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

Department of Computer Science
George Mason University
Fairfax, VA 22030, USA

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.

Keywords

Return Oriented Programming; Anomaly Detection; Program Characteristics; Directional Statistics .

*An extended version of this paper is available as a technical report at http://cs.gmu.edu/techreports/2017

CODASPY'17, March 22–24, 2017, Scottsdale, AZ, USA

DOI: http://dx.doi.org/10.1145/3029806.3029812

1. INTRODUCTION

Since its introduction by Shacham in 2007 [19], 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.

Present ROP detection solutions aim at detecting ROP attacks at runtime, via means of signature-based or anomaly-based detection. 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 [7, 12, 9] or by leveraging existing hardware branch tracing features [6]. 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 (e.g., [5]).

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. Recent anomaly-based ROP defenses utilized hardware characteristics to detect attacks [14, 8, 15, 22]. They generally 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 [10].

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, register traffic load, memory locality, 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 (see Section 5). 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.

We implemented a prototype of EigenROP on Linux, using the dynamic instrumentation framework Pin [13]. We conducted several experiments to quantify the accuracy of EigenROP, the effect of involved parameters and the incurred performance overhead (see Section 6). 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.

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, as well as the accuracy-performance tradeoff.

2. BACKGROUND

2.1 Return-Oriented Programming

Return Oriented Programming (ROP) 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.

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.

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 [23]. Also, it is worth mentioning that `ret`-based ROP is not the only way to launch or chain attacks. We discuss in Section 7 how other variants of ROP are detected by EigenROP.

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 [10]. 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 [10, 14, 22].

The main downside of solutions using microarchitecture-independent characteristics is that there is currently no kernel or hardware support to collect the characteristics. Therefore, our prototype implementation requires runtime instrumentation to measure the characteristics. However, the overhead decays over time as more efficient algorithms and tools are developed, as as hardware and kernel support becomes available [3].

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. Such direc-

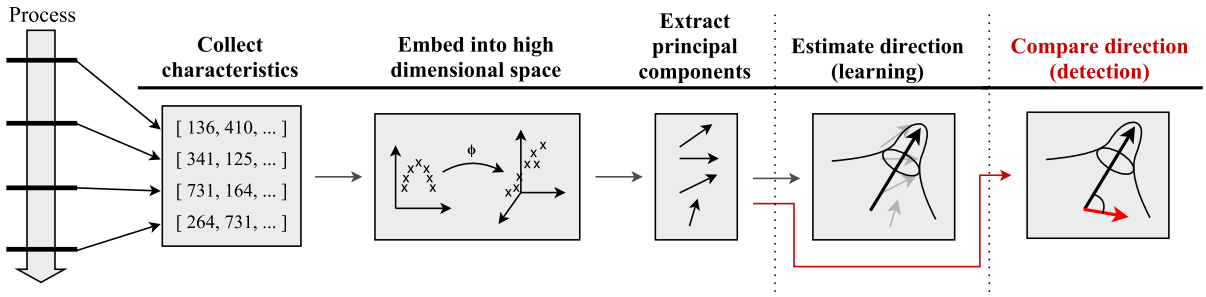

Figure 1: Workflow of EigenROP. It periodically interrupts the monitored process, measures the characteristics, embeds them into a high-dimensional space, extracts the principal directions in that space, and estimates a representative direction and density around the direction. In the detection phase, the principal directions of incoming measurements are test for significant deviation from the learned direction and density.

tions can be extracted using Kernel Principal Component Analysis (KPCA). 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 Figure 1. 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 6.2). We considered most of the characteristics used in previous program characterization work [10, 14, 22]. Then, we used the Fisher Score to quantify the discriminative power of each characteristic. The following is a brief description of 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. 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: 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 metric measures the number of unique cache blocks referenced between subsequent memory reads. For each memory read, the corresponding cache block is retrieved (assuming LRU cache). For each cache block, the number of unique cache blocks

Table 1: Top 15 characteristics sorted by discrimination power (highest to lowest). Chosen characteristics are marked with ⋆. 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.

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.

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:

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

for $j = 1 \ldots n$. 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 [17], which solves the following eigenvalue problem:

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

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 $i = 1 \ldots m; l = 1 \ldots m$. Here, k is the kernel function, which we set to the Radial Basis Function (RBF) given by:

$$k(\mathbf{x_1}, \mathbf{x_2}) = \Phi(\mathbf{x_1})\Phi(\mathbf{x_2})^T \quad (4)$$

$$= \exp\left(-\gamma \left\|\mathbf{x_1} - \mathbf{x_2}\right\|^2\right), \quad (5)$$

where $\gamma = \frac{1}{d}$. We assume K is centered, 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:

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

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:

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

The direction $\boldsymbol{\mu}$ is the representative direction for the entire trace of characteristics, where the extracted directions $\mathbf{v}^{(j)}$ distribute around $\boldsymbol{\mu}$. To handle multiple runs $\{T^{(i)}\}_{i=1}^{k}$, where each $T^{(i)}$ corresponds to a different run of the monitored program, we compute the family of sets of directions $\mathcal{V} = \{\{\mathbf{v}^{(j)}\}_{j=1}^{n^{(i)}}\}_{i=1}^{k}$, then compute $\boldsymbol{\mu}$ over $\mathcal{V}$.

Hence, the following similarity vector Z is constructed:

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

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:

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

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.

The resulting density is expected 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 effect of 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.

5.2 Detection Time and Space Complexity

Computing the anomaly metric requires performing the KPCA computation (Eq. (3)) in $O(m^3)$ [17]. Computing the resultant vector (Eq. (6)) takes $\Theta(m^2)$. The distance in Eq. (12) is computed in $\Theta(m)$. 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 $\Theta(md+c)$. Note that all terms in our prototype implementation of EigenROP are bounded: $d = 10$, $m \leq 10$ and $c \leq 1000$.

6. EVALUATION

We implemented EigenROP on top of MICA [11], a Pintool for collecting program characteristics. Pin [13] 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 [13]. The EigenROP module is implemented in ∼700 lines of Python, with the aid of the SciKit-Learn [2] machine learning toolkit.

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 [1] compiler. For performance evaluation, we used the UnixBench 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.

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

6.1 Dataset and Evaluation Procedure

We used two publicly available ROP exploits: OSVDB-ID:87289 and OSVDB-ID:72644, 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 [1], 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 [6]: 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 Receiver Operating Characteristics (ROC) and its Area Under Curve (AUC). The higher the AUC, the higher the detection accuracy. The AUC reaches its best value at 1 and its worst at 0. We stress that labeled measurements were collected strictly for testing; EigenROP uses only the clean measurements for training.

6.2 Detection Accuracy

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 (e.g., [14]). 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.

Figure 2 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. This is significantly higher than the state-of-the-art microarchitectural defenses [22, 14], where the detection accuracy (AUC) ranged from ranged from 49% to 68%.

The breakdown of the detection accuracy for different sampling intervals is shown in Figure 3. 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 7 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.

Figure 4 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.

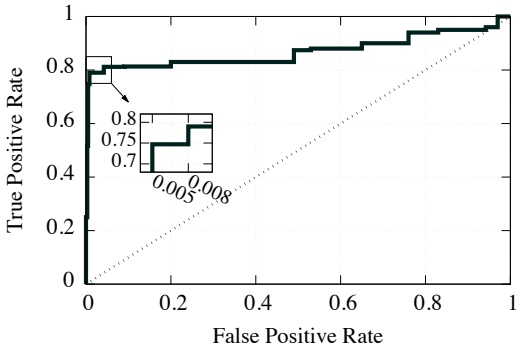

Figure 2: Overall ROC of EigenROP with 16k sampling interval. The AUC is 0.81.

6.3 Overhead-Accuracy Tradeoff

We quantified the overhead of EigenROP for different sampling intervals by measuring the overall percentage slowdown in execution of UnixBench. Figure 5 shows the overhead and accuracy tradeoff. The overhead incurred by Eigen-

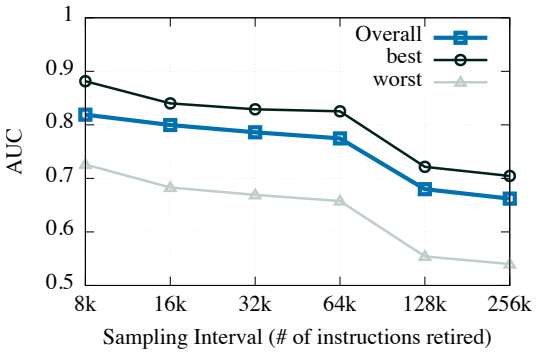

Figure 3: AUC for different sampling intervals. The higher the curve, the higher the accuracy.

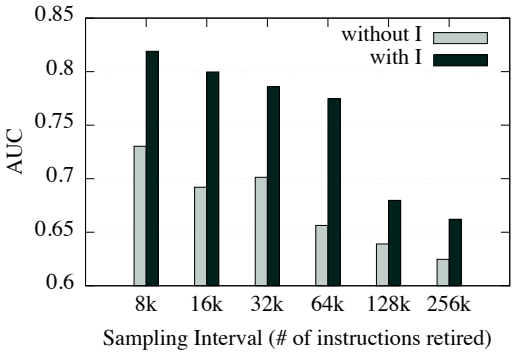

Figure 4: AUC for different sampling intervals, with and without the microarchitecture-independent characteristics.

ROP 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 [7, 22]. 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 Section 5.2).

7. DISCUSSION

7.1 False Positives and Negatives

The detection approach of EigenROP (and relevant HPC-based solutions [14, 8, 22]) 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

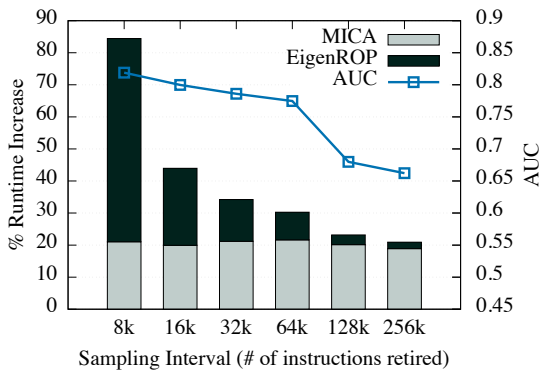

Figure 5: Overhead-accuracy tradeoff. The runtime overhead of MICA is measured relative to the overhead of Pin.

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 [15] and ROPecker [6]).

7.2 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 [4], Jump-Oriented Programming (JOP) was introduced where indirect jumps are employed to chain the gadgets rather than using return instructions. In [18], 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.

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.

7.3 Evasion and Mimicry Attacks

Three recent attack gadgets were presented [5] that bypass ROP defenses through evasion and mimicry: call-preceded gadgets, evasion gadgets, and history-flushing gadgets.

Call-preceded gadgets are violate a common key assumption made by defenses that depend on branch tracing [15, 6, 7, 12]: 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 use long gadgets to evade ROP detectors that look for short gadgets within an executing gadget chain (e.g., [15, 6]). Such solutions put constraints on the chain length, with the main presumption being 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 execution 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 [15] uses the Last Branch Record (LBR), a hardware feature that records *only* the most recent 16 taken branches by the processor. Therefore, it can be evaded by a ROP chain that executes any 16 valid indirect jumps to fill the LBR with legitimate branches [5].

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 [5, 16], history flushing comes at the expense of significant slowdown (reported 20-times slowdown) in the execution of the ROP payload. Randomization-based defenses against evasion and mimicry (e.g., [24, 20]) can also be employed to further increase the attack cost.

8. RELATED WORK

Due to space constraints we briefly discuss only related anomaly-based solutions.

One of the first works on using hardware architectural characteristics of programs was the work of Malone et al. [14]. 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. [8] ported

the idea to Android, and proposed hardware modifications to detect malware using HPC measurements from good and malicious samples. Stewin et al. [21] proposed detecting DMA attacks by monitoring the number of transactions on the memory bus. In [22], 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.

9. 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, and discussed limitations and potential improvements. EigenROP is unsupervised, fully transparent, and does not require any side information about the protected programs.

While our work demonstrates that ROP payloads can be detected using simple program characteristics, there are still needed improvements concerning detection accuracy of very short chains and overhead reduction. Future hardware support can help on both fronts by enabling low-cost fine-grained monitoring. Despite that, EigenROP raises the bar for ROP attacks, and can easily run in-tandem with complementary ROP defenses.

Acknowledgments

We thank the anonymous reviewers for their comments. This material is based on work supported by the National Science Foundation (NSF) under grant no. SATC 1421747, and by the National Institute of Standards and Technology (NIST) under grant no. 60NANB16D285. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NIST, or the US Government.

10. REFERENCES

[1] Ropc. https://github.com/pakt/ropc.
[2] Scikit. http://scikit-learn.org/stable/.
[3] Applications, tools and techniques on the road to exascale computing. In K. de Bosschere et al., editors, *Advances in Parallel Computing*, volume 22. 2012.
[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ASIACCS*. ACM, 2011.
[5] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.
[6] Y. Cheng et al. Ropecker: A generic and practical approach for defending against rop attack. In *NDSS*, 2014.
[7] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *6th ASIACCS*. ACM, 2011.
[8] J. Demme and others. On the feasibility of online malware detection with performance counters. *Computer Architecture News*, 41(3), 2013.
[9] I. Fratrić. Ropguard: Runtime prevention of return-oriented programming attacks, 2012.
[10] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarch.-independent characteristics. In *Workload Characterization*. IEEE, 2006.
[11] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 3, 2007.
[12] E. R. Jacobson and others. Detecting code reuse attacks with a model of conformant program execution. In *Engineering Secure Software and Systems*. Springer, 2014.
[13] C.-K. Luk et al. Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[14] C. Malone, M. Zahran, and R. Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *6th ACM workshop on Scalable Trusted Computing*, 2011.
[15] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, 2013.
[16] D. Pfaff et al. Learning how to prevent return-oriented programming efficiently. In *Engineering Secure Software and Systems*. Springer, 2015.
[17] B. Schölkopf, A. Smola, and K.-R. Müller. Kernel principal component analysis. In *Artificial Neural Networks - ICANN*, pages 583–588. Springer, 1997.
[18] F. Schuster et al. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security & Privacy*. IEEE, 2015.
[19] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th CCS*. ACM, 2007.
[20] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS*, 2016.
[21] P. Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platformâĂŹs main memory. In *RAID*. Springer, 2013.
[22] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *RAID*. Springer, 2014.
[23] M. Tran et al. On the expressiveness of return-into-libc attacks. In *RAID*. Springer, 2011.
[24] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *RAID*. Springer, 2006.