Appeared in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 367–382, Beijing, China, September 2019.

Fluorescence: Detecting Kernel-Resident Malware in Clouds

Richard Li[⋆] Min Du[†] David Johnson[⋆] Robert Ricci[⋆] Jacobus Van der Merwe[⋆] Eric Eide[⋆]

[⋆]*University of Utah* [†]*University of California, Berkeley*

Abstract

Kernel-resident malware remains a significant threat. An effective way to detect such malware is to examine the kernel memory of many similar (virtual) machines, as one might find in an enterprise network or cloud, in search of anomalies: i.e., the relatively rare infected hosts within a large population of healthy hosts. It is challenging, however, to compare the kernel memories of different hosts against each other. Previous work has relied on knowledge of specific kernels—e.g., the locations of important variables and the layouts of key data structures—to cross the "semantic gap" and allow kernels to be compared. As a result, those previous systems work only with the kernels they were built for, and they make assumptions about the malware being searched for.

We present a new approach to detecting kernel-resident malware within a "herd" of similar virtual machines. Our approach uses limited knowledge of the kernels under examination—e.g., the location of the page global directory and the processor's instruction set—to concisely *fingerprint* each kernel. It uses no kernel-specific semantics to *compare* the fingerprints and find those that represent anomalous hosts. We implement our method in a tool called Fluorescence and demonstrate its ability to identify Linux and Windows hosts infected with real-world, kernel-resident malware. Fluorescence can examine a herd of 200 virtual machines with Linux guests in about an hour.

1 Introduction

Kernel-resident malware is stealthy. Once a kernel-resident rootkit infects a machine, it will generally try to hide traces of its intrusion, disable security software, and install persistent backdoors for future unauthorized access [12, 22, 39, 40]. Despite recent advances in protecting kernel integrity, kernel rootkits remain a significant threat: for example, at Black Hat 2017, Bulygin et al. [7] demonstrated a successful kernel rootkit attack against Windows 10, which has multiple kernel-protection mechanisms enabled.

To persist and perform malicious activities, a kernel rootkit must inject code into the kernel's address space [21]. Thus, in principle, an analyst can detect the presence of a kernel rootkit by comparing a memory snapshot of a suspect host against a memory snapshot of a clean, uninfected host running the same kernel. Such a comparison is difficult in practice for three reasons. First, the analyst must locate the baseline: a host that is running the relevant kernel and that is guaranteed to be uninfected. A clever way to solve this problem is to leverage the fact that clouds commonly run many instances of a single virtual-machine (VM) image [6]. Given a large number of VMs running the same image and the assumption that infections are rare, an analyst can assume that the overwhelming majority of the VMs will be clean [3]. Second, kernel memory snapshots are large. While it is possible to transfer "raw" memory snapshots to a single point for analysis [3], the network cost of this collection can be high when many VMs are to be examined. Third and most significantly, the kernel-memory snapshots of two VMs that run "the same kernel" can differ widely, for a large number of reasons that do *not* indicate the presence of a rootkit infection. Across two snapshots, the routine differences due to divergent virtual-to-physical memory mappings, address-space layout randomization (ASLR), paravirtualization-related patching, and other factors can make it very challenging to identify the differences that are indicators of kernel malware.

To distinguish between benign differences and potentially important ones, previous work relies on knowledge of the kernels being inspected. For example, the Blacksheep system [3] uses knowledge of the Windows XP and 7 kernels—e.g., the identities and layouts of important data, the locations of kernel entry points, and the structure of Portable Executable (PE) files—so that it can give special attention to differences in Windows' key components. Bridging the "semantic gap" in this way is effective but has three practical weaknesses. First, the analyzer becomes specialized: it works only on the kernels for which it has special (and accurate) implementation knowledge. Second, the analyzer becomes more complex: it must include code to walk individual data structures, extract specific kernel features, assign weights to the extracted features, and so on. Third, by choosing to give special attention to certain parts of the kernel, the analyzer inherently makes assumptions about how malware will integrate with the kernel. These assumptions may or may not be accurate, and as malware evolves, the analyzer's assumptions may become less true over time.

In this paper, we present a new and alternative approach to detecting kernel-resident malware within a large group of similarly configured VMs (a "herd"). Our approach uses no malware-specific knowledge, e.g., no signatures or assumptions about how malware attaches to the kernel, except for

the assumption that the malware has code that resides in the kernel. Our approach uses *limited kernel-specific knowledge* to obtain, for each VM in the herd, a meaningful but concise *fingerprint* of the code in that VM's kernel. The knowledge used in this process is low-level: e.g., the location of the page global directory, allowing the rest of the kernel's memory to be located, and knowledge of the processor's instruction set, allowing the kernel's code to be disassembled. Each fingerprint is a collection of hashes that summarize the (normalized) contents of a kernel's code pages: we use fuzzy hashing [20] so that similar page contents map to similar hash values. The fingerprints are generated on the physical machines that host the VMs, and then they are sent to a central analysis server. The server uses *no kernel-specific knowledge* to carry out its task. It compares the fingerprints by first performing feature alignment (identifying the elements that "line up" over all the fingerprints); then putting the fingerprints into a space over which distances can be computed; and finally, computing clusters over the fingerprints. The fingerprints of most VMs form a single cluster, representing the healthy members of the herd. Fingerprints that fall outside the main cluster correspond to VMs with anomalous, possibly malware-infected, kernels.

We have implemented our approach in a tool called *Fluorescence* and evaluated its ability to detect VMs that are infected with real-world kernel rootkits. Fluorescence can examine both Linux (3.13–4.15, x64) and Windows 7 (x64) kernels. Because the kernel-specific knowledge needed for memory acquisition and normalization is minimal and low-level, it tends to be stable across many versions of a single kernel: in particular, we report that Fluorescence's single Linux-specific agent works correctly across the range of Linux kernels we have tested, 3.13.0–4.15.0. We also report on our experiments using Fluorescence to detect the presence of kernel rootkits within herds. Fluorescence was able to find all of the infected hosts in the Linux- and Windows-based herds that we created; in addition, when multiple types of malware were present, Fluorescence was able to correctly cluster the infected hosts by type. Finally, we report that the time taken by Fluorescence is reasonable, even for herds containing a few hundred VMs. In our experience, Fluorescence can analyze a 50-host herd in less than ten minutes, and 200 hosts in ∼60–80 minutes.

Our contributions are threefold. First, we present a new method for detecting kernel-resident malware within a group of VMs that run the same kernel. Unlike previous methods that require detailed knowledge of the kernel under examination, our method uses limited kernel-specific knowledge to construct fingerprints and no kernel-specific knowledge to analyze the fingerprints. Second, we describe the implementation of our method in Fluorescence. Our implementation shows that our approach is general: Fluorescence works with both Windows and Linux kernels, and a single Linux agent suffices for a wide range of Linux kernel versions. Third, we evaluate Fluorescence in terms of its detection abilities and speed. In our experiments, Fluorescence was able to detect all the real-world kernel rootkits in our VM herds. The time required by Fluorescence is reasonable for regular (e.g., daily) scanning of large herds, containing up to a few hundred VMs.

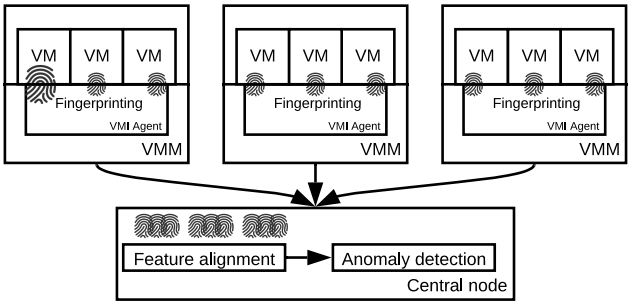

Figure 1: Fluorescence architecture.

2 Design

Figure 1 illustrates the overall design of Fluorescence, our system for detecting kernel-resident malware within a herd of similarly configured VMs. Fluorescence implements a three-step algorithm. First, it collects the current *fingerprint* of every VM in the herd (§2.1). A fingerprint summarizes the code pages within the kernel of a VM's guest; fingerprints are computed by agents that run on the physical machines that are being monitored by Fluorescence, and these agents send the fingerprints to a central node for analysis. Second, Fluorescence's central node performs *feature alignment* (§2.2). Each fingerprint is an unordered multiset that represents the content of one VM's kernel, and the feature-alignment step finds the elements that best correspond to each other across the multisets. The output of this step is a matrix. Each row encodes the fingerprint of one VM, and each column represents a *feature*; the elements of each fingerprint (row) are permuted so that the best-corresponding elements across all fingerprints are aligned (columns). Third, the central node performs *anomaly detection* over the data in the matrix (§2.3). Fluorescence does this in two steps: the first transforms the data in the matrix so that one can compute "distances" between the fingerprints, and the second uses machine learning—deep learning (§2.3.1) and clustering (§2.3.2)—to find anomalies. The fingerprints of most VMs form a single cluster. Given the assumption that malware infections are rare [3], that cluster represents VMs that are healthy. Outliers correspond to VMs with anomalous kernel-memory code, e.g., malware infections. The clustering pattern among the outliers can help an analyst determine if the outlier VMs are infected by a single kind of malware (one cluster) or different kinds (multiple clusters).

Fluorescence is designed to be general in two ways. First, it relies on no malware-specific knowledge: no signatures or assumptions about how malware works, except for the assumption that it must have code in order to stay resident. Second, it requires only very limited information about the kernels that are being monitored. As Table 1 shows, the

Analysis Step	Kernel/Arch.-specific Knowledge
Fingerprinting	
kernel page pinning (opt.)	debug info used by pinning tool
VM pause/resume	*none*
kernel page acquisition	page global dir. (KPGD) location x86_64 page table Intel Extended Page Tables
normalization	general ELF/PE loading layout OS-dependent addr. space layout x86 instruction set
hashing	*none*
Feature alignment	*none*
Anomaly detection	*none*

Table 1: Kernel- and architecture-specific knowledge needed by Fluorescence.

fingerprinting agents need some basic, low-level information in order to locate a kernel's pages (§2.1.1) and normalize their contents (§2.1.2). Fluorescence's central server, which does feature alignment and anomaly detection, needs no kernel- or architecture-specific knowledge at all.

2.1 Fingerprinting

Fluorescence's agents, which are co-located with the VMs being monitored, produce a fingerprint for every VM in the herd. Fluorescence's feature-alignment and clustering steps operate on these fingerprints, rather than kernel memory snapshots, which greatly reduces the amount of data that is transferred to the central server. The key goal of fingerprinting, therefore, is to preserve the most important characteristics of a VM's guest kernel code memory while also being concise.

Creating a fingerprint involves four steps. First, the agent pauses the target VM. Second, the agent uses virtual machine introspection (VMI) to locate the VM guest's kernel code memory pages. It copies the pages and their metadata into its own memory—a quick operation—and then resumes the target VM.[1] Third, the agent *normalizes* the contents of the copied pages to reduce expected sources of "noise," e.g., the effects of address-space layout randomization (ASLR). The agent has multiple ways to normalize the raw data, resulting in multiple *feature views* of each page. Fourth, the agent uses fuzzy hashing [20] to compute a hash for each feature view. A fuzzy hash function produces similar hashes for similar inputs, and is therefore a summarizer: the "distance" between the hashes of pages *A* and *B* can be used to estimate the similarity of the full contents of pages *A* and *B*.

The complete fingerprint of a VM guest's kernel is a multiset, and each element of the multiset is a tuple that describes one 4 KB-page of the kernel's code memory. The first element of the tuple is the hash for the first feature view of the page, the second is the hash of the second feature view, and so on. When it is complete, the agent sends the fingerprint to Fluorescence's central server for analysis.

2.1.1 Finding Kernel Code Pages

There are three main challenges that Fluorescence addresses in obtaining the kernel code pages of a monitored VM. The first is to ensure that all the code pages are in memory. Some kernels, including the Windows 7 kernel, can swap their own code pages to disk; on-disk pages cannot easily be read through VMI, and unreadable pages would result in incomplete fingerprints. For such "swappable" kernels, our Fluorescence implementation simply invokes a tool inside the VM guest to pin all of the kernel's code pages, prior to Fluorescence starting the fingerprinting process (§3.1). The second challenge lies in accessing the target VM's memory. For this, our implementation uses libVMI [31], a popular and open-source library that implements virtual machine introspection. The third challenge is to find all of the kernel code pages. To do this, the Fluorescence agent starts from the kernel page global directory (KPGD) and makes a breadth-first traversal of the page table to collect and copy all of the kernel's executable pages. The location of the KPGD is kernel-specific, but easily obtainable via libVMI (§3.1).

The x64 architecture supports multiple page sizes—4 KB, 2 MB and 1 GB—and kernels use pages of different sizes to improve memory management. So that fingerprint generation is not influenced by the use of huge pages (which changes over time), Fluorescence uses a uniform 4 KB page size. When the agent finds a huge kernel page, it divides that page into multiple 4 KB pages within its own representation of the kernel's memory. As described next, each 4 KB page becomes the basis of a feature in the kernel's fingerprint.

2.1.2 Normalization

Consider a single page of code that is loaded into the guest kernels of two VMs. One might assume that in the running kernels, the contents of the two pages would be identical, but this is often not the case. In particular, the kernel-loading process may patch the loaded code to replace symbolic references (e.g., to functions in other code) with actual (virtual) addresses. The two kernels may patch *different* addresses into the code for various reasons, including the use of ASLR, thus causing the two copies of the code to be slightly different across the two VMs. This difference is benign—expected, and not indicative of an anomaly. Unless differences like this are accounted for, however, they can make it difficult for Fluorescence to identify differences that *are* anomalous.

To reduce the effects of benign differences, the Fluorescence agent performs *normalization*: it applies a set of functions to reduce the "noise" introduced by factors such as ASLR. A perfect normalization function would effectively undo the benign changes, mapping every copy of "the same code page" across all the monitored VMs onto a single value that is similar to the originally loaded, unpatched code. With

[1] A future version of Fluorescence could use page sharing between the target and agent VMs, in conjunction with copy-on-write, to reduce the pause time for the target VM. We have not implemented this because the pause time is already short, and reducing pause time is not the focus of our research.

enough kernel information (e.g., debug symbols) such perfect normalization is feasible, but our aim is for Fluorescence to work with minimal knowledge of the kernels it monitors. Our implemented normalization functions (§3.2) therefore rely only on basic knowledge about ELF/PE loading and the x86 ISA. As a consequence, they are approximate, but still effective at reducing "noise."

The Fluorescence agent applies normalization to each of the 4 KB pages that it obtains from a monitored VM (§2.1.1). The agent supports multiple normalization functions and applies each one individually, resulting in multiple translations of each page. We refer to each of these translations as a *feature view*. By convention, the identity function is always one of the normalization functions: i.e., the "raw content" is always one of the feature views.

2.1.3 Hashing

Finally, the agent hashes every feature view of every page that the agent obtained from the monitored VM. Our implementation uses `ssdeep` [20], which is a fast fuzzy hash function. For each page, the agent collects the hashes of the page's feature views into a tuple. It then collects the tuples into a multiset, which is the completed fingerprint of the VM.

2.2 Feature Alignment

The fingerprints of all the monitored VMs are sent to Fluorescence's central server for analysis. The first step of the analysis is feature alignment, which aims to find the best correspondences—i.e., the best matches—of all of the pages across all of the VMs. Imagine putting all of the fingerprints into a 2D matrix, where each row contains the tuples from a single fingerprint. The goal of feature alignment is to permute each row so that the tuples in each matrix column represent versions of "the same page" across all of the VMs.

Feature alignment has three phases. The first simplifies the fingerprints by removing tuples that represent content present in all of the VMs. The second computes a *basis*, which is a vector that contains the most representative elements (tuples) across all of the fingerprints. The third phase translates all of the fingerprints into vectors, ordering the elements by matching them against the basis.

2.2.1 Remove Tuples for Ubiquitous Content

Recall that our ultimate goal is to find anomalous VMs within the herd. Pages that are identical across all of the VMs are not useful for anomaly detection, so their tuples can simply be removed from the fingerprints.

The algorithm for removing "ubiquitous tuples" considers each normalization separately, starting from the identity function. Let i be the number of the current normalization, and let F be the collection of n fingerprints. Let F_i be the "i-th projection of F": the fingerprints F, but replacing every tuple with just its i-th element. Now compute H as the (multiset) intersection of the members of F_i. The hashes in H represent tuples that represent the same content (under normalization i) across all of the fingerprints. Now we can remove those tuples from the fingerprints. For each f in F, for each h in H, remove the tuple whose i-th element is equal to h.[2]

The above process is repeated for each normalization function. By starting with the identity normalization, the algorithm considers exact page-content matches first: their tuples are matched and removed, allowing subsequent matching to be more accurate. In practice, the above algorithm removes a large fraction of the tuples from all of the fingerprints, greatly speeding up all of the subsequent analysis steps.

2.2.2 Compute a Basis

The next step is to compute a basis: a vector that Fluorescence can use to impose an order on the elements of all of the (simplified) fingerprints. The order is arbitrary; its only purpose is to allow similar pages across the VMs to be associated with each other, so that the fingerprints can sensibly be compared, and outliers identified.

Fluorescence creates the basis by selecting the most *representative* elements from the actual fingerprints. We say that an element of a fingerprint is representative if it is *similar* to an element in every other fingerprint, where the similarity of elements (i.e., tuples) is defined in terms of the hashes they contain. Given two hashes, `ssdeep` can compute a similarity score between 0 and 100: a high score means that the hashes represent highly similar strings (i.e., normalized page content) and a low score means that the hashes represent very dissimilar strings. We define the similarity σ of two fingerprint elements as the maximum similarity scores of their hashes for every feature view:

$$\begin{aligned}\sigma(e_1,e_2) &= \max(sim_1, sim_2, \dots)\\ \text{where } e_1 &= \langle \alpha_1, \alpha_2, \dots\rangle, e_2 = \langle \beta_1, \beta_2, \dots\rangle,\\ sim_i &= \text{ssdeep}(\alpha_i, \beta_i)\end{aligned}$$

Fluorescence collects the most representative elements into a basis, called Λ, by using the following algorithm. Initialize Λ to be a zero-element vector. For every element e in every fingerprint, find the element in every other fingerprint that is most similar to e. Call that set $neighbors_e$, and let sim_e be the sum of $\sigma(e,n)$ for all $n \in neighbors_e$. (If no element in a given fingerprint has a positive similarity score when compared to e, then that fingerprint contributes nothing to $neighbors_e$.) Now, from all the elements in all of the fingerprints, choose the element e that has the greatest sim_e: this is the most representative element. Extend Λ by adding e, and remove every element of $neighbors_e$ from further consideration. Now repeat: from the remaining eligible elements, choose the element e with the greatest sim_e; extend Λ, and remove e's neighbors from further consideration. Repeat until there are no more elements to consider.

[2] By construction, such a tuple is guaranteed to exist. If there is more than one tuple whose i-th element is h, arbitrarily choose one to remove.

In general, Λ may contain more elements than any of the actual fingerprints. (It must be at least as long as the longest fingerprint.) This is because Λ includes elements that represent all of the "rare" and unique elements that are present in any fingerprint.

2.2.3 Compute the Fingerprint Vectors

After computing the basis Λ, it is straightforward to transform the fingerprints into vectors that can be sensibly compared. Fluorescence does this by computing a 2D matrix, called T (for "tuples"), in which every row represents a fingerprint and every column corresponds to an element of Λ. Fluorescence fills each row from left to right, i.e., from the most representative to the least representative elements of the basis. Consider the translation of a fingerprint f. To fill the leftmost cell of f's row in the matrix, Fluorescence finds the element in f that has the greatest similarity with the leftmost value of the basis. (I.e., choose the element e from f that maximizes $\sigma(\Lambda_0, e)$.) Store that element in the leftmost cell of f's row, and remove the element from f. Repeat the selection process for the remaining cells, moving from left to right across the row. When filling the cell at index j, if no remaining element of f has a positive similarity score with Λ_j, leave cell j empty.

2.3 Anomaly Detection

At its central server, Fluorescence performs two analyses to detect anomalous VMs. The first (§2.3.1), based on deep learning, detects anomalies by measuring a neural network's ability to reconstruct (encoded) fingerprints. The network is able to reconstruct "typical" fingerprints more accurately than it can reconstruct anomalous ones. The second (§2.3.2) applies a clustering algorithm to distinguish between typical fingerprints (a large cluster, representing healthy VMs) and atypical ones (small clusters, representing anomalous VMs). The two algorithms have different strengths and weaknesses (§2.3.3) but generally agree in practice, so each serves to validate the other's results.

The versions of these algorithms that we use in Fluorescence operate on data points that are represented as vectors of numbers. The matrix T that Fluorescence computed in §2.2.3, however, has cells filled with tuples. To prepare for anomaly detection, therefore, Fluorescence computes a new matrix, called S (for "similarities"), in which the tuples of T are replaced by similarity scores as follows:

$$S_{ij} = \begin{cases} \sigma(\Lambda_j, T_{ij}) & \text{if } T_{ij} \text{ is not empty} \\ -1 & \text{otherwise} \end{cases}$$

That is, each tuple is represented by its similarity to the corresponding element of the basis, and empty cells are represented by -1. This transformation is not distance-preserving in principle,[3] but it preserves the essential qualities of the fingerprints in practice. If two fingerprints are "close to" the basis in a particular dimension (column) in T, they remain close to each other in that dimension in S. Similarly, fingerprints with unique and/or rare contents (i.e., that have tuples in columns of T where most fingerprints have none) continue to be distinguished in S.

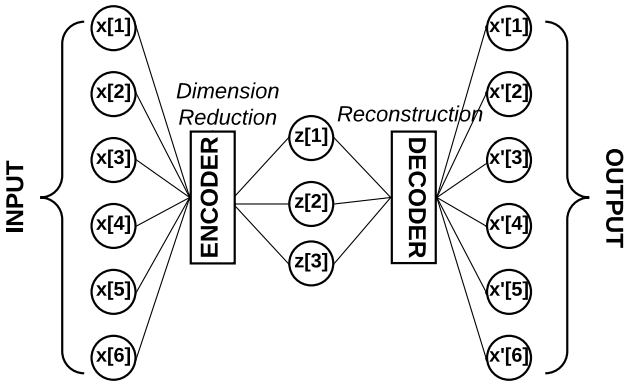

Figure 2: Autoencoder architecture.

2.3.1 Deep Learning Approach

Fluorescence's deep-learning approach to detecting anomalous VMs is based on an *autoencoder*. The purpose of an autoencoder is to learn, in an unsupervised manner, an efficient method for representing a dataset. As illustrated in Figure 2, an autoencoder contains an encoder, which reduces the number of dimensions in an input, and a decoder, which attempts to reconstruct the original input from its reduced representation. To minimize the error between the input and output, an autoencoder must learn to preserve the maximum information while encoding. The major patterns within the input dataset are learned and preserved, and as a consequence, the minor patterns—found in "outliers"—are lost.

Fluorescence leverages these properties to identify the rows in S that represent anomalous VMs. It trains an autoencoder over the rows of S. The encoder reduces the dimensionality of each row vector to $1/30$th of its original (with a minimum of two encoded dimensions), and the decoder attempts to reconstruct the original vector. The learning goal is to minimize the sum of mean square errors between all the corresponding inputs and outputs, i.e., to minimize $\sum_{i=1}^{m}\sum_{j=1}^{n}(S_{ij} - S'_{ij})^2$, where m is the number of VMs (rows of S), n is the number of features per VM (columns of S), and S'_{ij} is the "reconstructed value" that corresponds to S_{ij}.

After training the autoencoder, Fluorescence calculates the *error score* of each VM as the squared error between its original vector representation in S and the reconstructed vector in S'. The VMs that have significantly larger error scores are identified as anomalies. To find such scores, Fluorescence models the error score as a Gaussian distribution. The VMs

[3] If two fingerprints α and β have the same similarity to the basis in dimension j, then $\sigma(\Lambda_j, T_{\alpha j}) = \sigma(\Lambda_j, T_{\beta j})$. Thus $S_{\alpha j} = S_{\beta j}$ and in S, the distance between the fingerprints in dimension j is zero. In general, however, $T_{\alpha j}$ and $T_{\beta j}$ may not be identical.

that have error scores within an acceptable confidence interval of this distribution are considered to be normal, and all others are flagged as anomalous.

The encoder and decoder networks are 1-layer fully connected neural networks having dimensions $D \times 2$ and $2 \times D$, respectively, where D is the input encoding dimension. For optimization, we use ADADELTA [45] as the optimizer and cross-entropy loss as the loss function. The autoencoder model is trained on the entire input data for 50 epochs, with a mini-batch size of 2.

2.3.2 Clustering Approach

Fluorescence's clustering approach to identifying anomalous VMs uses DBSCAN [11], a density-based algorithm. (Unlike some other methods, such as k-means, density-based clustering does not require prior knowledge of the number of clusters to be formed.) The Euclidean distance d between two vector rows of S, representing two VMs, is computed in the usual way: $d(\alpha,\beta) = \sqrt{(S_{\alpha 1} - S_{\beta 1})^2 + (S_{\alpha 2} - S_{\beta 2})^2 + \cdots}$

DBSCAN requires two parameters: ε, which is a distance threshold, and m, which is the minimum number of points needed to form a cluster. DBSCAN builds a cluster by choosing an unvisited data point, "expanding" toward all of the neighboring points that are within ε of that point, and then recursively expanding from each of those neighbors. A point is marked as an outlier if it is not a member of a cluster that contains at least m points.

Fluorescence's goal is for all normal VMs to be contained in one or more large (many-member) clusters and for anomalous VMs to be either (1) contained in small clusters or (2) classified as outliers. To do this, we need to choose appropriate values for ε and m.

To illustrate how we choose ε, we performed experiments in which we deployed herds of similar VMs with some instances infected with malware, measured the distance between every pair of VMs, and plotted the distances as a CDF. Figure 3 summarizes six of these experiments: in each, we deploy 95 normal VMs and five that are compromised with one type of malware. Note the "plateau" in the CDF of each experiment. The slope to the left of the plateau corresponds to VM-pairs that are near each other and that we would like to cluster together. The slope to the right corresponds to VM-pairs where the VMs are far apart, and where we would like the VMs not to be clustered together. Thus, the plateau represents the difference between intra-cluster distances and inter-cluster distances for a given dataset; a good choice of ε is one that lies near the left edge of the plateau. Based on our tuning experiments, we set ε to 100. Tuned in this way, Fluorescence can collect normal VMs into a small number of clusters—ideally, a single cluster—each with many members.

We set m to three so that DBSCAN will cluster even small numbers of VMs. This allows Fluorescence to identify clusters of anomalous VMs, e.g., VMs infected by the same malware (§4.2.2). After clustering, Fluorescence reports all outliers and all members of small clusters as anomalous.

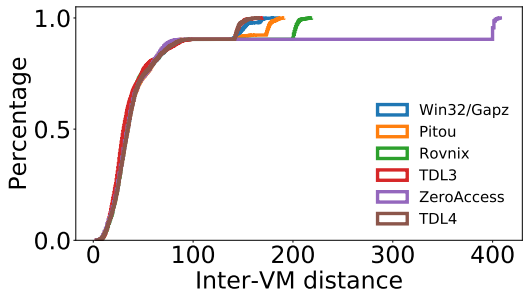

Figure 3: CDFs of all inter-VM distances in six tuning experiments. Each experiment involves 100 VMs, where five are infected with a single kind of malware.

2.3.3 Discussion

In cases where a small number of VMs are infected with malware, Fluorescence's deep-learning and clustering techniques are likely to identify the same VMs as anomalous, and thus serve to validate each other. In other circumstances, however, the two approaches have different strengths and weaknesses.

One advantage of the autoencoder-based approach is that it does not require one to tune the algorithm by choosing clustering parameters: the autoencoder simply learns the dominant patterns in the dataset and can identify the data points that are most atypical. However, this comes at a cost in three ways. First, in the common case where no VMs are infected, the autoencoder may be overly sensitive. If the error scores of all VMs are small, then the threshold for flagging anomalies is also small. In these cases, the clusters and intra-VM distances computed by the clustering method can be used as a sanity check. Second, in the rare case that many VMs are infected, the autoencoder may start to become insensitive: it may learn the patterns of the infected VMs, even if those VMs are in the minority (say, 10%). In comparison, the clustering method is less sensitive, and can be accurate even when the number of infected VMs is high, as long as the uninfected VMs constitute the largest cluster. Third, while the autoencoder method can detect anomalies, it cannot report which anomalies are similar to each other. DBSCAN, of course, can compute clusters among the anomalous VMs.

The main shortcoming of the clustering approach is that the threshold for building clusters, ε, may be too "generous." If the data points for normal VMs are actually much closer to each other than ε, then a poor choice of ε means that anomalous VMs that are nevertheless "within ε" will be classified as normal by DBSCAN. In contrast, in this situation, the autoencoder would essentially learn the threshold automatically. Similarly, consider a situation in which there are two types of VMs (e.g., two different kernels) that are similar, and the data points of these two types form "overlapping" regions when

measured in S. DBSCAN may group instances of these VMs into a single cluster and thereby obscure small anomalies in each type. Conversely, the autoencoder could potentially differentiate between the types and identify anomalous VMs at a finer grain.

3 Implementation

Fluorescence depends on its fingerprinting agents (§2.1) to (1) "snapshot" the kernel code pages within all of the monitored VMs and (2) perform normalization, so that benign differences do not mask actual anomalies. This section provides implementation detail about how the agent performs these tasks. The agent requires some kernel- and architecture-specific knowledge to carry out these tasks; our implementation works with VM guests that run Linux (3.13–4.15) or Windows 7 kernels for x64 within VMs managed by Xen 4.9. The agent is implemented in C and uses libVMI [31] to manage and access the monitored VMs.

3.1 Walking the Page Table

To obtain all of the executable pages from the kernel of a VM guest, the Fluorescence agent performs a breadth-first traversal of a page table, starting from the KPGD. Since hypervisors typically maintain a KPGD for each VM, the KPGD can be easily accessed through VMI libraries. (The kernel-specific knowledge about the location of the KPGD is encapsulated within libVMI.) Using knowledge of x86_64 page tables, it is straightforward for Fluorescence to find all of the kernel code pages.

In practice, ensuring that the kernel code pages are in memory is a concern. As previously described (§2.1.1), some kernels can swap their own code pages to external storage, where they are not easily accessible through VMI. In particular, we found that Windows 7 swaps its own code pages aggressively: if a page is not used for long time, Windows may move the page to external storage or mark it as "in transition" (another kind of invalid page table entry). If the Fluorescence agent encounters such a page, the resulting fingerprint will be incomplete, possibly leading to spurious anomaly reports.

To avoid problems caused by swapped-out pages, Fluorescence invokes a tool on VMs running Windows, prior to taking a fingerprint, that pins all of the guest kernel's code pages into memory. Fluorescence uses DRAKVUF [23] for this purpose. DRAKVUF is a system, built atop libVMI, that enables one to hijack a process to run an injected executable. Fluorescence injects code into the `winlogon` process, which is a privileged process present on every Windows machine, that loads and runs a kernel module that pins the kernel's executable pages. The Fluorescence agent does this before each fingerprint of a Windows guest, in order to catch any pages that might have been injected since the VM was last fingerprinted. We did not develop a similar page-pinning tool for Linux guests, because none of the Linux kernels we used ever swapped any of their code pages out.

3.2 Normalizing Code-Page Content

When a kernel is loaded into memory, the loader patches code pages to replace symbolic references (to code or data) with the actual addresses of the things being referenced. Due to ASLR and other factors, when the same kernel is loaded into two different VMs, the loaded copies of the kernel will be patched with different addresses (§2.1.2).

Normalization aims to reduce these differences by replacing patched addresses with constants, in a way that is *consistent* across all of the VMs. One can think of this as undoing the patching, replacing all resolved references with symbolic ones. Consider an original (unloaded) code page P. In every loaded copy of P, every patched reference to a function f should be replaced with a constant that represents f—the same constant in all copies of P. To preserve as much information as possible in the normalized pages, a "mostly unique" constant should be used for each referent: e.g., if a page refers to two functions f and g, the constants chosen to represent references to f and g should be different.

To perform this replacement, the Fluorescence agent must solve two problems. It must determine the constants it will use to replace patched references, and it must find the references.

Determine the constants. To find constants in a consistent way across all of the protected VMs, Fluorescence leverages the fact that a kernel image consists of many loaded "objects" (i.e., object files). A single object generally defines many functions and variables, and these things appear at different offsets within the loaded object. ASLR may randomly arrange the objects in the kernel's memory, but it does not rearrange the contents *inside* the objects. The offset of a function or variable within its containing object is thus constant across all of the kernels that have loaded that object. Fluorescence uses these constants to normalize loaded pages: given a resolved reference to a function or variable f, it replaces that reference with the byte offset of f within the object that contains f.

More specifically, Fluorescence divides the address space of a kernel into a number of 4 KB-page-aligned *regions* of virtually contiguous memory. (Certain address ranges are excluded, based on knowledge of how kernels use their address spaces: e.g., the "direct mapping" region in Linux, and the system cache region in Windows, are excluded.) Fluorescence makes regions that correspond to the kernel's loaded objects by searching for objects in the appropriate area of the kernel's address space. For Windows, the search is easy: the Portable Executable (PE) header value "MZ" appears at the start of each loaded object. For Linux, the search is also straightforward. Although ELF headers are not present in memory, loaded ELF objects follow a common pattern: a series of executable (code) pages, followed by some read-only (data) pages, and then by some writable (data) pages. Using these patterns, it is simple to create regions that correspond to the kernel's loaded objects.

After making regions for objects, Fluorescence creates re-

gions that cover the remainder of the kernel's address space (with some exceptions as previously noted). Whenever a normalization function wants to replace a resolved reference with a constant, Fluorescence determines the region that the reference points into. It replaces the reference with the difference between the referenced address and the region's start.

Find the references. Our implemented Fluorescence agent can provide up to three normalized versions, a.k.a. feature views (§2.1.2), of every kernel code page. The first is the page content just as it appears in the VM, or the *original* view. If a page is not modified at all by the kernel loader, than all of the copies of that page will be identical in this view. The second feature view, called *sub-base*, and the third, called *disassembly*, both modify the original page content by heuristically searching for resolved references and then replacing those references with constants as described above. The sub-base and disassembly views differ in how they attempt to find resolved references within the code.

The sub-base view is simple, treating the input page as an array of eight-byte elements. The sub-base normalization function examines each element: if the value of an element can be interpreted as an address that falls within a defined region, that element is replaced by a constant (i.e., the difference between the element value and the referenced region's start). This heuristic is very fast, correctly transforms constructs like jump tables that are eight-byte aligned, and "works" because the 64-bit address space is sparsely filled. In general, however, the sub-base method may overlook many resolved references (e.g., those that are not eight-byte aligned) and may modify values that are not actually addresses.

The disassembly view attempts to address the shortcomings of the sub-base view. Given a code page, the disassembly normalization function uses Capstone [33] to interpret the code found on the page. It searches the disassembled code for operands that appear to be (64-bit) absolute addresses and (32-bit) PC-relative addresses. Whenever one of these appears to point into a defined region, the normalization function replaces the address with the appropriate constant as described above. The disassembly view is often more precise than the sub-base view, but it is still heuristic; in our experience, it is still subject to false positives (incorrect modifications) and false negatives (overlooked references). This is why Fluorescence relies on multiple feature views of each code page: when one view "fails," another may succeed.

4 Evaluation

We evaluate our Fluorescence implementation by testing its ability to identify VMs that are infected with kernel-resident malware, picking those VMs out of a mostly healthy herd of similarly configured VMs running Linux or Windows 7. We focus on answering four questions. *First, can Fluorescence correctly identify the VMs in the herd that are infected with malware (§4.2)?* In our experiments, the answer is yes: Fluorescence accurately identified the infected VMs. We discuss the performance of the autoencoder (§4.2.1) and clustering (§4.2.2) detection methods. *Second, is normalization necessary and effective (§4.3)?* We find that the answer is yes: our implemented normalization process can drastically reduce the number of features that need to be considered during anomaly detection, and it allows different classes of VMs to be clustered for identification. *Third, what is the run-time performance of fingerprint generation (§4.4)?* In our experiments, fingerprint generation takes 11–13 s, and the monitored VM is paused for only 0.6 s. *Fourth, how does the run time of Fluorescence scale as the number of monitored VMs increases (§4.5)?* We find that the run time of Fluorescence is acceptable even for moderately sized herds: Fluorescence can process a herd of 200 VMs in about an hour. Larger herds can be handled by treating them as multiple subherds, each analyzed by a separate instance of Fluorescence.

4.1 Experiment Setup

We deploy herds containing various numbers of VMs running Linux or Windows. We use Xen 4.9 as the hypervisor and run Ubuntu 16.04 within dom0. Each Linux VM runs (64-bit) Ubuntu 16.04 as the guest, with Linux kernel version 4.4.0. Each Windows VM runs (64-bit) Windows 7. Each VM is configured with one virtual CPU and 1 GB RAM. The VMs are distributed over twenty "d430" physical machines within the Utah Emulab network testbed [43].[4] Each d430 has two 2.4 GHz Intel Xeon E5-2630v3 8-core CPUs and 64 GB RAM. Each VM-hosting machine runs an instance of the Fluorescence agent in its dom0. Fluorescence's central server, which performs feature alignment and anomaly detection, is located on a separate d430 (hosting no VMs) that runs Ubuntu 16.04.

We collected samples of kernel-resident malware to use in our experiments. We focus on malware that persists in the kernel by injecting or modifying kernel code, because that is the type of malware that Fluorescence is designed to detect. For Windows, we collected samples of Pitou, Rovnix (a.k.a. Cidox), ZeroAccess (a.k.a Sirefef), Win32/Gapz, and two variants of TDSS known as TDL3 and TDL4. Some of these have recently been active in the wild [41]. For Linux, we collected three rootkit samples: Diamorphine [26], Nurupo [29], and Reptile [1].

For experiments with Linux VMs, we configured Fluorescence to compute all three feature views of every page (§3.2). For experiments with Windows VMs, we configured Fluorescence to compute only the *original* and *sub-base* views. We did this because the Windows 7 kernel uses Position Independent Executables (PIE); as a result, most code pages are unpatched by the kernel-loading process and are already identical across VMs. We leave *sub-base* enabled in our Windows 7 experiments to handle the small number of code pages that do vary across loaded Windows 7 kernels.

[4] We used Emulab's "experiment firewall" feature to block traffic between the VMs and the Internet.

Fluorescence transfers VMs' fingerprints to its central server for analysis. For the VMs described above, we found that that size of a Windows VM fingerprint was approximately 2.7 MB. A Linux VM fingerprint was approximately 1.8 MB. These numbers compare favorably to prior work [3] that collects and analyzes physical memory dumps.

4.2 Malware Detection

We performed a set of experiments to test Fluorescence's ability to identify the VMs in a herd that are infected with kernel-resident malware. For each test, we set up the herd as follows. We created 50 VMs, running either Windows or Linux. To perturb the collection, we randomly selected ten VMs, logged into them, and performed some activities manually. On Windows, we opened some files, played some small games, and/or used a web browser to download some files. On Linux, we ran Apache or nginx and then downloaded random files from those servers. We then randomly installed malware on some of the VMs. For Windows, we chose a subset of our six samples to inject (1–6 samples); for each sample, randomly chose the number of instances to inject (1–4); randomly chose the set of VMs to be infected (one VM per instance); and then injected the samples. For Linux, we followed the same process, injecting 1–3 rootkit types, with 1–4 instances each.

After setting up the herd, we ran Fluorescence to test its ability to discover the infected VMs. For each herd configuration that we selected, we repeated the herd setup and Fluorescence test five times to check for repeatability.

In every test we performed, Fluorescence identified the infected VMs. The DBSCAN method always found the infected VMs, without false positives or negatives, and properly clustered the VMs that were infected with a common type of malware. The autoencoder method was also accurate but produced false positives at negatives when a large fraction of the VMs were infected (as discussed below, §4.2.1).

Figure 4 explains this result. It visualizes S, the similarity matrix (§2.3), for one of our tests involving all six of our Windows malware samples. Each row represents a VM, and each column represents a feature, i.e., a "matching page" across the VMs. We sort the rows and columns for clarity in the visualization. Each cell is colored according to its value: light cells contain high values (i.e., are similar to the corresponding basis value) and dark cells contain low values (are dissimilar to the corresponding basis value). In the visualization, it is clear that the infected VMs present distinct patterns in S, and each type of malware has a different signature. These are the patterns that Fluorescence detects. The figure is annotated to show the different malware families.

Figure 8 presents a similar visualization for one of our Linux tests, one involving all three of our Linux rootkits.

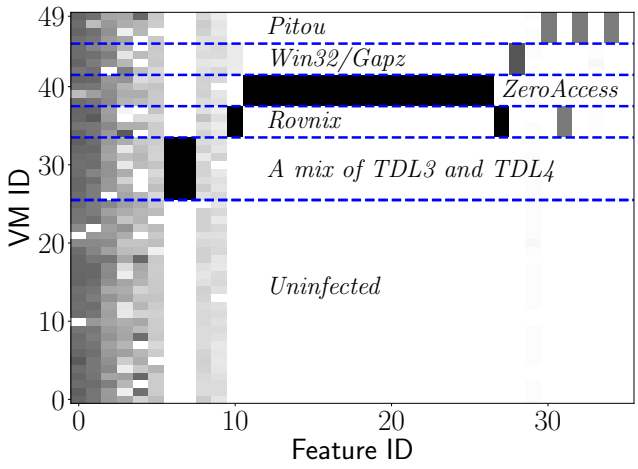

Figure 4: Similarity matrix for an experiment involving 50 VMs running Windows, many infected with malware. In this visualization, *higher* similarity scores have lighter colors.

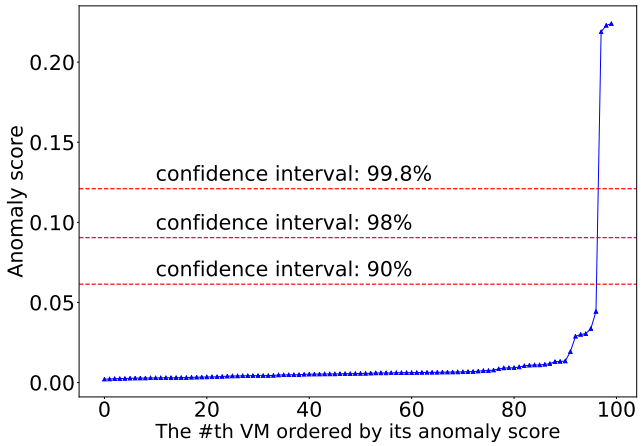

Figure 5: Error scores from the autoencoder-based analysis of an example herd of 100 VMs, three of which are infected with malware. The scores of the infected VMs (upper right) are flagged because they exceed a threshold, as determined by a confidence interval.

4.2.1 Anomaly Detection with Autoencoder

We conducted another set of experiments to better characterize the performance of Fluorescence's autoencoder-based method for detecting anomalies. Recall that the autoencoder method calculates an error score for each monitored VM (§2.3.1). It models the error score as a Gaussian distribution; if the error score of a VM lies outside a chosen confidence interval, Fluorescence flags the VM as anomalous. Figure 5 illustrates this idea. To generate this example, we created and processed a fingerprint set for 100 VMs, three of which were infected with malware. We sorted the resulting scores and plotted the results. The three points at the right side of the figure, with scores above 0.20, correspond to the infected VMs. For confidence interval choices between 90%

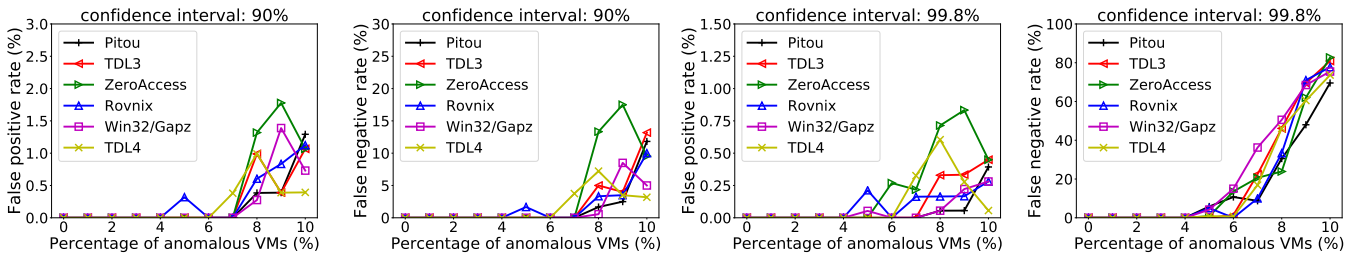

Figure 6: Autoencoder performance, with 90% and 99.8% confidence intervals, for different types of Windows kernel malware and rates of infection in a herd of 100 VMs. Points represent averages over twenty trials.

and 99.8%, all of these VMs are reported to be anomalous, and there are no false positives or negatives.

To perform a large set of experiments similar to the one described above, we reused the fingerprints of Windows VMs that we collected during the experiments described in §4.2. In each of these new experiments, we simulate a herd of 100 VMs by randomly selecting 100 fingerprints from our set of previously generated fingerprints. Each experiment involves one kind of Windows malware, and we vary the number of infected VMs from zero to ten. (I.e., in each configuration, we choose 0–10 fingerprints of VMs infected with the chosen malware, and 90–100 fingerprints of clean VMs.) We then perform anomaly detection over the assembled herd, using the autoencoder method, and record the number of false positives and false negatives reported at the 90% and 99.8% confidence intervals. To account for randomness introduced by the autoencoder—e.g., randomly initialized parameters—we repeat the analysis of each herd twenty times. For each herd, we compute and report the average false positive and false negative rates over the twenty trials.

Figure 6 presents the results of these experiments. (Note the varying ranges of the y-axes in the figure.) We make three observations about this data. First, if the rate of anomalous VMs was under 4%—a realistic threshold for large herds in practice—the autoencoder detected all anomalies without any false positives or false negatives. Second, when the rate of anomalous VMs increases beyond 4%, the autoencoder becomes less effective. Third, the higher confidence interval leads to more false negatives, but also to possibly fewer false positives. This suggests an incremental strategy to maintaining a herd: fix reported VMs, thus lowering the rate of infection, and then repeat analysis with Fluorescence. From our experiments, we conclude that the autoencoder method works well when few anomalies are present, which is likely to be the case in practice. Despite its limitations, it is widely applicable for detecting unknown anomalies.

4.2.2 Anomaly Detection with DBSCAN

We reuse the simulated herds described in §4.2.1 to test the performance of Fluorescence's clustering approach, utilizing DBSCAN, for detecting anomalies. Recall that each herd is

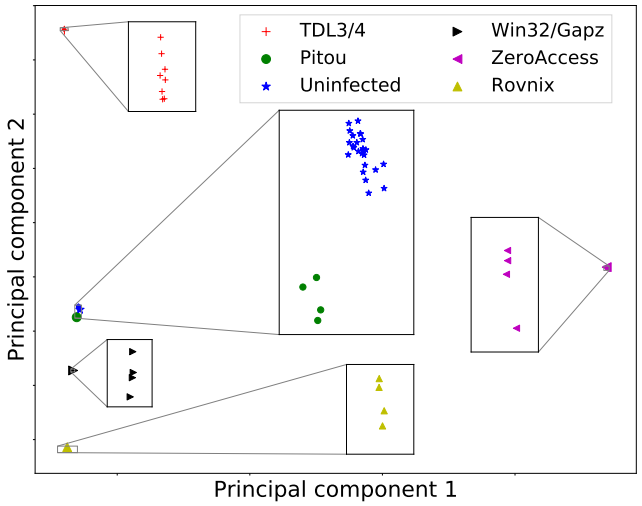

Figure 7: Visualization of clusters found by DBSCAN within a herd of 100 Windows VMs. Using principal component analysis (PCA), the data for each VM was reduced to a 2D coordinate.

represented by the fingerprints of 100 VMs with Windows 7 guests, where 0–10 of those guests have been infected by one type of malware. In each of these tests, DBSCAN correctly partitioned the normal and infected VMs into separate clusters, with no false positive or negatives.

To visualize the reason for DBSCAN's success in our tests, we simulated another herd of 100 Windows VMs. In this herd, we included fingerprints of VMs infected with all of our Windows malware samples: four instances of each malware sample, for a total of 24 infected VMs. We analyzed this herd with Fluorescence—again, all the VMs were properly classified and clustered by family—and obtained the similarity matrix S for the herd. We used principal component analysis (PCA) to reduce the number of features for each VM to just two. Finally, we used the two values for each VM to plot the VMs on the X-Y plane.

Figure 7 shows the result. Each color/shape represents one cluster identified by DBSCAN. The green-dot and blue-star clusters are relatively close to each other, but they are identified as separate clusters, as shown in a zoom-in view. The blue-star cluster has the greatest number of VMs among

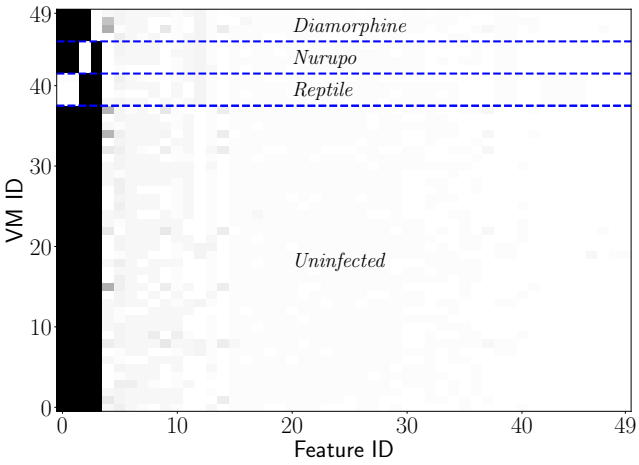

Figure 8: Similarity matrix for an experiment involving 50 VMs running Linux, many infected with malware. In this visualization, *higher* similarity scores have lighter colors.

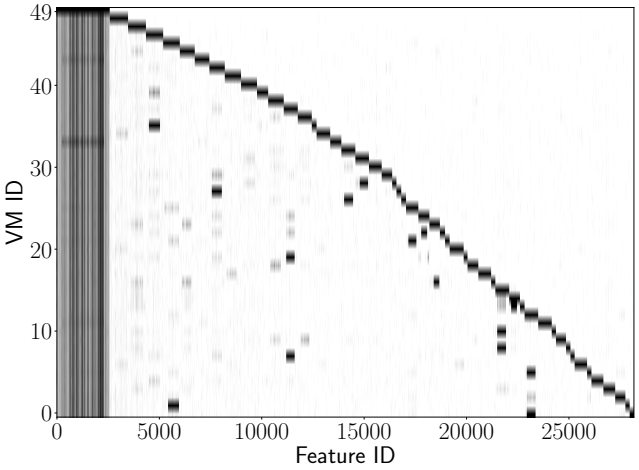

Figure 9: Similarity matrix for the 50 VMs portrayed in Figure 8, but with *sub-base* and *disassembly* normalization disabled. In this visualization, *lower* similarity scores have lighter colors.

all clusters, so the VMs in this cluster are identified as normal while the other, smaller clusters represent different kinds of anomalies (i.e., VMs infected by different families of rootkits). This analysis by DBSCAN matches the ground truth.

4.3 Impact of Normalization

Fluorescence performs normalization to reduce benign differences in the contents of acquired kernel pages. We performed an experiment to assess the effectiveness of our implemented normalization procedure.

Figure 8 visualizes the similarity matrix S for a herd of 50 Linux VMs, many of which are infected with malware. As explained previously for Figure 4, each row represents a VM, each column represents a feature, and cells are shaded according to their values. We have sorted the rows and columns for clarity. The visualization makes the different groups of VMs apparent; each has a pattern that Fluorescence detects.

Step	Windows 7	Linux
Pin kernel code pages	6.4	N/A
Copy kernel code pages	0.6	0.6
Compute kernel memory regions	0.6	0.6
Normalization	0.9	7.7
Hashing	2.7	4.2
Total	11.2	13.1

Table 2: Time (secs) to generate a fingerprint. The VM being fingerprinted is paused only while pages are being copied (step 2).

We removed the *sub-base* and *disassembly* feature views from the fingerprints of this herd—leaving only hashes for the *original* page contents—and analyzed the resulting fingerprints to produce another similarity matrix. Figure 9 visualizes this result. (In Figure 9, low values are light and high values are dark; this is the opposite of the convention used in Figure 8.) Two things are immediately apparent. First, compared to the original matrix, the number of features per VM has increased by more than two orders of magnitude. This happens because Fluorescence is no longer able to remove content that is ubiquitous across the VMs; normalization is necessary for finding such content and is very effective. Second, the patterns that are so clear in Figure 8 are not apparent in Figure 9. This suggests that Fluorescence would have a difficult time finding the malware-infected VMs in this data.

4.4 Fingerprint Generation Time

We measured the time needed for the Fluorescence agent to produce the fingerprint of a VM. We ran the fingerprinting procedure for a Windows VM and a Linux VM and recorded the elapsed time for each step of the process. We repeated the procedure ten times for each VM and computed the average elapsed times over the trials. Table 2 presents our results.

The VM being fingerprinted is paused only while Fluorescence copies its kernel code pages (§2.1). For both Windows 7 and Linux, the pause time is short, less than a second.

For Windows 7, the longest step is the one that uses DRAKVUF to pin the kernel's code pages into memory (§3.1). (The pinning step is unnecessary for our Linux VMs because the Linux kernel does not swap-out its code.) Although pinning for Windows takes several seconds, the VM continues to run during that time. The pinning procedure forces about 2,000 pages to be present in memory, or about 8 MB. Of those, it is common for half to be marked as "in transition," meaning that the page contents are in memory but marked as inaccessible; the kernel can quickly make those pages present again. In this way, the pinning procedure imposes about a 4 MB overhead on the Windows kernel.

For Linux, the most time-consuming step is normalization. For Windows 7, we configured Fluorescence not to compute the *disassembly* feature view (§4.1). Fluorescence spends sev-

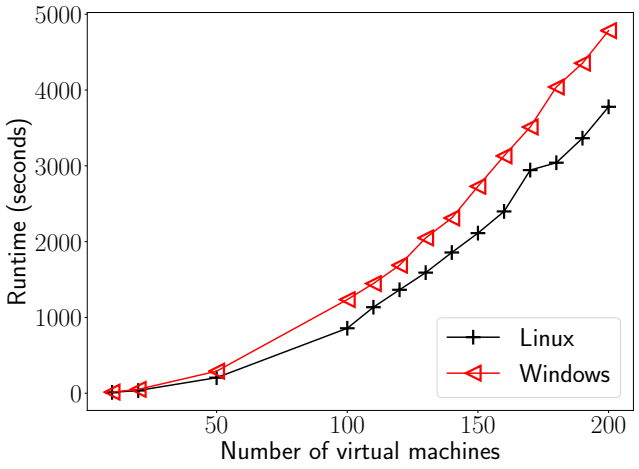

Figure 10: Time required for Fluorescence's central server to analyze VM herds of varying sizes.

eral seconds performing this normalization on Linux kernel pages, and in our experience, it is necessary in order to get good anomaly detection results (§4.3).

4.5 Scalability

Because Fluorescence operates on herds of VMs, it is important to understand the performance of Fluorescence on herds of VM of various sizes. Again, we reused the fingerprints of VMs that we collected in earlier experiments to simulate herds of Windows and Linux VMs of varying sizes, from 10 to 200 VMs. For each herd, we measured the time required for Fluorescence's central node to analyze the collected fingerprints, i.e., to perform both feature alignment and anomaly detection. Anomaly detection is fast—less than a minute in all of our tested configurations—so the majority of the time is spent on feature alignment.

Figure 10 presents the results of these experiments. In brief, Fluorescence required less than ten minutes to analyze each 50-VM herd. It analyzed our herd of 200 Linux VMs in approximately 63 minutes, and it analyzed our herd of 200 Windows VMs in approximately 80 minutes. We believe that this performance is reasonable for periodically measuring the health of a VM herd. Moreover, Fluorescence scales horizontally. For monitoring a herd containing more than a few hundred VMs, one can divide the herd into subherds, each monitored by a separate instance of Fluorescence.

To put these results in context, we note that Bianchi et al. reported [3] that their Blacksheep system, which looks for kernel-level anomalies in herds of Windows machines, needs ten minutes to compare two 1 GB memory dumps. In that time, Fluorescence can search for kernel-level anomalies in at least 50 VMs.

5 Security Analysis

Fluorescence aims to detect kernel-resident malware within a large group of similarly configured VMs, and its design is based on three main assumptions.

The first is that the malware to be detected within the VMs cannot compromise the virtual machine monitors (VMMs) on which those VMs run. In other words, Fluorescence assumes that the VMMs are trustworthy. This assumption allows Fluorescence to fingerprint the monitored VMs efficiently by running its agents on the same physical hosts as the monitored VMs (§2, Figure 1); the agents use virtual machine introspection to access the memory of the monitored VMs and read their kernel code pages (§2.1). If malware is able to compromise the VMMs, then the Fluorescence agents may be disabled or otherwise compromised as well, and the fingerprinting process may not be trustworthy. VMM integrity continues to be an important area of concern [30], but it is not the concern that Fluorescence is intended to address.

The second assumption is that the VMs being monitored are similar to each other in terms of configuration. They boot from a single "golden image," which is a common practice in cloud-based application deployments [6], and therefore they have the same kernel patches applied and the same kernel modules installed, at least at boot time. Like the VMM integrity assumption, the VM similarity assumption helps Fluorescence to be efficient: normalization accounts for anticipated but benign differences (§2.1.2), and normalized pages that are identical across all of the VMs can be removed from fingerprints (§2.2.1), greatly speeding up subsequent analysis. More significantly, the similarity assumption allows Fluorescence to automatically identify anomalous VMs (§2.3), because their fingerprints are most different from the basis that Fluorescence computes (§2.2.2).

The third assumption is also related to automatic anomaly detection. Fluorescence assumes that, in a large group of initially healthy VMs, malware-infected VMs will be the exception, not the rule (§2.3). This assumption, which is also made by prior work [3], allows Fluorescence to distinguish "healthy" VMs (the majority) from "abnormal" ones (the minority) without kernel-specific knowledge.

The second and third assumptions introduce the risk of misclassification. Our experiments showed, for example, that at infection rates above 4%, the autoencoder-based detector produced both false positives and false negatives (§4.2.1). While our experiments with DBSCAN produced no false positives or false negatives (§4.2.2), it is conceivable that an attacker could design malware in a way that would cause Fluorescence to overlook it. For example, an attacker could learn the distribution of pages known to be very different across benign VMs (e.g., ten pages in the Windows kernel) and figure out how to inject code only in those pages. The code injected into each VM would need to be unique to prevent Fluorescence from clustering the infected VMs; just a few similar features are

enough for Fluorescence to differentiate healthy VMs from infected ones (Figure 4, Figure 8). We believe that hiding kernel-resident malware from Fluorescence in this way would require a great deal of sophistication and effort.

If healthy VMs are dissimilar from one another, then Fluorescence would need reconfiguration—or additional information—in order for it to automatically identify anomalous VMs. Consider a herd of healthy VMs in which half have a particular kernel module installed and half do not. This might happen, for example, if the herd is in the middle of an upgrade. Such a split is likely to (1) decrease the effectiveness of the autoencoder-based detector and (2) cause the DBSCAN-based detector to divide the healthy VMs into two clusters. A cloud administrator could deal with such a split in two ways. The first way is to run two instances of Fluorescence, one for each class of VM; the administrator would migrate VMs from one instance to the other as the VMs are upgraded. The second way is for the administrator to *manually* label healthy clusters in some fashion, so that Fluorescence would not need to assume that only the largest cluster is healthy. (This would require a change to Fluorescence, and it would be a kind of kernel-specific knowledge being added to the classifier.) In practice, the first approach is likely to be preferable, because manual labeling would not improve the performance of the autoencoder-based classifier.

The current implementation of Fluorescence does not consider dynamically generated code ("JIT-compiled code"), which can cause the kernel code pages of healthy VMs to diverge. If a VM's kernel contains JIT-compiled code, it will likely be identified as an anomaly. Legitimately JIT-compiled kernel code, such as that produced by eBPF, is typically verified before is it executed, and as such can generally be considered to be benign. Code pages created by eBPF can be recognized by the magic code `0xeB9F` [18], but it would be an obvious security problem for Fluorescence to simply ignore pages marked with this code. We leave the development of appropriate normalization (§2.1.2) methods for JIT-compiled kernel code as future work.

6 Related Work

We classify existing rootkit detectors into three general categories: baseline-based approaches, integrity-protection approaches, and comparative anomaly-detection approaches. By contrasting Fluorescence's herd anomaly-detection approach to systems in these categories, and we show that Fluorescence advances the state of the art.

Baseline approaches. Many rootkit detectors [2, 5, 9, 28, 34, 42] require the computation or collection of a *baseline* prior to deployment, such as a sample of a known rootkit (a negative baseline) or a sample of an uninfected operating system (a positive baseline). Hancock [16] and Hamsa [24] generate signatures of rootkit samples; others [8, 19, 35, 36, 44] learn from known rootkits' behavior and generate patterns to match future rootkit execution. These approaches are limited because they can only detect rootkits similar to known signature or behavior patterns. Invariant-based detection [10, 14, 25, 32] establishes a correct view of key kernel structures or state, and detects anomalies when that state invariant is violated. These techniques require comprehensive knowledge of specific kernels and assume kernel source code availability. In contrast, Fluorescence requires no baseline and minimal kernel-specific knowledge, which enables it to detect anomalies in both Windows and Linux VMs.

Integrity protection. Some systems periodically check the integrity of kernel critical components, including code and key data structures. Although not yet fully adopted by the Linux kernel, Kernel Patch Protection [13] and Driver Signature Enforcement [17] have been applied to 64-bit Windows and have raised the bar for kernel rootkit development [37]. However, if a rootkit evades these defenses, it can simply disable the protection [7]. The `pitou` rootkit [41], which affects Windows XP through Windows 10, infects the MBR, and bypasses kernel-mode code signing to load a malicious kernel driver. In contrast to integrity protection within a VM, Fluorescence works outside the VM, and thus cannot be disabled by a rootkit unless it escapes the VM and executes code in the hypervisor; this is considerably more difficult.

Comparative approaches. Baseline-based tools may be difficult to deploy, due to the difficulty of constructing an appropriate baseline sample. Cross-view detection, applied in tools like GMER [15] and RootkitRevealer [38], compares multiple different views of the same system state to find inconsistencies [4]. These approaches require significant manual effort to identify the system state to be observed and compared. Diffy [27] is a live cloud triage tool that provides mechanisms for both baseline- and clustering-based anomaly detection. It runs a special agent in each monitored VM to collect OS-level information. In contrast, Fluorescence detects anomalies without relying on in-VM software and semantics.

Blacksheep [3] makes the assumption that infection begins in a minority of a set of homogeneous machines. It detects anomalies by collecting memory dumps from the machines, extracting selected system information, and comparing pairwise to measure distance via clustering. Fluorescence starts from the same assumption as Blacksheep, but is designed to be less reliant on OS semantics and to detect code injection and modification attacks.

Because Blacksheep makes heavy use of OS semantics to exclude "benign differences" between VMs, its approach is difficult to port to non-Windows systems. For example, Blacksheep requires the PE header to handle load-time relocations, but object headers in Linux are removed during object load. Furthermore, the Linux kernel performs additional load-time code patching beyond object relocation and kASLR, e.g., in the `.paravirtualization` ELF section, and this noise should also be considered benign. Instead, Fluorescence uses very limited OS semantics, and its approach is viable on mul-

tiple versions of both Windows and Linux. For each memory dump, Blacksheep must transport gigabytes of files across a network for analysis. In contrast, a Fluorescence fingerprint is less than 3 MB. Blacksheep weaves its data summary and comparison together while making use of a large amount of OS semantics. Fluorescence decouples its data-summary and VM-comparison processes: by comparing VM fingerprints, rather than "raw" memory snapshots, Fluorescence enables scalable monitoring and clustering.

7 Conclusion

Fluorescence is a novel tool that detects kernel-resident malware infections within a "herd" of similar virtual machines. It uses virtual machine introspection-based observations to identify anomalies, without the need for training over specific anomalies. Previous work relied on knowledge of specific kernels to cross the "semantic gap" and compare kernel state, or made assumptions about the malware being searched for. Fluorescence's more general approach to anomaly detection does not rely on information that is specific to a single target kernel: the fingerprinting procedure needs some low-level information to acquire and normalize kernel memory samples, but the anomaly-detection process needs no kernel- or architecture-specific information at all. Fluorescence is scalable because the examination and summarization of VMs happens in parallel across the VM hosts; the central server receives and operates on fingerprints, not full memory snapshots; and the feature-alignment and analysis algorithms are chosen and engineered to be fast. We report that Fluorescence can analyze a herd of 200 VMs in ∼60–80 minutes. Large herds can be divided into subherds for faster analysis.

Acknowledgments

We thank VirusTotal and VirusShare for making malware samples available to us, T. Roy from CodeMachine Inc. for sharing his expertise about Windows rootkits, and Mingbo Zhang at Rutgers University for helping us to debug our Windows kernel-memory-pinning driver. We thank the anonymous RAID reviewers and our shepherd, Andrea Lanzi, for their valuable comments and help in improving this paper. This material is based upon work supported by the National Science Foundation under Grant Numbers 1314945 and 1642158.

References

[1] Ighor Augusto. Reptile rootkit. Commit b0a2d0f, April 2018. URL https://github.com/f0rb1dd3n/Reptile.

[2] Rishi Bhargava and David P. Reese, Jr. System and method for passive threat detection using virtual memory inspection, March 14, 2017. U.S. Patent 9,594,881 B2.

[3] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proc. CCS*, pages 341–352, October 2012. doi: 10.1145/2382196.2382234.

[4] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Learning, 2009. ISBN 978–1598220612.

[5] Michael Boelen, John Horne, et al. The Rootkit Hunter project. Release 1.4.6, February 2018. URL http://rkhunter.sourceforge.net/.

[6] Ed Bukoski, Brian Moyles, and Mike McGarr. How we build code at Netflix. Netflix Technology Blog, March 9, 2016. URL https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15.

[7] Yuriy Bulygin, Mikhail Gorobets, Andrew Furtak, and Alex Bazhaniuk. Fractured Backbone: Breaking modern OS defenses with firmware attacks. Presentation at Black Hat USA, July 2017. URL https://youtu.be/ryKy9LvmSIs.

[8] Chen Chen, Darius Suciu, and Radu Sion. POSTER: KXRay: Introspecting the kernel for rootkit timing footprints. In *Proc. CCS*, pages 1781–1783, October 2016. doi: 10.1145/2976749.2989053.

[9] Amit Dang, Preet Mohinder, and Vivek Srivastava. System and method for kernel rootkit protection in a hypervisor environment, June 30, 2015. U.S. Patent 9,069,586 B2.

[10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007. doi: 10.1016/j.scico.2007.01.015.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. KDD*, pages 226–231, August 1996. URL http://www.aaai.org/Library/KDD/1996/kdd96-037.php.

[12] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet dossier. White paper, version 1.4, Symantec Corporation, February 2011. URL https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[13] Scott Field. An introduction to kernel patch protection. MSDN Blog, August 12, 2006. URL https://blogs.msdn.microsoft.com/windowsvistasecurity/2006/08/12/an-introduction-to-kernel-patch-protection/.

[14] Francesco Gadaleta, Nick Nikiforakis, Jan Tobias Mühlberg, and Wouter Joosen. HyperForce: Hypervisor-enforced execution of security-critical code. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research: SEC 2012*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 126–137. Springer, June 2012. doi: 10.1007/978-3-642-30436-1_11.

[15] GMER. GMER - rootkit detector and remover, 2016. URL http://www.gmer.net/.

[16] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection: RAID 2009*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer, September 2009. doi: 10.1007/978-3-642-04342-0_6.

[17] Ted Hudek and Cymoki. Driver signing. Microsoft Windows documentation, April 2017. URL https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing.

[18] The Kernel Development Community. BPF type format (BTF), 2019. URL https://www.kernel.org/doc/html/latest/bpf/btf.html.

[19] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proc. USENIX Security*, pages 351–366, August 2009. URL https://www.usenix.org/legacy/events/sec09/tech/full_papers/kolbitsch.pdf.

[20] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3(Supplement):91–97, September 2006. doi: 10.1016/j.diin.2006.06.015.

[21] Jesse D. Kornblum. Exploiting the rootkit paradox with Windows memory analysis. *International Journal of Digital Evidence*, 5(1):1–5, Fall 2006. URL http://www.utica.edu/academic/institutes/ecii/publications/articles/EFE2FC4D-0B11-BC08-AD2958256F5E68F1.pdf.

[22] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proc. NDSS*, February 2009. URL https://www.ndss-symposium.org/ndss2009/k-tracer-system-extracting-kernel-malware-behavior/.

[23] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proc. ACSAC*, pages 386–395, December 2014. doi: 10.1145/2664243.2664252.

[24] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proc. IEEE S&P*, pages 32–46, 2006. doi: 10.1109/SP.2006.18.

[25] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. NDSS*, February 2011. URL https://www.ndss-symposium.org/ndss2011/siggraph-brute-force-scanning-of-kernel-data-structure-instances-using-graph-based-signatures/.

[26] Victor Ramos Mello. Diamorphine rootkit. Commit ba97922, March 2018. URL https://github.com/m0nad/Diamorphine.

[27] Forest Monsen and Kevin Glisson. Netflix Cloud Security SIRT releases Diffy: A differencing engine for digital forensics in the cloud. Netflix Technology Blog, July 17, 2018. URL https://medium.com/netflix-techblog/netflix-sirt-releases-diffy-a-differencing-engine-for-digital-forensics-in-the-cloud-37b71abd2698.

[28] Nelson Murilo and Klaus Steding-Jessen. Chkrootkit. Version 0.52, March 2017. URL http://www.chkrootkit.org/.

[29] nurupo. Nurupo rootkit. Commit 78faabd, December 2017. URL https://github.com/nurupo/rootkit.

[30] Rajendra Patil and Chirag Modi. An exhaustive survey on security concerns and solutions at different components of virtualization. *ACM Comput. Surv.*, 52(1):12:1–12:38, February 2019. doi: 10.1145/3287306.

[31] Bryan D. Payne, Tamas K. Lengyel, Steven Maresca, Antony Saba, et al. LibVMI: Simplified virtual machine introspection. Version 0.12.0, April 2018. URL https://github.com/libvmi/libvmi.

[32] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. USENIX Security*, pages 289–304, July/August 2006. URL https://www.usenix.org/legacy/events/sec06/tech/petroni.html.

[33] Nguyen Anh Quynh. Capstone disassembly framework. Version 3.0.5-rc2, March 2017. URL https://github.com/aquynh/capstone.

[34] Jayakrishnan Ramalingam. Rootkit monitoring agent built into an operating system kernel, September 17, 2013. U.S. Patent 8,539,584.

[35] Junghwan Rhee, Ryan Riley, Zhiqiang Lin, Xuxian Jiang, and Dongyan Xu. Data-centric OS kernel malware characterization. *IEEE Trans. on Information Forensics and Security*, 9(1):72–87, January 2014. doi: 10.1109/TIFS.2013.2291964.

[36] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proc. EuroSys*, pages 47–60, April 2009. doi: 10.1145/1519065.1519072.

[37] T. Roy. Personal communication, October 2016.

[38] Mark Russinovich. RootkitRevealer v1.71. Microsoft Windows documentation, November 2006. URL https://docs.microsoft.com/en-us/sysinternals/downloads/rootkit-revealer.

[39] Dan Sullivan. Beyond the hype: Advanced persistent threats. White paper, 2011. URL https://www.realtimepublishers.com/book.php?id=197.

[40] Symantec Corporation. Advanced persistent threats: A Symantec perspective. White paper, 2011. URL https://www.symantec.com/content/en/us/enterprise/white_papers/b-advanced_persistent_threats_WP_21215957.en-us.pdf.

[41] Gianfranco Tonello. Bootkits are not dead. Pitou is back!, January 2018. URL https://www.tgsoft.it/english/news_archivio_eng.asp?id=884.

[42] Trend Micro Inc. OSSEC, open source host-based intrusion detection system, 2019. URL https://github.com/ossec/ossec-hids.

[43] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, December 2002. URL https://www.usenix.org/legacy/event/osdi02/tech/white.html.

[44] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proc. NDSS*, February 2008. URL https://www.ndss-symposium.org/ndss2008/hookfinder-identifying-and-understanding-malware-hooking-behaviors/.

[45] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. arXiv preprint arXiv:1212.5701, December 2012. URL https://arxiv.org/abs/1212.5701.