General Process Detection Through Physical Side Channel Characterization

Michael A. Taylor
Darwin Deason Institute
Southern Methodist University
Dallas, TX, USA
taylorma@smu.edu

Eric C. Larson
Darwin Deason Institute
Southern Methodist University
Dallas, TX, USA
eclarson@smu.edu

Mitchell A. Thornton
Darwin Deason Institute
Southern Methodist University
Dallas, TX, USA
mitch@smu.edu

***Abstract*—Physical sensors present in modern Systems-on-a-Chip (SoC) provide a rich source of side channel information that can be exploited to detect and characterize processes running concurrently on the device. Sensor data is periodically collected and machine learning classifiers are employed that predict the types of processes running under a variety of processor load conditions. Experimental results evaluate a number of different classifier models and identify the best types of classifiers for four different general process types; i) file I/O, ii) CPU/ALU intensive, iii) network I/O, and iv) virtualization. The process characterization classifiers are evaluated under a range of processor load conditions varying from light loads to heavy loads to ascertain their effectiveness in the presence of other concurrent and benign processes. Our results indicate that some process classes are less sensitive to background load conditions versus others and our suggested classifier architectures are devised to account for this variability. The detection of file I/O and CPU/ALU-intensive processes are shown to exhibit very high efficiency and robustness with respect to background load conditions. Virtualization process detection is shown to exhibit high accuracy under light loads with moderate degradation observed as load conditions increase. Network I/O detection is shown to have the lowest accuracy due to the relatively small number of sensors present in the network interface card (NIC) of the system under evaluation.**

***Index Terms*—side channel, physical sensors, malware detection and characterization**

I. Introduction

The ability to detect and characterize processes running within the working set of a computer system via side channels enables a variety of applications including malware detection, process classification, and white- and black-list adherence [1]. The use of side channels for general process detection likewise allows for such processes to be identified and characterized in a manner that is difficult to spoof since the physical characteristics of the process are used rather than signatures or other means [2].

While this work focuses on a method of using physical sensor side channels in detecting any arbitrary target process our previous work in this area was designed for the detection of a specific process such as ransomware [3]. In this past work certain physical sensors present in modern System-on-Chips (SoC) that comprise one or more CPU cores were used to gather real-time data that is provided as input to a machine learning classifier to determine if an instance of ransomware was present. This past approach involved training the classifier with instances of processes that exhibited ransomware behavior. Specifically, many ransomware processes can be generally characterized as first identifying victim files for encryption followed by a phase where the identified files underwent encryption [4]–[6]. From a general process point of view, one can view these activities as first comprising a phase of file I/O activity followed by a phase of CPU-intensive operations when the encryption algorithms are run.

The objective of this work is to perform general process detection at a finer-grained level such that sensor data can indicate and discriminate between processes such as those that are heavily biased toward file I/O activity or CPU/ALU-intensive activity. If a suitable set of general processes could be detected, then the sequence described by the ransomware process would involve first detecting file I/O intensive activity followed by the detection of CPU/ALU-intensive activity. Thus, given the sequential behavior of any type of application, whether it is an instance of malware or not, the ability to detect the sequence and type of processes running on a CPU-based system could represent a set of basic building blocks that allow for arbitrary types of applications to be detected via physical side channels by characterizing an application of interest as a sequence of activities that are dominated by a particular type of process. For example, when a process is identified as first consisting of file I/O activity followed by bursts of heavy CPU/ALU activity then the overall process can be detected that is first searching for target files followed by processing them through heavy computation, such as an encryption phase, as is the case for many instances of ransomware. Such a capability greatly generalizes the approach taken for ransomware detection described in [3] and furthermore does not require customized training of process classifiers for each different instance of applications to be detected. Rather, the system administrator can specify a particular sequence of processes to characterize a process of interest, and when such a sequence is detected, an alert can be issued that indicates a particular process sequence has been detected, with an associated probability value. Another benefit of this research is the evaluation of the effectiveness of using physical sensor side channels (PSSC) for different types of general processes.

The efficacy of the use of PSSC-based detection and char-

acterization is largely dependent upon the presence, or lack of presence, of appropriate sensors in the host architecture [7]. For example, if a particular host architecture does not have sensors present inside or near the network interface circuitry (NIC), then the detection of a process that comprises significant amounts of communication with the external network is more difficult to achieve with the PSSC-based approach.

Another contribution of this work is the incorporation of models that account for sensor output due to other processes concurrently running on a system. From a signal processing point of view, a particular sensor can be viewed as a data collector whose output is due to the composite set of processes that are instantiated at any instance. An analogous concept is the presence of unintentional in-band jamming signals in the same band as a signal of interest. It is thus of high interest to account for and to incorporate means to allow the PSSC-based approach to be effective in SoCs where a plurality of processes are concurrently executing. While it is the case that operating systems generally use temporal sequencing of multiple processes to provide the illusion of concurrent execution, the time slices are very small and modern architectures have significant levels of concurrency due to clever architectural features.

As an example of the presence of concurrent processes within an SoC, a file I/O intensive process may be accessing memory through use of a direct memory access (DMA) core while the CPU/ALU is concurrently engaged in heavy number-crunching activities. Furthermore, these two activities may have originated from different processes altogether. Additionally, the relatively low sample rates of many on-board physical sensors as related to the operating system context switching rate effectively causes the physical sensors to be viewed as supplying a low-pass filtered version of a composite signal representative of more than one process executing at the same time. For these reasons, the sensor is effectively providing composite information regarding the environment within the system. We have devised our classifier models to account for the presence of such background loads.

We chose four general classes of processes to consider and we report on the ability of a PSSC detection and characterization approach for each process category. These are: i) file I/O, ii) CPU/ALU intensive, iii) network I/O, and iv) virtualization instantiation. Our results indicate varying levels of detection ability for these classes of processes. Classes i) and ii) are shown to have high accuracy over all background load conditions whereas class iii) can be reasonably detected under low load conditions and class iv) has very good detection capabilities for low load conditions but accuracy decreases as load conditions increase. The worst performing process detection case was that for network I/O. We attribute this poorest performing network I/O class to the fact that the NIC in our system under evaluation did not comprise a suitable set of physical sensors. NIC's that contain more sensors or for cases where the NIC is physically located adjacent to other sensor-rich subsystems would perform better. This indicates that alternative side channels for detecting network I/O may be desirable in our target architecture such as the use of network traffic patterns [8], [9] or the use of programmed event monitors [10], [11].

A. *Physical Sensor Side Channels*

Modern computing devices often include many different sensors that monitor the device's physical state in order to avoid damage or to control resources such as cooling fans and power sources. These sensor readings are often considered to be innocuous and can be easily accessed via system OS calls. Individually, accessing these sensors would not pose much of a security risk. However, access to all of the sensors within the system enables behavioral patterns to emerge that can yield detailed information regarding the nearly-instantaneous operation of the system.

B. *Detection Models*

Our detection models utilize multi-modal machine learning (ML) algorithms that receive a fused set of sensor readings allowing for the generation of a binary-valued prediction metric that indicates whether a target process is, or is not, currently executing within the system [12]. In the context of this study a "Detection Model" (DM) is an obfuscation of a machine learning model that, when provided a vector of a system's current physical sensor readings, provides a binary output representing whether the target process is currently running on a system. We note that our models could easily be extended to also provide a confidence factor associated with each binary prediction metric. Once a DM has been trained and implemented, the resources required to perform the resulting predictions is minimal as there are no complex behavioral or computational analyses being necessary. Additionally, as all of the data that is used for process detection is side-channel data, there is minimal risk to privacy and user data leakage.

C. *Contributions*

Our prototype implementation and accompanying experimental results demonstrate the viability of detecting arbitrary target processes by measuring the physical state of a system through the use of existing system sensors. Previous work has shown that process detection often requires complex behavioral analysis that can be to be very accurate, but often at the cost of performance and incurring significant delay thus causing these approaches to not be viable for real-time usage [13]. In applying our technique to malware detection, our new detection model presented here is meant to serve as an initial rapid indication of a target process that could trigger the activation of safety measures to be taken until such time as more traditional behavioral analyses can be performed. In this way the detection method presented in this paper is meant to act in tandem with existing methods of process detection in an effort to augment their performance and to serve as a second line of defense. Because our approach can model threats as a sequence of basic and common process tasks, it is applicable to zero-day vulnerabilities as well as known instances of malware. Other applications could include an ability to report

the state of a system for system administration purposes using physical sensors that are based upon the computational environment rather than characteristics or knowledge of the actual working set. Thus, an alternative and difficult-to-spoof means for providing information to system administrative tools for event management comprises another viable application for this approach.

II. Training and Building Detection Models

A. Experimental Environment and Setup

We utilized an Apple Mac Mini MGEM2LL/A with a 1.4 Ghz Intel Core i5 processor, 4 GB of LPDDR3 RAM, and a 500 GB HDD as our evaluation platform. Using a software application named "Hardware Monitor," we can access the current sensor readings for 50 different system sensors through the command line [14]. The system's sensors comprised 16 temperature sensors, 6 voltage sensors, 12 current sensors, 15 power sensors, and 1 sensor for the exhaust fan RPMs. Testing scripts and automated data collection are implemented using the Continuum Anaconda Python 3 package and its associated libraries [15], [16]. The Scikit-learn library is used to transform data, create prediction models, and to perform target process predictions [17], [18]. In order to create additional CPU loads for training, the Go programming language was installed in order to run the Go script called "*go-cpu-load*" [19]. This script maintains a desired CPU load by instantiating a continuous loop that allows for adjusting a delay period after each loop body execution in order to adjust CPU usage. To create additional CPU loads for testing and evaluation, we use the Stress-ng project package that is capable of creating a number of different system stressors with a high level of control [20]. We created the resulting CPU load by using over 70 different methods that are included in Stress-ng. All of the different load tasks differ as compared to the methods used during DM training with the "*go-cpu-load* " script.

This experiment was carried out with four different target processes that represent different types of processes that are likely to be seen by users. The first target process involves heavy file I/O and utilizes the Iozone filesystem benchmarking tool in [21]. The second target process involves heavy CPU resource usage and utilizes the FFmpeg multimedia framework [22] containing a significant amount of ALU-intensive instructions. The third target process involves heavy network I/O and utilizes the Nmap network discovery and security auditing tool [23]. The final target process involves launching and running a virtual machine (VM) on the host system and utilizes the commercial version of the VMware Fusion 10 Pro desktop hypervisor [24].

B. Experimental Process

To perform the evaluations of our process detection tool, experiments were specified using a python script that performs a series of training data collection tasks and DM training activities followed by a series of performance evaluation data collection tasks for each process being considered. Prior to beginning the data collection for a new process, the system is power-cycled in order to free all resources. After every individual training and test cycle, a comma-separated data file is stored on the test system until all testing ends and the data can be collected. Once the training and test data are collected the training data is used to create all of the necessary DMs for the various combinations of machine learning algorithms, test processes, and DM types. Once all of the models are created, data is collected for each test process and is used to generate the detection predictions for each time interval. The same test data is used for all created DMs for each test process in order to directly compare their performance with a controlled data set. Once all of the detection prediction vectors have been generated, performance metrics are computed in order to analyze and draw conclusions about the new detection method.

C. Collecting Training Data

Based on the findings from [3], we use a 2 hour window to characterize classification performance. In [3], it was reported that models trained with 2 hours of sensor data were equally effective as models trained with up to 24 hours of data. Due to this knowledge, training data for each additional CPU load level was collected for two hours. Additional system loads ranged from 0% to 100% with one data set being collected at each interval of 10%. The additional load is achieved by creating a process on all hyper-threads which run an empty loop at a dynamic rate that maintains the desired overall CPU load. Maintaining the desired additional load level relative to each hyper-thread ensures that the additional load is balanced over the system's total CPU resources.

During each two hour training cycle, three threads are created. The first thread controls the operation of the process being tested, the second thread adds a desired additional CPU load onto the system, and the third thread runs a data logger. The test process control thread starts by running the test process for an entire cycle and recording the amount of time that it requires to complete. Afterward, the test process control thread waits 2 minutes for the system sensors to re-stabilize and then it sets a Boolean variable that indicates that the data logging thread should begin recording data. The test process control thread then begins a loop which first waits for the amount of time previously recorded for the test process to complete, then the test process is run, and finally the test process control thread waits again for the amount of time previously recorded for the test process to complete. The test process control thread then waits 2 minutes for the system sensors to re-stabilize and if the total time has met or exceeded the desired total training time the loop halts execution. This data collection routine results in two hours of data comprising a 2:1 ratio of the test process not running to the test process running. This ratio is not a realistic representation of what is likely to be experienced in a real-world environment; however, for the purposes of training the detection models this ratio is more beneficial.

D. Training Machine Learning Models

DMs are trained with a known additional system CPU load level present while the test process is both running and not running. Each DM is a ML model trained for one of the test processes and includes all of the training data collected at various additional CPU load levels.

III. Evaluating Process Detectors (DM)

A. Collecting Evaluation Data

Evaluation data is collected using the same method as that for the training data. However, the target process is not initially timed and there is no loop present for running the target process until the desired time has elapsed. Instead, the target process controls thread wait times as a random amount of time between 10 and 40 minutes, runs a single instance of the target process, and then waits for the remainder of the desired evaluation time. This method results in data that is collected over a very short amount of time whereas the target process is actively running and it is occurring at a random time.

1) Simple Random Additional Load: Evaluation data is collected with a randomly selected additional CPU load preset on the system beyond what is present due to regular background processes. Each target process is run a single time over the course of one hour at a randomly determined time. Prior to performing each evaluation run, an additional CPU load level is randomly selected from a specific range of values and is applied to the system using the same method implemented in training. Four values are selected from a range of ten (10) values before moving to the next range of ten (10) values starting with one through ten and ending with 91 through 100. This results in a data set comprising 40 evaluations with equal representation of random additional load levels at intervals between where training data was collected with known additional CPU loads. This data set allows for the possibility of randomly selected additional load levels to be the same as the known load levels used during training.

2) Advanced Random Additional Load: Test data is collected with a randomly selected additional CPU load present on the system beyond what is present due to regular background processes. Each targeted process is run a single time over the course of one hour at a randomly determined time. Prior to performing each evaluation run, an additional CPU load level is randomly selected from a specific range of values and is applied to the system using a different method in comparison to those implemented during DM training. The randomly selected load level is further randomly divided into unbalanced load levels maintained on each hyper-thread that cumulatively applies the overall desired additional system CPU load level. The method of maintaining the load level on each hyper-thread is randomly selected from 70 different CPU stress methods that do not include the empty loop method utilized in training. Four values are selected from a range of nine (9) values before moving to the next range of nine (9) values, skipping those used during training, starting with one through nine and ending with 91 through 99. This results in a data set of 40 tests with equal representation of random additional load levels at intervals between where training data was collected with known additional CPU loads. This data set does not allow for the possibility of randomly selected additional load levels that are the same as the known load levels used during training.

B. Target Process Detection Methodology

1) Process Detection With Unknown Additional Loads: Section III-A details test data collection with two different types of loads present on the system which are both unknown to the detector. The data set collected from each type of unknown additional load implementation is used for a separate test of process detector's performance ability.

Simple random additional load test data is realistic for testing the process detectors. However, the implementation of the same method of applying the additional balanced CPU load as well as allowing for the possibility of selecting random load levels which are the same as those used during training incorporate some favorable aspects for process detection. The test on this data set is used for showing the ability of the process detectors when the system is likely in an unseen state from what was used for training, but the system exhibits similar patterns of behavior. Analysis of detection ability with this data set allows for identifying process detectors which are capable of performing in the most basic realistic scenarios.

Advanced random additional load test data is much more realistic and rigorous for testing the process detectors than the simple random additional load data sets. The implementation of multiple unseen methods of applying the additional unbalanced CPU load as well as not allowing for the possibility of selecting random load levels which are the same as those used during training truly tests the detection predictors in a scenario where obvious behavioral similarities are not present. The test on this data set is used for showing the ability of the process detectors when the system is in an unseen state from what was used for training with comparable system states achieved through completely different methods. Analysis of detection ability with this data set allows for identifying process detectors which are capable of performing in unfavorable and unseen realistic scenarios.

C. Performance Evaluation Methodology

1) Binary Classification Evaluation: In this evaluation experiment machine learning algorithms are individually used to make a prediction concerning the binary indication that a target process is running on a system. The final prediction vector of a model and the actual system state vector are compared to obtain the distribution of the four types of binary classification. The distribution of the four classifications are used to compute five metrics that offer more insight into prediction performance. These metrics are *sensitivity*, *precision*, *specificity*, *fallout*, and *accuracy* [25].

2) Matthews Correlation Coefficient (MCC): The Matthews correlation coefficient (MCC) is a measure of the correlation between true and predicted values [26]. MCC values range

between -1 and +1. A coefficient of +1 represents a perfect predictor, 0 represents the same as random prediction, and -1 indicates total disagreement. The MCC is also symmetric which means the positive and negative binary metrics can be reversed and the result will be the same MCC. This is useful in cases with a large imbalance between positive and negative system states during evaluation such as the evaluation data in this experiment. The MCC values provided in the experimental results serve as a reference for how well the DMs perform purely as binary classifiers without accounting for the context or purpose of the DMs implementation.

3) Rate of Process Detection (RPD): During predictor evaluation, an actual target process's time of presence is in the form of a time series that defines the time periods during which the target process is actually running on the system. Target process runs start when a labeled system state of "normal operation" transitions to a labeled state of "process running." Conversely, a target process run ends when a labeled system state of "process running" transitions to a labeled state of "normal operation". If a positive prediction exists during the period of time representing a target process run, then the target process is considered detected. Ideally there should exist at least one positive prediction during each target process run that would result in a 1.0 or perfect rate of process detection.

4) Time to Process Detection (TPD): The initial labeled system state of "process running" for each target process run in the actual process presence time series represents the time interval at which the target process run began. The first instance of a positive prediction in the corresponding prediction time series at or after this initial "process running" state and before the next "normal operation" system state represents the initial target process detection. Ideally the initial "process running" state itself would be a positive prediction, but in practice it is more likely that the sensors would need a small amount of time to reach the values at which positive prediction occurs. This metric counts the number of time intervals until the first positive prediction is recorded for every target process run which was successfully recognized. Afterward, all values are averaged to determine the time to process detection.

5) Detector Performance Score (DPS): Our new detection model is designed to rapidly detect a target process while maintaining a low fallout. In other words we are most concerned with three aspects of the detector's performance. The first aspect is to maintain a low TPD which would mean the detector is able to quickly identify when the target process begins running on a system. Based on previous work we note that in order for our model to be effective, it must have a TPD of no more than 60 seconds [13]. The second aspect of performance, most important for our new DM, is that DMs should maintain a low fallout during operation. This would mean that the DM is not creating so many false positive detection alerts that it becomes a detriment to operation. In a 2020 report by Neustar Security Solutions it was found that on average 26% of security alerts experienced by organizations were deemed false positives [27]. Therefore, we infer that, in order for our model to be effective, it must have a fallout of no more than 0.26. The final aspect of performance, most important for our new DM, is that it should miss minimal test instances of a target process. We determined that if the detection model misses an instance of the target process during DM evaluation, it should be heavily penalized during assessment of its performance. Taking into account the three most important aspects of performance we devise and use a new metric called "Detection Performance Score" (DPS) in order to more appropriately assess our DM's performance in addition to the more traditional binary classification metrics outlined above.

$$DPS = 1 - \frac{\frac{TPD}{60} + \frac{fallout}{0.26}}{2}$$

IV. Experimental Results

The results for each of the four target processes are compared using the two different tests described in III-B. The results for the simple random additional load evaluation experiment and the advanced random additional load evaluation experiment are computed for each algorithm.

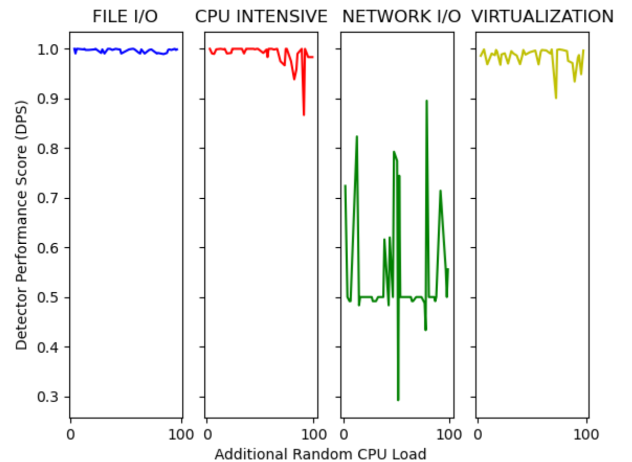

Fig. 1. Top Detector DPS for Each Process

A. File I/O Process

For this target process we are aiming to create a significant amount of file I/O on the system in order to determine if the new method is capable of detecting a specific pattern of file I/O with a random amount of CPU usage occurring at the same time. We implemented the filesystem benchmarking tool IOzone to act as our file I/O process. IOzone generates and measures a variety of file operations in order to measure a system's file I/O performance. The actual process we ran is given as follows:

```
$ iozone -a
```

1) File I/O Process Evaluation Results: The results of the simple random additional load test and the advanced random additional load test are present at the top of Table I. During the advanced random load evaluation, the detector that was trained using the logistic regression ML model exhibited the best performance. The detector correctly predicted all instances of the target process during evaluation within an average of 0.275 seconds of detection latency. All instances of the target process were correctly predicted by all ML algorithms with an average detection latency of less than half a second. These results indicate that our new method of process detection is highly successful when the process was mostly performing file I/O tasks.

While these results are encouraging, we anticipated high performance from the detection models during this experiment. The CPU is the component of the system that has an additional load placed on it. During the file I/O testing very little of the system's CPU resources are required, particularly for SoC that utilize DMA channels for file I/O operations. This type of target process demonstrates a situation where the target process predominantly utilizes an infrequently used system resource. These results demonstrate the viability of utilizing the new DMs for target processes that fall within this category.

B. CPU Intensive Process

For this target process our goal is to create a significant amount of CPU usage on the system to determine if the new method can detect a CPU intensive process with an additional random amount of CPU usage occurring at the same time. The actual target process comprised the multimedia tool ffmpeg to convert a video in .mov format to a video in .mp4 format. The target process uses ffmpeg to re-encode an entire sample video file thus using a large amount of ALU-intensive operations. We used a sample video that is 100 MB in size, 3 minutes in length, and with high definition resolution. The actual target process is as follows:

```
$ ffmpeg -i video.mov -c:v libx264 -crf
   10 video.mp4
```

This command requires roughly 6 minutes to complete on the Mac Mini system used in our evaluation.

1) CPU Intensive Process Evaluation Results: The results of the CPU intensive evaluation process are shown in the second section from the top of Table I. In observing the results of the random additional load test, it can be seen that the detector that was trained using the random forest machine learning algorithm performed the best with all target process instances being detected with an average latency of 1.275 seconds.

The results from this test are very encouraging as we had anticipated a drop-off in detection performance. We postulate that the high performance for this case is due to the CPU having a random additional load placed on it while also running a CPU intensive process. When we analyzed the feature importance scores determined by the ML algorithms, we found that CPU measurements are given less weight than measurements for the system memory. Due to the target process frequently utilizing a large portion of the CPU resources, patterns were discovered in the operation of the system memory that indicated the target process was currently running. This experiment demonstrates a situation where the target process predominantly utilizes a frequently used system resource. This experiment also demonstrates the viability of utilizing the new DMs for target processes which fit this category.

C. Network I/O Process

For this target process, our objective is to create a significant amount of network traffic on the system to determine if the new method can detect a process which generates heavy network traffic with an additional random amount of CPU usage occurring at the same time [28]. The actual process we ran utilized the network discovery and testing tool NMAP to carry out a full port scan of every IP address in the subdomain of the evaluation systems very quickly. We ran the actual process four consecutive times during each test cycle and is specified as follows:

```
$ sudo nmap -Pn -T4 10.10.10.0/24
```

This command required roughly 5 minutes to complete on the evaluation Mac Mini system.

1) Network I/O Process Evaluation Results: The results for the network I/O test process are present in the third section from the top of Table I. It is observed that the performance is significantly lower than either of the previously tested processes. The highest performing machine learning algorithm for the advanced random load test was K-nearest neighbor (KNN). Using the KNN algorithm, all instances of the target process were detected with an average detection latency time of only 1.95 seconds. However, the fallout for this detector was 0.472 which makes implementation undesirable as it would result in far too many false positives for most applications. This same trend is observed with the other ML algorithms that were considered.

Upon further analysis, the Mac Mini evaluation system did not have any physical sensors that directly monitored the network interface in the system. Instead, the only way to measure this part of the system involved using the physical sensors present on the main board. It is likely that this lack of resolution in measurements lead to the diminished performance for this target process. This target process demonstrates a situation where the target process predominantly utilizes a system resource that does not have a means of direct measurement through the system's physical sensors. Not surprisingly given the lack of physical sensors on the NIC, this experiment demonstrates the limited viability of utilizing the new DMs for target processes that fit this category.

D. Virtualization Process

For this target process, our goal is to launch a virtual machine (VM) using a type 2 hypervisor with a simulated user load running on the virtual machine to determine if

the new method can detect a VM instance running with an additional random amount of CPU usage occurring at the same time on the host system [29]. We utilized VMware Fusion 10 Pro as the type 2 hypervisor running on the Mac Minis. VMware Fusion includes a command line tool called vmrun that allows a virtual machine to be controlled and interacted with on the host system through the command line. We implemented commands to start and display, run a Bash script, and shutdown and close a specific VM. We use a VM that is running Xubuntu, an Ubuntu Linux OS variant that uses the XFCE desktop environment that is lighter weight and requires fewer resources. The VM is allocated one of the Mac Mini's processing cores, 2 GB of memory out of a total of 4 GB, and a pre-allocated virtual hard drive of 100GB. The command used to launch and display the virtual machine is:

```
$ vmrun start process.vmx gui
```

After initiating Xubuntu, we pass a command to the VM that includes a path to the Bash interpreter and a path to a script that we created to add a simulated user load to the VM. The command to run the bash script is:

```
$ vmrun -gu <USERNAME> -gp <PASSWORD>
   runScriptInGuest process.vmx -
   interactive "" "/bin/bash usersim 5"
```

The script creates one CPU stressor, one mixed I/O stresser, and one hard drive stressor. The usersim script is run for 5 minutes each time after the VM has fully started. Once the usersim script has completed the 5-minute run, the VM is shut down using the following command:

```
$ vmrun stop process.vmx soft
```

Instantiating and starting the VM, running the usersim script, and shutting down the VM requires approximately 7 minutes.

1) Virtualization Process Evaluation Results: The results for the virtualization test process are provided in the last section of Table I. Both the simple random load test and the advanced random load test resulted in a DPS higher than 0.9 for all machine learning algorithms except for one. The highest performing ML algorithm is *K*NN as it has the highest DPS for the advanced random load test and the second highest DPS for the simple random load test. The detection model that uses *K*NN detected all instances of the target process during the advanced random load test yielding an average detection latency time of 0.425 seconds while maintaining an average fallout of only 0.0065. It is important to note that the highest MCC score during the advanced random load test was only 0.6735 by the detector using the SVC ML algorithm. This result indicates that the new DM is very successful at quickly detecting the target process as well as keeping fallout very low which is the goal in the context of this experiment. However, for this target process the DMs had difficulty with continuing to correctly identify the process after the initial detection. The low MCC scores are a result of symmetric evaluation as the positive system state, which is very small relative to the negative system state, is incorrectly classified frequently after the initial positive prediction. In other words the DMs trained for detecting the VM are less than ideal in terms of being purely implemented for binary classification. Although, when considering their performance in the context of rapid process detection the negative aspects of their performance as pure binary classifiers are mitigated. This result is reflected in the DPS metric that is 0.9839 for *K*NN.

V. Conclusion

The results we describe in this paper demonstrates that the PSSC-based ransomware detection approach devised and evaluated in our previous research can be successfully extended for the general case of arbitrary process detection. Our new DM exhibits a high level of performance when trained to detect processes that predominately use resources that are directly monitored by a subset of the system's sensors. Not surprisingly, we show that the approach is less effective when the hardware directly involved in supporting a target process has no physical sensors as direct monitors. We hypothesize that as system complexity inevitably increases, there will likely be a corresponding increase in the number of physical system sensors present to monitor and regulate subsystem operation. The need to monitor system components with a higher level of resolution will almost also certainly increase the performance of the new DM and allow for additional use cases.

References

[1] D. F. Kune, A. B. Ransford, and D. E. Holcomb, "Anomoly and malware detection using side channel analysis," Jul 2016.

[2] N. Gattu, M. N. I. Khan, A. De, and S. Ghosh, "Power side channel attack analysis and detection," *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[3] M. Taylor, E. Larson, and M. Thornton, "Rapid Ransomware Detection Through Side Channel Exploitation," in *IEEE International Conference on Cyber Security and Resilience*, 2021, pp. 47–54.

[4] Correa R, ""How Fast Does Ransomware Encrypt Files? Faster Than You Think"," https://blog.barkly.com/how-fast-does-ransomware-encrypt-files, 2016, note: This is an electronic document. Date of publication: [April 2016];.

[5] Bracken B, ""What's Next for Ransomware in 2021?"," https://threatpost.com/ransomware-getting-ahead-inevitable-attack/162655/, Dec 2020.

[6] U.S. Department of Homeland Security, "Alert (TA16-091A) Ransomeware and Recent Variants," 2016. [Online]. Available: https://www.us-cert.gov/ncas/alerts/TA16-091A

[7] M. Alam, D. Mukhopadhyay, S. P. Kadiyala, S. K. Lam, and T. Srikanthan, "Side-channel assisted malware classifier with gradient descent correction for embedded platforms," *7th International Workshop on Security Proofs for Embedded Systems*, vol. 7, p. 1–15, 2018.

[8] S. Kadloor, X. Gong, N. Kiyavash, T. Tezcan, and N. Borisov, "Low-cost side channel remote traffic analysis attack in packet networks," *2010 IEEE International Conference on Communications*, pp. 1–5, 2010.

[9] D. G. X.H. Wu, H.T. Ma, "An automatic network protocol reverse engineering method for vulnerability discovery," *Network Security and Communication Engineering*, p. 65–70, 2015.

[10] R. Oshana, M. A. Thornton, E. C. Larson, and X. Roumegue, "Real-time edge prrocessing detection of malicious attacks using machine learning and processor core events," in *Proceedings of 2021 IEEE Systems Conference*. IEEE, 2021, pp. 1–8.

[11] C. Li and J.-L. Gaudiot, "Detecting spectre attacks using hardware performance counters," *IEEE Trans. on Comp.*, vol. early pre-print, p. 55, 2021.

[12] S. Picek, A. Heuser, A. Jovic, S. A. Ludwig, S. Guilley, D. Jakobovic, and N. Mentens, “Side-channel analysis and machine learning: A practical perspective,” *2017 International Joint Conference on Neural Networks (IJCNN)*, p. 4095–4102, May 2017.

[13] M. Rhode, P. Burnap, and K. Jones, “”early-stage malware prediction using recurrent neural networks”,” *Computers & Security*, vol. 77, p. 578–594, 2018.

[14] Bresink M, “”Hardware Monitor: Reference Manual”,” https://www.bresink.com/osx/216202/Docs-en/index.html, 2017, note: This is an electronic document. Date of publication: [2017];.

[15] “Individual edition.” [Online]. Available: https://www.anaconda.com/products/individual

[16] F. Chollet, *Deep Learning with Python*. Manning Publications, 2017.

[17] Pedregosa F, Varoquaux G, Gramfort A, Michel V, “”Scikit-Learn User Guide”,” Tech. Rep., 2010.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine Learning in Python ,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] Vikyd, “vikyd/go-cpu-load: Generate cpu load on windows/linux/mac.” [Online]. Available: https://github.com/vikyd/go-cpu-load

[20] ColinIanKing, “Colinianking/stress-ng: This is the stress-ng upstream project git repository. stress-ng will stress test a computer system in various selectable ways. it was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces.” [Online]. Available: https://github.com/ColinIanKing/stress-ng

[21] “iozone filesystem benchmark.” [Online]. Available: https://www.iozone.org/

[22] “ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video.” [Online]. Available: https://www.ffmpeg.org/

[23] “nmap security scanner.” [Online]. Available: https://nmap.org/

[24] “Vmware fusion 12 pro.” [Online]. Available: https://store-us.vmware.com/vmware-fusion-12-pro-5424173700.html

[25] A. Tharwat, “Classification assessment methods,” *Applied Computing and Informatics*, vol. 17, no. 1, p. 168–192, 2020.

[26] D. Chicco and G. Jurman, “The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation,” *BMC Genomics*, vol. 21, no. 1, 2020.

[27] H. N. S. M. 24, H. N. Security, and M. 24, “Increasing number of false positives causing risk of alert fatigue,” Mar 2020. [Online]. Available: https://www.helpnetsecurity.com/2020/03/24/alert-fatigue/

[28] P. Prasse, L. Machlica, T. Pevny, J. Havelka, and T. Scheffer, “Malware detection by analysing network traffic with neural networks,” *2017 IEEE Security and Privacy Workshops (SPW)*, p. 205–210, 2017.

[29] Y. Lusky and A. Mendelson, “Sandbox detection using hardware side channels,” *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, p. 192–199, 2021.

TABLE I
RANDOM UNKNOWN ADDITIONAL LOAD TEST RESULTS

		Average				
	Algorithm	**DPS**	**MCC**	**RPD**	**Fall**	**TPD**
File I/O Process						
Simple	SVC	0.996	0.9955	1.0	0.0013	0.175
	Rand Forest	0.9951	0.9943	1.0	0.0017	0.2
	Extra Tree	0.9944	0.9919	1.0	0.0019	0.25
	Log Reg	0.9926	0.9899	1.0	0.0032	0.15
	N Bayes	0.9897	0.9848	1.0	0.0042	0.275
	KNN	0.9865	0.9809	1.0	0.0064	0.15
	MLP	0.9857	0.9777	1.0	0.0069	0.125
	Dec Tree	0.9001	0.8785	1.0	0.051	0.225
Advanced	Log Reg	0.9968	0.9982	1.0	0.0005	0.275
	Rand Forest	0.9964	0.9984	1.0	0.0004	0.35
	KNN	0.9956	0.9972	1.0	0.0008	0.35
	Extra Tree	0.9952	0.995	1.0	0.001	0.35
	SVC	0.9949	0.9948	1.0	0.0014	0.275
	N Bayes	0.9948	0.9946	1.0	0.0011	0.375
	Dec Tree	0.9921	0.9963	1.0	0.0007	0.775
	MLP	0.6588	0.5639	1.0	0.3586	0.175
CPU Intensive Process						
Simple	Dec Tree	0.9988	0.9987	1.0	0.0002	0.1
	KNN	0.9987	0.999	1.0	0.0001	0.125
	Log Reg	0.9986	0.9991	1.0	0.0002	0.125
	SVC	0.9983	0.9989	1.0	0.0002	0.15
	Rand Forest	0.9983	0.9983	1.0	0.0004	0.1
	Extra Tree	0.9974	0.9951	1.0	0.0007	0.15
	MLP	0.9972	0.9944	1.0	0.001	0.1
	N Bayes	0.9267	0.8958	1.0	0.0844	0.25
Advanced	Rand Forest	0.989	0.7878	1.0	0.0002	1.275
	Dec Tree	0.9875	0.7613	1.0	0.0004	1.4
	Extra Tree	0.9861	0.5901	1.0	0.0005	1.55
	KNN	0.9776	0.6876	1.0	0.0008	2.5
	N Bayes	0.7896	0.6577	1.0	0.3329	0.6
	MLP	0.5484	0.4717	0.55	0.0003	0.2273
	Log Reg	0.4988	0.4756	0.5	0.0001	0.25
	SVC	0.4738	0.4738	0.475	0.0001	0.2632
Network I/O Process						
Simple	SVC	0.8363	0.3751	0.95	0.046	3.6842
	MLP	0.7829	0.2976	1.0	0.1874	0.975
	KNN	0.7641	0.1906	1.0	0.1234	0.875
	Log Reg	0.7568	0.2759	0.875	0.0492	4.0571
	Extra Tree	0.7296	0.1318	0.975	0.1662	3.3333
	Dec Tree	0.6966	0.1754	1.0	0.1725	1.8
	N Bayes	0.4965	0.0479	0.775	0.4243	2.8065
	Rand Forest	0.4318	0.1693	0.5	0.0327	2.9
Advanced	KNN	0.5484	0.017	1.0	0.472	1.95
	Dec Tree	0.5114	0.0114	1.0	0.4957	3.7
	Log Reg	0.5103	0.0094	1.0	0.8753	0.2
	MLP	0.4888	-0.0636	1.0	0.8323	1.65
	SVC	0.4811	0.015	0.975	0.906	1.1795
	Extra Tree	0.3904	-0.0756	0.775	0.5175	5.129
	N Bayes	0.3625	-0.0337	0.725	0.7077	0.0
	Rand Forest	0.3354	0.0135	0.575	0.3946	2.7826
Virtualization Process						
Simple	Log Reg	0.9896	0.9402	1.0	0.004	0.325
	KNN	0.9865	0.9375	1.0	0.006	0.225
	SVC	0.9863	0.9365	1.0	0.0061	0.225
	Rand Forest	0.9753	0.884	1.0	0.0106	0.525
	Extra Tree	0.9721	0.8756	1.0	0.0136	0.2
	MLP	0.9609	0.8591	1.0	0.0194	0.225
	Dec Tree	0.9481	0.8228	1.0	0.0259	0.25
	N Bayes	0.8566	0.6393	1.0	0.1339	0.8
Advanced	KNN	0.9839	0.6658	1.0	0.0065	0.425
	SVC	0.9836	0.6735	1.0	0.0071	0.325
	Rand Forest	0.9822	0.6697	1.0	0.0081	0.275
	MLP	0.9753	0.5257	1.0	0.0073	1.275
	Extra Tree	0.9566	0.5926	1.0	0.021	0.375
	Log Reg	0.9386	0.6506	1.0	0.037	0.425
	N Bayes	0.9029	0.5785	1.0	0.0772	0.3
	Dec Tree	0.7266	0.2298	1.0	0.2829	0.2