detection phase. During the learning (offline) phase, programs are executed over benign inputs under EigenROP, where it collects different characteristics. The characteristics are measured periodically, every N instructions retired. A model is then constructed using Kernel Principal Component Analysis (KPCA) [36] and directional statistics (see Section 5). EigenROP uses a temporal model, where both the current snapshot of characteristics and the history are taken into account. This concludes the learning phase. In the detection phase, EigenROP monitors the execution of the target program, collects the characteristics, and tests for deviation from the trained model.

We implemented a prototype of EigenROP on Linux, using the dynamic instrumentation framework Pin [29]. We conducted several experiments to quantify the accuracy of EigenROP, the effect of involved parameters and the incurred performance overhead (see Section 7). In our experiments, microarchitecture-independent characteristics resulted in 11% increase on average in detection accuracy, relative to using only microarchitectural characteristics. EigenROP achieved an overall accuracy of 81%, 80% true positive rate, and only 0.8% false positive rate. The incurred performance overhead decayed exponentially as the sampling interval increases, and faster than the deterioration in accuracy. Overall, the overhead incurred matches with prior Pin-based solutions (see Section 9).

To summarize, we make the following contributions:

- We study the effectiveness of combining microarchitecture-independent program characteristics with typical hardware characteristics for the detection of ROP attacks.
- We propose a novel anomaly detection algorithm using directional statistics of program characteristics, embedded in high-dimensional space.
- We present EigenROP, a working prototype of our approach.
- We quantify the security effectiveness of EigenROP using in-the-wild ROP attacks against common Linux programs.
- We quantify the runtime accuracy-performance tradeoff of EigenROP.

2 Background

2.1 Return-Oriented Programming

Return Oriented Programming (ROP) [38] enables attackers to execute arbitrary code *without* injecting new code into the victim process, by returning to arbitrary instruction sequences in the executable memory of the program.

The basic idea is to use indirect jumps (e.g., `ret` instructions) to return to arbitrary points in the executable process memory that execute sequences of instructions ending in another indirect jump instruction. The last indirect jump instruction allows executing one such sequence after another. Multiple sequences can be combined into "gadgets" that perform an atomic task, such as load, store and system call. The attacker then "chains" the gadgets together, to perform the intended malicious functionality. Typically, gadgets end with a `ret` instruction, which returns to the stack. The attacker chains the gadgets by hijacking the stack and writing appropriate addresses to the beginning of the desired instruction sequence.

A typical ROP attack operates as follows: first, the attacker overwrites the stack contents with addresses of the desired ROP gadgets. Once the `ret` instruction of the current routine is executed, the first return address of the current stack frame is used as a return target. Instruction sequences at that address will execute, till the next `ret` instruction. Upon execution of the `ret` instruction, control is transferred to the next gadget. This process repeats, jumping from one gadget to the next, till the gadget chain terminates.

In Fig. 1, we give an example of a gadget that stores a constant `0x1` at the target memory address `0xa0de1b6e`. The gadget starts by loading the constant `0x1` from the stack to the register `eax`. It then loads the target memory address to register `ebx`. Finally, it moves the contents of `eax` back to the memory pointed at by `ebx`.

Conventional ROP attacks use `ret` instructions to chain the gadgets [38]. In [11], a ROP variant was presented that uses indirect jump (e.g., `jmp eax`) instructions to chain the gadgets. While we mainly evaluate EigenROP using conventional ROP attacks, our solution is applicable regardless of how the gadgets are chained. We discuss in Section 8 how different variants of ROP payloads can be detected by EigenROP.

It has been shown that ROP can perform Turing-Complete computations if the attacker can find sufficient gadgets to perform memory, arithmetic, logical operations and system calls [42]. An infamous example on that is the recent ROP-only Adobe Reader exploit [1]. We refer the reader to [38, 35] for more details on ROP.

Finally, it is worth mentioning that overwriting the return address on the stack is not the only way to hijack the execution of the target process. Other vulnerabilities such as format string and integer overflow vulnerabilities can allow the attackers to write arbitrary values to function pointers that are used as jump targets by the program, thereby redirecting the execution to the attacker's instructions of choice. For example, a very common approach is overwriting the Global Offset Table on Linux systems, which holds absolute addresses to functions in dynamically linked libraries.

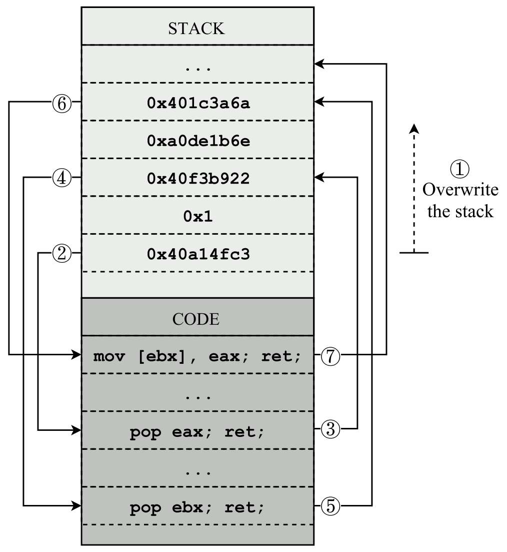

Figure 1: Example of a ROP gadget that stores a constant value `0x1` at memory location `0xa0de1b6e`.

2.2 Microarchitecture-independent Characteristics

It has been shown that microarchitecture-independent characteristics have higher discrimination power between different inherent program behaviors, compared to architectural and microarchitectural characteristics [20, 45]. Microarchitecture-independent characteristics are program characteristics that are unique to a given instruction set architecture (ISA) and a given compiler but are independent of a given microarchitecture. In other words, the characteristics are *invariant* of the underlying hardware cache size, pipeline size, branch predictors size and algorithm, number of cores and their configurations, and so on. In the context of ROP detection, several microarchitecture-independent characteristics can prove useful in discriminating between benign execution behavior and gadget execution, such as memory locality and reuse distance, and register traffic (see Section 4 for details). Note that while characteristics dependent on the ISA, i.e., architectural characteristics, can be regarded as a subset of microarchitecture-independent characteristics, we keep them distinct in this work as is the trend in prior program characterization work [20, 45, 30, 41].

The main downside of using microarchitecture-independent characteristics is that it requires runtime instrumentation to measure the characteristics. However, the overhead decays over time as more efficient algorithms and tools are developed [8].

In the following section, we outline the big picture of how EigenROP works.

3 Overview of EigenROP

The key idea of EigenROP is to identify *anomalies* in program characteristics, due to the execution of ROP gadgets. In this context, it is difficult to precisely define what anomalies are since that depends on the characteristics of both the monitored program and the ROP. However, it is reasonable to assume that some unexpected change occurs in the relationships among the different program characteristics due to the execution of the ROP. By extracting and learning arbitrary relationships among the program characteristics, EigenROP detects ROP by looking for unexpected changes in the learned relationships.

Given our definition of anomaly, strong relationships among the measured program characteristics should appear as principal directions in some high-dimensional space [36]. Such directions can be extracted using Kernel Principal Component Analysis (KPCA) [36]. More specifically, the principal component vectors of the measurements mapped into the high-dimensional space can be interpreted as the *relationships* among the program characteristics.

The general workflow of EigenROP is illustrated in Fig. 2. First, the target program is loaded and executed. During execution, EigenROP takes a snapshot of the different program characteristics, every N instructions retired. Each snapshot is a $d-$dimensional vector of characteristics. The snapshots are pushed to a buffer that EigenROP iterates over using a sliding window.

In the learning phase, the target program is executed over benign inputs. For each window of measured characteristics, EigenROP maps the measurements into a high-dimensional space and extracts the principal components of the measurements in that space. EigenROP then estimates a representative *direction* from all the principal components, and estimates the density of the distances of all principal components around the representative direction. Recall, the idea here is that any strong relationships among the measured characteristics will appear as principal components in the high-dimensional space. In the detection phase, EigenROP computes the distances of the principal components of incoming measurements, in the high-dimensional space, to the representative direction. If the distance exceeds some threshold, then an alarm is raised.

In the following, we define the characteristics used by EigenROP and explain in detail how learning and detection work.

4 Which Characteristics to Measure?

To choose the most relevant characteristics for ROP detection, we conducted several experiments to collect clean and infected measurements from a variety of programs and exploits (see Section 7.3). We considered most of the characteristics used in previous program characterization work [20, 45, 30, 41]. Then, we used the Fisher Score to quantify the discriminative power of each characteristic. The following is the shortlisted categories of characteristics we measured. The letters between brackets denote the type of the characteristics: **A**rchitectural [A], Microarchitecture-**I**ndependent [I], and **M**icroarchitectural [M]. We emphasize that all the characteristics used in this work are computed in software.

- **Branch predictability [M].** Since ROP attacks disturb the normal control flow of execution, they may increase the number of mispredicted branches by the processor branch predictor.
- **Instruction mix [A].** This is a traditional architectural characteristic that measures the frequency of different classes of instructions (branch, call, stack, load and store, arithmetic, among others). Since ROP attacks depend on chaining blocks of instructions that load data from the hijacked program stack to registers, and for returning to the stack, they may exhibit different usage of `ret` and `call` instructions as well as stack `pop` and `push` instructions.
- **Memory locality [I].** Given a set of instructions, memory locality is the difference in the data addresses between subsequent memory accesses [27]. Here, it is typical that a distinction is made between memory reads (loads) and writes (stores). Since ROP attacks depend on chaining gadgets from arbitrary memory locations, the attacks may exhibit low memory locality when compared to clean execution. The memory distance between subsequent reads and writes may indicate the execution of a ROP attack.
- **Register traffic [I].** Two useful register traffic characteristics can be measured [17]: 1) the average number of register input operands to an instruction; and 2) the register reuse distance, i.e., the number of instructions between writing a register and reading it. ROP attacks load data from the hijacked stack to registers typically using pop instructions that take a single operand. Therefore, the number of instruction operands could be an indicator of the presence of a gadget chain. Additionally, the usage degree of the registers themselves could be different from that of clean execution.
- **Memory reuse [I].** This is an important metric that characterizes the cache behavior of programs. It

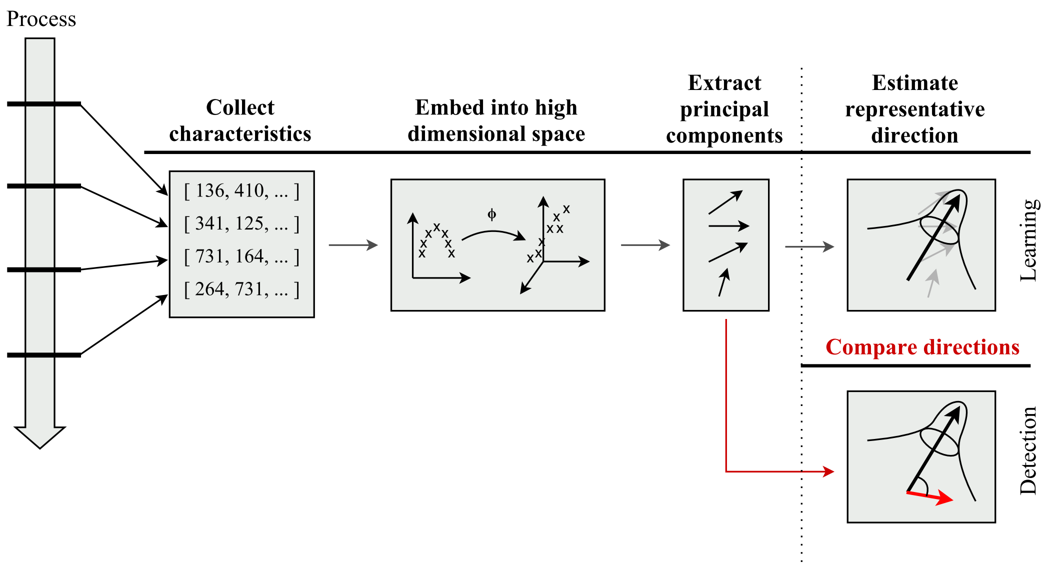

Figure 2: Workflow of EigenROP. EigenROP periodically interrupts the monitored process to measure the characteristics. It embeds each window of measurements into a high-dimensional space and extracts the principal directions in that space. Then, in the learning phase, it computes a representative (mean) direction and estimates the density of distances of all principal directions to the mean direction. In the detection phase, the principal directions of incoming measurements are compared to the mean direction for significant deviation.

measures the number of unique cache blocks referenced between subsequent memory reads [47]. For each memory read, the corresponding cache block is retrieved (assuming LRU cache). For each cache block, the number of unique cache blocks accessed since the last time it was referenced is determined. Since ROP attacks operate by using the stack for chaining the gadgets, and the gadgets are typically spread out across the memory of the program, they shall exhibit abnormal reuse of the same memory blocks when compared to clean execution.

Table 1 shows the top 15 characteristics, ranked by their Fisher scores. For each characteristic i, its Fisher Score is computed by:

$$score_i = \frac{m^{(+)}\left(\bar{\mathbf{x}}_i^{(+)} - \bar{\mathbf{x}}_i\right)^2 + m^{(-)}\left(\bar{\mathbf{x}}_i^{(-)} - \bar{\mathbf{x}}_i\right)^2}{m^{(+)}s_i^{2(+)} + m^{(+)}s_i^{2(-)}}, \quad (1)$$

where $(+)$ and $(-)$ are the infected and clean classes of measurements, respectively; $\bar{\mathbf{x}}_i^{(y)}$ and $s_i^{2(y)}$ are the mean and variance of characteristic i in class $y \in \{+,-\}$, and $\bar{\mathbf{x}}_i$ is the overall mean of feature i over both the infected and clean measurements. The Fisher Score is a widely established feature filtering method that assigns higher scores to features that result in greater separation between the means of clean and infected samples. Note that we used infected and clean measurements here to quantify the discriminative power of the selected characteristics. The infected measurements are **not** used during the learning phase of EigenROP.

Table 1: Top 15 characteristics sorted by discrimination power (highest to lowest). Chosen characteristics are marked with ⋆. Types A, I and M stand for "architectural," "microarchitecture-independent" and "microarchitectural," respectively. All counts are for instructions (insns) retired.

Rank	Type	Name	Description
⋆ 1	A	INST_RET	# `leave` and `ret` insns.
⋆ 2	A	INST_CALL	# near `call` insns.
⋆ 3	I	MEM_REUSE	Memory reuse distance.
⋆ 4	A	INST_STACK	# `pop` and `push` insns.
⋆ 5	I	MEM_RDIST	Memory read distance.
6	A	INST_LOAD	# memory read insns.
⋆ 7	I	REG_OPS	Avg. # register operands.
⋆ 8	M	MISP_CBR	Mispredicted branches.
9	A	INST_ARITH	# arithmetic insns.
⋆ 10	M	MISP_RET	Mispredicted `ret` insns.
11	A	INST_STORE	# memory write insns.
⋆ 12	I	MEM_WDIST	Memory write distance.
⋆ 13	A	INST_NOP	# `nop` insns.
14	I	REG_REUSE	Register reuse distance.
15	I	ILP	Instruction level parallelism.

Since the Fisher Score ignores mutual information, some of the scored characteristics might be redundant. Therefore, we picked 10 features out of the top 15 as follows. First, we excluded Instruction Level Parallelism (a measure of how many instructions of a program can be executed in parallel) since it added significant performance overhead and is highly dependent on the type of application. For example, cryptography applications may exhibit low instruction level parallelism, while a scientific computation program may exhibit high parallelism. Similarly, we excluded INST_LOAD and INST_ARITH. Via experimentation, we found that REG_REUSE does not increase the accuracy of the model, so we excluded it as well.

5 Learning and Detection

Given a sequence T of d-dimensional measurements, we divide T into n subsequences using a sliding window of width m. Let us denote the resulting subsequences by:

$$S^{(j)} = \begin{bmatrix} \mathbf{x}_1^{T(j)} \\ \mathbf{x}_2^{T(j)} \\ \vdots \\ \mathbf{x}_m^{T(j)} \end{bmatrix}, \tag{2}$$

for $j = 1 \dots n$. Note that each $\mathbf{x}_i^{(j)}$ is a vector of d measured characteristics.

Next, each $S^{(j)}$ is embedded (implicitly mapped) into a higher dimension space $\mathcal{H}$ with $\Phi : \mathbb{R}^d \to \mathcal{H}$, and the principal component vectors of $S^{(j)}$ in $\mathcal{H}$ are extracted. This is done using Kernel PCA [36], which solves the following eigenvalue problem:

$$\lambda_i^{(j)} \mathbf{v}_i^{(j)} = K \mathbf{v}_i^{(j)}, \tag{3}$$

where $\lambda_i^{(j)}$ are the eigenvalues of K, $\mathbf{v}_i^{(j)}$ are the normalized eigenvectors of K, and K is the $m \times m$ kernel matrix $\left[k\left(\mathbf{x}_i^{(j)}, \mathbf{x}_l^{(j)}\right)\right]$ for $i = 1 \dots m; l = 1 \dots m$. Here, k is the kernel function, which we set to the Radial Basis Function (RBF) given by:

$$\begin{aligned} k(\mathbf{x_1}, \mathbf{x_2}) &= \Phi(\mathbf{x_1})\Phi(\mathbf{x_2})^T \quad (4) \\ &= \exp\left(-\gamma \left\| \mathbf{x_1} - \mathbf{x_2} \right\|^2\right), \quad (5) \end{aligned}$$

where $\gamma = \frac{1}{d}$. We assume K is centered [36], i.e., $K = K - \mathbf{1}_m K - K\mathbf{1}_m + \mathbf{1}_m K \mathbf{1}_m$, where $\mathbf{1}_m$ is an $m \times m$ matrix for which each element takes the value $\frac{1}{m}$.

Using the eigenvalues and eigenvectors in $\mathcal{H}$, the *resultant* direction $\mathbf{v}^{(j)}$ of the data $S^{(j)}$, embedded in $\mathcal{H}$, is then computed by:

$$\mathbf{v}^{(j)} = c \sum_{i=1}^{m} \lambda_i^{(j)} \mathbf{v}_i^{(j)}, \tag{6}$$

where c is a normalizing factor such that $\mathbf{v}^{(j)T}\mathbf{v}^{(j)} = 1$. This direction can be perceived as a representative direction of all the principal axes of $S^{(j)}$ in the kernel space $\mathcal{H}$.

We then compute the mean direction $\boldsymbol{\mu}$ of T by:

$$\boldsymbol{\mu} = \frac{\sum_{j=1}^{n} \mathbf{v}^{(j)}}{\left\| \sum_{j=1}^{n} \mathbf{v}^{(j)} \right\|}. \tag{7}$$

The direction $\boldsymbol{\mu}$ is the representative direction for the entire trace of characteristics, where the extracted directions $\mathbf{v}^{(j)}$ distribute around $\boldsymbol{\mu}$.

Hence, the following similarity vector Z is constructed:

$$Z = \begin{bmatrix} \mathbf{v}^{(1)T}\boldsymbol{\mu} \\ \mathbf{v}^{(2)T}\boldsymbol{\mu} \\ \vdots \\ \mathbf{v}^{(n)T}\boldsymbol{\mu} \end{bmatrix}, \tag{8}$$

where each row corresponds to the angular distance between each direction $\mathbf{v}^{(j)}$ and $\boldsymbol{\mu}$.

Next, a kernel density is estimated over Z using the standard normal kernel density estimator, given by:

$$f_h(z) = \frac{1}{nh} \sum_{i=1}^{n} \mathsf{N}\left(\frac{z - z_i}{h}\right), \tag{9}$$

where h is the smoothing parameter (the bandwidth), $z_i \in Z$, and N is the standard normal function. In our implementation, we chose the value of h using grid search.

We expect the resulting density to be close to exponential since the directions extracted from clean measurements are expected to be concentrated (tightly distributed around $\boldsymbol{\mu}$), resulting in a skewed density with a peak around high similarity values. Therefore, we reduce the skewness of f_h by applying the following logarithmic transform:

$$\hat{f}_h(z) = f_h(z) \log\left(f_h(z)\right), \tag{10}$$

where the area under the curve of $\hat{f}_h(z)$ gives the entropy η of $\hat{f}_h$. This transforms the bulk of the density towards the peak, resulting in a shorter (easier to threshold) tail.

This concludes the learning phase. The following subsection explains the anomaly metric and the detection phase of EigenROP.

5.1 Anomaly Metric

Given an incoming subsequence of measurements $S'^{(j)}$, an anomaly is detected if the direction of $S'^{(j)}$, in the $\mathcal{H}$ space, is significantly different from the learned directions around $\boldsymbol{\mu}$. The decision r is computed by:

$$\mathbf{v}'^{(j)} \text{ from Eq. (6)} \tag{11}$$
$$z'^{(j)} = \mathbf{v}'^{(j)T}\boldsymbol{\mu} \tag{12}$$
$$\zeta = \int_{-1}^{z'^{(j)}} \hat{f}_h(z)\,dz \tag{13}$$
$$r = \operatorname{sgn}(\zeta - \theta\eta)\,, \tag{14}$$

where $\theta \in (0, 1)$ is the detection threshold, which sets the fraction of the entropy that the model leaves out for detecting attacks. This concludes the detection phase.

To summarize, EigenROP operates as follows:

Learning Phase

1. Periodically, collect program characteristics $\{S^{(j)}\}_{j=1}^{n}$ of the target program.
2. Extract the principal directions $\{\mathbf{v}^{(j)}\}_{j=1}^{n}$ in a higher-dimension kernel space.
3. Compute a representative direction $\boldsymbol{\mu}$ from $\{\mathbf{v}^{(j)}\}_{j=1}^{n}$.
4. Estimate η of the distance between the principal directions and $\boldsymbol{\mu}$.

Detection Phase

5. Repeat steps 1 and 2.
6. Compute the anomaly metric r, if r equals -1 then an attack is present.

5.2 Detection Time and Space Complexity

Computing the anomaly metric requires performing the KPCA computation (Eq. (3)) in $O(m^3)$ [36]. Computing the resultant vector (Eq. (6)) takes $O(m^2)$. The distance in Eq. (12) is computed in $O(m)$. Thus, it takes a total time of $O(m^3)$ to compute the anomaly metric. Our model requires space $m \cdot d$ for the incoming measurements window $S^{(j)}$, m for the representative direction $\boldsymbol{\mu}$, and c for the transformed density (Eq. (10)), where c is the number of points of the density. Thus, it takes a total space of $O(md + c)$. Note that all terms in our prototype implementation of EigenROP are bounded: $d = 10$, $m \leq 10$ and $c \leq 1000$.

5.3 Handling Multiple Runs

The algorithm discussed so far focused on a single run of the monitored program. To handle multiple runs, we proceed as follows. Given a set $\{T^{(i)}\}_{i=1}^{k}$ of sequences, where each $T^{(i)}$ corresponds to a different run of the monitored program, we compute the family of sets of directions $\{\{\mathbf{v}^{(j)}\}_{j=1}^{n^{(i)}}\}_{i=1}^{k}$, then compute $\boldsymbol{\mu}$ over the entire family. Here, storing the entire set of directions is not necessary, since $\boldsymbol{\mu}$ and the distance density can be computed iteratively using online (streaming) mean and density algorithms.

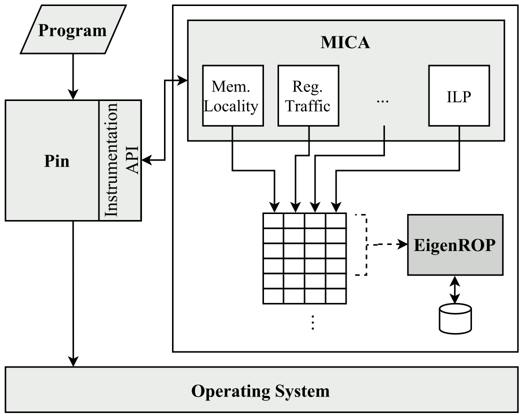

Figure 3: Architecture of EigenROP within Pin.

6 Implementation

We implemented a proof-of-concept prototype of EigenROP on top of MICA [21], a Pintool for collecting program characteristics. The EigenROP module is implemented in ∼700 lines of Python, with the aid of the SciKit-Learn [6] machine learning toolkit. Pin [29] is a generic dynamic instrumentation framework with a rich API that Pintools use to specify own instrumentation code. Pintools are written in C/C++. We chose Pin since it achieves the best performance among various dynamic instrumentation platforms [29].

Fig. 3 shows the architecture of EigenROP within Pin. MICA uses the instrumentation API of Pin to specify its own instrumentation code, which computes the different characteristics. As the program executes, the JIT compiler in Pin intercepts the program traces and compiles the instrumentation code into the program, where the characteristics are computed over the program traces. A program trace is a chain of multiple basic blocks that end with an unconditional jump. The measurements reported by MICA are stored in a d-dimensional circular buffer, one row at a time. The EigenROP module consumes and processes the buffer using a sliding window as explained in Section 5. Finally, the learned directions and densities are stored on disk for usage in the detection phase, where the same procedure is followed in addition to computing the anomaly metric. If a ROP is detected, EigenROP logs an alarm and terminates the target process.

7 Evaluation

We evaluate the security effectiveness, the added value of using microarchitecture-independent characteristics, and the tradeoff between runtime overhead and the detection accuracy of EigenROP. For security evaluation, we conducted several experiments using in-the-wild ROP attacks and attacks generated by the ROPC [5] compiler. For performance evaluation, we used the UnixBench [7] systems benchmark. We ran our experiments on an Intel Core i7-4870HQ 2.5 GHZ machine with 4 GB of RAM, running 32-bit Linux Ubuntu 12.04, Intel Pin version 2.14, MICA version 0.40 and GCC version 4.6.3.

7.1 Evaluation Metrics

To evaluate our approach, we use Receiver Operating Characteristics (ROC) curves and the area under the curve (AUC) scores. The x-axis of the ROC curve gives the false positive rate (FPR), and the y-axis gives the true positive rate (TPR). The FPR (eqv. with $1-$ specificity) represents the probability of false alarm, i.e., the likelihood of mislabeling a clean execution as an attack, given by $FP/(FP+TN)$. The TPR (eqv. with sensitivity) represents the probability of correct detection of ROP execution, given by $TP/(TP+FN)$. Each point on the ROC curve corresponds to the FPR and TPR, for a specific value of $\theta \in (0,1)$. The area under the curve (AUC) of the ROC is also computed, which provides a quantitative single value measure of the accuracy of the system for a variable θ. The higher the AUC, the higher the detection accuracy. The AUC reaches its best value at 1 and its worst at 0.

Table 2: Data set used in our experiments.

Program	Avg. Payload Length	# of Samples
`cmp`	800	80
`cpio`	650	210
`diff`	910	140
`file`	700	315
`grep`	631	150
`hteditor`	60	100
`openssl`	1021	195
`php`	400	265
`sed`	570	350
`sort`	712	110
`stat`	673	110
`wget`	813	90
	Total Samples:	2115

7.2 Dataset and Evaluation Procedure

We used two publicly available ROP exploits: OSVDB-ID:87289 [2] and OSVDB-ID:72644 [4], for the Linux Hex Editor (`hteditor`) version 2.0.20 and PHP version 5.3.6, respectively. We also used a number of exploits generated by the ROP gadgets finder and compiler ROPC [5], for common Linux programs (4 different exploits per program). Table 2 shows the programs used in our evaluation, the average payload length (the number of instructions) of each exploit, and the number of samples per program.

We collected clean samples for each target program by running the functionality tests that shipped with the program. In the case of `hteditor`, as it did not ship with functionality tests, we ran it on 100 random PDF files downloaded from the web. We collected infected samples following a similar approach to [13, 34]: assume that the attacker had successfully compromised the target process, and inject code into the target process to load a given exploit payload into memory and execute it. The payload (gadgets) is executed by directly jumping to the beginning of the payload at random points during the execution of the process. Each payload execution was considered an infected (attack) sample.

For each program, we used 5-fold cross-validation: 4 clean folds for training, and 1 clean fold for testing along with infected samples. We used the same number of clean and infected samples in the testing fold. The mean of the resulting five TPRs and FPRs is then used in computing the ROC and its AUC. We stress that labeled measurements were collected strictly for testing; EigenROP uses only the clean measurements for training.

7.3 Detection Accuracy

7.3.1 Hteditor OSVDB-ID:87289 and PHP OSVDB-ID:72644

EigenROP successfully detected the `hteditor` ROP exploit with sampling intervals up to 16k instructions retired and detected the PHP ROP with sampling intervals up to 32k. In both cases, EigenROP resulted in *zero* false positives. We emphasize that the focus here is on the detection of the ROP stage of the exploits, i.e., the execution of a gadget chain, rather than the execution of a shell code or a different process (both were shown to be easily detectable [41, 30]). Despite the very small ROP length (only ~60 instructions in the case of `hteditor`) when compared to the sampling window size, EigenROP still detected the deviation in the programs characteristics.

7.3.2 Overall Detection Accuracy

Fig. 4 shows the overall ROC of all experiments, for a sampling interval of 16k instructions. EigenROP achieved an overall accuracy (AUC) of 81%. The best point of performance had 80% TPR and 0.8% FPR. Note that EigenROP focuses on the detection of ROP. This is different from relevant prior work [41, 30], where it was assumed that the attacks undergo multiple stages such that *only the first stage* is a ROP chain, while the rest are normally injected code or a different process. Since the ROP chain length is usually in the order of a few hundred instructions, it is significantly more challenging for

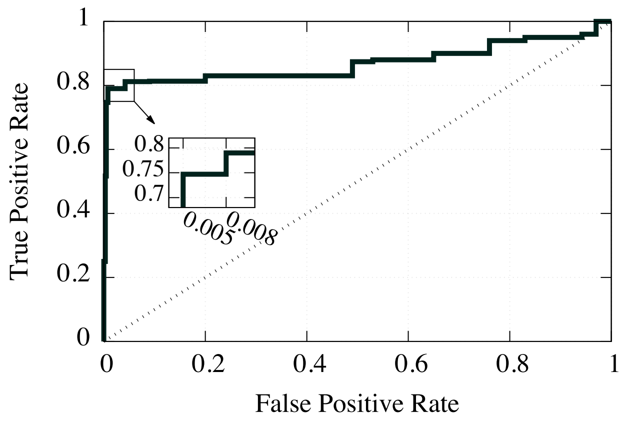

Figure 4: Overall ROC of EigenROP. The sampling interval was set to 16k instructions. The AUC is 0.81.

it to be detected. While the authors in [41, 30] detected the non-ROP stages of the attack with high accuracy, and as they noted, their proposed models performed poorly in the detection of the ROP chains alone (AUC ranged from 49% to 68%). In contrast, EigenROP focuses on the detection of the execution of the ROP gadget chain itself.

7.3.3 Sampling Granularity

The breakdown of the detection accuracy for different sampling intervals is shown in Fig. 5. As expected, the accuracy drops for very large sampling intervals, given the small number of instructions of the attacks. Out of all the programs, `wget` had the worst detection accuracy due to excessive use of signals, which exhibits poor locality and reuse (see Section 8 for discussion). The density estimate of `wget` was very heavy-tailed, which resulted in low discrimination between clean runs and attacks. On the other hand, `openssl` had the highest detection accuracy, as its characteristics had higher concentration around the mean direction. The bulk of the distribution of the AUC curves neared the best accuracy curve (the AUC was skewed towards the worst accuracy curve), indicating that the behavior of `wget` was possibly an outlier.

7.3.4 Microarchitecture-independent vs. Other Characteristics

Fig. 6 shows the difference in accuracy with and without the microarchitecture-independent characteristics. By including microarchitecture-independent characteristics, an increase of 9% to 15% in accuracy was achieved. This indicates that microarchitecture-independent characteristics contribute significantly to the detection performance of EigenROP.

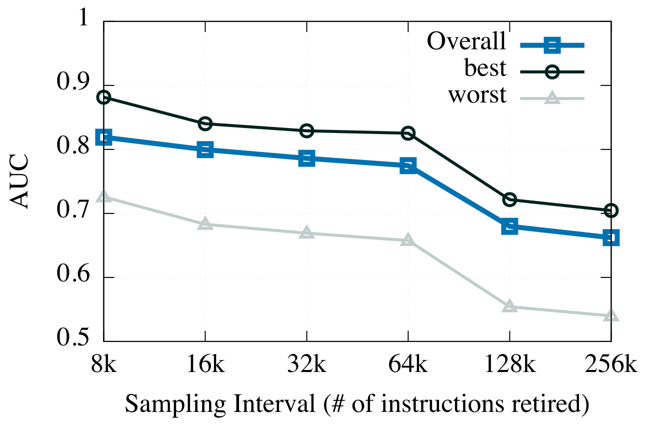

Figure 5: AUC for different sampling intervals. The higher the AUC curve, the better the detection accuracy.

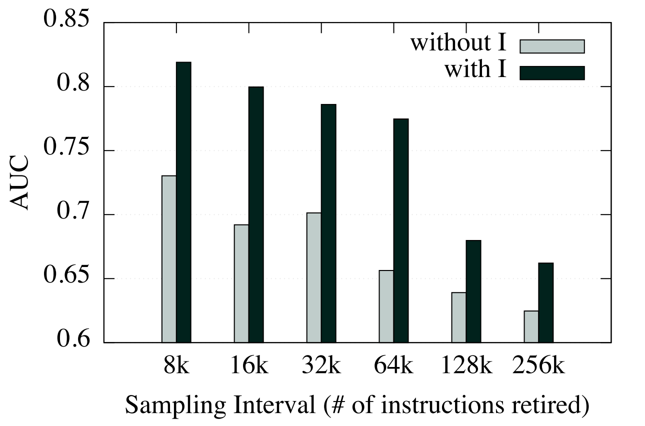

Figure 6: AUC for different sampling intervals, with and without the microarchitecture-independent characteristics.

7.3.5 Sliding Window Size

Fig. 7 shows the effect of changing the sliding window size m on the detection accuracy. Note that the window size controls the amount of temporal information available to the model. We observe that the effect of the window size on accuracy goes through three stages. First, too small window sizes hurt the detection accuracy, since small windows give higher variances in principal directions, resulting in higher FPR. Second, as the window size increases, the detection accuracy improves since the directions become more stable around μ. Finally, the accuracy deteriorates for too large window sizes since the influence of clean measurements on the principal directions dominates that of the ROP payload, resulting in lower TPR.

7.4 Overhead-Accuracy Tradeoff

We quantified the overhead of EigenROP for different sampling intervals by measuring the overall percent-

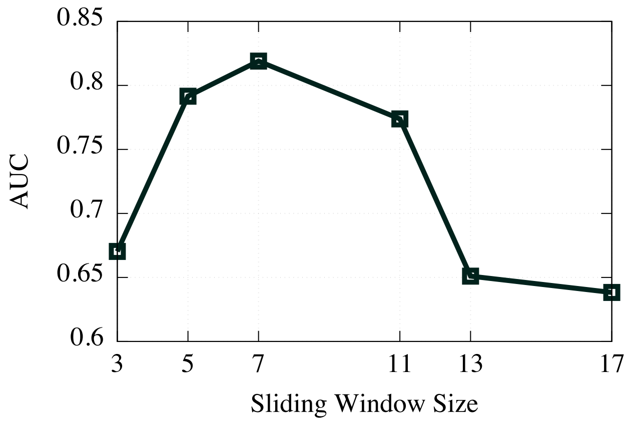

Figure 7: AUC for different sliding window sizes. Both too small and too large windows result in lower detection accuracy.

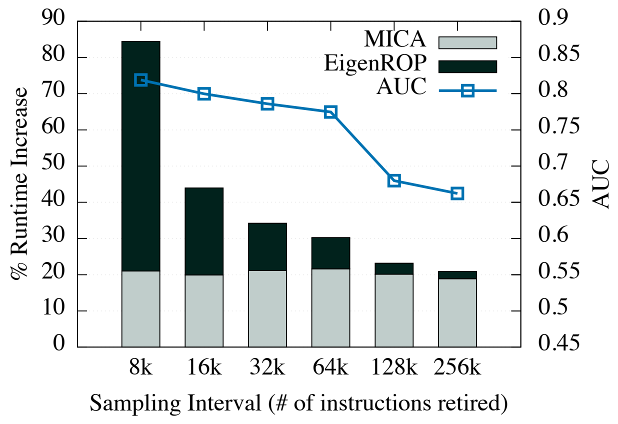

Figure 8: Overhead-accuracy tradeoff. The runtime overhead of MICA is measured relative to the overhead of Pin.

age slowdown in execution of UnixBench [7]. Fig. 8 shows the overhead and accuracy tradeoff. The overhead incurred by EigenROP exponentially decreases as the sampling interval increases. We also observe that the reduction in overhead *outpaces* the decay in accuracy. The overhead incurred by MICA is approximately constant as MICA analyzes the individual instructions of target programs, and the total number of instructions of each execution is invariant of the sampling interval. Overall, the incurred runtime overhead is comparable to similar dynamic instrumentation and HPC-based defenses [15, 34, 41]. Note that we did not perform any optimization attempts to reduce the overhead of EigenROP or MICA. Our work is orthogonal to how the program characteristics are collected. While we used MICA and Pin in our prototype implementation of EigenROP, they may not be the best tools for full build-out and full production. Finally, we emphasize that the memory and space overhead incurred by EigenROP are bounded and negligible (see Sections 5.2 and 6).

8 Discussion and Improvements

8.1 False Positives and Negatives

The detection approach of EigenROP (and relevant HPC-based solutions [30, 16, 41]) is based on the hypothesis that programs exhibit characteristics that are relatively concentrated around some statistic – in our case, the mean direction. However, if a program exhibits behavior that has a large spread, it becomes harder to separate anomalies from benign executions, resulting in a higher false positive rate (or a lower true positive rate).

From our experience with EigenROP, we observed that programs that use far jumps (e.g., `setjmp`/`longjmp`, `signal`) or extensively multiplex between data sources (e.g., using `select` for socket multiplexing) are more likely to suffer from false positives. The reason is that such programming constructs access far code and data, which inherently exhibits poor branch predictability, memory locality, and reuse. A possible workaround is to identify the entry and exit points of such code sites and build a separate model for the characteristics exhibited by those code sites. ROP chains missed by EigenROP were very short chains (<40 instructions) with small gadgets (2-4 instructions per gadget). This is mainly due to the relatively large sampling interval compared to the chain size. To handle such very short chains, EigenROP can be complemented by low-overhead solutions that target short gadgets and chains (e.g., kBouncer [33] and ROPecker [13]).

8.2 ROP Variants

In our evaluation of EigenROP, we used conventional ROP payloads that use return instructions to chain the gadgets. However, several variants of ROP were discovered by researchers. For example, in [11], Jump-Oriented Programming (JOP) was introduced where indirect jumps are employed to chain the gadgets rather than using return instructions. In [37], COOP was introduced where a loop in the program code that invokes attacker-controlled virtual function calls in C++ binaries is used to dispatch and chain the gadgets.

The goal is to simulate `ret` using a sequence of instructions that pops an address from the stack then jumps to that address using an indirect jump instruction, i.e., a pop-jump gadget. To use the pop-jump gadget, other gadgets have to end in an indirect `jmp` that transfers control to the pop-jump gadget, e.g., [`add; mov; ...; jmp eax; pop ebx; jmp ebx;`] where [`jmp eax;`] jumps to the pop-jump gadget, and [`pop ebx; jmp ebx;`] executes the pop-jump gadget and transfers control to the next gadget. In EigenROP, we picked the characteristics that cover the behavior of all ROP variants (branches, calls and returns, memory locality and reuse, stack usage, and `nop` sleds) regardless of how the gadgets are chained. Also, it is easy and straightforward to include other relevant characteristics if need be, such as the number of indirect jump instructions retired. Overall, EigenROP has the advantage that the detection is robust against attack variations, since it captures the execution behavior of benign runs, and does not put strong assumptions on how the gadgets are chained at the ISA level.

8.3 Evasion and Mimicry Attacks

Three recent attack gadgets were presented [12] that bypass ROP defenses through evasion and mimicry: call-preceded gadgets, evasion gadgets, and history-flushing gadgets.

Call-preceded gadgets are constructed from sequences of instructions that are preceded by a call instruction in the program memory. Such gadgets violate the assumption made by the majority of defenses [33, 13, 15, 23] that a sequence ending in `ret` must be legitimate, if it was preceded by *any* `call`. Since EigenROP does not depend on branch tracing, it is not vulnerable to attacks based on call-preceded gadgets. Moreover, the return address will be mispredicted, regardless of the gadget type, unless the call-ret are strictly paired. Since EigenROP takes the misprediction rate of returns into account (see Section 4), call-preceded gadgets will result in abnormal mispredictions, potentially increasing the detection accuracy.

Evasion gadgets were introduced for evading ROP detectors that use heuristics based on the length of the gadget chain (e.g., [33, 13]). Such detectors detect ROP by identifying gadget chains within some window of the execution trace. The heuristics are based on the length of the gadgets within the chain, with the presumption that short gadgets are likely part of an executing ROP. Evasion gadgets violate that assumption by using long enough gadgets to violate such constraints. Since EigenROP does not depend on the gadget chain length, rather on the characteristics of the gadgets, it is not vulnerable to attacks based on evasion gadgets.

History-flushing gadgets target defenses that only keep a limited history about execution (typically dependent on the available hardware buffer size where the history is recorded). History is flushed by utilizing innocuous gadgets to fill up the history. For example, kBouncer [33] uses the Last Branch Record (LBR), a hardware feature that records the most recent 16 taken branches by the processor. While kBouncer is very efficient against short ROP chains, it can be evaded by a ROP chain that executes any 16 valid indirect jumps to fill the LBR with legitimate branches completely[12].

In our context, flushing the history means manipulating *all* affected characteristics by the ROP, such that they appear normal. The attacker would need to chain gadgets that exhibit similar characteristics to benign code, in addition to achieving the attack goal. While this is theoretically possible, we argue that it is hard to realize such attacks in practice. First, chaining more gadgets would require larger attacker-controlled memory space. Second, if the attacker includes benign code in the ROP to mimic normal behavior, the benign code would be required to either have no effect on the actual ROP execution or be undone by chaining, even more, gadgets. Third, and As noted in [12, 34], history flushing comes at the expense of significant slowdown (reported 20-times slowdown) in the execution of the ROP payload.

Randomization has been proposed as a defense against evasion and mimicry attacks in anomaly-based intrusion detection systems [44, 43], and more recently [39] where it was shown that mimicry attacks could be efficiently detected by judging the quality of detection using an ensemble of classifiers. In the context of EigenROP, a potential defense strategy is to randomize the set of characteristics measured by EigenROP and build multiple detectors using different subsets of characteristics. The detectors can be constructed using different models, where a subset of the models is chosen at runtime at random. Additionally, we can randomly choose between the models at different points in the program. For example, using 15 characteristics and 5 models where each model randomly uses 5 characteristics, there will be $5 \cdot \binom{15}{5} = 15015$ possible configurations. Since the attacker does not have direct control over the program characteristics, she would need to craft ROP payloads that bypass *all* possible configurations of detectors and characteristics, significantly increasing the cost to attack.

8.4 Overhead Reduction

The current downside of using microarchitecture-independent characteristics is the need for dynamic instrumentation to compute the characteristics. As shown in Section 7.4, this may incur a non-negligible overhead penalty. However, this is an active research area, and more efficient program characterization algorithms and tools are being developed [8]. The need for dynamic instrumentation can be eliminated if the hardware or the kernel provide support by computing the required characteristics. Rather than instrumenting the process in user space, the characteristics can be computed (by the kernel or the hardware) and written to a memory-mapped ring buffer that is readable in user space. In case the buffer is not consumed quickly enough, an interrupt can be triggered to pause the monitored process. A similar approach is adopted by the Linux performance counter subsystem [3], which already provides support for a wide range of architectural and microarchitectural characteristics.

8.5 Input Coverage

In the learning phase of EigenROP, the target program is executed over benign inputs. Sufficient input coverage could arguably be a challenging task for the deployment of EigenROP. In our evaluation, we used the positive functionality tests that shipped with the programs to train EigenROP, which are integral to the software development lifecycle. In addition to functionality tests, EigenROP models can be constructed from successful dry runs during internal acceptance and pre-release testing. Additionally, EigenROP can even be trained by end users. To avoid learning *bad* behavior, the learned models can be aggregated from clusters of users and averaged (by computing the mean directions and densities), then filtered (cleaned) from outliers. Further, EigenROP can continue learning even after deployment by iteratively updating the learned directions and densities. This can be a privilege that is tied to the user group, for instance, update the models only from processes owned by admin users. The effectiveness of training by end users is currently in our future work.

9 Related Work

While the literature on ROP is vast, due to space constraints we briefly only mention solutions that used hardware or software characteristics, as well as anomaly-based solutions.

9.1 Binary Rewriting and Instrumentation

Some solutions were presented that used binary rewriting and dynamic instrumentation to detect ROP attacks. ROPDefender [15] enforces call-ret pairing by maintaining a shadow stack of `call` and `ret` targets. When a `ret` instruction is executed, ROPDefender compares the shadow stack to the actual system stack. If the two stacks do not match, then a ROP is detected. ROPStop [23] uses static binary rewriting to insert instrumentations that check two main constraints on the program counter and the call stack: 1) the program counter must point to a valid intended instruction, and 2) the call stack height is valid. The 2nd constraint is checked by analyzing the CFG and computing the set of all possible call stack heights from function entry points to branching points. If any of the constraints is not satisfied, a ROP is detected.

Similarly, ROPGuard [18] checks a set of constraints over the call stack, `call` and `ret` instructions at entry points to system calls, e.g., `ret` instructions must be preceded by `call` instructions, the `call` instruction must lead back to current entry point, etc. While such solutions are easy to deploy and require no system modifications, they are limited by some factors: 1) using CFGs is constrained by the speed and accuracy of binary disassembly and CFG construction. 2) Binary rewriting breaks self-checksumming and signed code. 3) Frame pointers that are required to traverse the stack are usually omitted by compilers during optimization. And, 4) call-ret pairing restricts valid, commonly used, call-without-return assembly constructs, e.g., using `[setjmp; ...; longjmp;]` for exception handling, and `[call; pop;]` for retrieving the program counter.

9.2 Hardware Branch Tracing

Recently, ROP defenses that leverage existing hardware branch tracing features were introduced. kBouncer [33] uses the Last Branch Record (LBR) on modern Intel processors to check for sequences of consecutive `call-ret` instructions. The LBR stores the most recent 4-16 indirect branches executed by the processor. kBouncer checks, at the entry of every system call, if 1) `call` instructions preceded `ret` targets, and 2) there is no `call-ret` sequence of length greater than 8. ROPecker [13] extends kBouncer by also checking at arbitrary points during the program execution, and counting the number of possible gadget-like sequences ahead of the program counter. Similarly, Eunomia [46] utilizes the Branch Trace Store (BTS) to check for unpaired `call-ret` sequences. Unfortunately, while these approaches incur very low overhead, attackers have bypassed them by violating the length based heuristics [14, 12, 19]. Nevertheless, they provide a solid defense against very short ROP chains (due to the limited hardware buffer sizes) and are complementary to this work.

9.3 Anomaly-based Solutions

In [26], Krugel et al. introduced an application specific approach that uses network traffic to detect malicious activities. Mazeroff et al. [31] described methods for inferring and using probabilistic models for detecting anomalous sequences of API calls. In [24], Jyostna et al. proposed a system for detecting anomalous program behavior by clustering critical system calls. While network traffic and system call defenses are simple and easy to deploy, they are susceptible to mimicry attacks [25].

One of the first works on using hardware architectural characteristics of programs was the work of Malone et al. [30]. They showed that hardware performance counters (HPC) could be utilized to detect unauthorized software changes. The authors recorded HPC measurements of the original programs and used linear regression to detect if the program was modified at runtime. Demme et al. [16] ported the idea to Android, and proposed hardware modifications to detect malware using HPC measurements from good and malicious samples. Stewin et al. [40] proposed detecting DMA attacks by monitoring the number of transactions on the memory bus.

In [41], Tang et al. combined microarchitectural characteristics with architectural characteristics to detect drive-by attacks. They assumed that attacks consist of three stages: ROP stage disables DEP, stage 1 downloads a malicious program, and stage 2 executes the malicious program. By training a one-class Support Vector Machine (oc-SVM) over the architectural and microarchitectural characteristics of benign samples, they showed that stage 1 of the attacks could be detected with high accuracy, while their model performed poorly on stage 2 of the attacks. This is because the oc-SVM is very sensitive to tuning parameters, and the chosen features did not have sufficient discrimination power to detect the execution of ROP chains. This is different from EigenROP since we solely focus on stage 2 of the attack. Similarly, in [10, 34], two solutions were presented that trained a two-class SVM using the architectural characteristics of both clean executions and attacks.

In contrast, EigenROP does not require any analysis of the binaries and operates using measurements only from clean executions. It does not need source code or debug information, and does not depend on branch tracing. EigenROP introduces a new class of anomaly-based detectors that utilize both hardware characteristics and microarchitecture-independent characteristics of monitored programs.

10 Conclusion

We presented EigenROP, a novel anomaly-based ROP detector that utilizes program characteristics and directional statistics. To the best of our knowledge, we are the first to study the effectiveness of using microarchitecture-independent program characteristics versus typical architectural and microarchitectural characteristics, in the detection of ROP. We demonstrated the ability of EigenROP to detect both in-the-wild and pure ROP exploits, despite the short payload length. EigenROP is unsupervised, fully transparent, and does not require any side information about the protected programs. One limitation of using microarchitecture-independent characteristics is the need for dynamic instrumentation to collect the measurements. One potential avenue to significantly reduce the overhead is by implementing the run-time monitors in hardware. Also, hardware support would probably increase the detection accuracy by enabling low-cost fine granularity monitoring. While our work demonstrates that ROP payloads can be detected using simple program characteristics, there are still needed improvements concerning detection accuracy and overhead reduction. Despite that, EigenROP raises the bar for ROP attacks, and can be easily coupled with hardware-based defenses to detect ROP transparently without program changes [28, 22].

References

[1] Analyzing the first ROP-only, sandbox-escaping PDF exploit. `https://blogs.mcafee.com/mcafee-labs/analyzing-the-first-rop-only-sandbox-escaping-pdf-exploit`.

[2] HT Editor 2.0.20 - Buffer Overflow (ROP). `https://www.exploit-db.com/exploits/22683/`.

[3] Linux performance counter subsystem. `https://github.com/torvalds/linux/blob/master/tools/perf/design.txt`.

[4] PHP 5.3.6 - Buffer Overflow PoC (ROP). `https://www.exploit-db.com/exploits/17486/`.

[5] ropc: A turing complete ROP compiler. `https://github.com/pakt/ropc`.

[6] Scikit. `http://scikit-learn.org/stable/`.

[7] Unixbench. `https://github.com/kdlucas/byte-unixbench`.

[8] Applications, tools and techniques on the road to exascale computing. In K. de Bosschere, E. H. D'Hollander, G. R. Joubert, D. Padua, and F. Peters, editors, *Advances in Parallel Computing*, volume 22. IOS Press, 2012.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[10] M. B. Bahador, M. Abadi, and A. Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 703–708. IEEE, 2014.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[12] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[13] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG, et al. Ropecker: A generic and practical approach for defending against rop attack. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[14] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[15] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[16] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[17] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *ACM SIGMICRO Newsletter*, volume 23, pages 236–245. IEEE Computer Society Press, 1992.

[18] I. Fratrić. Ropguard: Runtime prevention of return-oriented programming attacks, 2012.

[19] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium, San Diego, CA*, pages 417–432, 2014.

[20] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 83–92. IEEE, 2006.

[21] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 3:63–72, 2007.

[22] J. F. Hughes, A. Van Dam, M. Morgan, D. F. Sklar, J. D. Foley, and S. K. Feiner. *Computer Graphics: Principles and Practice*. Pearson Education, 2013.

[23] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller. Detecting code reuse attacks with a model of conformant program execution. In *Engineering Secure Software and Systems*, pages 1–18. Springer, 2014.

[24] G. Jyostna, P. Himanshu, and P. Eswari. Detecting anomalous application behaviors using a system call clustering method over critical resources. In *Advances in Network Security and Applications*, pages 53–64. Springer, 2011.

[25] H. G. Kayacik et al. Mimicry attacks demystified: What can attackers do to evade detection? In *Privacy, Security and Trust, 2008. PST'08. Sixth Annual Conference on*, pages 213–223. IEEE, 2008.

[26] C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 201–208. ACM, 2002.

[27] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 57–67. IEEE, 2004.

[28] G. Lavou'e and R. Mantiuk. Quality assessment in computer graphics. In *Visual Signal Quality Assessment*, pages 243–286. Springer, 2015.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005.

[30] C. Malone, M. Zahran, and R. Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 71–76. ACM, 2011.

[31] G. Mazeroff, J. Gregor, M. Thomason, and R. Ford. Probabilistic suffix models for API sequence analysis of Windows XP applications. *Pattern Recogn.*, 41(1):90–101, Jan 2008.

[32] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[34] D. Pfaff, S. Hack, and C. Hammer. Learning how to prevent return-oriented programming efficiently. In *Engineering Secure Software and Systems*, pages 68–85. Springer, 2015.

[35] M. Prandini and M. Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6):84–87, 2012.

[36] B. Schölkopf, A. Smola, and K.-R. Müller. Kernel principal component analysis. In *Artificial Neural Networks - ICANN*, pages 583–588. Springer, 1997.

[37] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[38] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[39] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *Network and Distributed System Security (NDSS) Symposium*, 2016.

[40] P. Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platforms main memory. In *Research in Attacks, Intrusions, and Defenses*, pages 1–20. Springer, 2013.

[41] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses*, pages 109–129. Springer, 2014.

[42] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[43] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Advances in Intrusion Detection*, pages 226–248. Springer, 2006.

[44] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *RAID*, pages 21–38. Springer, 2004.

[45] J. J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The exigency of benchmark and compiler drift: designing tomorrow's processors with yesterday's tools. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 75–86. ACM, 2006.

[46] L. Yuan, W. Xing, H. Chen, and B. Zang. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 6. ACM, 2011.

[47] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.