Received 14 March 2023, accepted 24 April 2023, date of publication 1 May 2023, date of current version 8 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3271999

RESEARCH ARTICLE

Decoy Processes With Optimal Performance Fingerprints

SARA M. SUTTON[1] **AND JULIAN L. RRUSHI**[2], **(Member, IEEE)**
[1]School of Computing and Information Systems, Grand Valley State University, Allendale, MI 49401, USA
[2]Department of Computer Science, Oakland University, Rochester, MI 48309, USA

Corresponding author: Sara M. Sutton (suttosar@gvsu.edu)

This work was supported by the USAFA under Agreement FA7000-18-2-0022.

ABSTRACT Decoy targets such as honeypots and decoy I/O are characterized by a higher accuracy in detecting intrusions than anomaly, misuse, and specification-based detectors. Unlike these detectors, decoy targets do not attack an activity classification problem, i.e. they do not attempt to discern between normal activity and malicious activity. By design, decoy targets do not initiate system or network activity of their own, consequently any operation on a decoy target is unequivocally detected as malicious. However, we have found that this innate characteristic of decoy targets can be exploited by malware-initiated probes to detect them quite reliably. As a proof of concept, we describe red team tactics that collect and analyze live performance counters to detect decoy targets. To counter these threats on machines in production, we developed a defensive countermeasure that consists of decoy processes, with dynamics that are regulated and guarded by convolutional neural networks. Our deep learning approach characterizes and builds the performance fingerprint of a real process, which is then used to feed a performance profile into its decoy counterpart. Decoy processes emulate the existence of system activity, which is crafted to enable decoy I/O on machines in production to withstand malware probes. We evaluated the interplay between red team tactics and decoy processes integrated with a decoy Object Linking and Embedding for Process Control (OPC) server, and thus discuss our findings in the paper.

INDEX TERMS Malware interception, operating system kernel, deep learning, decoy, OPC.

I. INTRODUCTION

Malware is on the rise with the ever increasing amount of data transferred in modern communication networks. Malware impacts both general-purpose computing and industrial control systems. In response, many defensive tools and techniques have been utilized against them. Honeypots are among those defensive measures. Honeypots can detect malware quite reliably because they do not initiate any activity on their own. Nevertheless, decoy-savvy malware checks for inconsistencies in its targets prior to pursuing them. Should malware suspect a target to be a decoy, it will fall back, erase itself and hence disappears before the defender sees any clues at all.

As a proof of concept, we developed a red team approach that leverages live performance counters to detect a honeypot. The approach is also applicable against 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 [1]. Performance counters are data that characterize the performance of a process, kernel driver, or the entire operating system (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 [2].

In our previous works [3], [4], we designed decoy processes that enable a decoy to qualify as a valid target of attack and hence affect malware's target selection. We worked on two factors related to decoy processes, namely existence and performance consistency. Existence is achieved via instrumentation of data structures related to performance counters in the OS kernel. Performance consistency is achieved via deep learning, namely a convolutional neural network that can learn the performance fingerprint of a real process.

The associate editor coordinating the review of this manuscript and approving it for publication was Mohamed Elhoseny.

We use the extracted knowledge to protect its decoy counterpart from malware probes. To the best of our knowledge, we are the first to propose a solution that uses OS-level performance data to create the existence of a decoy process and protect it from adversarial probes.

In the past, we have explored data structure instrumentation to emulate the existence of a decoy process [5]. Nevertheless, we instrumented data structures that were strictly related to processes and threads, consequently we only created the partial existence of a decoy process without any run-time performance dynamics. Saldanha and Mohanta from Juniper Networks proposed HoneyProcs, which is a deception methodology based on decoy processes [6]. HoneyProcs' objective is to detect malware that injects code into other processes. HoneyProcs creates a real decoy process that mimics another real process. Once the decoy process reaches a steady state, it stops making progress with its execution, which leaves its state immutable.

HoneyProcs uses the fixed state of a decoy process as a baseline against any changes, including those made by malware's code injection. Our red team tactics, which we describe in detail later on in this paper, can detect HoneyProcs without making any changes to the fixed state of a decoy process. A simple analysis of real-time performance counters shows that the resource utilization of a decoy process freezes to constant. For example, the size of the working set of a decoy process, which is 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 that represents memory localities. Similarly, the page fault rate of a decoy process swiftly drops to 0, whereas a continuous 100% page hit rate is simply not possible due to demand paging in virtual memory.

Our research on camouflaged user space is slightly larger than the contribution of this paper, therefore a few elements of this research are out of scope for this paper. The deep learning in this work needs to be hidden and protected from malware, otherwise threat actor may manipulate its computations and evade it. One solution is to run the deep learning code 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 alternative is to run the deep learning code on a hardware sideboard that is physically isolated from the host machine.

A honeypot's lack of network activity can be leveraged to detect it remotely. A threat actor could only attack machines that show network activity. Given that no machine in production communicates with a honeypot, a threat actor will never pursue a honeypot. We have described tactics for remote detection of honeypots in [7], therefore we do not treat this subject in this paper.

The remainder of this paper is organized as follows:

- Section III describes the threat model for this work.
- Section IV describes our red team tactics that collect and analyze live performance counters to detect decoy targets.
- Section V fingerprints the performance counter via control flow graph.
- Section VI describes the model that protects a decoy process from malware probes.
- Section VII describes synthetic I/O channel.
- Section VIII presents implementation, testing, and validation of this work.
- Section II discusses research related to various aspects of this work.
- Section X summarizes our findings and concludes the paper.

The contribution of this article can be listed as follows:

- Discussing the key concepts of the existence of decoy processes through operating system techniques.
- Discussing attacks we developed to detect deep inconsistencies in the resource utilization of a decoy process provided by neutral network.
- Identifying input/output data inconsistency in decoy process.
- Proposing a solution to improve the visibility of decoy process.

The OS reference in this work is Microsoft Windows. The foundations of the method have been established in prior work [3], [4], but the main features are summarized here to make this paper self-contained.

II. RELATED WORK

The development of deception strategies in the manufacture of machines is not recent. We discuss various technologies to malware detection and compare them to our proposed solution.

A. BEHAVIOUR ANALYSIS

Intelligent malware can detect if it is in an emulator environment. After detecting that it is in a virtual machine or decoy machine, it can easily halt its activities and exit the system without a trace [8]. Malware can check the behavior of a system, including its running processes and OS level behaviors, to identify suspicious activity in its environment. Chen et al. demonstrated that malware can perform consistency checks and infer if a process is running under a decoy environment because some instructions take different paths to finish in an emulated machine than in a real one [9]. In order to camouflage our decoy environment, we apply machine learning and a Control Flow Graph (CFG) to provide the consistency of a real process to a decoy.

Various detection techniques have been researched in order to counteract anti-honeypot technology. Hayatle et al. studied

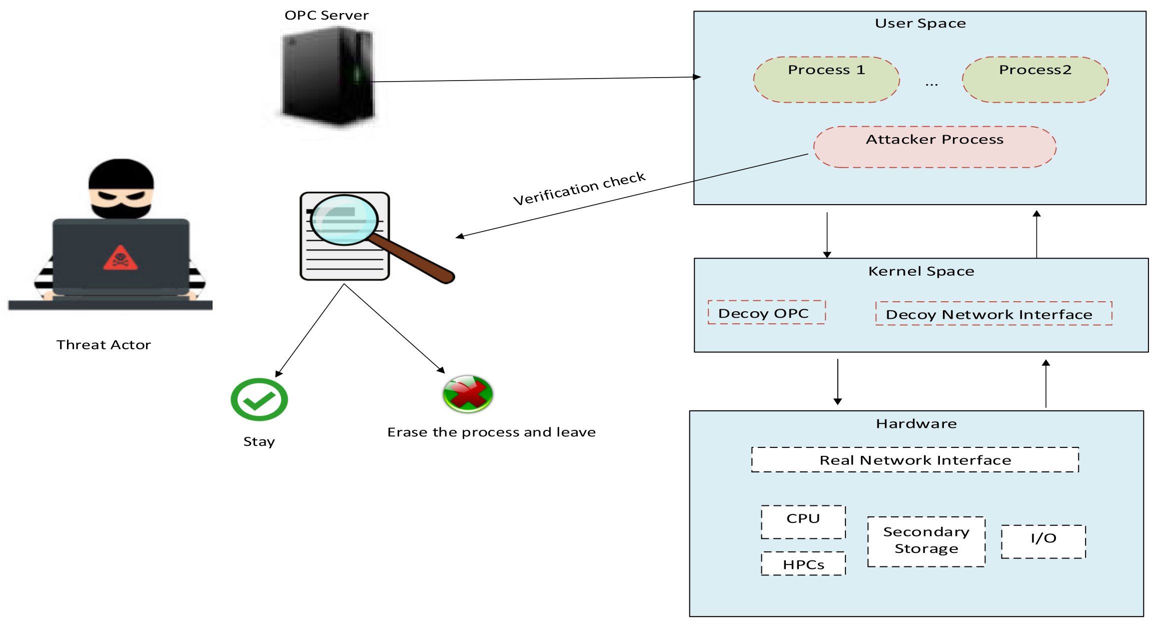


FIGURE 1. **Adversarial probing of decoy processes on a compromised machine.**

the interaction of malware and honeypots through a Markov Decision Process (MDP) model to determine an optimal defense strategy that minimizes the probability of attack [10]. Jiang and Xu proposed a catering honeypot architecture called BAIT-TRAP, which dynamically identifies services to use as "bait" for incoming attackers in order to quickly trap subsequent exploitation [11]. Since both ignore system-level activity, we employ a different technique through decoy processes which represent the activity of a real machine.

Rrushi proposes an anti-malware solution to detect Object Linking and Embedding for Process Control (OPC) malware on machines in production. This is accomplished by configuring a Decoy Network Interface Card (DNIC) on a computer in production, i.e., one being used by a human for real work tasks. The DNIC is a decoy connection to a decoy network. No such network exists in reality, but it appears to be real to malware. Therefore, the malware is detected with no prior knowledge of its behavior or code [5].

Behavior-based detection has been studied recently. Demme et al. uses performance counters to capture micro-architectural features [12], [13]. The authors apply these counters to detect abnormal malware behavior. Singth et al. expanded on this idea by specifically testing kernel rootkits using performance counters [14]. Wang et al. focus on similar detection of kernel rootkits [15]. They employ a custom, code-based solution which collects the number of system-specific call events during the execution of a process. However, threat actors can use these counters to subvert a system to find inconsistencies. We use these features to show the consistency of our decoy process.

Bruschi et al. use control flow graphs to detect self-mutating malware and its malicious code fragments and therefore detect malware [16]. Jaffar et al. also use this type of graph to study paths sensitive to where there are multiple nodes to execute a process [17]. Anomalous run-time behavior detection is another method used by researchers. Nadi et al. use execution logs to form a control flow graph [18]. Our research not only uses CFGs to improve the heatmap training mechanism, but also to evaluate the unexpected black swan values in order to predict resource utilization. Chirkin et al. estimate bounds on workflow processes for all paths in workflow graph [19].

Murphy explores graphical models and relates them into Bayesian networks. By understanding the graphs associated with algorithms, machine learning can be leveraged to produce better results [20]. With the assumption that a hot path has minimal effect on the program state, Ali uses a Bayesian network approach in order to infer whether a program's code path is categorized as hot or cold. Their modified model enumerates static paths for each method to determine the best hot paths to increase performance [21]. Based on these methodologies, we use hot paths to find inconsistencies of a decoy process.

B. HONEYPOTS

Honeypots provide a line of defense against incoming attackers. However, techniques have been researched to show that they also have many vulnerabilities. These techniques commonly aim towards retrieving and analyzing key attributes on the decoy which do not align with those on a real system. In the past, techniques have been used to fingerprint the performance of network links based on Neyman-Pearson decision theory [22] and open proxy relays [23], [24] show that honeypots are vulnerable by focusing on a performance analysis technique over networks. They show that attackers can measure features set to clearly distinguish honeypots and real systems.

Our approach seeks to resolve another common honeypot vulnerability: resource utilization consistency of decoy processes. Instead of focusing on network attributes, we seek to intercept and deceive malware against machines in production based on the consistency of resource utilization of decoy process with its counterpart.

C. DECOYS

Decoy techniques have been an effective defense method to secure our computing systems and detect attacks. The unique characteristic of decoys is the fact that they are deployed within the true system. Decoys have the advantage in that they can expose stealthy attacks effectively and the legitimate users are not required to login to the decoys.

Park et al. proposed software decoys to generate Java source code using code obfuscation techniques [25]. Therefore, the code appears real to an attacker. In [26], Lee et al. proposed a method to create decoy files based on the analysis of the Ransomware. Sun et al. developed a hybrid decoy system to display a decoy over the network [27]. The system separates light weigh front and end decoy proxies from servers.

In our approach, we use the operating system performance counters to display a decoy process to the attacker.

III. BACKGROUND AND THREAT MODEL

The research described in this paper delivers a survivability capability that protects a machine after it has fallen in the hands of threat actors. At this point, malware has penetrated and landed on a machine, and is now probing its environment to determine if all or a part of the machine is a decoy. Thus, probes originate from the inside. These factors are illustrated in Figure 1. We have observed that target validation in malware is commonly a precursor to further attack operations. These operations are implemented as separate malware modules that follow the exploit modules. Malware modules are brought into a compromised machine primarily in two ways.

A single-stage malware dropper incorporates these modules and brings them along. The dropper itself is downloaded over the network from another machine controlled by threat actors. A multi-stage dropper operates slightly differently. A multi-stage dropper is first brought into a compromised machine. Subsequently, the multi-stage dropper downloads these modules into the compromised machine. Figure 1 shows a threat actor in pursuit of a target OPC server. Since the latter may be a decoy, the threat actor avoids contacting it right away. If no processes in the compromised machine are seen to communicate with the OPC server for a long time, then the OPC server is likely a decoy.

A process that accesses the OPC server over the network indirectly becomes a viable validation instrument. If that process is real, then the target OPC server ought to be real. No real processes access decoy OPC servers. By the same token, a decoy process will appear to access a decoy OPC server. Furthermore, a decoy process cannot mask a real process while the latter accesses a real OPC server. Now that a decoy process is at the center of the threat actor's attention, it is important to define what actions the threat actor could and could not take. First and foremost, it is disadvantageous to the threat actor to access the virtual address space of the target process. Although this virtual address space is mapped to inexistent physical addresses, all accesses to it are easily monitored.

As with the decoy OPC server, any memory contact with the decoy process results in immediate detection. Consequently, the threat actor is forced to engage in passive probing techniques against a decoy process. In this paper, we consider probes that use performance counters, however other types of probing techniques are feasible. Performance counters are a set of special-purpose registers, which are built into the performance monitoring units of modern microprocessors to store counts of system activities. Performance counters are commonly used to gather low-level details of events that occur in the hardware during code execution.

The intended purpose of performance counters is to help system administrators with performance tuning and system diagnostics. They have minimal overhead, and their use requires no modifications to the OS or the underlying hardware. Overall, performance counters are a powerful tool for system administrators, but also for threat actors, to gain insight into the behavior and resource utilization of a process.

Figure 1 conveys the key message that engineering decoy processes which can withstand probes is of significance value to the defender. Decoy processes and their performance consistency, along with other types of consistency, are decisive on whether malware falls into a trap, or steps away from a decoy target, erases itself and hence disappears even before the defender sees any clues at all. An ineffective decoy results in none of the malware modules or even the dropper ever being brought onto the machine.

Post-exploit defense is a necessity. The initial exploit may go undetected, such as when it leverages a 0-day vulnerability to land on a target machine. Malware operations that unwittingly involve decoy I/O are the defender's opportunity to detect them.

IV. ADVERSARIAL PROBING

We now describe experiments in which we leveraged live performance counters to detect honeypots, decoy I/O, and later decoy processes.

A. HONEYPOT EXPERIMENT

We did an experiment based on performance counter analysis to stress test decoy covertness. The goal was to practically assess the ability of honeypots and decoy I/O to protect their decoy function. We performed the experiment separately on a Windows honeypot, and then on a Windows machine in production that was running decoy I/O devices. The research testbed consisted of two desktops and a laptop. We connected

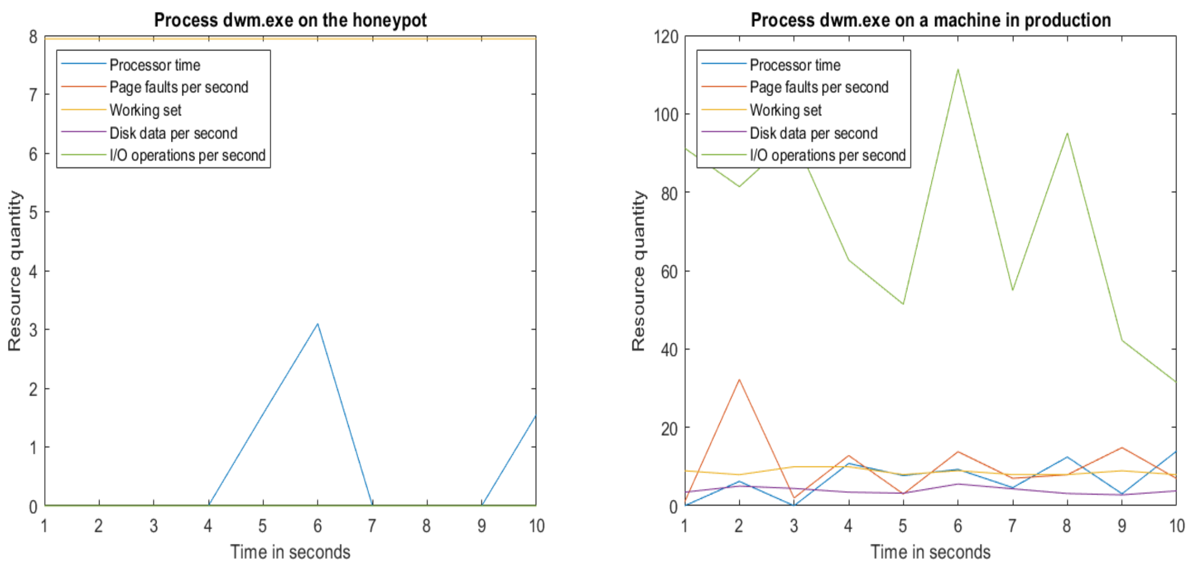


FIGURE 2. Contrast between the performance of a process on a honeypot and the performance of the same process on a machine in production [3].

all these machines to a local area network (LAN) and we logically and physically isolated them from any other networks.

We ran metasploit [28] against the honeypot to exploit a publicly known vulnerability in one of its network services. The exploit injected and ran code that returned a shell on the honeypot, which was then usable to fetch and run decoy probing code. The honeypot mechanisms detected the exploit as soon as its first packet reached the honeypot machine. Nevertheless, neither the exploit, nor the vulnerability, were of any value to the defender because they are both publicly known already. In these circumstances, the defender could benefit by patiently waiting for the testers' next steps, namely operations like those that we referenced in the threat model earlier in this paper.

Advanced malware will not proceed with any further operations on the compromised machine until they have validated the compromised machine is not a honeypot. To this end, given our red team position, we wrote a PowerShell script to collect performance data in real-time. The data that we collected included host processor and memory utilization, and secondary storage activity. The PowerShell script continuously filled a data repository with samples. The data repository enabled us to view a table of the names and process identifiers of all processes currently running on the system. We could also view and store all details and attributes of a specific process of our choice.

We analyzed performance counter data to search for patterns of either low or completely absent resource utilization in [3]. We found that per-process performance analysis is much more accurate in spotting inactivity than machine-wide performance analysis. Let us start with the processor time, which refers to the percentage of elapsed time that the processor spends executing an active thread. Host processor time on the honeypot was somewhat comparable to a machine in production that is idle or in low use. We reached similar conclusions about the amount of time the processor spent executing user space code and kernel space code, respectively.

The only machine-wide activity that we generated in this experiment consisted of our PowerShell script in user space, and the honeypot monitoring tools in kernel space. The resulting interrupt arrival rates and page fault rates were hardly distinguishable from their counterparts on a machine in production that is idle or in low use. It may occur that multiple independent threat actors land on a honeypot. Furthermore, it is common for malware to compete with each-other. In these cases, resource utilization rises. Together with the lack of attribution in machine-wide performance parameters, high resource utilization by multiple malwares makes inactivity detection more challenging. These findings formed the basis for directing deep learning towards the performance profile of specific processes rather than the machine.

After directing our PowerShell script towards specific processes, we obtained performance counters that indicated a total lack of any resource utilization on the honeypot. Occasionally, performance counters revealed existent but low resource utilization, which came from our own operations on the honeypot. We had pre-computed the resource utilization of all operations that we had planned to run on the honeypot, consequently self-generated activity and related jitter was easy to filter out. The processor time of processes on the honeypot stays flat at 0, because those processes do not make progress with their execution.

New pages in memory are not referenced, consequently no page faults occur. Human-machine interaction is absent, consequently 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. The data plot on the left of Figure 2 shows some of the performance parameters of a Desktop Windows Manager (DWM) process on the honeypot. Patterns of absent or low resource utilization are clear. Apart from the working set parameter, all other performance parameters visualized on the data plot are constantly 0. A few processor time spikes occurred; 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 of Figure 2, never changed from a specific constant value. This comes as no surprise, given that with no page faults occurring, the working set could not change. In the data plots of Figure 2, we have applied a log_{10} reduction of the working set to make this parameter fit within the same plot as the other performance parameters. The data plot on the right of Figure 2 shows the same performance parameters of the DWM process, but this time these parameters come from a machine in production. The working set and the disk data per second have both been reduced log_{10}. They are high and variable.

We repeated the honeypot experiment on decoy I/O. This time, we directed the red team probes against a decoy process, which was hidden in a set of 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 [5]. The decoy process in our experiment was an OPC client, which, just like HoneyProcs, maintained a consistent appearance. The decoy process appeared to load the same libraries, had the same size on disk, and created the same number of threads, as its real counterpart. Nevertheless, after we probed the decoy process over performance counters, we could detect it immediately due to obvious patterns of low or absent resource utilization.

We collected and analyzed multiple performance samples to improve the accuracy of our red team approach. Periods of inactivity or low use may be quite common even for a machine in production. By extending data collection over a large time window, we made sure that if production activity existed, we could collect it sooner or later.

B. DECOY PROCESS EXPERIMENT VIA INFLUENCE ANALYSIS

Influence-based probing consists of running code to consume resources on a machine in production, and subsequently studying how the consumption of those resources affects the resource utilization of a target process. This probing technique is illustrated in Figure 4. Threat actors acquire a profile of the compromised machine which consists of detailed hardware specifications along with their speed or capacity. This includes CPU, main memory, secondary storage, OS type and version, and a list of processes and installed software along with their respective configurations. Threat actors send the machine profile to another machine under their control.

The attacker can mimic the compromised machine and create a replica of it once the archived package including the machine profile is received on the attacker's external machine. The replica on the external machine will have the same CPU, memory, secondary storage and running processes. Now that they have all the needed information about the compromised machine, they create their own process on the replica by using an executable file of their own and collect the performance data to characterize the resource utilization, which includes memory usage over time, time spent in user space, time spent in kernel space, etc. After collecting and evaluating the performance counters, this information is sent to the compromised machine under the attacker's control.

At this point, the attacker creates the same process used on the replica and runs the executable file to collect the performance counters on the compromised machine. The threat actors now can compare the data from the compromised machine against that collected via performance counter on the external replica regarding their specific process. The threat actors make the decision to continue the attack based on these comparisons. If they do not see any meaningful discrepancy between the two sets of data they stay and continue their operation; otherwise, they will likely erase all malicious code and disappear before the defender is alerted to their presence.

Should they fail to uncover the decoy, the defender gets an opportunity to detect and analyze the threat actor's actions and methods based on their contact with the decoy. On the other hand, an ineffective decoy results in none of the malware modules ever being brought onto the machine.

V. FINGERPRINTING THE PERFORMANCE COUNTERS OF A REAL PROCESS USING CONTROL FLOW GRAPH

The consistency of a decoy process is obtained by fingerprinting the performance counters of a real process. Although OS behavior can be deterministic and predictable, the resource utilization of a process can always changes depending on its Control Flow Graph (CFG), which is not always a set of specific values. Processes contain executable programs that are connected using their appropriate state transition instructions. Therefore, a process can go through different paths on its control flow graph. Since these paths are not sequential, the same process and input variables on that process can produce different processor times in user and kernel space on each execution. An effective decoy process should adapt this dynamic behavior of execution flow so the threat actors cannot distinguish it from a real process.

When a process is executing it can go through many conditional and unconditional statements in the graph, possibly leading to a state explosion problem. As the length of the graph grows infinitely, it becomes challenging to fingerprint the entire range of performance counter values. Furthermore, we understand that a real process can have a varying set of performance counter values alongside a possible set of black swan values. These black swan values are unexpected, but not unpredictable; therefore, in our approach, we also capture the frequency by which black swan values occur. Otherwise, the

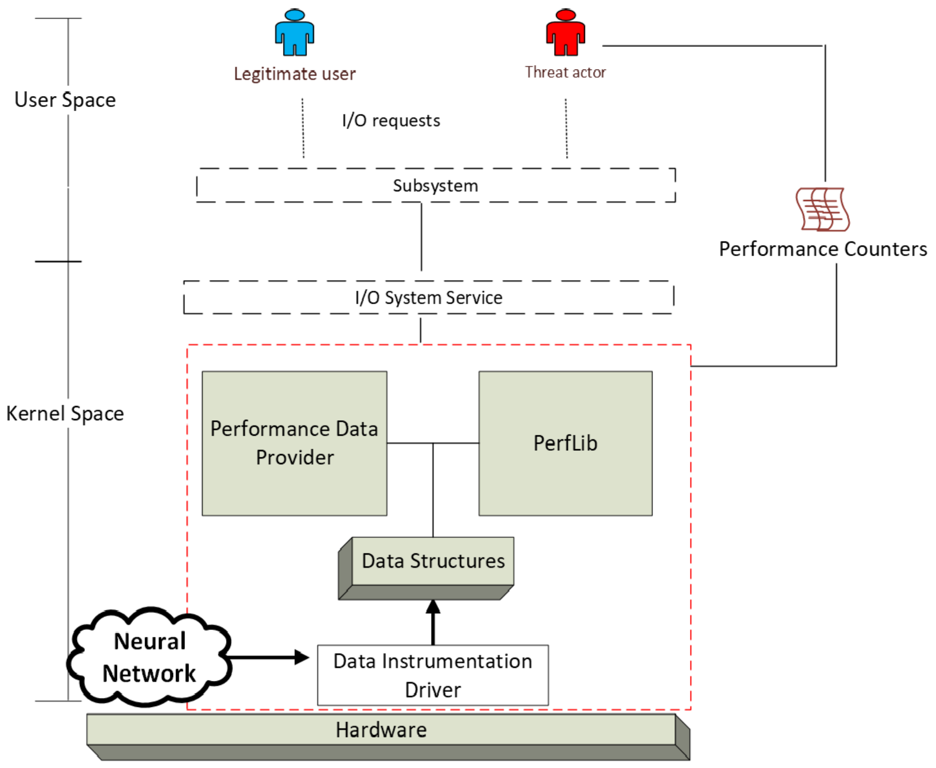


FIGURE 3. **System settings collected by attacker.**

decoy is going to exhibit a stationary performance counter that will never break out of a set of values.

A CFG is made up of blocks of code in a certain program. When the program is executed as a process, the execution flows from one block to another block of code. However, the neural network does not have awareness of the flow of execution outlined in the CFG and so does not learn the value for the path that reaches a block of code less frequently. Therefore, for a similar heatmap the neural network disregards the flow of execution outlined in the CFG and reports the performance counter data with no regard to the current location of the process in its CFG and learns from the more frequent code locality.

Moreover, the neural network has no awareness of a time factor. Below are factors that cause the neural network to report inconsistent performance counters:

1) UNBOUNDED PERFORMANCE COUNTER DATA

The neural network needs to take into the account the relation between performance counter values and time. Therefore, it reports performance counter data without regard to the total resource utilization. For example, a real process may issue I/O write or read operations which can be spread out across the execution time of the process. The neural network does not have the sense to report the exact value that is consistence with the total number of I/O operation performed by the process and will only report when there is a match with the heatmaps. However, the reported values of performance counter values that match with heatmaps are not in the same frequency of the real process. To get the correct consistency, our neural network needs to be capable of reporting the maximum resources which a process is capable of drawing from.

2) UNTIMED PERFORMANCE COUNTERS

The neural network needs to consider the progress of a process' execution over time when reporting performance counter data. The neural network reports performance counter values when there are matches to a heatmap image. However, the time in the execution of a process is not represented in the heatmaps and the neural network reports when there are matches to the heatmaps without considering the time in which the heatmaps were created. For example, if the decoy process reports values $\{\alpha, \beta, \beta, \gamma\}$ over the course of 15 seconds, then the real process may have reached only $\{\alpha, \beta\}$ in the same amount of time.

3) UNCORRELATED PERFORMANCE COUNTERS

This issue arises when the reported performance counter data are out of sequence and are not correlated with each other. The neural network can accurately read heatmap images individually but not when viewed as adjacent or consecutive heatmap reading. The execution flow of a real process over the control flow graph follows sequences which the heatmap images by

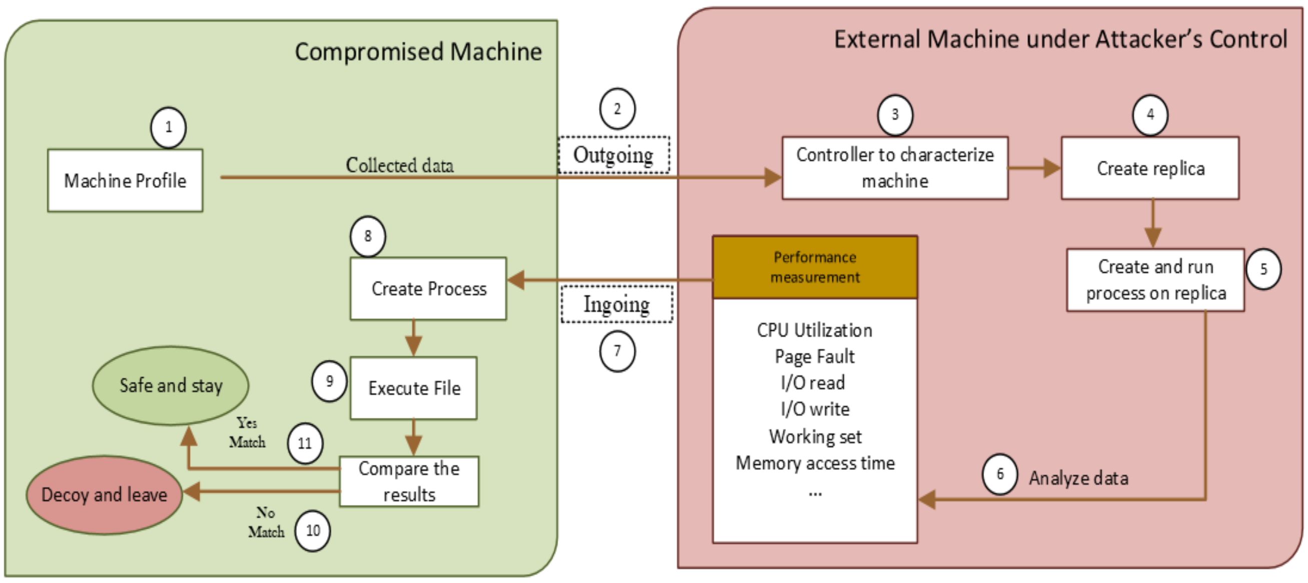


FIGURE 4. Illustration of influence analysis against a decoy process on a machine in production.

themselves cannot recognize. A real process executing over the control flow graph can have a sequence of values $\{\alpha, \beta, \beta, \gamma\}$. However, because the adjacent values could be read too fast or slow the returned values could be only $\{\alpha, \beta, \gamma\}$ or $\{\alpha, \gamma, \beta\}$ in the same amount of time. Some performance counter data would be ignored because the two performance counters are out of sequence with one another.

In our experimentation section VIII, we show how an attacker can use techniques to detect inconsistencies in performance counters of decoy process reported by a naive neural network that does not consider the automaton model.

A. SCHEDULING INCONSISTENCY ATTACK

Scheduling is a main operating system function. CPU scheduling decides which of the processes is ready to execute. On windows and all time-shared modern OSs, where there is more than one process in the ready queue waiting its run to be assigned to the CPU, the operating system decides through the scheduler the order of execution. Therefore, when we have a large population of processes running on the machine that needs CPU cycles. The CPU scheduler will apply an algorithm to schedule the allocation of processor cycles to the hungry process. This can cause a specific process to run slower as it must wait for other processes to be serviced first.

In Windows OS, the scheduling code is implemented in the kernel. Windows implements a priority-driven, preemptive scheduling system [29] where the highest priority process in ready queue always runs. When a process is selected to run, it runs for an amount of time called quantum. A quantum is the amount of time a process is allowed to run before another process in the ready queue can run. Already processes are kept in a queue. The CPU scheduler picks the highest priority process from the queue, sets a time to interrupt after the time interval of quantum, and dispatches the process. Because Windows implements a preemptive scheduler, a process might not complete its quantum. Therefore, the currently running process might be preempted before finishing its time cycle. Consequently, the CPU takes the next process to run. If the process finishes before the end of the quantum, the process itself releases the CPU.

We detected an inconsistent scheduling issue by loading a large population of processes admitted to the system demanding CPU cycles. In our CFG, we apply a model with the time transition criteria at which each transition from one state to another state can occur only at certain times. However, if we do not consider the load of processes and the number of CPU hungry processes, we can display a progress of decoy process over time that is inconsistent with the current load process. Therefore, a threat actor can simply perform a load analysis to indicate that the decoy process is running too fast or slow.

We found the root cause is the inconsistency of the load of processes requesting CPU cycles. This situation is graphically represented in Figure 5. The two data plots represent the I/O operations per second for both the decoy process and its real counterpart. Roughly, in the first third of the graph, the two data plots run simultaneously. This indicates that the decoy is accurately imitating the real process' rate of I/O operations. Then, at a time between 10 and 15, the data plots diverge; the synthetic process begins its I/O operations a few seconds before the real process. At this time, a batch of other processes began execution and compete for CPU cycles, causing the real process to have a delay in its own execution as it waits to be scheduled. The decoy does not account for this delay and continues, and thus a noticeable discrepancy occurs between itself and the real process.

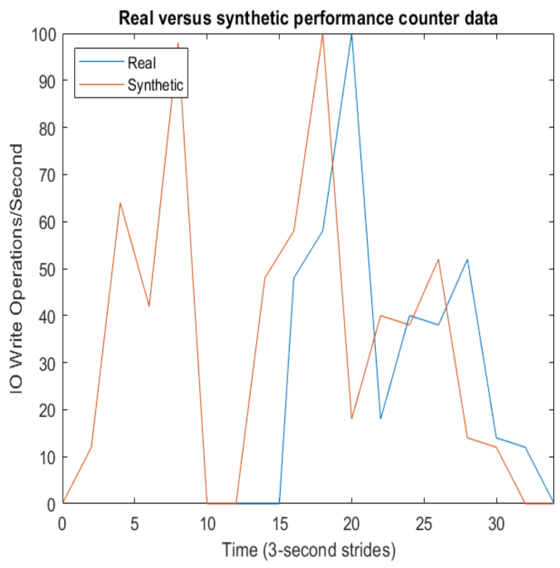


FIGURE 5. **Architecture of CPU Scheduling model [4].**

B. SOLUTION TO THE SCHEDULING INCONSISTENCY

To provide a solution to the scheduling inconsistency problem, we first create a pre-computed hash table in which to store the various possible states in the control flow graph in which the performance counter of our decoy process may be in its execution. A hash table is a data structure that can be used inside an OS kernel. This information is provided to a threat actor who is asking for performance counter data pertaining to the decoy process, which must align with a real process operating in a system hosting many other processes that affect each other's performance. In other words, the hash table stores an indication of the cluster in the cluster graph, as well as the specific state in the inner automaton, where the decoy process currently resides. This hash table is created and populated ahead of time before any querying for performance counter data has been done. This is done by running CPU scheduler simulations under various conditions and is described in detail below. Once this is finished, the table is ready to be queried when needed using a three-part key.

The pre-computed hash table is, as with any hash table, queried with a key. In our approach as shown in the Figure 6, we use a key that consists of three components: a path identifier, arrival time, and simulation identifier. The path identifier is simply a number that corresponds to a specific path on the control flow graph at which the number of paths is given by automaton. The arrival time refers to the time in which a threat actor asks for performance counter data of the decoy process. This time begins counting when the decoy process' execution is begun. Finally, a simulation identifier is assigned to each simulated CPU scheduler scenario that is performed and its state recorded. When the hash table is queried with this three-part key, it returns the relevant position on the control flow graph where the decoy process would be located given the time in which it has been executing and the load of other processes slowing execution.

1) SIMULATION-IDENTIFIER

Several CPU scheduling simulators have been previously developed for CPU scheduling algorithm evaluation. We use CPU scheduling simulators from [30] to simulate varies circumstances. The simulator enables us to study the behavior of the windows CPU scheduling when running different processes and simulate the normal work for the system. We assign a time quantum of one for CPU scheduling simulator to execute each process. In our experiment, we found that the code inside each cluster is normally executed entirely with one time quantum which is enough to make the decoy process indistinguishable from its counterpart. A simulation-identifier is assigned for each simulation that pertains to specific simulation situation which represents the number of processes used during the simulation and what processes they are. We also emulate the CPU scheduling by running real processes which for our approach will not matter. What we get at the end is a quantification of how other processes compete with the decoy process for CPU.

When a threat actor makes a request for the performance counter of decoy process. At the time of arrival, our approach identifies the simulation that complies with the current situation of the load of processes. We use the OS kernel data which is the real CPU scheduler trace file to find the current situation of running processes on the system. For example, reading the trace file can show how many processes the decoy process is competing with and since we use pre-simulated CPU scheduling and real CPU scheduler trace file, our approach can identify what simulation identifier should assign to.

2) REMOVING THE ROOT CAUSE OF CPU SCHEDULING INCONSISTENCY

We now discuss the effect of our improved heatmap training mechanism on the failure exploited by the CPU scheduler. Based on the time of requests for performance counters for a decoy process, our models now consider the large population of processes and the CPU scheduler at which the time that takes for the CPU to enter the node or the cluster in inner automaton.

VI. BUILDING MODEL

This section discusses how our approach learns the performance fingerprint of a real process to carry out performance recognition tasks in support of the process' decoy counterpart. We express details of our approach through the lens of deep learning. The reader is referred to [31] for a detailed discussion of deep learning.

A. DECOY PROCESSES

We now describe the method for creating dynamic display for decoy processes. The objective of this work is to project a decoy process as a new process in a way that the process appears to the attacker to be fully functional and have the

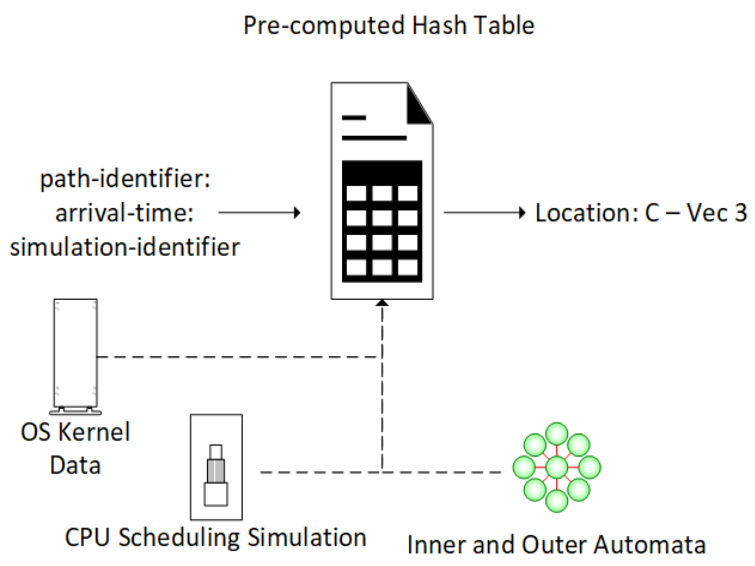


FIGURE 6. Architecture of CPU Scheduling model.

same performance values as its real counterparts. However, it does not exist, so our process does not consume real resources on the machine. The creation of our process is different than spawning. When spawning a real process, there is an overhead involved for consuming resources, data, code, and libraries to store the decoy process in the main memory. Spawning a process requires a freeze of execution so the CPU scheduler does not run the process. However, as we discussed earlier in this paper, freezing execution to create a fixed state is not effective because it still consumes CPU cycles and secondary storage.

A dilemma exists in distinguishing our process from any malicious processes. Malicious actors hide their process through manipulating the normal execution of system calls. They modify the system calls and process manager in such a way that information about the malicious processes will not appear in the returned list and therefore the process manager bypasses it. However, there are known techniques to show the existence of process that has been obscured in this way, such as the task manager tool, the tasklist command and the PS command.

In our previous research [3], we wrote data structure instrumentation code that deposits synthetic data in the repository of performance counters. Figure 3 shows how a decoy process is visible to malware in user space. The figure presents the performance counters which track the counter sets that are provided by OS Kernel components. They are in the drivers of the OS kernel which operate as a Performance Counter Library (PERFLIB). We can acquire the performance data by querying this library. The performance data is linked in data structures, including linked lists. For our decoy process, we have written data structure instrumentation code to project the existence of a decoy process. We then inserted synthetic performance data in the repository of performance counters. The synthetic and real performance counters are provided to consumers in user space, including any potential malicious actors or programs. This way, a decoy process has a dynamic synthetic resource utilization and appears to the malicious intruder to have utilization values in line with the decoy's real counterparts. However, the synthetic performance data values need to be consistent with those given by the system's performance counters, which we address later in this paper.

The OS kernel counts performance data at specific time windows when events occur so the performance data in user space is not updated at the same time as in kernel space and there is a delay until the counting is completed. For instance, a counter for page faults increments anytime a page is referenced that is not in the physical main memory. The counter is buffered until the counting completes and is not stored immediately in the repository of performance counters. Therefore, the consumers of performance counters in the user space do not see fresh counter data until after the counting is done. As a result, it is important that the data instrumentation driver in Figure 3 does not store the synthetic performance data generated by the neural network too fast or too slow in the repository of performance counters.

We use a fully trained neural network to generate synthetic performance counter values so the process of producing performance counters is rapid. For the neural network to perform performance recognition, it needs to learn from the performance counters of real processes. The training will help the neural network produce synthetic performance data. The synthetic performance data is stored in the repository where the data instrumentation driver can access it directly. As the neural network delivers the synthetic performance data to the data instrumentation driver, the data will be buffered until the counting period is done.

We need to mention that it is safe for honeypots to have synthetic performance data for a decoy process that does not exist. Honeypots lack the human factor. Since there is no human interaction on a honeypot, the risk of a user interacting with a decoy process is null. However, the risk of a user in production equipped with decoy I/O is considerable. We rely on a safety measure from previous work [5], 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. Since we know the name and the performance data of the decoy process, we can have them filtered out from the user's visual interface.

B. DESIGN OF PERFORMANCE PROFILE

In this section, we present the approach we use to learn the performance counters of real processes. We use deep learning to learn the performance counters of real processes and to be able to predict the performance counter of a decoy process. The reader is referred to [31] for a detailed discussion of deep learning. The reasoning of this work is specific to the OPC client process. However, it is generally applicable to all processes. We selected OPC client for the deep learning approach because of its integration with a decoy I/O capability that we have developed in previous work [5].

We use the performance counter of real processes in production because the environment is more complex compared to the honeypot environment. Honeypots are decoy computer resources so we can configure most or all processes to be decoy processes in the honeypot and choose their performance counters ourselves. Therefore, it is easier to calculate their resource utilization. However, we cannot control the performance of real processes in production, especially while malware is probing for performance inconsistency. Therefore, we tailor our work for a production environment to ensure it will work with any possible values.

C. AUTOMATON MODELS OVER TRANSFORMED CONTROL FLOW GRAPHS

In this section, we present the automaton approach we used to make our machine learning approach more effective. We divide the computation executed by a real process into individual tasks and assign an operation code (opcode) to each of them. Our machine learning approach is then applied to these tasks. That is, the neural network learns and recognizes one task at the time within the larger functionality delivered by a process.

1) GRAPH CLUSTER

We build a CFG of a real process and apply clustering of the vertices, which represent blocks of code in the program. Graph clustering is the task of grouping the blocks of code of the CFG into clusters such that there should be many edges between blocks assigned to a particular cluster, and relatively few between clusters. Blocks in each cluster have similar functionality or is connected in some predefined sense. In the CFG, the vertices are ordered in a complex way; there is no clear structure in the adjacency matrix. Therefore, it is challenging to interpret the number or quality of clusters inherent in the graph. Thus, we applied techniques from [32], the Schaeffer algorithm [33], and graph partitioning [34]. The result of this clustering work is what we call a cluster graph.

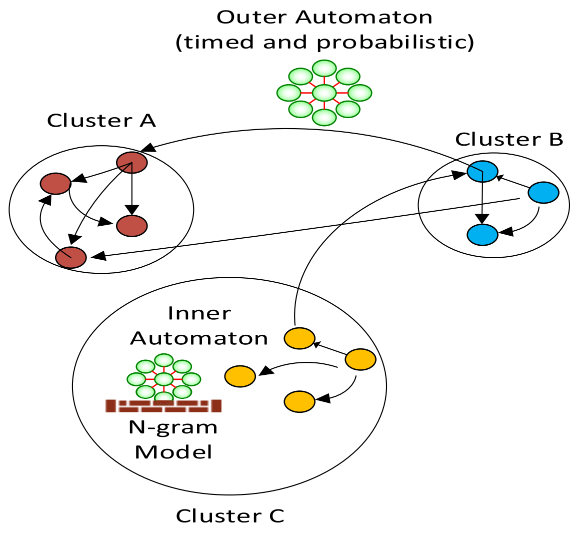


FIGURE 7. Architecture of automaton models over a transformed control flow graph.

As shown in Figure 7, each vertex of the cluster graph consists of a grouping of blocks of code that the processor transitioned between frequently. In other words, those vertices of the control flow graph that are more densely connected via edges with each other than they are with other vertices are clustered together. These clusters become the vertices of the cluster graph. There is a directed edge from a node in a cluster B to a node in another cluster.

2) BUILDING INNER AUTOMATON

We use an automaton model for our machine learning approach. We apply an outer automaton on top of the control flow graph, and an inner automaton at each vertex of the control flow graph. These are timed probabilistic finite state automata with a few additions. They contain a time module that limits when a transition may be taken. A clock increases as the automaton is fed input. When a transition is successfully taken, the clock may or may not be reset to zero and begin its count anew. Furthermore, a probability is assigned to each transition that defines how likely that transition is to be taken.

We use the k-means clustering method [35] to cluster vectors of performance counter samples within each vertex of the cluster graph. Clusters of vectors of performance counter samples from the states of the inner automaton for their corresponding vertex in the cluster graph. These clusters are the states of the inner automaton. As a processor runs code within a vertex of the cluster graph, the type and amount of resources consumed by the process evolve. Accordingly, in our inner automaton model, the process transitions from one state, i.e., cluster of vectors of performance counter samples, to another.

We observe the times at which a state transition occurs relative to the start time of the task at hand. We subsequently turn those observations into a time condition for that specific transition in the inner automaton. We study the execution of real process based on the recorded time. As the processor runs code within a cluster, we collect performance counter samples every second to determine the time the transition from one cluster of vectors of performance counter data to another occurred. Furthermore, the input of each transition from one cluster of vectors of performance counter data to another is defined as the major difference between these two clusters of vectors of performance counter samples that are connected by the transition.

Given that, by definition, the transitions between states in the inner automaton are probabilistic, we developed *n*-gram models [36] to estimate their probabilities as well as possible sequences of transitions. For each sequence of *n* previous states of the inner automaton, our *n*-gram model predicts the next possible states along with their respective probabilities of occurrence. As a result, for each inner automaton, we now have states, transitions between those states, input, time conditions, and a probability of occurrence for each state transition.

3) BUILDING OUTER AUTOMATON

The states of the outer automaton are defined as the vertices of the cluster graph. Each cluster holds blocks of code that branch into each other regularly, and more rarely branch into a block that is in another separate cluster. These inter-cluster branches form transitions between states in the outer automaton. The input of a transition in the outer automaton is the cluster of vectors of performance counter samples that the process is going into upon completing the transition. For example, if we have outer automaton transition from C to B, then the input associated with that transition is the cluster of vectors of performance counter data that is observed the moment the processor starts running block of code in cluster B.

To build the outer automaton, we record the frequency of each state transition as the real process is running. We use our *n*-gram model to estimate a probability of occurrence for each transition in the inner automaton. As in the case of the inner automaton, we are interested in the time conditions needed to transition from one state to another in the outer automaton. We run the real process and record the times at which state transitions in the outer automaton occur. Based on those time observations, we define conditions on the points in time at which a transition is allowed to occur.

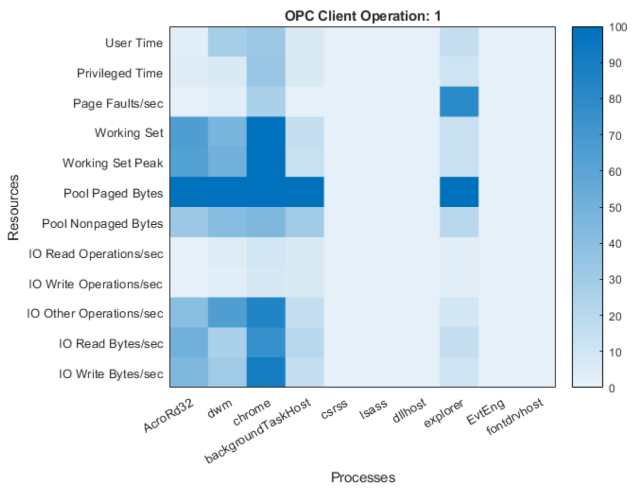


FIGURE 8. A performance heatmap for neural network consumption [3].

TABLE 1. List of performance counters visually assembled in heatmaps.

Counter name	Description
user Time	CPU used by user mode
privileged Time	CPU used by kernel mode
page Faults	memory page fault
working Set	process working set
working set Peak	maximum size of page in virtual address
pool Paged Bytes	memory used when page faults are allowed.
pool Nonpaged Bytes	memory used when page faults are not allowed.
IO Read Operations/sec	IO read bytes
IO Write Operations/sec	IO write bytes
IO Other Operations/sec	process IO device activities
IO Read Bytes/sec	IO read bytes per second
IO Write Bytes/sec	IO write bytes per second

D. HEATMAP MODEL

We use heatmaps to visualize the machine's resource utilization. Therefore, color coding is used to represent performance parameters. Figure 8 shows an example of the performance counter heatmap display. The *x*-axis corresponds to the real processes whereas the *y*-axis corresponds to the performance parameters. The coloring scale coding indicates high to low counter values. The higher a performance parameter, the stronger its color in the heatmap.

Depending on the selected hardware counters, each heatmap cell visually represents the value of the performance counter for a specific real process. Table 1 lists some of the performance parameters that we used in this work. We select some parameters of the whole resource utilization spectrum for training proposes in the hope that our approach can learn the performance fingerprint of a process.

In our neural network training, many heatmaps are given. The neural network reads the heatmap and produces output of a class label. Each heatmap is labeled as a class label where each label is an array of color strengths for each performance parameter of the decoy process. Based on the class label, our

TABLE 2. **List of operations of an OPC client used for generating heatmaps.**

OPC operation level	OPC operation description
server level	- view OPC server properties - add OPC group to server - add subscription - add an alarm
group level	- add item to an existing group - delete item from group - view group properties - change group properties - create a new group - delete an existing group
tag level	- list tags of an OPC object - add a tag to an OPC object - edit a tag to be added to a group - delete a tag from a group - clear all the tags - read a tag - write a tag

approach can specify values to the performance counters of the decoy process.

Values that the decoy process reveals to the malicious users depends directly on the resource utilization of real processes on the machine. When malware probe our machine, we take a screenshot of the performance counters of all processes in the machine, including real processes and processes created by malware, and turn them into a heatmap for recognition by the neural network. Figure 8 shows a performance heatmap of all processes running on the machine.

We refer to $foreign\_process_n$ for any resource utilization of processes created by malware in heatmap, regardless of what they are named by the threat actor. That way, we distinguish them from standard internal names to prevent the neural network from getting confused. Furthermore, our approach is aware of the current resource utilization load on the machine. When a user opens several chrome tabs, each tab creates a separate process. We include the performance of all such processes in the heatmaps.

E. DEEP LEARNING METHODOLOGY

1) LABELING PERFORMANCE COUNTERS

We first need a data set to train the neural network. For that purpose, we collect performance counters of all processes running on the machine, including an OPCExplorer process. We only collect performance counters of the OPCExplore process for the operations of Table 2 one at a time.

The next task is labeling. We use the collected performance counters from the OPC process and all other machine processes to build heatmaps and establish a class label. We save these labels for later use.

2) MODEL TRAINING

We also need the labels generated during the heatmap creation to train the neural network. Therefore, we repeat the same steps and use the labels for training the neural network. The training of the convolutional neural network is given in Algorithm 1.

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 units (ReLU), pooling layers, a fully connected layer, and a softmax function. The hidden layers are used to adjust and scale the activation of given features from the heatmap images. Thus, the number of layers is critical. We use a standard softmax function:

$$Y : R^C \rightarrow R^C \tag{1}$$

which is defined by the following formula for $k = 1 \ldots, C$ and $\phi = \phi_1, \ldots, \phi_c$:

$$(y)_k = \frac{\exp(\phi_k)}{\sum_j^c \exp(\phi_j)}$$

We start with a common input layer to load and initialize the heatmap from the training set for further processing. Then we add several ReLU layers to achieve accuracy. ReLU layers allow faster and more effective training of the performance learning. The layers zero out negative values and maintain positive values in a heatmap undergoing processing. We also add pooling layers to simplify the output by reducing the number of heatmap image parameters that the network needs to learn about.

We then add a fully connected layer to produce a vector with size equal to the number of class labels. This layer looks at the output of the previous layer and determines which features most correlate to a particular class. Each element of this vector is the probability for a class label of the heatmap image that the neural network just processed. Some of these probabilities may be negative. Also, adding the probabilities may not be *1.0*. Softmax corrects this idea and assigns decimal probabilities to each class in a multi-class problem and the sum of those decimal probabilities must be *1.0*. Thus, it normalizes the vector in question into a probability distribution. The classification layer allocates the accurate class label to a heatmap image that the neural network just processed, based on that probability distribution. We implemented the softmax layer is implemented just before the output layer.

After training, we calculate the accuracy of the heatmap model by running the neural network to classify heatmaps from the data set. Then we compare the results with the known class labels from heatmaps. Our goal is to have high accuracy. So, if the level of accuracy is low, we adjust the accuracy of the network by adding more layers and retrain from scratch. We try altering the neural network design until we get a much better accuracy result.

3) ACTION OF DECOY DRIVER

In Algorithm 2, we show how the performance counters generated by neural network are reported to malware in response to their probes. At this point, we have a fully trained neural network with high accuracy. When malware sends

Algorithm 1 Heatmap Labelling and Neural Network Learning Mode

```
Function Learn-Performance-Fingerprint ($G$, $V$);
Input  : Set of Performance Counters $G$, set of
         transactions over CFG $V$.
Output: Convolutional neural network $\Pi$, heatmap
         recognition accuracy $\delta$.
Build a Cluster graph $\beta$;
$\delta \leftarrow 0$;
for $\forall$ task $\eta$ executing over nodes in CFG do
    $t$ = $execution_time$ ($\eta$);
    while $t$ != 0 do
        build inner and outer automaton;
        label the heatmap $\nu$ with location $\tau$;
        Read $\nu$ into array $\alpha$ in memory;
        Add Label($\nu$) to $\alpha$;
while $\delta < 90$ do
    Empty $\Pi$ if any layers present;
    Define the input layer of $\Pi$;
    Add $count_1$ ReLU layers to $\Pi$;
    Add $count_2$ pooling layers to $\Pi$;
    Add $count_3$ batch normalization layers to $\Pi$;
    Add a fully connected layer to $\Pi$;
    Add a softmax layer to $\Pi$;
    Add a classification layer to $\Pi$;
    Select $\Pi$'s training options;
    trainNetwork($\Pi$);
    for $\forall$ heatmap $\epsilon \in V$ do
        $\delta \leftarrow \Pi(\epsilon)$
    Increase $count_1$, $count_2$, and $count_3$;
```

a consumer query, the request is sent to the decoy driver. Then we take a screenshot of the performance counter of a real process and build a heatmap for recognition by the neural network to produce class labels. Based on the labels, the synthetic performance counters are produced. The synthetic performance data are buffered in the repository until the counting period is done. The decoy data instrumentation driver accesses the repository to insert performance data into the data structure. We query the data structure instrumentation code which contains a heatmap that is representative of the resource utilization of all processes on the machine, 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. This data is reported to the malware in the form of performance counters in response to their probes.

VII. SYNTHETIC I/O CHANNEL

In this section, we outline the I/O data that can be captured by a threat actor to detect inconsistency in decoy process. To solve the issue, we define some sets of input data and

Algorithm 2 Algorithm to Describe the Action of Decoy Driver

```
Function Display-Synthetic-Performance ($Q$, $P$);
Input  : Consumer query $Q$, Performance Counter $P$.
Output: Performance counters generated based on
         labels $\omega$, decoy driver $\beta$.
$\omega \leftarrow 0$, $\beta \leftarrow 0$
$CNN$ neural network
$\iota_1, \iota_2, .., \iota_i$ labels of heatmap
for $\forall$ Consumer query $\alpha \in Q$ do
    Read $\alpha$ into array in memory;
    If $P$ =! empty then build heatmap $\gamma$;
    Call $CNN$ and give the heatmap $\gamma$ to it;
    Read labels $\iota_i$ generated from neural network;
    Adjust performance counter $\rho \in P$ based on
    classification label produced by the neural network;
    while buffering of performance counters is not done
    do
        $\omega+ \leftarrow$ synthetic performance data generated
        based on labels;
    $\beta \leftarrow \omega$;
    Insert $\beta$ into Data Structure
```

create synthetic I/O channel. This channel will provide the same functionality as a real I/O channel.

A. IMPACT OF I/O DATA ON DECOY PROCESS CONSISTENCY

A real process has I/O read and write activities where it can read data and produce output. These I/O operations come from files, keyboards, monitors, hard disks, and network interface cards. Therefore, even though the performance counters give details about I/O activities, threat actors can monitor the process interaction with I/O devices. They can observe all I/O traffic and notice the total inactivity of I/O traffic to and from a target process. Therefore, the consistency of resource utilization of decoy processes should also align with I/O traffic to and from I/O devices to support the effectiveness of decoy process in redirecting malware.

We applied the automaton model that is comprised of a cluster graph, outer and inner automaton being probabilistic and timed to improve the resource utilization consistency of our decoy process. Therefore, we model the execution of real process which is tied to specific input in a way that any changes to the input data will require different model with different probability of transition. However, the model may not change for very small input changes. For example, with given history, for specific input the probability of transition from cluster C to cluster B is *0.5*. However, if the input changes drastically, the processor may not even visit nodes in cluster C because of the instruction branches the input will take while the process is executing over the CFG. Therefore, changing the input data requires modifying the model with different probability of transitions.

Some input data are characterized by similar probabilities and times of transition whereas others are not. For example, we have the task of adding a tag in Table 1 for OPCExplorer client where we are required to name the tag. Let us say for adding a tag to OPCExplorer objects A and B, the change of the length of tag name for OPCExplorer objects A and B may not necessarily require a different model in terms of transitions. However, our model may no longer hold true for naming the tag with the same character size for an OPCExplorer object C. The moment we identify an input data that is no longer advised by the current parameters of our model, then the probability of transitions should be recalculated using the same technique.

We may not have a model per set of input data, but we could have the same model for possibly different probabilities, and times of automaton state transition per set of input data. We perform an input analysis that ties the input data to our model. If our model no longer holds true, then we build a new model in the sense that the CFG and the cluster graph are the same but the probability of transitions changes. However, the probability and time changes in our model do not affect the neural network and heatmap model. When we build a heatmap with location labeled to it and feed it to the neural network, based on the heatmap, it reports an accurate classification with the range of possible values for each performance counter of the decoy process to malware.

There are two main parameters that are affected by the revised model: Firstly, probability of transitions from one state to another are different in the outer automaton as well as the inner automaton. Secondly, the time to transition from one state to another is also different. Therefore, these two parameters must be adjusted per input. An attacker can detect inconsistency in how the process transitions over the model and generates input over the synthetic I/O channels. Therefore, that input will never cause the program to be in the given cluster by the model.

B. DEFINING INPUT DATA SET

We now describe how we define sets of input data $\{X, Y\}$ such that they are adjustable to the main parameters of probability of transitions and time from one state to another state. We can get different model based on the definition of input data sets which are same model, but their time and probability of transitions are different.

We decomposed the functionality of OPCExplorer process into tasks. As a result, the decomposition of the process functionality into tasks defines the input data $\{X, Y\}$ where X is comprised of input data values that are characterized by the same model with the same parameters of time and probability of transitions. Y contains data values that do not comply with the current parameters and need to adjust the time and probability. To clarify, the task of viewing OPC server properties from Table 1 will have different code path and hence, different probabilities and times of transitions in outer and inner automaton compared to the task of removing a tag from a group or reading a tag.

However, the variation between each task in terms of times and probabilities of transitions are tolerated by our model. Some state transitions at specific times can happen more frequent than other state transitions. We capture every state transition with their probability of transitions and times so our approach will not be intimated by variation.

C. SYNTHETIC I/O CHANNEL INTEGRATION

We use the idea of emulation and mimicry of I/O devices from [1] to create synthetic I/O channels. We use decoy I/O devices to feed synthetic data into decoy process. These I/O channels need to be emulated under a decoy process so the data can communicate between the decoy process and decoy I/O devices. Furthermore, the decoy process needs to be consistent with the resource utilization of a process. The timing factor is a main factor in the support of the consistency of I/O activities. As a real process performs I/O operations, there is a timing factor associated at which time the operation was performed in connection with the I/O device. The overhead of the creation of synthetic I/O channels is a small fraction compared to the creation of real I/O devices and processes.

Decoy I/O devices for OPCExplorer process consist of decoy keyboard, monitor, mouse, and power system which are the source of the synthetic input data. For I/O device and network interfaces to communicate, windows supplies a stack of device drivers for each. The stack is responsible for handling I/O data that exists in a class driver. These drivers perform generic functions that are necessary to process the input data request, irrespective of the specific device hardware manufacturer. The drivers that reside close to the hardware perform processes that are specific to the device hardware. Each write or read operation is handled by a Controller Driver which interfaces with the lowest level driver in the stack as depicted in Figure 9.

Each of these drivers has some degree of information regarding the I/O device. Similarly, the Network interface card (NIC) has its own stack of drivers that process the input/output data through the network interface. In short, data that is generated by a process traverses the stack of network interface drivers and is sent over the network. The reverse is also true. The NIC receives incoming packets and sends them to be processed by the stack before being delivered to the associated process. The process reads this data in the same fashion as if it were reading from a file stream. The I/O device appears as a text file to the process.

The I/O director has a section called the plug-and-play manager, which is responsible for keeping track of what different I/O devices are on the computer. The plug-and-play manager provide updates on I/O devices installed and removed from bus drivers. A real keyboard or mouse is usually connected to a USB hub, it is the bus driver that controls the USB hub to report the installation or removal of a keyboard or mouse to the plug-and-play operator [37]. A driver from the stack known as the Filter driver receives traffic from the keyboard or mouse and processes it as normal.

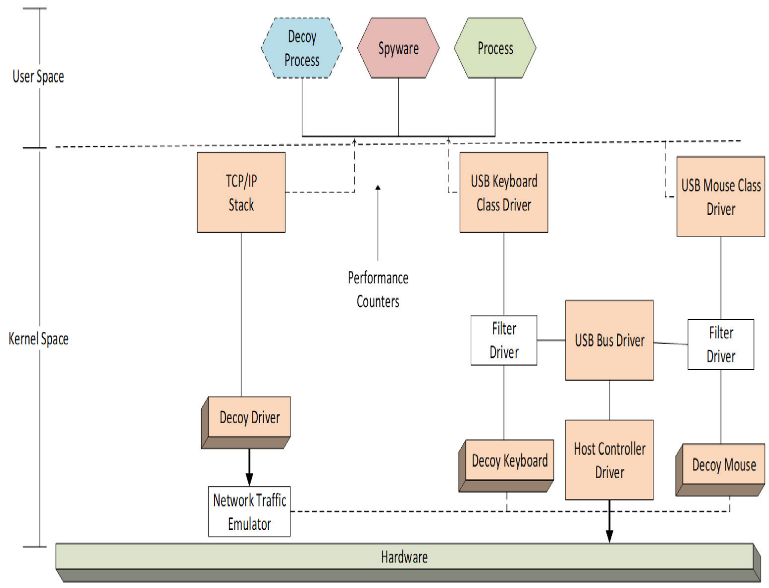


FIGURE 9. Synthetic I/O channel and its integration with decoy processes in the OS kernel.

During times at which there is no activity coming from the I/O devices for the process, the filter driver assumes that the process is not in use and is programmed to send a signal to the decoy I/O devices that it is to begin its processes.

The decoy channel produces prerecorded synthetic I/O traffic. This traffic is always identical to authentic process data but has no meaningful content. The decoy process as shown in the figure is comprised of a minimal amount of code that reads traffic from the decoy I/O devices. The decoy I/O device exposes synthetic traffic through the user space (where malware operates) and delivers this decoy data to the decoy process. The decoy process will then appear to send the data over the network but, the synthetic data will traverse the user space the same way the real I/O traffic does. It is eventually caught at the bottom of the stack where the network emulator exists.

In the stack of drivers that receive the data exists a shadowed driver. This driver shadows the network interface and is connected to the network interface system filter. The filter driver is programmed to sense when there is a lull in network traffic. When this occurs, it redirects any incoming traffic away from the network interface card and toward the decoy driver [7].

VIII. TESTING AND VALIDATION

A. EXPERIMENTAL OBJECTIVE

Can simple performance parameters be used to redirect malware? To answer this question, we make twofold experimentation. The experiments were performed many times. First, we collect live performance counter data and produce heatmap labels to use for training the neural network. Secondly, as a red team, we develop attacks that can detect inconsistencies in the resource utilization of a decoy process as provided by the neural network with a heatmap training mechanism. Those attacks show that the neural network in charge of protecting the resource utilization consistency of a decoy process is insufficiently trained. As a result, we develop machine learning over control flow graphs to improve the heatmap training mechanism of the neural network.

B. IMPLEMENTATION OF EVASION STRATEGY

A typical malware delivers exploits to the victim system. The exploit can be sent to a victim from any model described in the threat model section. To maintain information about the system, they download the payload in memory. The payload is the piece of code that the adversary executes on the target to collect data from a compromised machine to build a replica and perform a malicious task.

As mentioned in the evasion strategy, attackers can create their own process and evaluate the results to see if the system is a decoy. For that purpose, they need to get a list of all running processes with details on the machine before their evaluation, including the system configuration.

We now describe how attackers build a replica on Windows. In this work, we performed our experiment on a Windows machine in production. To collect the necessary information for building a replica, we wrote a PowerShell script on the compromised machine with a function to list all the running processes on Intel i7 multi-core processor model. The process monitoring can be set by the operator to take any specified number of data samples at an interval also set by the operator. The result of the process monitoring function was a table of process names, identifiers, CPU, secondary storage, and memory usage of each process running on the system at the time function is called.

We define running processes as any programs executed before the function was called and that are running or waiting at the instant the function is called. Any process terminated before or begun after the function is called within the script will not be captured by the monitor as these processes do not exist and are not consuming any resources. For example, during data collection, there was a chrome application running with multiple tabs and the script was able to collect the identifier, CPU, and secondary storage usage for each open tab of the chrome.

The script also examined process details so we can filter the measurements based on specific processes running on the machine. The script gave the details of any operator specified process running on the machine. For instance, we analyzed the process created by the Chrome web browser, the output contained the details for each tab including the identifier, total processor time consumed, working set, and user processor time.

Furthermore, we need the details of the system's operating system and hardware. The script provided us the system information including OS name and type, OS total memory size, paging files, etc. We saved each output list in files. All the directories to save the files are created by the script at runtime.

C. HEATMAP AND NEURAL NETWORK IMPLEMENTATION

We wrote two MATLAB scripts for the heatmap generation and deep learning approach. We extended the PowerShell script that we used in the honeypot experiment to collect live performance counters from all processes running on the machine and store them in files.

We wrote a script to automatically collect the performance counter values of all processes and create a heatmap of the counter values for all processes excluding OPCExplorer. This is accomplished by directing the log file output of our PowerShell script into a MATLAB script we have designed which generates heatmaps from our data and saves them as image files to be utilized later for training the neural network. 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.

For the neural network training purposes, we need to assign a label to each performance parameter of OPCExplorer operation. Although, the Intel processor we use for our measurements permits hundreds of parameters to be monitored using performance counters, not all of them are equally useful for the labeling. We examine performance parameters mentioned in Table 1.

To create heatmaps for each OPCExplorer operation, we ran our PowerShell script to collect performance counter samples of all processes during each operation on OPCExplorer at the time the script is running and saved them in a file. The script continued collecting samples until the designated number of samples were reached. We then used an additional script to go through the files and find the maximum and minimum range for each parameter to create intervals for our labels. The second script allowed the operator to input a log file of the OPCExplorer performance counters for a specific OPC client function and return only the values for a specific counter like the counter values for percentage of privileged time used by the process. After collecting samples for an operation and saving them in a file, we ran the filtering script to read the file and find the range of percentage of privileged time used for the operation. We then used this range to define labels.

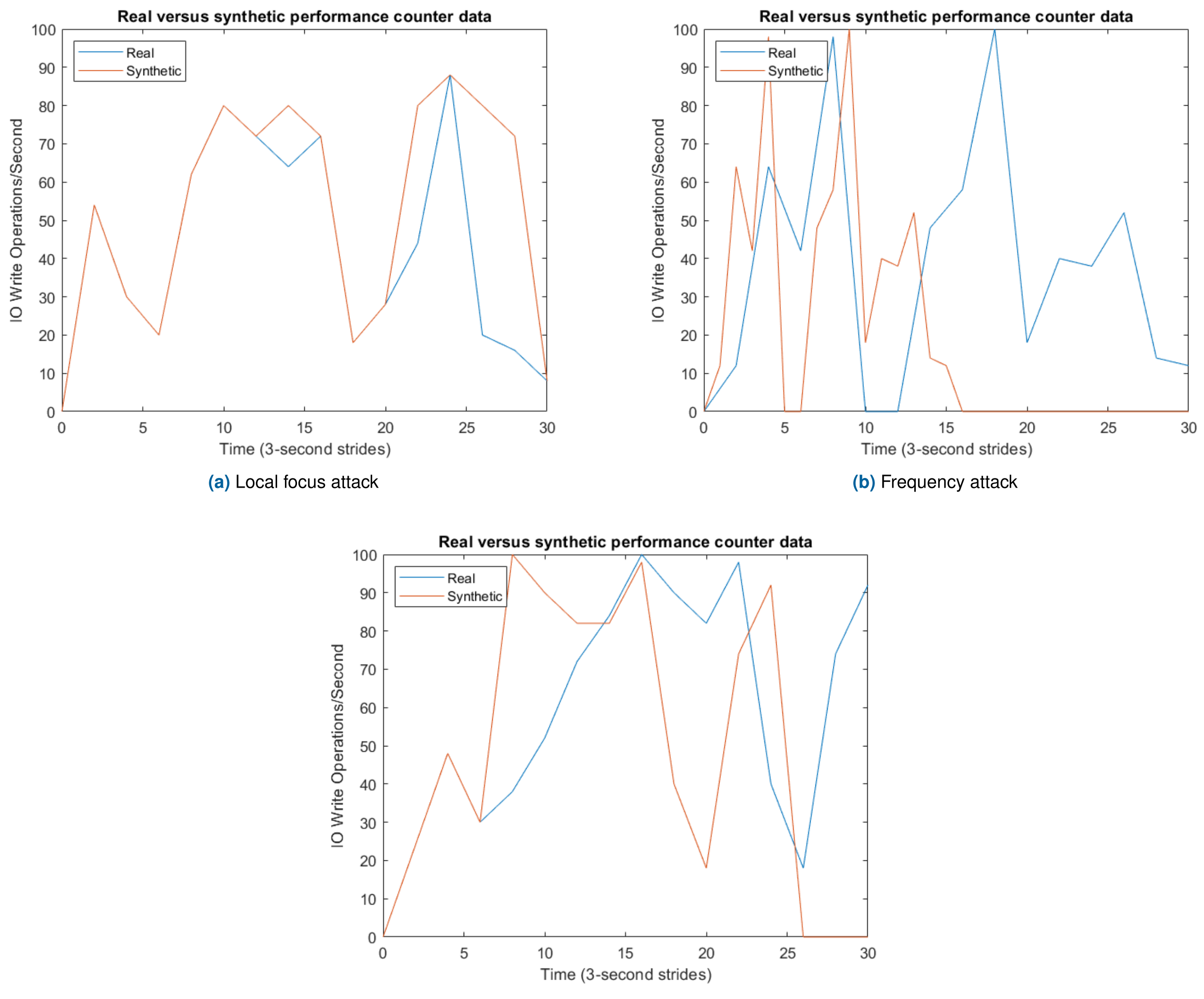
To create labels for our heatmaps, the original data collection script parses the data files where we have saved the raw output from the performance counter sampling for values related to the OPCExplorer process. It then filters these results to only include the performance parameters that have been selected as significant in the context of this paper. Null values are then stripped from this data set and averages are calculated from the remaining data points. These averages are then compared against our previously defined ranges and assigned a corresponding label component. These components are combined into one label string which is used to name the sub-directory where all heatmaps that fall within the defined ranges will be stored.

During the testing experience we noticed the I/O performance parameters, including I/O write and read operations from secondary storage, are always 0 value because these processes did not 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.

D. EXPERIMENTAL ATTACKS AGAINST DECOY PROCESS CONSISTENCY

The use of performance counters to detect decoy processes is a newly emerging technique. Since no known malware utilizes it, it's not possible to test the technique in a real-world scenario. We had to return to the honeypot experiment [5], which was successful in detecting honeypots and decoy I/O processes. We performed the experiment many times. Since we do not have access to malware that utilize our technique, we developed a red team that takes advantage of the findings of our work. This means that the red team may be aware that the processes it is interacting with may in fact be decoys. Further, they know that the decoy performance parameters may be based on heatmap-fed neural networks.

When our red team compared the performance counter of a real process with a decoy process, the real process shows a stronger sequential pattern. If our decoy process returns an incomplete set of values outside of what is expected of a real process, we again are at risk of exposing the decoy nature of the decoy process to threat actors. The outstanding problem was that performance counters for our decoy did not adapt to changes in the runtime environment. This can be exploited by threat actors to expose the decoy and avoid detection by the defenders. We illustrated this by acting as the attacker.



(a) Local focus attack

(b) Frequency attack

(c) Rate of change attack

FIGURE 10. Experimental attacks against decoy process consistency [4].

We researched attack techniques that consider the execution process over the control flow graph. Different paths they take depend on different factors, including input. To this end, given our red team position, we wrote PowerShell scripts to collect performance counter data and analyzed those performance counter data reported via our naive machine learning code that we wrote in MATLAB. The PowerShell scripts collected performance counters for both decoy processes and their real counterparts. We detected three types of attacks, namely, exploiting local focus, frequency, and rate of change attack.

Local focus attack happens when the neural network learns the performance counter values of real process from paths on the control flow graph that flow more frequently than other paths. The neural network might not observe infrequent paths during training. With reference to Figure 10, the local focus attack happens under similar heatmaps where the neural network favors the resource utilization that it has learned from the frequent paths. As a result, the neural network produces the most frequently occurring performance counter values for a given heatmap, and therefore fails to produce the rare performance counter values for the same or similar heatmap. In an overall lookup view, the produced performance counter values are correct, but only if the process is executing within a local situation within execution of process which appear to happen more frequently. Black swan performance counter values are not captured by our neural network.

The frequency attack happens when the neural network returns the correct performance counter data at a frequency that diverges from the real process. In this case, the decoy's frequency is too high or slow compared to the real process, and this may reveal the decoy process as a synthetic process. This is demonstrated in Figure 10b for the performance counter values of the I/O write operations per second. As shown in the figure, performance counter data from the real process span from $t = 0$ to $t = 30$. However, the decoy process begins at $t = 0$ and slightly pasts $t = 15$, and then zeroes out. It is important for our approach to report performance counter values of decoy processes within an exact time window and frequency that their real counterparts report. If the flowing path of the decoy process does not match the execution path of the real process, the decoy process will be at risk of being discovered by threat actors.

The rate of change attack is a new form of attack when performance counter data from the decoy process and the real process are assessed as a sequence. Therefore, the performance counter data provided by the neural network for a decoy process appear to be accurate when compared to those of its real counterpart. However, those data diverge from each other. More specifically, two heatmap classifications of the performance counter data are consistent when considered individually, but there could be inconsistencies when viewing them as adjacent performance counter readings. With reference to Figure 10c, the performance counter of decoy and its real counterpart align in the beginning from $t = 0$ to $t = 5$, a shift in the rate of change occurs the rest of the time. In our frequency attack, if the real process had a sequence of values $\{\alpha, \beta, \beta, \gamma\}$, these would be returned either too fast or too slow. However, for the rate of change attack, the returned values could possibly only be α and γ for that same period.

To improve the effectiveness of our heatmap training mechanism, we decomposed the overall functionality of OPC software into tasks. Each task is denoted by an operation code. We detected those tasks share many vertices in the control flow graph, but also have their own separate vertices. As we performed each task of the OPC client process, independently and one at a time, we collected its performance counters via PowerShell scripts and ran our approach to build the inner and outer automata. We used the interactive disassembler tool (IDA Pro) with Python code to extract the control flow graph, trace the execution paths of each task over the control flow graph, and record the time of transitions from one state to another in both the inner automata and the outer automaton. We ran each task several times to generate a large set of performance counter data and state transition timestamps.

E. EFFECT OF THE AUTOMATON MODEL

We now describe how automaton approach improves the heatmap training mechanism. The heatmaps are learned by explicitly indicating the location of the vertex of the cluster graph in the outer automaton. The heatmap will include the location of the state of the outer automaton at the time the request for performance counter data arrives. This location property is attached to a heatmap before feeding it to the neural network. Therefore, since the neural network reads heatmaps with this labeling, it no longer gets confused when similar heatmaps arrive. The location will clearly differentiate the heatmaps and the neural network will record each instance accurately based on the performance counter data.

To prevent the frequency attack, we rely on our implementation of an inner automaton and timing conditions. The inner automaton corrects the frequency attack via its representation of clusters of vectors of performance counter data as states/vertices. The transitions between these states of the inner automaton cannot occur at any time. Therefore, the transition from one state to another cannot happen unless the timing conditions are met. Since the inner automaton is considered before the data is reported to a request for performance counter data of a decoy process, data for the process cannot be reported too slowly nor too quickly to a threat actor.

That is, the neural network will not report performance counter data until the inner automaton has finished processing the time conditions for each transition. In the rate of change attack, we observed that the neural network originally had trouble producing performance counter data that was in sequence with a real process in our decoy process. To defeat this attack, we used both our inner automaton and n-gram model to enforce a sequence from one state to another. Since performance counter data inside the cluster of our inner automaton cannot occur irrationally, it must occur via the order enforced by the n-gram model which achieves this behavior probabilistically.

F. EXPERIMENTAL RESULTS

We wrote a dummy PowerShell program as a red team to do I/O, CPU and memory intense tasks and collected the performance counters of all the processes, including the intense usage processes during execution of each process on the compromised machine. We ran the same dummy program on the replica to collect the performance counters of all processes running on the system. We then compared the values to see how close or far the performance values are from each other and checked for consistency between the results generated on both machines. Figure 11 compares 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.

As the attacker, we created processes that created a large amount of CPU demand, generated heavy I/O traffic, and high rates of memory consumption. The neural network tolerated these changes and generated class labels with no noticeable changes. We ran local focus attacks, frequency attacks, and rate of change attacks that examined a data set holding solely performance counter samples extracted from numerous different runs of the real OPC client process. We called this dataset clean. The purpose of these tests was to identify false positives in how those attacks work prior to using them to estimate the effectiveness of our decoy process.

We ran the same attack, but this time on a dataset extracted from the performance counter of a decoy process, which relied on the naively trained neural network and real process. We called this dataset dirty to identify false negatives in those attacks. We also ran these attacks on a dataset which consisted of mixed performance counter samples from a real OPC client process and samples from imaginary executions of the decoy OPC client process. This time it had its execution dynamics

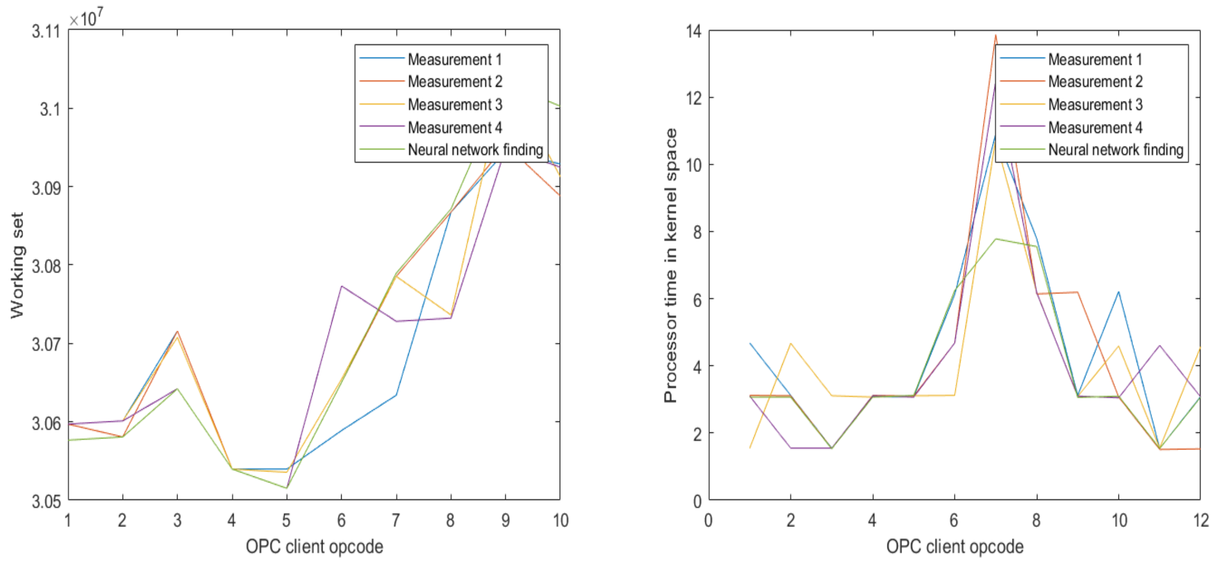


FIGURE 11. Empirical measurements of performance data versus deep learning class labels.

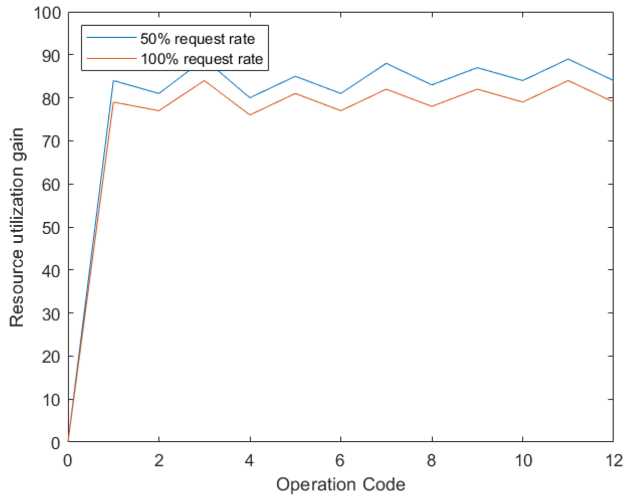


FIGURE 12. The resource utilization gain of our decoy process relative to the resource utilization of a decoy and existent OPC client process.

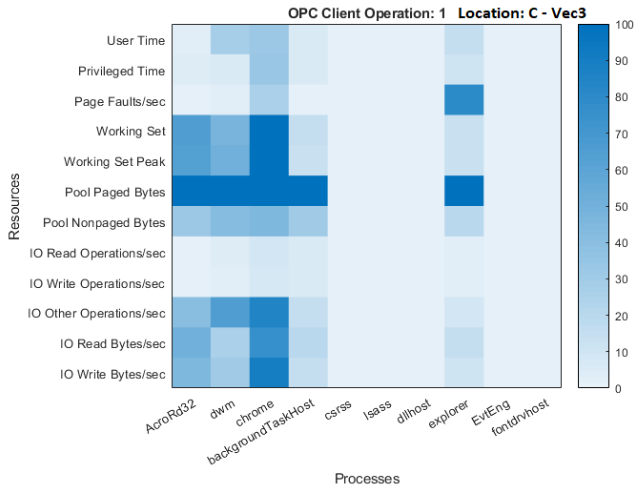


FIGURE 13. Illustration of a heatmap used in the improved neural network training.

regulated by the neural network as trained by our automaton model approach.

The three attack techniques, namely local focus, frequency, and rate of change attacks did not find resource utilization inconsistencies in the clean dataset. However, we detected resource utilization inconsistencies in approximately 38% of the performance counter samples in the dirty data set and no resource utilization inconsistencies in the mixed data set.

The data plots in Figure 12 show the resource utilization of our decoy process and OPC client process. We noticed the highest rate that our approach could encounter for the incoming requests to read and display at a time were every second. This is the case that tasks consume the highest amount of resources.

An illustration of a heatmap generated by our approach is given in Figure 13. Based on the time of arrival of requests for performance counters for a decoy process, our models determine the states of the outer and inner automata that match with that specific time.

IX. FUNCTION COMPARISON OF THE PROPOSED MODEL

The comparison of different deception models with each other is often challenging due to how a particulate model process and train data may be different from the other model being compared to it. Additionally, there aren't any proposed models based on the consistency of resource utilization of decoy process. Therefore, there is no numerical metric of a benchmark to compare our model with others. Thus, we used

TABLE 3. Functional comparison of related works with the proposed model.

Features	[15]	[27]	[10]	Proposed model
0-Knowledge detection	Yes	No	No	Yes
symantic behaviour (consistency)	No	Yes	Yes	Yes
performance counter	Yes	No	No	Yes
automated model	No	Yes	No	Yes
path analysis	Yes	No	No	Yes

functional comparison to compare the performance of the existing models with the proposed model.

This method is used to identify whether the performance of proposed method is the best or not. Table 3 clearly shows the performance comparison of various models with respect to proposed model respectively. As shown in the table, we used features such as 0-knowledge detection, consistency, automated model and path analysis, and low-level performance counters to compare our proposed work to other existing techniques. Our results show that our proposed work apples all of these features to optimize the decoy process, distinguishing it from others.

X. CONCLUSION

Our red team approach leveraged live real-time performance counters to see deep into the run-time dynamics of a process. Such deep insight enabled performance analyses that led to detection of several activity-related inconsistencies in high interaction honeypots and decoy real processes. We developed a countermeasure to protect decoy I/O from these threats on machines in production. The countermeasure consists of mechanisms in the OS kernel that project the existence of decoy processes. The work is done without spending resources on creating actual processes. We devised a convolutional neural network that can learn the performance fingerprint of a process in support of its decoy counterpart. Thus, a decoy process is given a performance profile that makes it indistinguishable from its real counterpart. Decoy processes affect malware's target selection to redirect them towards decoy I/O. In conclusion, we validated and quantified the ability of such decoy processes to sustain a realistic resemblance with a valid target of attack, and hence thwart detection probes directed at decoy I/O on machines in production.

In addition, we would like to clarify that the primary objective of our paper was not focused solely on the performance of our machine learning models, but rather on the development and validation of decoy processes as an effective means of redirecting malware. However, we recognize the importance of thoroughly validating our models and will make every effort to include more detailed information on this aspect in our future work.

ACKNOWLEDGMENT

The experiments presented in this article were originally conducted as part of the Ph.D. thesis of Sara Sutton [38]. The authors would like to thank Benjamin Bond for the careful proofreading.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] J. L. Rrushi, "Phantom I/O projector: Entrapping malware on machines in production," in *Proc. 12th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2017, pp. 57–66.

[2] *Performance Counters*. Accessed: Feb. 23, 2019. [Online]. Available: https://docs.microsoft.com/

[3] S. Sutton, G. Michilli, and J. L. Rrushi, "Redirecting malware's target selection with decoy processes," in *Proc. Annu. Conf. Data Appl. Secur. Privacy*. Cham, Switzerland: Springer, 2019, pp. 398–417.

[4] S. Sutton, B. Bond, S. Tahiri, and J. Rrushi, "Countering malware via decoy processes with improved resource utilization consistency," in *Proc. 1st IEEE Int. Conf. Trust, Privacy Secur. Intell. Syst. Appl. (TPS-ISA)*, Dec. 2019, pp. 110–119.

[5] J. L. Rrushi, "DNIC architectural developments for 0-knowledge detection of OPC malware," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 1, pp. 30–44, Jan. 2021.

[6] *Honeyprocs: Going Beyond Honeyfiles for Deception on Endpoints*, 2019.

[7] J. Rrushi, "Honeypot evader: Activity-guided propagation versus counter-evasion via decoy OS activity," in *Proc. 14th Int. Conf. Malicious Unwanted Softw.*, 2019, pp. 1–10.

[8] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proc. Int. Conf. Inf. Secur.* Cham, Switzerland: Springer, 2007, pp. 1–18.

[9] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC (DSN)*, Jun. 2008, pp. 177–186.

[10] O. Hayatle, H. Otrok, and A. Youssef, "A Markov decision process model for high interaction honeypots," *Inf. Secur. J., A Global Perspective*, vol. 22, no. 4, pp. 159–170, Jul. 2013.

[11] X. Jiang and D. Xu, "BAIT-TRAP: A catering honeypot framework," Tech. Rep., 7, 2019.

[12] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, 2013.

[13] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Cham, Switzerland: Springer, 2014, pp. 109–129.

[14] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 483–493.

[15] X. Wang and R. Karri, "NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proc. 50th Annu. Design Autom. Conf.*, May 2013, pp. 1–7.

[16] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2006, pp. 129–143.

[17] J. Jaffar and V. Murali, "A path-sensitively sliced control flow graph," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 133–143.

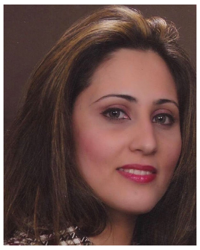
[18] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 215–224.

[19] A. M. Chirkin, "Towards better workflow execution time estimation," *IERI Proc.*, vol. 10, pp. 216–223, Jan. 2014.

[20] K. Murphy, "An introduction to graphical models," Tech. Rep., 96, pp. 1–19, 2001.

[21] K. Ali, "Using Bayesian learning to estimate how hot an execution path is," Tech. Rep., cs886, 2010.

[22] X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham, "On recognizing virtual honeypots and countermeasures," in *Proc. 2nd IEEE Int. Symp. Dependable, Autonomic Secure Comput.*, Sep. 2006, pp. 211–218.
[23] N. Krawetz, "Anti-honeypot technology," *IEEE Secur. Privacy*, vol. 2, no. 1, pp. 76–79, Jan. 2004.
[24] S. Mukkamala, K. Yendrapalli, R. Basnet, M. K. Shankarapani, and A. H. Sung, "Detection of virtual environments and low interaction honeypots," in *Proc. IEEE SMC Inf. Assurance Secur. Workshop*, Jun. 2007, pp. 92–98.
[25] Y. Park and S. J. Stolfo, "Software decoys for insider threat," in *Proc. 7th ACM Symp. Inf., Comput. Commun. Secur.*, May 2012, pp. 93–94.
[26] J. Lee, J. Lee, and J. Hong, "How to make efficient decoy files for ransomware detection?" in *Proc. Int. Conf. Res. Adapt. Convergent Syst.*, Sep. 2017, pp. 208–212.
[27] J. Sun, S. Liu, and K. Sun, "A scalable high fidelity decoy framework against sophisticated cyber attacks," in *Proc. 6th ACM Workshop Moving Target Defense*, Nov. 2019, pp. 37–46.
[28] *Metasploit Framework*, 2019.
[29] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals*. London, U.K.: Pearson, 2012.
[30] M. Rodríguez-Cayetano, "Design and development of a CPU scheduler simulator for educational purposes using SDL," in *Proc. Int. Workshop Syst. Anal. Model.*, 2010, pp. 72–90.
[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[32] S. E. Schaeffer, "Graph clustering," in *Computer Science Review*. Berlin, Germany: Springer, 2007.
[33] S. E. Schaeffer, "Stochastic local clustering for massive graphs," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining*. Cham, Switzerland: Springer, 2005, pp. 354–360.
[34] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proc. 36th Annu. ACM Symp. Theory Comput.*, Jun. 2004, pp. 81–90.
[35] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, vol. 1, 1967, pp. 281–297.
[36] B. Roark, M. Saraclar, and M. Collins, "Discriminative *n*-gram language modeling," *Comput. Speech Lang.*, vol. 21, no. 2, pp. 373–392, 2007.
[37] R. M. Vergaray and J. Rrushi, "On sustaining prolonged interaction with attackers," in *Proc. IEEE 15th Int. Conf Dependable, Autonomic Secure Comput., 15th Int. Conf Pervasive Intell. Comput., 3rd Int. Conf Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Nov. 2017, pp. 478–485.
[38] S. M. Sutton, "Decoy processes with optimal performance fingerprints," Ph.D. thesis, Oakland University, Rochester, Michigan, 2020.



SARA M. SUTTON received the master's degree in computer science and information systems from the University of Michigan and the Ph.D. degree in computer science and informatics from Oakland University, USA. She is currently an Assistant Professor of cybersecurity with the School of Computing, Grand Valley State University. Her research interests include cybersecurity, operating systems, and AI.

JULIAN L. RRUSHI (Member, IEEE) received the Ph.D. degree from the University of Milan, in 2009. He is currently a Faculty Member with the Department of Computer Science and Engineering, School of Engineering and Computer Science (SECS), Oakland University, Rochester, MI, USA. He works on operating systems, computer architectures, AI, and security and privacy.