Redirecting Malware's Target Selection with Decoy Processes

Sara Sutton, Garret Michilli, Julian Rrushi

▶ **To cite this version:**

Sara Sutton, Garret Michilli, Julian Rrushi. Redirecting Malware's Target Selection with Decoy Processes. 33th IFIP Annual Conference on Data and Applications Security and Privacy (DBSec), Jul 2019, Charleston, SC, United States. pp.398-417, 10.1007/978-3-030-22479-0_21 . hal-02384597

HAL Id: hal-02384597

https://inria.hal.science/hal-02384597

Submitted on 28 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Redirecting Malware's Target Selection with Decoy Processes

Sara Sutton, Garret Michilli, and Julian Rrushi

Department of Computer Science and Engineering,
Oakland University, Rochester MI 48309, USA
{smsutton2, gdmichilli, rrushi}@oakland.edu

Abstract. Honeypots attained the highest accuracy in detecting malware among all proposed anti-malware approaches. Their strength lies in the fact that they have no activity of their own, therefore any system or network activity on a honeypot is unequivocally detected as malicious. We found that the very strength of honeypots can be turned into their main weakness, namely the absence of activity can be leveraged to easily detect a honeypot. To that end, we describe a practical approach that uses live performance counters to detect a honeypot, as well as decoy I/O on machines in production. To counter this weakness, we designed and implemented the existence of decoy processes through operating system (OS) techniques that make safe interventions in the OS kernel. We also explored deep learning to characterize and build the performance fingerprint of a real process, which is then used to support its decoy counterpart against active probes by malware. We validated the effectiveness of decoy processes as integrated with a decoy Object Linking and Embedding for Process Control (OPC) server, and thus discuss our findings in the paper.

Keywords: Malware interception · Decoy processes · Operating system kernel · Deep learning.

1 Introduction

Malware keep wreaking havoc in both general-purpose computing and industrial control systems, despite various types of defense tools deployed against them. Amongst those tools, honeypots showed exceptional promise. Malware detection on honeypots is straightforward and unequivocal, since they have no activity of their own. Any system or network operation is indicative of intrusion. High interaction honeypots, in particular, provide the utmost protection. They run operating system (OS) services that are identical to those on machines in production. They also intentionally allow malware to run on the decoy machine. These factors contribute to a deep insight into malware's exploits and rootkit operations, which the defender can turn into signature or rule-based detectors to protect machines in production from the same or somewhat similar malware.

Nevertheless, advanced malware select their targets wisely. They probe their targets for inconsistencies that reveal decoys. In order to design better decoys, we experimented with a practical approach that uses live performance counters to detect a honeypot, as well as decoy I/O on machines in production. Decoy I/O consists of phantom I/O devices and supporting mechanisms that are deployed on machines in production [16]. Performance counters are data that characterize the performance of a process, kernel driver, or the entire OS. Their intended use is to help determine performance bottlenecks and fine-tune machine performance. Performance counters are provided by the OS and hardware devices [3].

Contribution. This work defensively affects malware's target selection by means of decoy processes. It causes changes in malware's findings in order to enable a decoy to qualify as a valid target of attack. The existence of a decoy process is projected onto a machine via instrumentation of data structures related to performance counters in the OS kernel. The performance consistency of a decoy process is attained via deep learning. We design and train a convolutional neural network that can learn the performance profile of a real process, which we use to support its decoy counterpart against active probes by malware. The OS of reference in this work is Microsoft Windows.

Novelty. To the best of our knowledge, this work is the first to leverage OS-level performance data to project a decoy process and protect it from adversarial probes. We explored data structure instrumentation in our previous work to emulate the existence of a decoy process [17]. Nevertheless, those data structures were strictly related to processes and threads and hence only created partial decoy process existence without run-time performance dynamics.

Saldanha and Mohanta from Juniper Networks proposed a deception methodology based on decoy processes called HoneyProcs, with a patent pending [1]. HoneyProcs aims at detecting malware that inject code into other processes. HoneyProcs works by creating a real process, which tries to mimic a legitimate process. Once the decoy process reaches a steady state, it stops making progress with its execution, which leaves its state immutable. HoneyProcs uses such fixed state as a baseline against any changes, including those caused by code injection. HoneyProcs is vulnerable to the very same decoy detection technique that we used in our honeypot experiment, which is discussed in detail later on in this paper.

Real-time performance counters show that the resource utilization of a decoy process freezes to constant or 0 values. For example, a simple analysis of the working set of a decoy process reveals that its size, namely the number of its memory pages that are currently present in physical main memory, remains constant or decreases due to the global memory frame replacement algorithm. This is abnormal, given that the working set is a moving window representing memory localities. Similarly, the page fault rate of a decoy process swiftly drops to 0, while a continuous 100% page hit rate simply is not possible due to demand paging in virtual memory.

Organization. The remaining of this paper is organized as follows. Section 2 describes an experiment that reveals a detectability weakness of honeypots stemming from their complete lack of activity. Section 3 visits the OS mechanisms behind the display of decoy processes on a machine. Section 4 describes the deep learning approach that protects a decoy process from malware probes. Section 5 reports on implementation, testing, and validation of this work. In Section 6 we discuss research related to various aspects of this work. Section 7 summarizes our findings and concludes the paper. The appendices desribe the threat model, define what is out of scope, and discuss additional related works.

2 Honeypot Experiment

Stress testing decoy covertness. The purpose of this experiment was to assess the ability of honeypots and decoy I/O to protect their decoy function in practice. The experiment was performed from a red team's perspective. It was done separately on a Windows honeypot, and then on a Windows machine in production equipped with decoy I/O. The testbed was comprised of two desktops and a laptop machine, all three of which were connected to a local area network that was logically and physically isolated from any other networks.

0-value exploit. We simply ran metasploit [2] to launch a publicly known exploit against the honeypot, leveraging a publicly known vulnerability. The exploit returned a command prompt, i.e. a shell, which was usable to fetch and run additional code. The test was detected as soon as the first packet reached the honeypot machine. Nevertheless, none of the exploit, nor the vulnerability, was of any value to the defender by virtue of all of this material being public and hence already well known. What was left for the defender was to wait for the testers' next steps, namely operations like those referenced in the threat model.

Engaging performance probes. At this point, we actively collected performance data regarding host processor utilization, memory use, and secondary storage activity. We wrote a PowerShell script with the purpose of gathering those performance data live and in real-time. A large number of samples are collected every second until a data repository is filled. The script enabled us to view a table of the names and process identifiers of all processes currently running on the system, as well as view and store all the details and attributes of a specific process of our choice.

Searching for patterns of absent/low resource utilization. We found that per-process performance analysis is much more accurate in spotting inactivity than machine-wide performance analysis. The honeypot was characterized by host processor time that was somewhat comparable to a machine in production in low use. Processor time refers to the percentage of elapsed time that the processor spends executing an active thread. A similar observation holds for the amounts of time the processor spent executing user space code and kernel space code. In this experiment, the code that generated all this machine-wide activity consisted of our own script in user space, and honeypot monitoring tools in kernel space.

The execution of all this code overall also generated interrupt arrival rates and page fault rates that were hardly distinguishable from their counterparts on a machine in production in low use. A complicating factor is that, at times, multiple independent threat actors may land on a honeypot. Often cases mal-

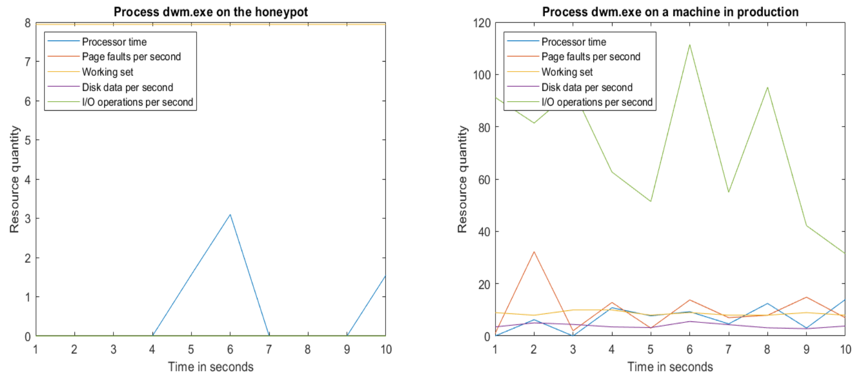

Fig. 1. Visual comparison between a process' performance on the honeypot and its performance on a machine in production.

ware even compete with each-other. The lack of attribution in machine-wide performance parameters hinders honeypot inactivity detection. These findings informed our decision to direct deep learning towards the performance profile of specific processes rather than the machine as a whole.

When directing our script towards specific processes, in most cases we obtained performance counters that indicated a total lack of any resource utilization. In a few cases, performance counters revealed existent but low resource utilization, which we deemed to be related to our own moves on the honeypot. Processes on a honeypot simply do not make progress with their execution, consequently their processor time is 0. New pages in memory are not referenced, consequently no page faults occur. Human-machine interaction is absent and thus interrupts do not occur. Secondary storage is not accessed, consequently the data rate and the number of I/O operations per second are both null.

Our findings are illustrated in Figure 1. The data plot on the left shows some of the performance parameters of the Desktop Windows Manager (DWM) on the honeypot, where patterns of absent or low resource utilization are clearly evident. The visualized performance parameters, except the working set, are constantly 0. There are a few processor time spikes, however those are very minimal. It is interesting to see how the working set, which is represented by the flat horizontal line at the very top, never changed from 87420928. Of course, with no page faults occurring, the working set could not change. In the data plot, we have applied a log_{10} reduction of the working set to make it fit within the same plot as the other performance parameters.

The data plot on the right shows the same performance parameters of the DWM process, but this time on a machine in production. The working set and the disk data per second have both been reduced log_{10}. They are high and variable.

Experiment repeated on decoy I/O. This time, the probes were directed against a decoy process amongst real processes on a machine in production.

Decoy I/O consisted of a decoy network interface controller, which projected a decoy network providing connectivity to a decoy Object Linking and Embedding for Process Control (OPC) server, as in [17]. The decoy process was an OPC client, which, just like HoneyProcs, maintained a consistent appearance. More specifically, it appeared to load the same libraries, had the same size on disk, and created the same number of threads, as its real counterpart. Nevertheless, when probed over performance counters, the decoy OPC process was immediately detected as in the honeypot experiment.

Multiple performance samples are needed for accuracy. It is normal for a machine in production, and hence a valid malware target, to have periods of inactivity or low use, which may be quite common. In this experiment, performance probes were collected over an extensive time window to make sure that production activity was observed if existent.

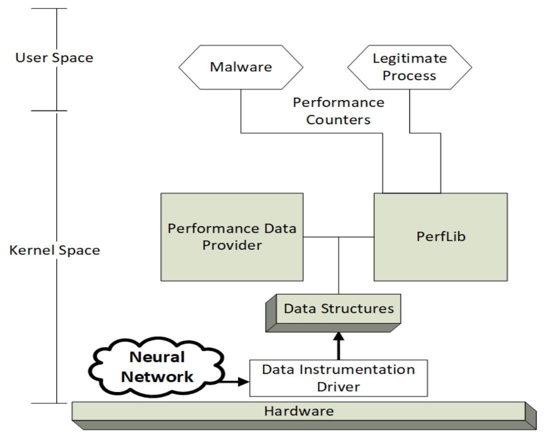

Fig. 2. Decoy process visibility via performance data instrumentation.

3 Decoy Processes

Make visible, but do not create. In this work, the idea is to only project or display the existence of a process for malware to see rather than concretely create that existence. In other words, we aim at making a process that is as visible as its real counterpart, without having to spawn it. The rationale is simple. Spawning a real process, albeit for use as a decoy, consumes real resources on the machine, which adds to the overhead of running detection tools specific to decoy processes. The data, code, and libraries of a decoy process would need to be stored in real frames in main memory. As we discussed earlier in this paper, freezing execution to create an immutable state is ineffective, therefore there would have to be real activity, which would consume CPU cycles and secondary storage.

Own real activity comes with its own complexities, since it needs to be distinguished from malicious activity, thus bringing the malware detection challenge almost in a form similar to conventional intrusion detection. The browsing presence of a decoy process is achieved by inserting a synthetic entry in the process table in the OS kernel. Furthermore, most of the known techniques to hide a malicious process are usable to create exactly the opposite effect, namely show the existence of a process that does not exist. The task manager tool, the tasklist command, and the ps command, all display an entry for the nonexistent process in their output.

The visibility of a decoy process is attained as illustrated in Figure 2. Performance counters originate in drivers in the OS kernel. These drivers operate as performance counter library (PERFLIB) providers, which furnish performance data in response to queries. Performance data are accumulated in data structures, such as linked lists. We have written data structure instrumentation code, which deposits synthetic performance data for a decoy process in the repository of performance counters. These performance data, real and synthetic, are then provided to a consumer in user space, including possible malware. This way we project the existence of a decoy process by means of synthetic resource utilization dynamics. Clearly the synthetic performance data need to be consistent, which we address via a neural network and discuss in detail later on in this paper.

Timing the replies to performance queries. Performance data are counted in the OS kernel during specific time windows as related events occur. For example, a counter of page faults is incremented each time a trap to the OS kernel is made as a result of a reference to a page that is not present in physical main memory. The counter's value is not stored immediately in the repository of performance counters. Instead, it is buffered until the counting period is complete. Consumers of performance data in user space will not receive fresh counter data until after the counting period. It is of paramount importance that the data instrumentation driver depicted in Figure 2 does not deposit the synthetic performance data too fast or too slow in the repository of performance counters.

In this work, the synthetic performance data are decided and produced by a neural network. This process takes relatively little time, since the neural network is already fully trained at the time it is utilized as a source of such data. The neural network needs the performance counters that pertain to all real processes on the machine in order to function. The data instrumentation driver collects these performance counters by accessing directly the repository where they are stored. Once the neural network delivers the performance counters for a decoy process to the data instrumentation driver, the latter buffers them until the end of the counting period, at which point it stores them in the repository.

Safety. Making a nonexistent and hence a decoy process visible via synthetic performance data is safe on honeypots. There are no humans who interact with honeypots while the latter are in operation, consequently the risk of a user interacting with a decoy process is null. The risk on a machine in production equipped with decoy I/O is considerable. We rely on a safety measure from related previous work [17], which is a filter driver integrated into the driver stack of the monitor device. The driver filters out decoy entries from frames of bytes bound for the monitor, before those data have traveled far enough to be displayed on the monitor. Since we know the name and the performance data of the decoy process a priori, we can have them filtered out from the user's visual.

4 Performance Support for a Decoy Process

We now discuss how our approach learns the performance fingerprint of a real process, to be able to perform performance recognition tasks in support of the process' decoy counterpart. We express details of our approach through the lenses of deep learning. The reader is referred to [7] for a detailed discussion of deep learning. In this paper, we base our reasoning on an OPC client process on a machine in production. The rationale for selecting an OPC client as a subject of deep learning and decoy process is connected to its integration with a decoy I/O capability, which we developed in our previous work [17]. Nevertheless, we deem that this work is applicable to all processes.

The rationale for solving the decoy process performance challenge on a machine in production is that the latter presents an environment that is much more complex than the environment of a honeypot. On a honeypot, most or all processes can be configured to be decoy processes, whose performance parameters we can choose ourselves. This makes it easier to calculate their projected resource utilization. On a machine in production, the performance of real processes is beyond our control, therefore our approach needs to be robust enough to work with any possible values they may have. And all this while malware are probing for performance inconsistencies.

4.1 Heatmaps

Heatmap design. We model the machine's resource utilization as a heatmap, where performance parameters are represented as a color with a given strength. An example heatmap used in this work is depicted in Figure 3. The higher a performance parameter, the stronger its color in the heatmap. An excerpt from the set of performance parameters that we used in this work is given in Table 1. These parameters are taken from the whole resource utilization spectrum, in the hope that they can enable our approach to learn the performance fingerprint of a process. With performance parameters and real processes aligned along the ordinate and abscissa, respectively, each heatmap cell visually indicates the value of a performance parameter for a specific real process.

The idea is to train the neural network by feeding it a large number of images generated by heatmaps. Each image of heatmap is labeled in such a way that its class label, i.e. an object type associated with the heatmap, is an array of color strengths, namely one color strength for each performance parameter of the decoy process. If training succeeds, the neural network can be used for heatmap recognition. The neural network reads a heatmap, which most likely was not seen during the training phase, and produces in output a class label. The class label,

Table 1. Some of the performance counters visually assembled in heatmaps.

CPU	Memory	Secondary Storage
User Time	Page Faults/sec	IO Read Operations/sec
Privileged Time	Working Set	IO Write Operations/sec
-	Working Set Peak	IO Other Operations/sec
-	Pool Paged Bytes	IO Read Bytes/sec
-	Pool Nonpaged Bytes	IO Write Bytes/sec

in turn, informs our approach as to what specific values to give the performance counters of the decoy process.

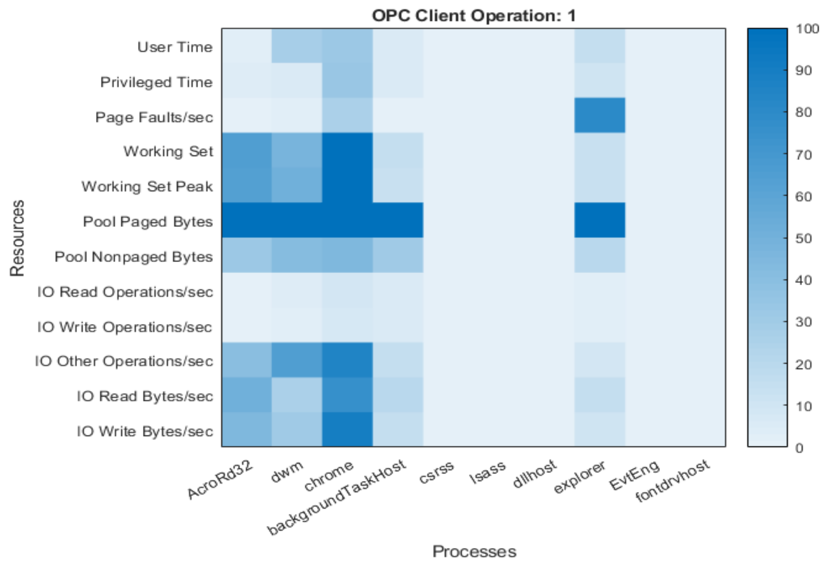

Fig. 3. A performance heatmap for neural network consumption.

Adapting to the performance of real processes. The decoy process, in our case the decoy counterpart of an OPCExplorer process, exhibits performance parameters that depend directly on the resource utilization of real processes on the machine. When probed by malware, we take a screenshot of the performance counters of real processes, metaphorically speaking, and turn them into a heatmap for recognition by the neural network. All processes are taken into account when estimating the performance parameters of the decoy process. We only show a few in Figure 3 to make the heatmap fit within the page borders. In reality, heatmaps are much larger.

The resource utilizations of any processes created by malware are also included in the heatmaps. Those processes are referred to as foreign_process$_n$ in the heatmaps, regardless of how they are named by the threat actors. Standard internal names for such processes prevent the neural network from getting confused. Multiple processes may be created off the same executable file. For example, the user may be running several chrome tabs, each of which runs as a separate process. Of course, we include the performance of all such processes in the heatmaps. This is what makes our approach cognizant of the current resource utilization load on the machine.

Performance correlation with input data. The heatmap of Figure 3 contains an explicit process activity indication at the very top. This is for the neural network to include in its internal heatmap processing. The activity indicator ties the performance fingerprint of the process to be mimicked with the input data of that process. When fed with different input data, a real OPC-Explorer process may have totally different resource utilizations for the same resource utilization load on the machine. The same heatmap leads to different performance parameters for different input data. We noticed that the performance of a process is insensitive to small variations of input data. Instead, a better input categorization is needed, which can indeed cause a visible change in resource utilization.

All processes have well defined operations in their design, which we find to be meaningful enough to resource utilization to cause changes. In this work, we use such operations to relate input with resource utilization. The operations that we used within heatmaps pertaining to the OPCExplorer process are summarized in Table 2. OPC consists of objects that are based on the Microsoft Distributed Component Object Model (DCOM). COM enables objects on the same machine to exchange data with each-other. DCOM is basically COM, but with the added functionality of enabling objects that reside on different machines to exchange data with each-other over the network.

An OPC object is a DCOM object. As all objects, an OPC object has methods and attributes. The attributes are also known as tags, or data points, which represent parameters of a physical system. Examples include voltage, phase, and current. An OPC server hosts OPC objects, which an OPC client can access in reading or writing over the network. The reader is referred to [10] for a detailed specification of OPC. Some of the operations in Table 2 refer to groups. These are sets of tags, possibly from different OPC objects, which the system operator has reasons to gather together when performing a given OPC task.

We assign numerical values to OPC operations, which we refer to as opcodes. These are identifiers that we use to differentiate OPC operations from each-other. Opcodes are then included in heatmaps for the neural network to process along with the other data. For example, the heatmap of Figure 3 shows an opcode of 1, which corresponds to viewing OPC server properties on the OPC client.

During testing experience we noticed the I/O performance parameters including I/O write and read operations from secondary storage are always 0 value as those processes didn't consume any I/O resources during our testing experiment. These counters are included in the label construction, however they are always of value zero and therefore do not effect classification. The collected label measurements are used for training our neural network.

Table 2. Categories of operations on an OPC client as used in heatmaps.

OPC Server Ops	**Group-level Ops**	**Tag Related Ops**
View OPC server properties	View group properties	List tags of an OPC object
Add an alarm	Change group properties	Add a tag to an OPC object
-	Create a new group	Include a tag in a group
-	Delete an existing group	Remove a tag from a group
-	-	Read a tag
-	-	Write a tag

4.2 Deep Learning of Performance Fingerprints

Training set and labeling. As we run the OPCExplorer process to perform the operations of Table 2 one at a time, we collect the performance counters of all other processes on the machine. Those performance data enable us to build heatmaps. We also collect the performance counters of the OPCExplorer process, which collectively enable us to unequivocally establish a class label for each heatmap. All these labeled heatmaps are used to train the neural network.

Test set. We repeat the previous steps, but this time do not include the labeled heatmaps in the actual training of the neural network. We set aside these labeled heatmaps for later use, once the neural network is fully trained.

Algorithmic approach. A convolutional neural network has multiple layers of neurons which include at least one input layer and one output layer and some number of hidden layers including Rectified liner unit, pooling, fully connected and softmax. The hidden layers are used to adjust and scale the activation of given features from the heatmap images. Thus, the number of layers are critical.

The inner workings of the convolutional neural network are given in Algorithm 1. One of the most critical steps is the configuration of the layers of this neural network. We add a standard input layer to load and initialize the heatmaps from the training set for further processing. Several rectified linear unit (ReLU) layers are also added to the neural network. ReLU layers increase the pace and effectiveness of the performance fingerprint learning. They zero out negative values and maintain positive values in convolved heatmaps undergoing processing. We also add several pooling layers, which reduce the number of heatmap image parameters that the neural network needs to work with.

The neural network includes several batch normalization layers, which adjust and scale the activations of given features from the heatmap images. The fully connected layer produces a vector with size equal to the number of class labels. Each element of this vector is the probability for a class label of the heatmap image that was just processed by the neural network. Some of these probabilities may be negative. Furthermore, the sum of all these probabilities may not be 1.0. The softmax layer corrects such anomalies, and thus normalizes the vector in question into a probability distribution. The classification layer assigns the correct class label to a heatmap image that was just processed, on the basis of that probability distribution.

Once the training is complete, we run the neural network to classify heatmaps from the test set. We compare the known class labels for those heatmaps with the class labels produced by the neural network, in order to calculate the heatmap recognition accuracy. If the attained level of accuracy is low, we add more layers to the neural network and retrain it from scratch. We keep revising the neural network design until we attain a satisfactory accuracy.

Algorithm 1: Algorithm to train and test a convolutional neural network for heatmap classification.

```
1  Function Learn-Performance-Fingerprint ($(G, V)$);
   Input  : Training set of heatmaps $G$, testing set of heatmaps $V$.
   Output: Convolutional neural network $\mathit{\Pi}$, heatmap recognition accuracy $\delta$.
2  $\delta \leftarrow 0$
3  for $\forall$ heatmap $\nu \in G$ do
4      Read $\nu$ into array $\alpha$ in memory;
5      Add Label($\nu$) to $\alpha$;
6  end
7  while $\delta < 90$ do
8      Empty $\mathit{\Pi}$ if any layers present;
9      Define the input layer of $\mathit{\Pi}$;
10     Add $count_1$ ReLU layers to $\mathit{\Pi}$;
11     Add $count_2$ pooling layers to $\mathit{\Pi}$;
12     Add $count_3$ batch normalization layers to $\mathit{\Pi}$;
13     Add a fully connected layer to $\mathit{\Pi}$;
14     Add a softmax layer to $\mathit{\Pi}$;
15     Add a classification layer to $\mathit{\Pi}$;
16     Select $\mathit{\Pi}$'s training options;
17     trainNetwork($\mathit{\Pi}$);
18     for $\forall$ heatmap $\epsilon \in V$ do
19         $\delta \leftarrow \mathit{\Pi}(\epsilon)$
20     end
21     Increase $count_1$, $count_2$, and $count_3$;
22 end
```

Usable oracle. At this point, a fully trained neural network with high accuracy can be queried by the data structure instrumentation code. A query contains a heatmap that is representative of the resource utilization of all processes on the machine, of course excluding the decoy OPCExplorer process. The response by the neural network contains a class label, which the data structure instrumentation code can easily convert into performance data for the decoy OPCExplorer process. Those data are reported to malware in the form of performance counters in response to their probes.

5 Experimental Testing and Validation

Implementation. We wrote Matlab code to implement the deep learning approach. We also wrote other Matlab code to generate heatmaps. We extended the PowerShell script that we used in the honeypot experiment to collect live performance data from all processes running on the machine. These data are stored in files, which are then read by Matlab code to produce heatmaps. The sample interval and number of samples collected are specified by the operator. Increasing the number of samples collected per interval creates a heatmap with greater density of data points. Labeling the heatmaps was a tedious task, which we completed manually one heatmap at a time. To that end, we exercised the OPC client operations referenced in Table 2 manually by interacting with the OPC client software similarly to a system operator. As we ran those operations, we measured the performance counters of the real OPCExplorer process, which we then used for labeling heatmaps.

The need for manual and hence time consuming work limited the number of heatmaps that we could label and use to train the neural network, which in turn affects negatively the accuracy of the neural network itself. As an aside note, in terms of future work, an artificial intelligence approach that uses a virtual keyboard and mouse to drive the functionality of the OPC client software would be most useful to improve the feasibility of this work.

Testing against live malware. A large set of OPC malware samples involved in the Dragonfly malware campaign have been publicly available for academic research for quite some time. Those malware samples come in many versions. Nevertheless, none of these samples appeared to analyze system or network activity on the compromised machine prior to attacking an OPC server. They perform a network search for servers, identify those specific servers that host OPC objects, and then simply pursue the tags in those objects over the network. BlackEnergy style of malware attacks also seem to ignore system or network activity prior to initiating keystroke interception, or prior to making VPN connections over the network.

Extended/revised honeypot experiment. Since the use of performance counters to detect decoys is new, and thus there are no malware that use it, we do not seem to have the means of testing this work against real-world malicious code, as we have in our previous works. We had to return to the honeypot experiment, which, at the beginning, had succeeded to detect the honeypot and decoy I/O. We repeated the various experimental trials on a machine in production equipped with decoy I/O. This time, the red team approach was equipped with the details of the entire contribution made in this paper. In other words, the red team was assumed to have awareness of the fact that system activity on the machine might be due to decoy processes, and that the performance data of those decoy processes are regulated by a deep learning algorithm based on heatmap recognition.

Visually, decoy processes resemble their real counterparts. Most importantly, they also have performance dynamics, which we assessed by putting the red team in the best attack conditions possible. The thought processes are illustrated in Figure 4. More specifically, in our red team role, we had a replica of the machine to be protected, with the only difference being that the OPCExplorer process was real. We measured empirically the performance parameters of all processes, including those of the real OPCExplorer process. As we were performing those measurements on the replica, in some cases we intentionally left all processes in low or moderate use, except the real OPCExplorer process.

Since most of the performance parameters of the other processes were low or near constant, they perturbed the performance of the real OPCExplorer process by a lesser amount than on a usual machine in production. We called these performance measurements group 1 (G1). In other cases, we drove the other processes such as to perform average or higher load tasks, and called the corresponding performance measurements group 2 (G2) and 3 (G3), respectively. In G2 and G3 conditions, the other processes affected the performance of the real OPCExplorer process by a larger amount than in G1 conditions. Overall, these maneuvers enabled us in our red team role to collect measurements that statically tied the performance of all processes on the replica with the performance of the real OPCExplorer process.

Our convolutional neural network is trained to cope with any arbitrary amount of effect that the other processes may have on the performance of the real OPCExplorer process, and thus reproduce consistent performance parameters on its decoy counterpart under all circumstances. The reason we are emphasizing specific G1, G2, and G3 conditions, is that, from a testing or probing perspective, our experience with this research suggests that the likelihood of seeing performance parameters on a machine in production that are similar to those observed and recorded on the replica varies across the G1, G2, and G3 spectra. After all, the feasibility of these testing/probing techniques depends on the portability of resource utilization dynamics from the replica onto the compromised machine, i.e. the machine in production.

G1 conditions are the most favorable to a threat actor, since their occurrence is statistically more common, especially on client machines in production. At times, users commonly interact with a few application programs at a time. Some users place higher demand on their machine, in which case G2 and G3 conditions take place. Nevertheless, we found that, even when G2 and G3 conditions occur, they are hardly stationary enough to resemble a specific predefined resource utilization pattern characterized on the replica. A threat actor may attempt to interact with processes in order to force their resource utilization to get close to a precalculated resource utilization signature. However, we deem the following adversarial actions to be out of reach:

– Non-invasively reduce the resource utilization of a process that is taking input from the legitimate user. Thus, a threat actor may be able to adjust the resource utilization of a process by increasing it. If the adjustment requires a decrease, the threat actor is impotent.
– Interact with a process that has a graphical user interface (GUI). The reason is simple, namely the legitimate user will certainly notice. Making process AcroRd32 load a portable document format (PDF) file and scrolling over the pages, or making Chrome browse a website, will display the respective GUI components on the screen.

Instead of requiring our red team approach to adjust the resource utilization of processes, if possible, and/or wait for a lucky resource utilization combination to occur, we facilitated the red team assessment by creating usable G1, G2, and G3 dynamics on the compromised machine. This is what we meant with putting the red team in the best attack conditions possible earlier in this section. The appearance of usable G1, G2, and G3 dynamics on the compromised machine may be a rare event, but we assume it to be possible in order to favor the highest strength of the red team assessment.

In some of the tests, we left most of the processes on the compromised machine in low or moderate use. In other tests, we used same or similar stimuli as on the replica to create G2 and G3 conditions that were close enough to the performance signatures taken on the replica.

This is the culmination of the target validation on the compromised machine. Since in our red team role we had prior knowledge of the performance of the real OPCExplorer process within a precalculated resource utilization signature, we could simply compare the expected performance of the OPCExplorer process with the performance collected on the compromised machine. If the two diverged by a non-negligible amount, the conclusion would be that we had landed on a decoy. Some of the findings of these trials are depicted in Figure 5.

The performance fingerprint of a process is not fixed. The data plots in Figures 5 and 6 show that, often cases, we get slightly different performance data for a real process such as OPCExplorer, although the performance data of the other processes do not change or change minimally. For example, under identical or similar underlying performance dynamics, we measured a processor user space time for real process OPCExplorer equal to 3.61. A few seconds later, without any change of conditions, we measured 3.05. Consequently, a decoy performance inconsistency has to be a considerable departure from patterns of resource utilization, since small departures are normal.

Overall, our work is able to keep the performance data of a decoy process within the normal variability of the performance fingerprint of its real counterpart. We had cases of incorrect class labels produced by the neural network, however those were relatively infrequent. We deem that those misses were due

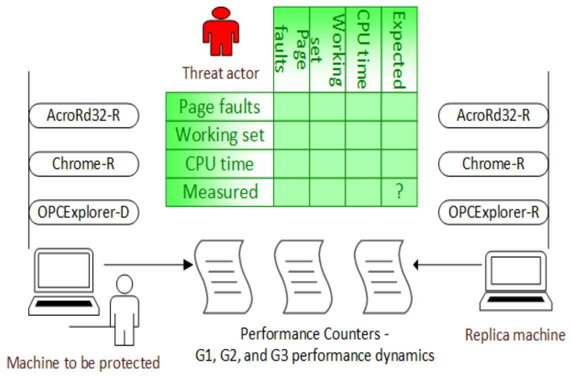

Fig. 4. Assessing the accuracy of our convolutional neural network in protecting a decoy OPCExplorer process from malware probes.

to the small number of heatmaps in the training set. With a larger training set, this work may attain a higher accuracy. We also had a few challenges during the actual measurements of performance data. The OPC client software that we worked with displayed hints and other help via pictures and other graphics on its graphical user interface. We noticed that the reading and displaying of those graphics one at a time, and for specific periods of time, did affect the performance parameters that we were measuring.

Load disturbance attempts. In our red team role, we created processes that requested large amounts of memory, consisted mostly of CPU bursts, or generated heavy I/O traffic. We also created processes that changed the amount of resources abruptly and quickly, from very high to very low, and then back to very high. The neural network tolerated these disturbances, with no noticeable class label changes.

6 Related Work

Several works have explored prediction models to estimate resource utilization at runtime. Matsunaga et al. surveyed supervised machine learning to train data points and predict execution time. However, the authors only attain detailed estimated in relation to fixed input data [12]. In contrast, in our work we consider any input data. Miu et al. examined features extracted from input data to find specific instances that maximize the accuracy of predicted execution time of a process. They used a combination of input features to learn regression models using C4.5 decision tree builders. Their method depends on learning from historical data [13]. Li et al. predicted scheduling in a Round Robin manner in the distributed stream data processing. For scheduling, a greedy algorithm is used to assign threads to machine under the guidance of prediction results [11]. Amiri et al. reviewed prediction models, including machine learning methods, to estimate performance and workload in the cloud [4].

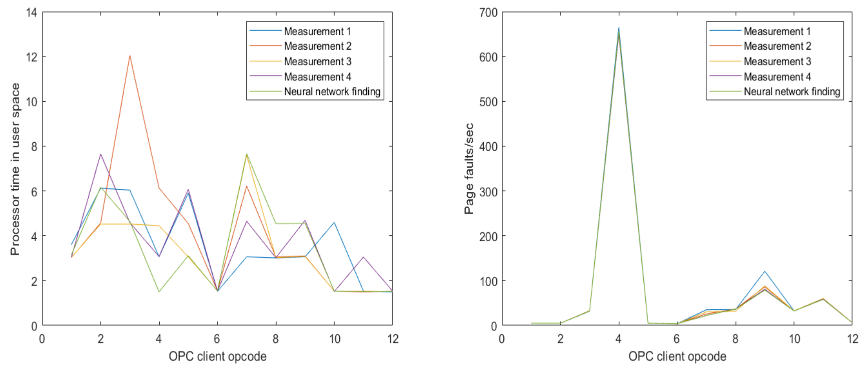

Fig. 5. Sample I - Empirical measurements of performance data versus deep learning class labels.

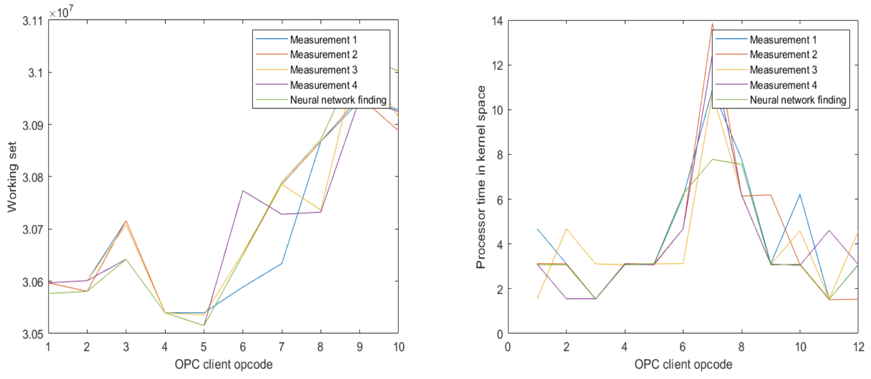

Fig. 6. Sample II - Empirical measurements of performance data versus deep learning class labels.

Pietri et al. proposed a method to predict execution time for a parallel workflow based on its structure in the cloud [14]. Their approach divides tasks to various levels based on their data dependency. Other related works focus on using machine learning to build a framework for mobile devices that can find features related to computational resource consumption from the input data that are given to a program [9]. These works use program slicing and sparse regression to extract pertinent information from program execution. Our work is different in that it is load dependent, and hence can predict the resource utilization parameters of a decoy process as other real processes continuously change their own performance parameters.

7 Conclusions

Live real-time performance counters enable a deep insight into the performance of a process. Our honeypot experiment showed that performance analysis of processes can catch the many inconsistencies of high interaction honeypots and decoy real processes, and can also be a threat to decoy I/O if left unaddressed. We described interventions in the OS kernel that project the existence of a decoy process, without having to spend resources on creating an actual process. We devised a convolutional neural network that can learn the performance fingerprint of a process in support of its decoy counterpart. In conclusion, we validated and quantified the ability of such decoy processes to sustain a realistic resemblance with a valid target of attack, thus possibly causing changes to malware's target selection.

Acknowledgment

This material is based on research sponsored by the USAFA and Oakland University under agreement number FA7000-18-2-0022. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
The opinions, findings, views, conclusions or recommendations contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the USAFA or the U.S. Government.

References

1. Honeyprocs: Going beyond honeyfiles for deception on endpoints. https://forums.juniper.net/t5/Threat-Research/HoneyProcs-Going-Beyond-Honeyfiles-for-Deception-on-Endpoints/ba-p/385830, accessed: 2019-02-23
2. Metasploit framework. https://www.metasploit.com/, accessed: 2019-02-23
3. Performance counters. https://docs.microsoft.com/, accessed: 2019-02-23
4. Amiri, M., Mohammad-Khanli, L.: Survey on prediction models of applications for resources provisioning in cloud. J. Netw. Comput. Appl. **82**(C), 93–113 (Mar 2017)

5. Butler, J., Undercoffer, J.L., Pinkston, J.: Hidden processes: the implication for intrusion detection. In: IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003. pp. 116–121. West Point, NY, USA (June 2003)
6. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 177–186 (2008)
7. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
8. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Vmm-based hidden process detection and identification using lycosid. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 91–100. New York, NY, USA (2008)
9. Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B.g., Huang, L., Maniatis, P., Naik, M., Paek, Y.: Mantis: Efficient predictions of execution time, energy usage, memory usage and network usage on smart mobile devices. IEEE Transactions on Mobile Computing **14**(10), 2059–2072 (Oct 2015)
10. Lange, J., Iwanitz, F., Burke, T.: OPC - From Data Access to Unified Architecture. VDE VERLAG GMBH, 4th edn. (2010)
11. Li, T., Tang, J., Xu, J.: Performance modeling and predictive scheduling for distributed stream data processing. IEEE Transactions on Big Data **2**(4), 353–364 (Dec 2016)
12. Matsunaga, A., Fortes, J.A.B.: On the use of machine learning to predict the time and resources consumed by applications. In: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. pp. 495–504. Washington, DC, USA (2010)
13. Miu, T., Missier, P.: Predicting the execution time of workflow activities based on their input features. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 64–72. Washington, DC, USA (2012)
14. Pietri, I., Juve, G., Deelman, E., Sakellariou, R.: A performance model to estimate execution time of scientific workflows on the cloud. In: 9th Workshop on Workflows in Support of Large-Scale Science. pp. 11–19 (Nov 2014)
15. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Proceedings of the 10th International Security Conference. pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (Oct 2007)
16. Rrushi, J.: Phantom i/o projector: Entrapping malware on machines in production. In: 12th International Conference on Malicious and Unwanted Software (MALWARE). pp. 57–66. Fajardo, Puerto Rico, USA (Oct 2017)
17. Rrushi, J.: Dnic architectural developments for 0-knowledge detection of opc malware. IEEE Transactions on Dependable and Secure Computing (2018)
18. Tsaur, W.J., Chen, Y.C., Tsai, B.Y.: A new windows driver-hidden rootkit based on direct kernel object manipulation. In: Hua, A., Chang, S.L. (eds.) Algorithms and Architectures for Parallel Processing. pp. 202–213. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

Appendix A: Threat Model

Probes originate from the inside. A malware sample has compromised a machine, and is now assessing whether or not it is a decoy. We have observed that this target validation assessment is commonly a precursor to attack operations such as the following:

– Launching local exploits to escalate the current privilege.
– Installing rootkits to preserve access.
– Installing I/O interceptors to capture keystrokes, webcam traffic, file system and network traffic.
– Accessing data and sending them to a threat actor over the network.
– Launching an exploit on the compromised machine against another target over the network.

These operations are typically implemented as separate malware modules, which follow the initial exploit. A multi-stage dropper downloads them onto the compromised machine over the network from another machine under threat actor's control. A single-stage dropper comes with these modules incorporated in it. The dropper itself is downloaded over the network similarly to the malware modules.

The initial exploit may yield partial or 0 value. On several occasions, the initial exploit may go undetected, consequently the malware operations referenced previously are the defender's opportunity to detect the malware based on its contact with decoys. A common case of this occurrence is when the initial exploit leverages a 0-day vulnerability on a machine in production equipped with decoy I/O. When targeting a honeypot, the same exploit is certainly detected upfront. Nevertheless, as we wrote earlier in this paper, it is possible to avoid making network contact with a honeypot on the basis of its lack of network activity. Some initial exploits yield no value to the defender, as in our honeypot experiment.

Withstanding probes is of significance. Decoy processes and their performance consistency, along with other types of consistency, are decisive on whether malware fall into a trap, or step away from a decoy target, erase themselves and hence disappear even before the defender sees any cues at all. An ineffective decoy results in none of the malware modules or even the dropper ever being brought onto the machine.

Appendix B: Out of Scope

The deep learning in this work needs to be hidden and protected from malware, otherwise threat actors may manipulate its computations and evade it. One solution is to run the deep learning on a virtual machine (VM), which is managed by a hypervisor and is isolated from the host machine. The overhead of a VM solution needs to be carefully assessed. Another solution is to run the deep learning on a hardware sideboard physically isolated from the host machine. This other solution comes with an added cost, which could be kept as low as under $50 with the right hardware design.

A honeypot's lack of network activity can be leveraged remotely to avoid attacking it. A threat actor operating on a compromised machine in production may select the next targets to be only those machines that the compromised machine is observed to communicate with. Since, by definition, no machine in production communicates with a honeypot, the threat actor will never hit a honeypot.

Because of room limitations, and to be able to describe the main contribution thoroughly, we do not include these efforts in this paper.

Appendix C: Additional Related Works

Several related works use stealth techniques to hide computer resources. Hooking, which prevents a request from accessing resource usage, and Direct Kernel Object manipulation (DKOM), which manipulates specific data in the OS kernel. Butler et al. described a non-hooking method to implement a device to hide and unlinked processes in EPROCESS blocks on Microsoft Windows [5]. On the other hand, Tsai et al. identified DKOM that can target all resources of an object directory, and thus alter and hide kernel objects that are commonly used by the OS in memory [18].

Jones et al. presented a technique to detect a virtual machine monitor (VMM)-based process that is maliciously hidden. This technique uses cross view validation, and then patches the executable code in order to affect its execution. The authors can detect any hidden processes that are running within a guest virtual machine. Their technique leverages CPU inflation, which is the CPU time consumed by each process within VMM and the guest operating system [8]. Unlike all these related works that we just discussed, our research stands out through the emphasis placed on creating a decoy process in EPROCESS to appeal to a threat actor, while hiding the decoy process from legitimate users.

As far as resource utilization prediction goes, we also use machine learning to predict performance parameters for a decoy process. However, our work is different than related works. As we mentioned earlier in this paper, our approach is load dependent. The other related works do not make load dependent estimations. Secondly, our work leverages input categorization based on process operations. Along with heatmap design and deep learning, these factors provide for a high level of accuracy, which is adequate to withstand malware probes.

Malware have a history of validating their targets prior to carrying out their operations. Some of these malware detect debuggers and/or virtual machines. An active debuger may be indicative of an execution environment operated by defenders in support of dynamic code analysis. Furthermore, a virtual execution environment is commonly used to host honeypots [6]. Similarly, some malware detect CPU emulators, which are also used for dynamic code analysis and honeypots [15]. As we wrote earlier in this paper, no other works appear to leverage OS-level performance data to detect decoys as of this writing.