ROPDetect : Detection of Code Reuse Attacks

Yifan Lu
Stanford University

Christopher Hansen
Stanford University

1 Introduction

Software exploitation, as used by malware and other kinds of attacks, require the attacker to take control of code execution. Historically, this involves injecting code into memory and using a software vulnerability to execute it. This works because both ARM and x86 uses a modified Harvard architecture which allows code and data memory to be shared.

ARMv6 introduced the "execute never"[1] feature and Intel introduced the "execute disable" feature with their "Prescott" processors[3]. Both of these implementations ensure that memory pages are never mapped as both writable and executable (unless specified explicitly by the OS). This mitigates code injection attacks that relies on redirecting execution to attacker-controlled code stored in data memory. In response to this, attackers rely on "code reuse" or "return orientated programming" (ROP).

The idea behind ROP is that the attacker cannot map her own code into the target's executable memory, but she can "reuse" the code already in executable memory as well as control the target's program stack (through some vulnerability). The program stack is typically used to store (along with other data), return pointers when a function call is made. The return pointer allows the program to resume execution at the caller after a function call returns. When the attacker overwrites the return pointer, she can redirect control flow to anywhere in executable memory. If the attacker finds useful "gadgets", or instructions that perform a single useful operation (such as a memory load or store) and then jumps to the next return pointer in the stack, she can inject a large number of return pointers into the stack and control execution that way.

Even though ROP attacks are very powerful, most attackers only use it as the first stage of a multi-stage attack where ROP is used to bypass operating system restrictions in order to escalate control of the compromised process to control of the entire system. If we can differentiate ROP execution with normal code execution, then we can terminate a process before the control is escalated and stop the attack. Heuristically, we can see that ROP execution is different because it uses a large number of "return" instructions while normal (optimized) code does not perform as many returns in the same sort period of time. Additionally, in normal code, "return" instructions are often matched with "call" instructions. Finally, processors optimized to run normal code will likely perform better with normal execution than ROP execution (which for example, makes branch prediction hard). However, even in normal execution, there are situations such as tail recursion in a tight function that may make classification using just heuristics difficult. We choose to implement unsupervised machine learning to solve the classification problem.

2 Background

A large inspiration for this comes from work done earlier this year by Demme *et al.*, who talked about the usage of data from CPU performance counters as features for anomaly detection[2]. Teng *et al.* provided an implementation on x86 machines for detecting malware[7]. In their implementation, they "[sampled at a] rate of every 512,000 instructions since it provides a reasonable amount of measurements without incurring too much overhead." This meant that their model "does not perform well in the detection of the ROP shellcode, likely because the sampling granularity at 512k instructions is too large to capture the deviations in the baseline models."

Our implementation differs mainly in two ways. First, we sample at a much higher rate of 10,000 instructions. This is possible because we take advantage of the multiprocessor implementation of ARM Cortex A7, and we can use a separate core to perform the measurements and not have to pay the overhead of interrupts required by

measuring on the same processor core. Second, we focus on ARM architecture rather than x86. We believe that the RISC model implemented by ARM results in less noise in measurements.

3 Setup

We choose to perform our tests on a Raspberry Pi 2, which runs a Broadcom implemented ARM Cortex A7 processor system[6].

We collect data from the Performance Monitor Unit (PMU) with our custom Linux module. Because of hardware limitations, we can only chose four events to monitor. We decided to capture instruction cache fetch, branch taken, return taken, and branch mis-predict based on our knowledge of the architecture and what affects code execution. Next, we generate a random ROP payload which takes gadgets from our workload simulation application. Once the data is captured, we process it offline.

Our simulation infrastructure simulates three types of workload. Data-bound workload is the "best case" for detection because most events triggered normally are data related. Branch-bound workload is the "worst case" because a lot of instruction related events are triggered. Mixed workload is the "average case" and triggers a mixture of both types of events. To generate our dataset, we run each workload multiple times both normally and with a randomly generated ROP payload to execute at a random time-step.

4 Model

Features

We first define our non-temporal features to be a 4-dimensional vector defined by the number of each of the four events we sample.

There are also temporal information we wish to consider. As a motivating example, consider that a normal execution has the following properties: a spike in the number of function call leads to a spike of return calls after some cycles. In a ROP execution, we have a spike of return calls without the preceding function call spike. Therefore, it would be useful to consider the concatenation of N vectors from N time-slices such that we have a $4N$-dimensional vector.

Training

Motivated by Teng *et al.*, we first attempted to use a one-class Support Vector Machine (oc-SVM) classifier that uses the non-linear Radial Basis Function (RBF) kernel[7]. When trained on a set of data from one class, tested points will be classified as belonging to the trained class or not based on the decision boundary calculated during training. With this in mind, we speculate a oc-SVM can be applied to our problem of detecting ROP anomalies by first training the SVM on non-payload performance data (ie: "normal" operation). Then, any point that does not fall into the class when tested on the SVM is said to be an "attack" point, in other words data collected while the payload is being executed.

While this approach sounds like an applicable method, we find in practice that SVM is not practical for our problem. Besides exhibiting mediocre to poor results in our initial rounds of testing, the training overhead given the size of our data sets was unfeasible. Considering the problem, we observe that it is best modeled using unsupervised learning, as there are no strong labels on data collected.

Mixtures of Gaussians[4] is a specific instance of the EM algorithm that makes the assumption that the conditional distribution of the data points given the data points latent random variables is normally distributed. Intuitively, it can be thought of as a soft clustering algorithm, where each data point is probabilistically assigned to a Gaussian. It should be noted that the number of Gaussian components in the model is a key parameter, and corresponds mathematically to the number of values that the latent random variables can take on. Thus, assigning an observed data point to a Gaussian can be thought of as assiging that point to its most likely labeling.

With these properties of the model in mind, we develop our method of leveraging the model to detect anomalies. We first observe that, given a sufficient number of Gaussian components, we should be able to determine which Gaussian a point can be labeled as with high probability for non-anomalous data. If we observe the posterior distribution of the model for a specific point and find that the model is not "sure" which Gaussian to label the point as (for example, 0.5 for Gaussian 1, 0.5 for Gaussian 2), then the point can be consider anomalous, which in this case means its from a payload. This approach relies on the assumptions that the Gaussian components are sufficiently separated and that the data can be clustered into the Gaussians.

Using this approach, we see that a key aspect of this model is determining a threshold value that is sufficient for determining anomalous points. If the threshold is to low, then data from normal operation would be considered anomalous, but if its to hight then anomalous data will not be detected. We test various threshold values to determine the best results.

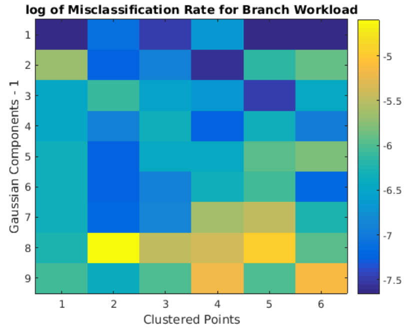

Figure 1: Branch workload error

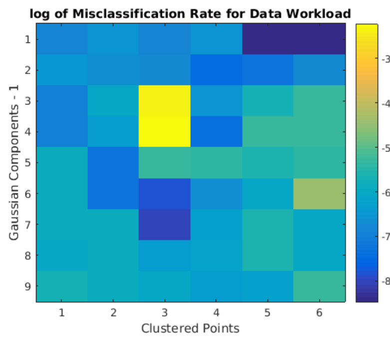

Figure 2: Data workload error

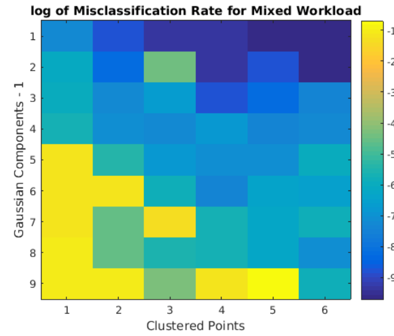

Figure 3: Mixed workload error

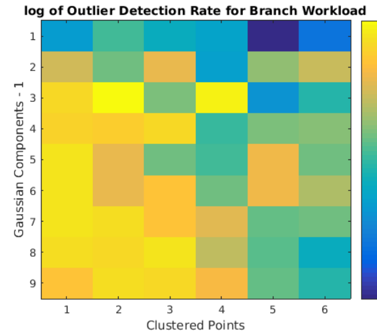

Figure 4: Branch workload success

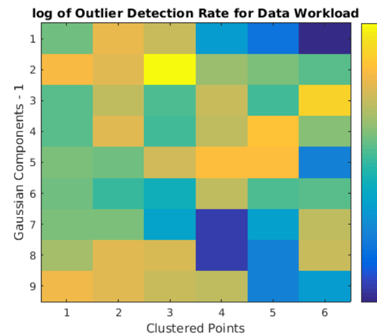

Figure 5: Data workload success

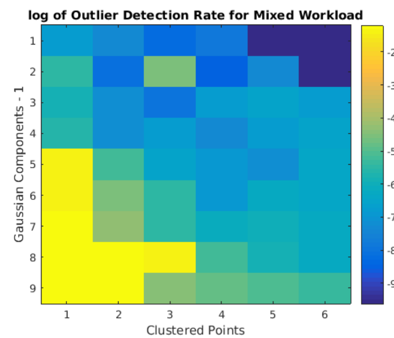

Figure 6: Mixed workload success

5 Experimentation

The majority of experimentation was conducted to find the optimal values for model parameters. In particular, three main parameters were found: number of Gaussian components in the model, data point clustering (number of points clustered together to create a higher dimensional, causally related set of features), and the threshold for determining if a point is an anomaly. We first optimize components and clusters parameters, since we rationalize that these parameters are invariant relative to each other to changes in the anomaly threshold. The primary statistics we use throughout this investigation are the false positive rate on non-payload test sets and the approximated true positive rate on payload test sets.

From a technical standpoint, we use the Gaussian Mixture Model (GMM) implemented in the scikit-learn[5] package as the basis for the model. We use Python scripting to iterate over different parameter values and test the model on each of them. It should be noted that we constrain the model to use a diagonal covariance matrix, since that showed the best results in our preliminary investigation.

Components and Clusters Parameters

We wish to find the pair of values for these parameters that both minimizes the false positive rate and maximizes the true positive rate, where payload points are designated to be positive. Difficulty in this approach arises from the fact that we do not actually know the exact number of data points in payload test sets that are anomalous. However, using graphs of the time series of the data, we predict that payload points make up approximately 5% of the points in data collected using 10MiB payloads. Experimentation was conducted by training our GMM using non-payload data from each workload as training data. False positive rate for each workload was found by testing the trained GMM on a non-payload data set from each workload. We repeat this training-testing patern for each pair of parameters in the range of 2-10 Gaussian components and 1-6 points clustered. A similar procedure was conducted for determining the positive rate, except we test on 10MiB payload data sets for each workload. For operations in this section, the threshold parameter was held at 0.05. Figures 1,2,3 show the aggregated false positive rate for workloads, while figures 4,5,6 show the anomaly detect rate. True positive rate can be estimated by taking the difference between the false positive rate and the anomaly detect rate. Note that the logarithm of the experimental results was taken to create a larger color distinction in the resulting heatmaps.

Observation of these heatmaps allows us to generate the following heuristics when determining parameters. First, as is most apparent in the false positive rate for the mixed workload (Figure 3), but can also be seen in the other figures, there is a substantial increase in the false positive rate once the number of Gaussian components exceeds five. There is little variation among values less than or equal to five. This suggest that the number of Gaussian components should be parameterized to five. The second trend that can be seen is that even though the false positive rate generally decreases as clustering increases, so does the anomaly detection rate. The implication is the overall true positive rate decreases as clustering increases. By inspection, a good value for the clustering parameter would be in the range of 1 to 3. It is our opinion, however, that 2 is the best value.

Threshold Parameter

We use the parameters found in the previous section to perform a similar procedure as before. Holding components at 5 and clusters at 2, we iterate over threshold values in the range of 0.05 to 0.40 in increments of 0.05 and record the anomaly detect rate. It should be stated that the threshold parameter operates by labeling a point as an anomaly if any of the posterior probabilities associated with that point are greater than THRESHOLD or less than 1 - THRESHOLD. The results of this experiment using the 10MiB payload data can be seen in figure 7.

Since we are targeting a detect rate of around 0.05 for payload workloads (this is a very rough estimate), it seems that 0.10 may be a good choice for the threshold parameter. However, the fact that anomaly rate for the branch and data workloads with payload were higher for 0.05 without a significant sacrifice in the false detect rate implies that these may have been caused by payload data points. For future investigations, it is essential to determine a better way of detecting true positives and false negatives.

Results

In this investigation, based on the statistics that we chose for the data, we find that we are able to obtain low false positive rates, on the order of 0.1% in many cases. However, despite the discussed uncertainty in the exact values for true positive rate in payload data sets, we observe that the mixed workload experiences low true positive rate relative to those displayed by the other workloads invariant of (reasonable) parameter values. From this we conclude that ROP payload detection on workloads similar to that of mixed would not work, whereas there would be a decent chance of detecting the payload on branch and data characterized workloads, especially if the threshold parameter where further tweaked.

6 Future Work

There's three main areas where we can improve the results. First, we would like to collect better data. To improve this, we can improve the simulation platform to better simulate common tasks. Ideally, instead of a simulation, we can test it on actual applications that people run. Instead of simulating a workload, we can instead inject vulnerabilities into common applications and test our algorithms that way. We can also improve the ROP payload generation to support JOP (jumps instead of returns) as well as making function calls (something common to regular ROP payloads that we do not account for). We also should work on recognizing multiple processes and multiple cores, which would represent a more realistic execution environment.

Second, we would like to speed up reaction time and also use less resources for the detection. It is impossible to realize both of these goals at once, so there will have to be a trade-off. Right now, we reserve a CPU core just for monitoring, but detection might be more effective with a dedicated hardware that performs both the data collection and processing. Creating a FPGA implementation might result in better performance. We have

Workload	t=0.05	t=0.10	t=0.15	t=0.20	t=0.25	t=0.30	t=0.35	t=0.40
Branch nopayload	0.0010	0.0007	0.0006	0.0005	0.0004	0.0003	0.0001	0.0001
Data nopaylod	0.0018	0.0013	0.0011	0.0009	0.0007	0.0004	0.0003	0.0002
Mixed nopayload	0.0008	0.0005	0.0004	0.0002	0.0002	0.0002	0.0001	0.0001
Branch payload	0.0988	0.0441	0.0229	0.0138	0.0082	0.0053	0.0035	0.0022
Data payload	0.0751	0.0546	0.0417	0.0323	0.0251	0.0181	0.0137	0.0084
Mixed Payload	0.0007	0.0006	0.0004	0.0003	0.0002	0.0002	0.0001	0.0001

Figure 7: Anomaly detect rate for different threshold values

not attempted to perform the classification locally (we process the data externally) because we focused on getting high quality data rather than tractability. Once we start to optimize for performance, we will lose the fine granularity of the sample collection. The challenge will be to still get good data as will as process it in time to make a decision.

Lastly, we wish to improve our classification algorithm. In our experiments, we did not make use of feature selection at all; instead we manually chose features based on our knowledge of computer architecture. There may be events we ignored from consideration that actually work well in our model. There may also be other ways of combining the event counts into features we have not considered. Since we were limited to four features by the CPU architecture, we would ideally like to collect all possible raw features and then run a rigorous feature selection. We also should consider other supervised and unsupervised algorithms to see if we can get better performance or reliability. Specifically, to recognize the mixed (and similar) workload. One immediate approach that comes to mind is to try to use a different conditional distribution for the EM algorithm, since we question whether the data is accurately modeled using Gaussians.

7 Conclusion

In continuing the work done earlier this year, we made another step towards a future of malware detection by classifying how code executes (versus the current standard of classifying what code executes). We showed that the PMU built into all modern ARM processors allow for fine grained access to architectural events that allows for detection of ROP payloads that are too small to recognize on x86[7]. We then used GMM to classify the execution traces.

Although our results are currently constrained to a simplified and unrealistic model (single process running on a single core) and our classification is done offline, we showed that there is promise in this avenue of work. Though the machine model we used is perhaps not the most suited, since we assume that the data is Normally distributed, we were able to obtain reasonably successful results for two out of our three workload types. Though it is concerning that the workload that failed was the mixed one, which we speculate is the one most likely to appear in actual systems, the overall results show that there is promise in the system. With more advanced data collection and feature selection, it would not be surprising to see this approach succeed on workloads more general than any we tested.

We hope that our detection method, along with recent work on control-flow integrity (which attempts to block ROP execution) will help defend computer systems against modern attackers.

References

[1] ARM. *ARM Architecture Reference Manual*, armv6 ed.

[2] DEMME, J., MAYCOCK, M., SCHMITZ, J., TANG, A., WAKSMAN, A., SETHUMADHAVAN, S., AND STOLFO, S. On the feasibility of online malware detection with performance counters. *SIGARCH Comput. Archit. News 41*, 3 (June 2013), 559–570.

[3] HP. *Data Execution Prevention*, 1.2 ed.

[4] NG, A. Mixtures of Gaussians and the EM algorithm, 2015.

[5] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[6] PI, R. Raspberry Pi 2 Model B, 2015.

[7] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. J. Unsupervised anomaly-based malware detection using hardware features. *CoRR abs/1403.1631* (2014).

Notes

[1] Our ARM Cortex A7 supports collecting up to four events

[2] We actually found a CPU bug in the Broadcom BCM2836 (ARM Cortex A7) device. The DBGDSAR register was implemented incorrectly, storing an absolute address instead of an offset. This is indication that debug registers may not verified as strenuously as the rest of the system.