Barnum: Detecting Document Malware via Control Flow Anomalies in Hardware Traces

Carter Yagemann[1(✉)], Salmin Sultana[2], Li Chen[2], and Wenke Lee[1]

[1] Georgia Institute of Technology, Atlanta, GA 30332, USA
yagemann@gatech.edu, wenke@cc.gatech.edu
[2] Security and Privacy Research, Intel Labs, Hillsboro, OR 97124, USA
{salmin.sultana,li.chen}@intel.com

Abstract. This paper proposes Barnum, an offline control flow attack detection system that applies deep learning on hardware execution traces to model a program's behavior and detect control flow anomalies. Our implementation analyzes document readers to detect exploits and ABI abuse. Recent work has proposed using deep learning based control flow classification to build more robust and scalable detection systems. These proposals, however, were not evaluated against different kinds of control flow attacks, programs, and adversarial perturbations.

We investigate anomaly detection approaches to improve the security coverage and scalability of control flow attack detection. Barnum is an end-to-end system consisting of three major components: (1) trace collection, (2) behavior modeling, and (3) anomaly detection via binary classification. It utilizes Intel® Processor Trace for low overhead execution tracing and applies deep learning on the basic block sequences reconstructed from the trace to train a normal program behavior model. Based on the path prediction accuracy of the model, Barnum then determines a decision boundary to classify benign vs. malicious executions.

We evaluate against 8 families of attacks to Adobe Acrobat Reader and 9 to Microsoft Word on Windows 7. Both readers are complex programs with over 50 dynamically linked libraries, just-in-time compiled code and frequent network I/O. Barnum shows its effectiveness with 0% false positive and 2.4% false negative on a dataset of 1,250 benign and 1,639 malicious PDFs. Barnum is robust against evasion techniques as it successfully detects 500 adversarially perturbed PDFs.

Keywords: Malware · Automated analysis · Classification · Deep-learning

1 Introduction

Control flow hijack attacks are still prevalent with nearly one million new exploit malware being reported in Q2, 2018 [3]. These attacks typically exploit memory corruption vulnerabilities to redirect the target program's control flow and gain

© Springer Nature Switzerland AG 2019
Z. Lin et al. (Eds.): ISC 2019, LNCS 11723, pp. 341–359, 2019.
https://doi.org/10.1007/978-3-030-30215-3_17

arbitrary code execution. They continue to exist despite protection mechanisms like control flow integrity (CFI) and code-pointer integrity (CPI) [22] due to compromises between accuracy and performance.

A serious concern to these systems is that they have been proven to only detect deviations from their models, which does not account for all control flow anomalies. Beyond exploits, over 97% of document malware rely on macros and ABIs that do not violate program integrity [29,34]. *Such abuse requires anomaly detection rather than integrity enforcement to detect and analyze.*

In this context, neural networks able to automatically learn features show great promise in detecting complex malware with high accuracy and scalability. Deep learning based malware detection has mostly focused on analyzing executable files and runtime ABI calls. The static analysis approaches use the headers, instruction opcodes, or raw bytes of an executable file to build models and classify the file before execution [10,35,36]. The dynamic approaches profile user or kernel ABI call sequences during execution [19,21,39]. Despite their potential, these classifiers are vulnerable to adversarial attacks [15,17,37].

We investigate anomaly detection approaches to build a robust offline system to detect control flow attacks. To be particular, we want to extend detection to unknown attacks against binaries that lack source code such as commercial off-the-shelf software, third-party libraries, and legacy programs. For security use cases, anomaly detection is more suitable than classification, since it is neither practical nor scalable to create behavior models for all possible attacks. In this paper, we describe Barnum, an anomaly detection based system that applies deep learning on hardware execution traces to build a per-application behavioral model and detect control flow attacks. The recent advancement in deep learning behavior modeling and hardware execution tracing enables us to efficiently trace many executions of a program with different inputs and build an automated system to identify expected behaviors from this vast volume of data.

We utilize Intel® Processor Trace (Intel® PT), a low overhead tracing feature in the CPU, to get the complete control flow audit of a program execution. Intel® PT records the non-deterministic control flow transfers, contextual, timing, etc. information, which combined with the program binary can be used to reconstruct the executed instruction sequences. We summarize the instructions into *basic blocks* (BBs), a sequence of linear instructions ending with a branching instruction, and assign each BB a unique BBID, creating a long sequence. We develop a hypervisor-based framework that makes the trace collection and processing secure and portable across OSes. Utilizing low level tracing also adds to the portability of Barnum across different OSes and hardware.

Barnum divides the control flow attack detection into two layers: (1) control flow modeling and (2) anomaly detection via binary classification. First, we train the normal behavior model of a program via self-supervised learning on benign traces. We then apply this model on unlabeled traces to predict the next BBIDs and use the prediction accuracy and confidence to learn the classification threshold of benign vs. anomalous traces.

We evaluate Barnum against 8 families of attacks (labeled by AVClass [42]) to Adobe Acrobat Reader and 9 to Microsoft Word on Windows 7. Both readers are complex programs with over 50 dynamically linked libraries, frequent use of just-in-time (JIT) compilation, network I/O for auto-updating and fetching remote content, and have known bugs that have been successfully exploited in the wild by attackers. Barnum shows its effectiveness with 0% false positive (FP) and 2.4% false negative (FN) on a dataset of 1,250 benign and 1,639 malicious PDFs and 0% FP, 10% FN on 200 benign and 379 malicious Word documents. The latter dataset is more challenging because 94% of the samples rely on ABI abuse, not exploits, which is outside the scope of related dynamic analysis systems. 2 detected Word malware samples are fully undetected on VirusTotal (VT). Barnum is able to handle programs that utilize JIT compilation by observing the control flow into and out of JIT regions without having to analyze the JIT code execution. Additionally, we use Mimicus [45] to perturb 500 malicious samples and confirm that the performance of Barnum does not degrade.

To summarize, we make the following contributions:

- We develop an offline anomaly detection based control flow attack detection system that applies deep learning on fine-grained control flow traces of an application. We describe the design challenges and architecture of Barnum.
- We utilize Intel® PT to collect control flow traces. The *hypervisor based* processing of *low level traces* make Barnum secure and portable across systems.
- We represent the control flow trace as a sequence of basic blocks and develop a multi-layer system to model program behavior for anomaly detection.
- We extensively evaluate Barnum against 8 families of attacks to Adobe Reader and 9 to Microsoft Word on Windows 7 64 bit. The experimental results show 0% FP, 2.4% FN and 0% FP, 10% FN to classify benign and malicious PDF and Word documents, respectively. We show that Barnum is resilient to adversarial attacks like Mimicus.

The rest of the paper is organized as follows: the next section describes the problem and provides background information. Section 3 details the design of Barnum. Our evaluation of Barnum on document malware is presented in Sect. 4. Section 5 covers related work and we conclude in Sect. 6.

All the Barnum source code, malware hashes, and data for reproducing results are available at the project homepage[1] or by contacting the first author.

2 Problem and Background

This section describes the problem Barnum is designed to address. We discuss our assumptions in the context of our adversary model and evaluation scenarios.

[1] https://tinyurl.com/y27clrfl.

2.1 Threat Model and Assumptions

In this work, our goal is to detect document malware via control flow anomalies. Most attacks against programs rely to some degree on changing the execution flow of the target program. Although it is possible to construct data-only attacks [8], most adversaries still rely on techniques like Return-Oriented-Programming to craft exploit payloads [6]. With this in mind, we consider a target program that may contain memory corruption vulnerabilities that an adversary can exploit to run a control flow manipulation attack. We also consider patterns like ABI abuse, which do not violate integrity. For example, a malicious script may invoke one ABI to save a file followed by another to execute it. This is allowed by the software specification, but is not the typical usage pattern.

We also assume that the libraries imported by the program contain vulnerabilities and that there may be dynamic code generation, even when the program is not under attack. For example, Adobe Acrobat Reader performs just-in-time (JIT) compilation on JavaScript.

2.2 Document Malware

We focus on offline analysis rather than online detection or prevention. We rely on dynamic analysis that, similar to related systems [38], executes the given sample for a fixed duration of time. It is possible for malware to employ techniques that detect the analysis environment [12,31] or delay execution beyond the observed time frame [20]. However, unlike general executables, malicious *documents* begin in a viewer program and rely on either scripts or malformed elements (e.g. CVE-2018-4990) to gain control. Thus, their options for environment detection are limited and can in themselves create a detectable signal. If the viewer is closed before the malware has gained control, the attack will be prematurely terminated. Thus, *document* malware cannot stall or inject benign activity for the durations general malware can. *Therefore we acknowledge the known limitations of virtualized dynamic analysis, but argue the compromises are reasonable for the document malware context.*

An additional challenge to document malware analysis is that even when an application is under attack, most of the overall activity can still be benign. For example, a trace of Acrobat Reader opening a malicious PDF will contain behaviors like creating the GUI, which our analysis must be robust to.

Lastly, not all document malware rely on exploits. While many do use vulnerabilities in the document viewer to hijack the program execution, some rely instead on combining the provided ABIs in abusive ways. For example, several PDF malware use `exportDataObject` to save and execute an attachment. The user is warned about such behavior with a message window, but if they click accept or disable the warning, the attack will succeed. Since ABI abuse does not violate control flow integrity (i.e. the program is functioning as intended), mechanisms like CFI and memory safety are not appropriate solutions. These ABI invocations, however, are reflected in the resulting control flow trace and

hence, still create a signal. In Sect. 4 we show that compared to exploits, ABI abuse is harder for Barnum to detect, but is still distinguishable in many cases.

3 Design

Figure 1 shows the system architecture of Barnum. It consists of three major components: (1) execution trace collection, (2) program behavior modeling, and (3) anomaly detection via binary classification. Barnum is evaluated on document malware targeting Acrobat Reader and Microsoft Word on Windows 7, but due to its OS and program agnostic design, the methodology can easily be expanded to cover other programs that process discrete inputs, such as web services.

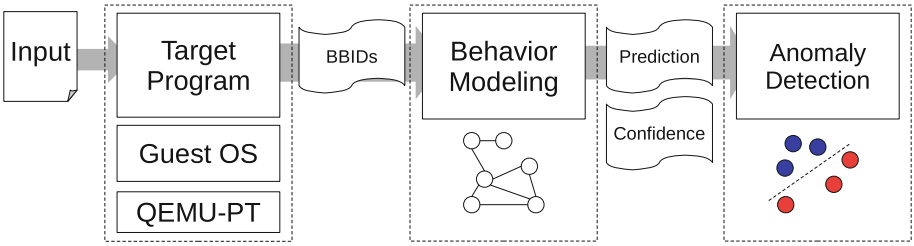


Fig. 1. Barnum has three components: (1) trace collection, (2) behavior modeling, and (3) anomaly detection.

3.1 Control Flow Tracing

The analysis of a program input begins with the collection of a trace. For this component, there are several design challenges that need to be addressed: *(a) how do we efficiently generate traces of the program execution, (b) since we need a trusted component to collect traces, how does it bridge the semantic gap to the rest of the system, and (c) how do we collect the traces and process them into a useful representation?* Underpinning this is the obvious security requirement that traces must be complete, untampered with, and difficult to evade.

Efficient Tracing. We need an efficient and secure way to collect control flow traces so we can analyze document malware. For this reason, we decide to leverage Intel® PT at the hypervisor level. Intel® PT is a hardware tracing feature found in recent Intel® CPUs. Using Intel® PT, developers can trace the CPU to get a rich stream of data including branching taken-not-taken (TNT), target instruction pointer (TIP) for indirect calls and jumps, power events, hardware interrupts, etc. These events are recorded asynchronously as a stream of packets directly into memory. The CPU guarantees that packets will be recorded

in the correct order upon instruction retirement. In other words, an Intel® PT trace only records what is actually executed. Due to its asynchronous nature, Intel® PT introduces minimal performance overhead; under 4% in our measurements. This is several orders of magnitude faster than approaches like binary instrumentation, allowing Barnum to analyze and classify more samples.

Semantic Gap. Intel® PT has built-in filtering options to control what is traced. Of particular interest to this work are CR3 and current privilege level (CPL) filtering. As these names imply, they configure Intel® PT to only enable tracing when a particular value is loaded into the CR3 register and when the CPU is in a particular CPL. We combine these filters to trace only the user space activity of our target process. We use a configurable agent inside the virtual machine (VM) to start the target program and open the document sample.

One caveat to this approach is we need a way to associate CR3 values with processes. To overcome this challenge, we leverage virtual machine introspection (VMI) to monitor the guest kernel's process list.

For security and portability, we control Intel® PT from a hypervisor and configure it to write the trace into reserved memory. Since the hypervisor has exclusive control over the Intel® PT configuration, the traces are ensured to be complete and untampered with. The trace is then saved to storage for analysis.

Data Preprocessing. Unfortunately, an Intel® PT trace alone does not tell the full story about the program control flow. For example, TNT packets record a single 1/0 bit for conditionals, which is insufficient to detect loops. To construct meaningful representations of the control flow, we need the corresponding instructions so we can calculate all the branch targets. This requires additional sideband data. Specifically, we use VMI to read the memory mapping from the guest kernel's data structures and recover the executable pages *immediately before opening the document sample*. Thus, we only miss dynamically generated code or late loaded libraries. Once collected, we combine the program binary with the Intel® PT trace to reconstruct the exact sequence of instructions executed and their corresponding addresses. We then use the memory mapping to normalize addresses as offsets within libraries and executables.

We summarize the instructions into basic blocks (BBs), defined as a sequence of linear instructions ending with a branching instruction (i.e. indirect calls and jumps, direct branches, and returns) and assign each basic block a universally unique *BBID*. We represent the control flow trace as a long sequence of BBIDs for program behavior modeling. Our implementation extends kAFL [41] to support user space tracing and we use `libipt` [1] for trace decoding.

Just-In-Time Compilation. For many programs, dynamic code generation occurs due to JIT compilation of scripting languages like JavaScript. Since it is not feasible to enumerate a representative set of possible scripts for documents, we instead elect to disregard this code in our analysis. When the program jumps into a code region that is dynamically generated, we stop disassembling the Intel® PT trace until the execution returns to a region that was not dynamically generated,

Last Instr	inc	movzx	icall	pop	ret	push	push	icall	pop	pop	pop	ret	pop	inc
Subseq #1	701	224	612	968	511	332	172	82	179	20	721	33	422	187
Subseq #2	701	224	612	968	511	332	172	82	179	20	721	33	422	187
Subseq #3	701	224	612	968	511	332	172	82	179	20	721	33	422	187
Subseq #4	701	224	612	968	511	332	172	82	179	20	721	33	442	187

Fig. 2. An example of sub-sequencing the BBIDs for a sliding window of size 3. The cells contain BBIDs and the top row is the last instruction in each BB. Each row shows a subsequence contained in the same 14 BBIDs. The lighter cells are the features and the darker cells are the labels.

upon which we resume disassembly. This means that while we do not attempt to analyze the control flow inside the JIT code, we still capture the points at which the program enters and exits it. Similarly, while we do not record kernel space, the trace captures where entries and exits occur. This allows us to observe the boundary between these worlds, which is sufficient to detect patterns like ROP chains and shellcode injection because programs typically have a binding layer that the control flow always passes through [33]. In other words, observing transitions that do not go through the binding layer of the program, enter system libraries directly from JIT code, etc., are indicative of an attack.

3.2 Control Flow Behavior Modeling of a Program

To model the normal control flow of a target program, we have to address the following: *(a) how do we slice the long BBID sequence into manageable subsequences, (b) what model should we use to represent normal control flow paths, and (c) what do we do about code coverage?*

Data Slicing. Since control flow hijacking only occurs at indirect calls, jumps, and returns, we only need to analyze subsequences that end on one of these instructions. Figure 2 shows how we use a fixed sliding window with variable width steps so that each frame ends on such instructions. The next BBID after the frame will become its label, which we discuss later. The optimal window size is experimentally found to be 32 BBIDs for our model.

Deep Learning Model Selection. While we have the intuition that knowing past control flow is useful for predicting future execution, shadow stacks being one such example, it is not trivial to utilize this history to achieve accurate predictions. Using heuristics would be neither practical nor scalable. Instead, we turn to machine learning (ML) to find these patterns automatically.

We structure our behavior model as a supervised learning problem where, given a fixed window of past BBIDs, we want to predict the next BBID. In machine learning terms, the features are the sequence of past BBIDs and the label is the next BBID in the trace. As we explain earlier, we do not directly

analyze JIT code. Therefore, the max number of BBIDs is fixed and can be over-approximated. Since the trace encodes both the features and labels, this type of learning falls into the subcategory of *self-supervised learning. The advantage of this approach is it does not require manual ground truth annotation, allowing for better scalability. This is essential given how much data Intel® PT generates.* To train a program behavior model, we only use traces from the benign dataset since we want to learn paths under normal conditions.

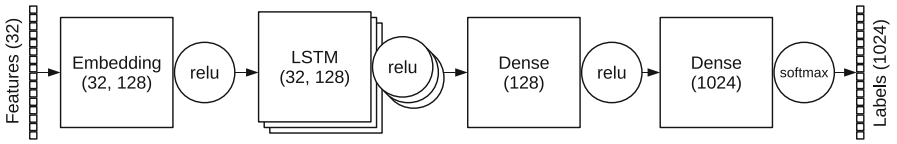


Fig. 3. The layers, shapes, and activations of our model. Recall that the subsequence length is 32, hence 32 features. Not shown is a 50% dropout between the two dense neural layers. We bucket the BBID predictions into 1,024 labels to reduce model size.

Since our features are temporally related, we center our model around recurrent neural networks, specifically Long Short Term Memory (LSTM). Our model consists of an embedding layer followed by three LSTM layers and a final dense neural network, as shown in Fig. 3. We find the exact layout and hyper-parameters experimentally. To reduce model size and improve runtime performance, we map the BBIDs to buckets, which shrinks the input and output spaces. To find the ideal number of buckets, we start with each BBID being its own bucket (i.e. no reduction) and merge buckets until accuracy degrades. For our datasets, the ideal number is 1024. We decide not to use classical ML models like n-gram because small n values miss features and large values produce sparsity leading to performance degradation. In short, picking n is hard whereas deep learning simplifies feature selection. The ability of LSTM to automatically extract both long and short-term features makes it well suited to our context. By contrast, convolutional neural networks have yet to demonstrate higher accuracy, but are known to be susceptible to minimal perturbations [4]. In Sect. 4, we evaluate simpler approaches to this problem to demonstrate the added value of our ML technique.

Code Coverage. A legitimate concern is whether the training dataset can cover all possible benign execution paths of a target program. This is known as the *code coverage* problem [11]. Code coverage is known to be difficult to guarantee, especially when source code is not available. We approximate the coverage of our dataset in Sect. 4 by counting the number of executed instructions.

3.3 Anomaly Detection

Now that we have trained a behavior model to predict normal control flow paths, we move on to the third major component of Barnum: anomaly detection. The

major challenge here is how to go from path prediction to anomaly detection. We use the accuracy and confidence of the model on benign traces to determine a decision boundary for classifying benign vs. anomalous executions. Given the BBID subsequences for a trace and the behavior model, we feed each subsequence into the model, get the prediction and confidence for the next BBID, and check whether the prediction is correct or not. This process results a sequential list of $\langle confidence\, percentage, prediction\, correctness \rangle$ tuples. Since Barnum is a dynamic analysis framework that records traces for a fixed duration of time, for a given trace, we average the prediction accuracy and confidence corresponding to wrong predictions. This gives us two dimensions to examine with our intuition being that *wrong* predictions with *high* confidence are a signal for anomalies.

Given these dimensions, we create a linear decision boundary to express a threshold. Any data points above the threshold are considered normal while points below the threshold are anomalous. It is possible to use other kernels to express the decision threshold, but in practice we find a linear boundary to be sufficient to achieve high classification accuracy.

4 Evaluation

In this section, we present our evaluation results for Barnum. Specifically, we evaluate each of the three components of Barnum in the context of analyzing document malware targeting Adobe Acrobat Reader 9.3 and Microsoft Word 2010 on 64 bit Windows 7. We analyze two distinct programs to demonstrate Barnum is not overly tailored to a particular program.

4.1 Overview

To evaluate Barnum, we ask several questions:

- **Path prediction.** How well can Barnum learn the control flow of complex programs like Acrobat Reader? How does it compare to simpler methods like rote learning?
- **Document malware classification.** How accurately can Barnum classify previously unseen traces to separate benign and malicious documents? How does it compare to other related work in malware classification?
- **Resource consumption.** How much memory and storage does Barnum consume to analyze samples? How long does it take to perform analysis?

4.2 Datasets and Experimental Setup

For our PDF dataset, we consider 3,660 samples, of which 1,250 are benign and 2,410 are malicious. These samples are picked randomly from a malware feed spanning several years. The benign samples are from several sources including past conference proceedings and the 2013 Contagio dataset [30]. We confirm that our benign documents contain embedded JavaScript, Flash, and Shockwave so simply detecting active content will not trivially lead to accurate classification.

For our malicious dataset, we start with 2,410 malicious samples that match known signatures from anti-virus companies. Since our system is built on runtime tracing, we have to manually verify that these samples exhibit their malicious behavior in our target VM. For example, some malicious PDFs carry exploits for particular versions of Acrobat Reader and will not perform any malicious activity if this requirement is not satisfied. We use older program versions to maximize the chance of triggering the malware, but even then there are unreliable exploits that simply crash. Lastly, some attacks are embedded in remote content that the document references. Once the hosting server is taken down, the document becomes benign. After manual filtering, our malicious dataset contains 1,639 samples; about 68% of the original set. In the context of evaluating dynamic malware analysis systems, this is typical. For comparison, PlatPal [48] had only 320 malware after filtering, of which their system detected 75.9%.

PDF							
pdfjsc	perferd	name	singleton	tiff	swrort	pidief	pdfka
2	2	4	7	10	99	201	1,314

Word								
powload	emotet	powdow	sagent	valyria	sload	donoff	obfuse	singleton
3	5	8	8	23	25	33	61	206

Fig. 4. AVClass label counts. Singleton labels are merged into a single category.

In total, AVClass produces 8 unique labels for this dataset with 7 samples producing no family label (i.e. AVClass classifies them as singletons). The distribution is shown in Fig. 4. We also randomly perturb 500 of our malicious samples using Mimicus [45] to evaluate the robustness of Barnum.

To demonstrate that Barnum is not tailored to Acrobat Reader, we also evaluate a dataset of 200 benign and 379 malicious Word documents. AVClass produces 9 unique labels for this dataset with *206 samples classified as singletons*. The high number of singletons is due to low detection rates and matches to signatures with uninformative names on VT. For example, 19 malware are detected by 3 or less of the 60 anti-virus products used by VT and 2 samples are fully undetected at the time of writing. Several matched signatures are simply named *heuristic*. In short, our Microsoft Word dataset is more challenging for existing anti-virus than the PDF dataset.

A point worth stressing is *Barnum handles Acrobat Reader and Microsoft Word without modifying any lines of source code*. We only adjust two settings to select which program the agent starts and the process name to trace.

Our experiments are performed on a single desktop with an Intel® i7-6700K CPU and Nvidia 1080-Ti GPU. The GPU is only used by the behavior model.

4.3 Baseline for Comparing PDF Detection

Most existing solutions for PDF malware analysis examine static features extracted from the document [9,26,28]. These systems are very accurate (>95%), but are also known to be vulnerable to perturbations like the one proposed in Mimicus. For dynamic analysis, solutions like CWXDetector [47] achieve upwards of 93% detection, but have limitations like not being able to detect exploit based attacks [40,44,46] or code reuse [25]. A more recent work, PlatPal [48], avoids these limitations, but only detects 75.9% of their 320 samples as malicious or suspicious. We compare Barnum accordingly:

1. We compare the behavior model to a database that performs rote learning to measure the value our LSTM model adds to Barnum.
2. We show that our detection accuracy is better than CWXDetector and PlatPal, which also perform dynamic analysis on PDF malware.
3. We show that our detection is robust against Mimicus [45], which evades systems based on static document input features.

	[25]		[47]		[48]		**Barnum**	
TP	91.7%	(917)	93.2%	(6,781)	75.9%	(243)	**97.8%**	(1,600)
FN	8.3%	(83)	6.8%	(497)	24.1%	(77)	**2.4%**	(39)
TN	100%	(994)	100%	(7,278)	100%	(1,030)	100%	(375)
FP	0%	(0)	0%	(0)	0%	(0)	0%	(0)

Fig. 5. Comparison of Barnum against related dynamic PDF analysis systems in terms of true positive (TP), false negative (FN), true negative (TN), and false positive (FP) rates. Values are shown as counts and percentages. Barnum has the best TP and FN.

A comparison of our results to related work is presented in Fig. 5.

4.4 Path Prediction for PDF Dataset

To evaluate the accuracy of our behavior model, we randomly split our benign PDF samples into a training set of 875 and a testing set of 375. Recall that we do not use the malicious PDFs in this stage of the pipeline.

We also use these same sets to train and test a model based on rote learning. Specifically, for each subsequence given to the rote learner during training, it looks up the corresponding row in its database and increments the label (i.e. the next BBID that follows the subsequence) by one. This creates a database that counts how often each label occurs for any subsequence seen during training. To conserve memory, we store this database in Redis, which uses a compressed key-value encoding and only create rows for subsequences seen during training. For testing, the model checks if it has seen the given subsequence before. If it has, it

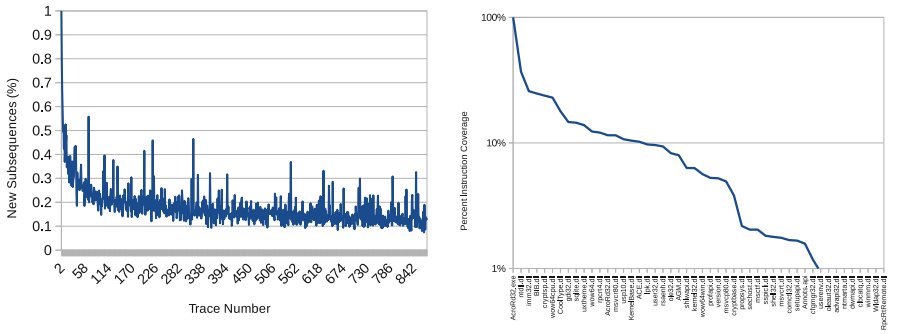
makes a weighted random guess based on the counts of all the possible labels. If the subsequence never occurred in training, it makes a completely random guess.

Note that we can adjust the learning capacity of the LSTM model by adding and removing nodes, which impacts accuracy and in-memory size. For comparison, we adjust the LSTM to achieve the same accuracy as the rote learner. Under this condition, the rote learner's database takes up 2,030 MB of memory whereas the LSTM model only needs 2.7 MB. In short, back-propagation enables LSTM to extract the most important patterns instead of memorizing everything.

4.5 Training Coverage of PDF Dataset

As part of the evaluation for our behavior model, we consider how well our benign PDF training dataset covers Acrobat Reader. First, we measure the novelty each additional trace adds to training. Specifically, for each trace, we extract the BBID subsequences and calculate the percentage that do not occur in previous traces. The result is summarized in Fig. 6(a). Note that within a single trace, if the same subsequence occurs multiple times, it is only considered once in the percentage calculation. As the figure shows, by the end of training we are still encountering traces where 15% of the subsequences are new. Unfortunately, the plot also appears to fit a power curve. If this trend is accurate, it implies that while getting decent coverage is doable, achieving excellent coverage is hard. Recent work in ML theory supports our finding [16]. However, even with only 875 benign training traces, we achieve better detection accuracy than the related work considered in Fig. 5.

In addition to the previous metric, we also consider code coverage in terms of how many unique instructions we encounter across Acrobat Reader and all its loaded libraries. The result is summarized in Fig. 6(b). Note that this figure is shown in log scale. We successfully cover 100% of the instructions in



(a) Novelty Curve (b) Instruction Coverage

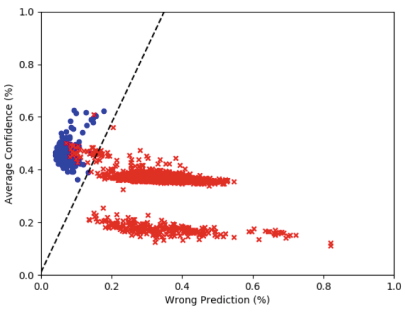
Fig. 6. The percentage of new subsequences appearing in each subsequent trace (left) and the coverage of instructions by library (right) for the Acrobat Reader dataset. Note that we do not expect full coverage of third-party libraries.

`AcroRd32.exe`, but on closer inspection we notice this binary is fairly small; only a few thousand machine instructions. It is likely that this is mostly bootstrapping logic and the core functionality is actually located in `AcroRd32.dll`, where we cover only 12% of the instructions. This low coverage is not surprising given all the features Acrobat Reader contains that we do not invoke like automatic updating and document editing tools. It is unclear to what degree (if any) adding more traces would increase coverage. Similarly, low coverage in libraries is expected since no program uses every function in every linked library. Interestingly, we cover 37% of `ntdll.dll`, which is used to interact with the Windows kernel.

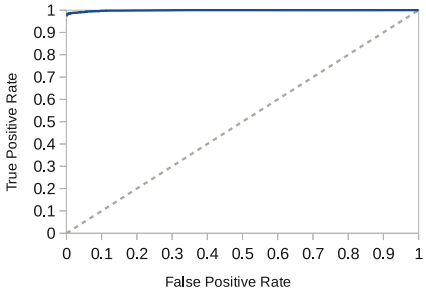
4.6 Anomaly Detection for PDF Dataset

As described in Sect. 3, once Barnum has a model trained for predicting the normal paths of our target program, it can be used by the next layer to perform binary classification between normal and anomalous traces. To start, we take our benign training samples and calculate the behavior model's average accuracy and misprediction confidence for each. We then set a threshold expressed as a linear decision boundary such that all these samples fall on one side. This becomes the normal side. Anything that falls on the other side is an anomaly. Figure 7(a) shows that Barnum classifies the PDFs with 0% false positive and 2.4% false negative. *This translates to 98.1% accuracy, 100% precision, and 97.6% recall.* These results are significantly better than CWXDetector (6.8% false negative) and PlatPal (24.1% false negative). In Fig. 7(b), we show the ROC curve for different thresholds.

To investigate, we randomly pick some misclassified samples and manually analyze them. Our finding is the malicious samples near the threshold boundary tend to abuse the Acrobat Reader ABIs (e.g. calling ABIs to save and execute an attachment) whereas those further from the boundary use exploits. For example,



(a) PDF Dataset

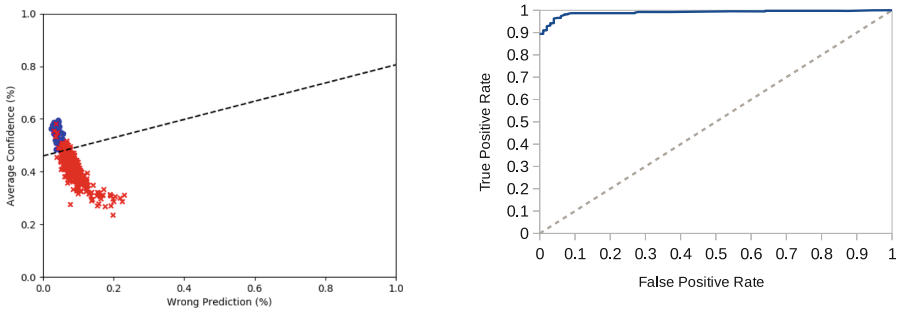


(b) ROC Curve

Fig. 7. Classification of our testing PDF dataset of 375 benign, 1,639 malicious, and 500 Mimicus perturbed PDFs. At 0% false positive, the false negative rate is 2.4%. The accuracy is not degraded by Mimicus.

all the `pdfka` traces contain an indirect transfer that appears hundreds of times, which is mispredicted by the model, causing this family to fall further from the decision boundary. The transfer site does not appear in any other traces and we believe it is indicative of an exploit against Acrobat Reader's TIFF parser (CVE-2010-0188). We could not find or create a benign PDF that invokes the same path.

To further demonstrate the value of Barnum, we randomly pick 500 samples from our set of 1,639 malicious PDFs and perturb them using Mimicus [45]. This is an evasion technique that adds additional DOM elements before the PDF trailer and modifies meta-data fields (e.g. author) to change the sample's appearance without altering its runtime behavior. As a result, it is effective against systems that rely on static features of the input document. Since Mimicus guarantees that the runtime behavior (e.g. exploit) is preserved, it is not effective against dynamic systems like Barnum. However, we *do not* claim that our system cannot be evaded just because it resists Mimicus. Rather, the point of including these samples is to show that *Barnum achieves accuracy that is comparable to systems based on static features while also achieving robustness comparable to dynamic systems*. For the adversary, changing the malicious PDF's behavior is more difficult than manipulating the static document features used by existing solutions because the former requires tweaking the exploit, without breaking it, while the latter is achievable with DOM element appends and meta-data tweaks.



(a) Word Dataset (b) ROC Curve

Fig. 8. Barnum classifies 100 benign and 379 malicious Word documents with 0% false positive and 10% false negative. The right figure shows the ROC curve.

4.7 Anomaly Detection for Word Dataset

To demonstrate that Barnum is not limited to Adobe Acrobat Reader, we also evaluate a set of 100 benign training, 100 testing, and 379 malicious Word documents using Microsoft Word 2010. The resulting classification and ROC curve are shown in Fig. 8. Barnum achieves 0% false positive and 10% false negative

rates on this dataset. The lower performance compared to the PDF dataset is due to most of the attacks relying on macros. Only 25 of the Word malware do not contain macros. Exploits are more prevalent in the PDF dataset and easier for Barnum to detect because their control flow overlaps less with benign traces. If we consider only the 25 Word samples without macros, our system's accuracy becomes 100% with no false positives or negatives. Two recent related work, ALDOCX [32] and the work by Bearden et al. [5], achieve lower false negative rates; 5.6% and 3.7% respectively. However, they use static document features, making them vulnerable to evasion techniques like Mimicus.

4.8 Runtime and Space Performance

The majority of our runtime cost comes from training the behavior model and using it to make predictions. It takes 3 epochs and 1.5 days to train on 875 benign PDFs. The low number of epochs is due to traces being long and repetitive, meaning fewer iterations are needed to converge. At test time, about 25% of the runtime is preprocessing, 75% is querying the path prediction model, and less than 1% is calculating the classification decision. Testing 1,783 PDFs takes 5.7 hours, which equates to about 7,500 samples per day per GPU. Traces are 30 MB on average when saved to storage (i.e. at rest).

5 Related Work

In this section, we summarize the related work in anomaly detection and document malware analysis.

5.1 Machine Learning Based Malware Detection

Anomaly detection has been applied to numerous domains such as networking [14,24], videos [27], and programs [13]. More recently, researchers have started exploring the value machine learning can add to anomaly detection [2,7]. LSTM in particular models anomalies well because it can handle sequences.

Barnum relates most closely to HeNet [7] in that both systems apply deep-learning on hardware traces to detect control flow attacks. Our work, however, surpasses HeNet in several regards. First, HeNet focuses solely on detecting exploit based attacks whereas Barnum expands its scope to include other attacks such as ABI abuse. Second, HeNet directly encodes its traces as images and uses transfer learning, whereas our behavior model is trained from scratch on BBIDs. While transfer learning makes training faster, training from scratch ensures that the layers only inherit deep latent features of trace data. Transfer learning cannot make this guarantee. Third, HeNet has not been extensively evaluated against a large dataset or adversarial inputs.

5.2 Document Malware

The proposals for document malware classification fall into two broad categories: analysis of static input features and dynamic execution. Analysis based on static input features has received the most attention [9,18,23,28,43] because it is faster, easier to scale, and takes advantage of the formal format structure of documents. Unfortunately, while they have solved problems like de-obfuscating scripts, they are known to be vulnerable to ML evasion techniques [45,49]. As we demonstrate in Sect. 4, since Barnum is based on control flow, altering the static structure is not enough to evade our system. Although we cannot claim our system cannot be evaded, evading our system is harder than evading systems based on input features because doing so requires alteration of the exploit.

On the other hand, there are fewer systems that classify documents based on dynamic behavior [26,47]. Most interesting is PlatPal [48], which runs a PDF in two different OSes and uses differential analysis to detect exploitation. Unfortunately, this approach is error prone due to sources of nondeterminism and subtle differences between implementations of the same program for different OS. It is also expensive due to the need for two virtual machines per sample.

6 Conclusion

This work introduces a methodology for collecting, modeling, and detecting control flow anomalies in an OS and program agnostic manner. We present Barnum, a prototype end-to-end system for collecting and analyzing traces of document editors opening benign and malicious PDF and Microsoft Word documents on Windows 7. We show that Barnum can classify documents with higher accuracy than other dynamic analysis frameworks and resists perturbations that thwart systems using static input feature.

Acknowledgement. This research was supported, in part, by the Intel Science and Technology Center for Adversary-Resilient Security Analytics. Some malware samples were provided by the Georgia Tech Research Institute Apiary framework. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

1. 01org: libipt (2018). https://github.com/01org/processor-trace
2. Aditham, S., Ranganathan, N., Katkoori, S.: LSTM-based memory profiling for predicting data attacks in distributed big data systems. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1259–1267. IEEE (2017)
3. C.B., et al.: McAfee Labs Threat Report. Technical report, McAfee Labs, September 2018
4. Athalye, A., Carlini, N., Wagner, D.: Obfuscated gradients give a false sense of security: circumventing defenses to adversarial examples. arXiv preprint arXiv:1802.00420 (2018)

5. Bearden, R., Lo, D.C.T.: Automated microsoft office macro malware detection using machine learning. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 4448–4452. IEEE (2017)
6. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: Proceedings of the 23rd USENIX Conference on Security Symposium (2014)
7. Chen, L., Sultana, S., Sahita, R.: HeNet: a deep learning approach on Intel processor trace for effective exploit detection. arXiv preprint arXiv:1801.02318 (2018)
8. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th USENIX Security Symposium (2005)
9. Corona, I., Maiorca, D., Ariu, D., Giacinto, G.: Lux0R: detection of malicious PDF-embedded Javascript code through discriminant analysis of API references. In: Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop, pp. 47–57. ACM (2014)
10. Davis, A., Wolff, M.: Deep learning on disassembly data. In: BlackHat USA (2015)
11. Fallah, F., Devadas, S., Keutzer, K.: OCCOM-efficient computation of observability-based code coverage metrics for functional verification. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **20**(8), 1003–1015 (2001)
12. Ferrie, P.: Attacks on more virtual machine emulators. Symantec Technol. Exch. **55**, 1–17 (2007)
13. Gao, D., Reiter, M.K., Song, D.: On gray-box program tracking for anomaly detection, p. 24. Department of Electrical and Computing Engineering (2004)
14. Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., Vázquez, E.: Anomaly-based network intrusion detection: techniques, systems and challenges. Comput. Secur. **28**(1–2), 18–28 (2009)
15. Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P.D.: Adversarial perturbations against deep neural networks for malware classification. CoRR abs/1606.04435 (2016)
16. Hestness, J., et al.: Deep learning scaling is predictable, empirically. arXiv preprint arXiv:1712.00409 (2017)
17. Hu, W., Tan, Y.: Black-box attacks against RNN based malware detection algorithms. CoRR abs/1705.08131 (2017)
18. Karademir, S., Dean, T., Leblanc, S.: Using clone detection to find malware in acrobat files. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, pp. 70–80. IBM Corporation (2013)
19. Kim, G., Yi, H., Lee, J., Paek, Y., Yoon, S.: LSTM-Based System-Call Language Modeling and Robust Ensemble Method for Designing Host-Based Intrusion Detection Systems. CoRR abs/1611.01726 (2016)
20. Kolbitsch, C., Kirda, E., Kruegel, C.: The power of procrastination: detection and mitigation of execution-stalling malicious code. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 285–296. ACM (2011)
21. Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C.: Deep Learning for Classification of Malware System Call Sequences, pp. 137–149 (2016)
22. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (2014)
23. Laskov, P., Šrndić, N.: Static detection of malicious Javascript-bearing PDF documents. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 373–382. ACM (2011)
24. Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., Srivastava, J.: A comparative study of anomaly detection schemes in network intrusion detection. In: Proceedings of the 2003 SIAM International Conference on Data Mining, pp. 25–36. SIAM (2003)

25. Liu, D., Wang, H., Stavrou, A.: Detecting malicious Javascript in PDF through document instrumentation. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 100–111. IEEE (2014)
26. Lu, X., Zhuge, J., Wang, R., Cao, Y., Chen, Y.: De-obfuscation and detection of malicious PDF files with high accuracy. In: 2013 46th Hawaii International Conference on System Sciences (HICSS), pp. 4890–4899. IEEE (2013)
27. Mahadevan, V., Li, W., Bhalodia, V., Vasconcelos, N.: Anomaly detection in crowded scenes. In: 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1975–1981. IEEE (2010)
28. Maiorca, D., Giacinto, G., Corona, I.: A pattern recognition system for malicious PDF files detection. In: Perner, P. (ed.) MLDM 2012. LNCS (LNAI), vol. 7376, pp. 510–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31537-4_40
29. Microsoft: New feature in office 2016 can block macros and help prevent infection (2016). https://cloudblogs.microsoft.com/microsoftsecure/2016/03/22/new-feature-in-office-2016-can-block-macros-and-help-prevent-infection/
30. Mila: 16,800 clean and 11,960 malicious files for signature testing and research (2013). http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html
31. Miramirkhani, N., Appini, M.P., Nikiforakis, N., Polychronakis, M.: Spotless sandboxes: evading malware analysis systems using wear-and-tear artifacts. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 1009–1024. IEEE (2017)
32. Nissim, N., Cohen, A., Elovici, Y.: ALDOCX: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. IEEE Trans. Inf. Forensics Secur. **12**(3), 631–646 (2017)
33. Niu, B., Tan, G.: RockJIT: securing just-in-time compilation using modular control-flow integrity. In: Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (2014)
34. Proofpoint: The human factor report 2016 (2016). https://www.proofpoint.com/sites/default/files/human-factor-report-2016.pdf
35. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.: Malware Detection by Eating a Whole EXE. ArXiv e-prints, October 2017
36. Raff, E., Sylvester, J., Nicholas, C.: Learning the PE Header. Malware Detection with Minimal Domain Knowledge, ArXiv e-prints, September 2017
37. Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y.: Generic black-box end-to-end attack against RNNs and other API calls based malware classifiers. ArXiv e-prints, July 2017
38. Sandbox, C.: Cuckoo sandbox (2018). https://cuckoosandbox.org/
39. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20, October 2015
40. Schmitt, F., Gassen, J., Gerhards-Padilla, E.: PDF scrutinizer: detecting Javascript-based attacks in PDF documents. In: 2012 Tenth Annual International Conference on Privacy, Security and Trust, pp. 104–111. IEEE (2012)
41. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: KAFL: hardware-assisted feedback fuzzing for OS Kernels. In: 26th USENIX Security Symposium, USENIX Security 2017, pp. 167–182. USENIX Association (2017)

42. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
43. Smutz, C., Stavrou, A.: Malicious PDF detection using metadata and structural features. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 239–248. ACM (2012)
44. Snow, K.Z., Krishnan, S., Monrose, F., Provos, N.: SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In: USENIX Security Symposium, pp. 183–200 (2011)
45. Šrndic, N., Laskov, P.: Mimicus: a library for adversarial classifier evasion (2016)
46. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.P.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the Fourth European Workshop on System Security, p. 4. ACM (2011)
47. Willems, C., Freiling, F.C., Holz, T.: Using memory management to detect and extract illegitimate code for malware analysis. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 179–188. ACM (2012)
48. Xu, M., Kim, T.: PlatPal: detecting malicious documents with platform diversity. In: 26th USENIX Security Symposium, USENIX Security 2017, pp. 271–287. USENIX Association (2017)
49. Xu, W., Qi, Y., Evans, D.: Automatically evading classifiers. In: Proceedings of the 2016 Network and Distributed Systems Symposium (2016)