Hindawi
Security and Communication Networks
Volume 2021, Article ID 8933681, 23 pages
https://doi.org/10.1155/2021/8933681

Research Article

Real-Time Malware Process Detection and Automated Process Killing

Matilda Rhode,[1,2] **Pete Burnap**,[2] **and Adam Wedgbury**[1]

[1]*Airbus, Newport, UK*
[2]*Cardiff University, Cardiff, UK*

Correspondence should be addressed to Matilda Rhode; matilda.rhode@airbus.com

Received 1 October 2021; Accepted 15 November 2021; Published 6 December 2021

Academic Editor: Mamoun Alazab

Perimeter-based detection is no longer sufficient for mitigating the threat posed by malicious software. This is evident as antivirus (AV) products are replaced by endpoint detection and response (EDR) products, the latter allowing visibility into live machine activity rather than relying on the AV to filter out malicious artefacts. This paper argues that detecting malware in real-time on an endpoint necessitates an automated response due to the rapid and destructive nature of some malware. The proposed model uses statistical filtering on top of a machine learning dynamic behavioural malware detection model in order to detect individual malicious processes on the fly and kill those which are deemed malicious. In an experiment to measure the tangible impact of this system, we find that fast-acting ransomware is prevented from corrupting 92% of files with a false positive rate of 14%. Whilst the false-positive rate currently remains too high to adopt this approach as-is, these initial results demonstrate the need for a detection model that is able to act within seconds of the malware execution beginning; a timescale that has not been addressed by previous work.

1. Introduction

Our increasingly digitised world broadens both the opportunities and motivations for cyberattacks, which can have devastating social and financial consequences [1]. Malicious software (malware) is one of the most commonly used vectors to propagate malicious activity and exploit code vulnerabilities.

Due to the huge numbers of new malware appearing each day, the detection of malware samples needs to be automated [2]. Signature-matching methods are not resilient enough to handle obfuscation techniques or to catch unseen malware types and as such, automated methods of generating detection rules, such as machine learning, have been widely proposed [3–6]. These approaches typically analyse samples when the file is first ingested, either using static code-based methods or by observing dynamic behaviours in a virtual environment.

This paper argues that both of these approaches are vulnerable to evasion from the attacker. Static methods may be thwarted by simple code-obfuscation techniques whether rules are hand-generated [7] or created using machine learning [8]. Dynamic detection in a sandboxed environment cannot continue forever, either it is time-limited [9] or ends after some period of inactivity [10]. This fixed period allows attackers to inject benign activity during analysis and wait to carry out malicious activity once the sample has been deemed harmless and passed on to the victim's environment. The pre-execution filtering of malware is the model used by antivirus but this is insufficient to keep up with the ever-evolving malware landscape and has led to the creation of endpoint detection and response (EDR) products which allow security professionals to monitor and respond to malicious activity on the victim machine. Real-time malware detection also monitors malware live on the machine thus capturing any malicious activity on the victim machine even if it was not evident during initial analysis. This paper proposes that once a threat is detected, due to the fast-acting nature of some destructive malware, it is vital to have automated actions to support these detections. In this paper, we investigate automated detection and killing of malicious processes for endpoint protection.

There are several key challenges to address in detecting malware on-the-fly on a machine in use by comparison with detecting malicious applications that are detonated in isolation in a virtual machine. These are summarised below:

(1) **Signal Separation:** Detection in real time requires that the malicious and benign activities are separated in order that automated actions can be taken on only the malicious processes.

(2) **Use of Partial Traces:** In order to try and mitigate damage, malware needs to be detected as early as possible, but, as shown in previous work [11], there is a trade-off between the amount of data collected and classification accuracy in the first few seconds of an application launching and the same may be true for individual processes.

(3) **Quick Classification:** The inference itself should be as fast as possible in order to further limit the change of malicious damage once the process is deemed malicious.

(4) **Impact of Automated Killing in Supervised Learning:** Supervised learning averages the error rate across the entire training set but when the classification results in an action, this smoothing out of errors across the temporal dataset is not possible.

This paper seeks to address these key challenges and provides preliminary results including a measure of "damage prevented" in a live environment for fast-acting destructiveware. As well as the results from these experiments, this paper contributes an analysis of the computational resources against detection accuracy for many of the most popular machine-learning algorithms used for malware detection.

The key contributions of this paper are as follows:

(i) The first general malware detection model to demonstrate damage mitigation in real-time using process detection and killing

(ii) Benchmarking of commonly used ML algorithm implementations with respect to computational resource consumption

(iii) Presentation of real-time malware detection against more user background applications than have previously been investigated; increasing from 5 to 36 (up to 95 simultaneous processes)

The next section outlines related work, followed by a report of the three methodologies that were tested to try and address these challenges 3 in which the method for evaluating these models is also explained (6.5). The experimental setup is described in Section 5.2.1, followed by results in Section 6.

2. Related Work

2.1. Malware Detection with Static or Post-collection Behavioural Traces

2.1.1. Static Sources. Machine learning models trained on static data have shown good detection accuracy. Chen et al. [5] achieved 96% detection accuracy using statically extracted sequences of API calls to train a Random Forest model. However, static data have been demonstrated to be quite vulnerable to concept drift [12, 13]. Adversarial samples present an additional emerging concern; Grosse et al. [14] and Kolosnaji et al. [8] demonstrated that static malware detection models achieving over 90% detection accuracy could be thwarted by injecting code or simply altering the padded code at the end of a compiled binary, respectively.

2.1.2. Post-Collection Dynamic Data. Dynamic behavioural data are generated by the malware carrying out its functionality. Again machine learning models have been used to draw out patterns between malicious and benign software using dynamic data. Various dynamic data can be collected to describe malware behaviour. The most commonly used data are API calls made to the operating system, typically recorded in short sequences or by frequency of occurrence. Huang and Stokes's research [3] reports the highest accuracy in recent malware detection literature with a very large dataset of more than 6 million samples to achieve an accurate detection rate of 99.64% using a neural network trained on the input parameters passed to API calls, their return values, and the co-occurrence of API calls. Other dynamic data sources include dynamic opcode sequences (e.g., Carlin et al. [9] achieved 99% using a Random Forest), hardware performance counters (e.g., Sayadi [15] achieved 94% on Linux/Ubuntu malware using a decision tree), network activity and file system activity (e.g., Usman et al. [16] achieved 93% using a decision tree in combination with threat intelligence feeds and these data sources), and machine activity metrics (e.g., Burnap et al. [17] achieved 94% using a self-organising map). Previous work [18] demonstrated the robustness of machine activity metrics over API calls in detecting malware collected from different sources.

Dynamic detection is more difficult to obfuscate but typically the time taken to collect data is several minutes, making it less attractive for endpoint detection systems. Some progress has been made on early detection of malware. Previous work [11]) was able to detect malware with 94% accuracy within 5 seconds of execution beginning. However, as a sandbox-based method, malware which is inactive for the first 5 seconds is unlikely to be detected with this approach. Moreover, the majority of dynamic malware detection papers use virtualised environments to collect data.

2.2. Real-Time Malware Detection with Partial Behavioural Traces. OS = operating system; HPCs = Hardware performance counters; DT = Decision Tree; MLP = Multi-layer perceptron; NN = Neural Network; RF = Random Forest.

Previous work has begun to address the four challenges set out in the introduction. Table 1 summarises the related literature and the problems considered by the researchers.

To the best of our knowledge, challenge *(1) signal separation* has only previously been addressed by Sun et al. [23] using sequential API call data. The authors execute up to 5 benign and malicious programs simultaneously achieving 87% detection accuracy after 5 minutes of execution and 91% accuracy after 10 minutes of execution.

Challenge *(2) to detect malware using partial traces as early as possible* has not been directly addressed. Some work has looked at early run-time detection; Das et al. [20] used an FPGA as part of a hybrid hardware-software approach to detect malicious Linux applications using system API calls which are then classified using a multilayer perceptron. Their model was able to detect 46% of malware within the first 30% of its execution with a false-positive rate of 2% in offline testing. These findings however were not tested with multiple benign and malicious programs running simultaneously and do not explain the impact of detecting 46% of malware within 30% of its execution trace in terms of benefits to a user or the endpoint being protected. How long does it take for 30% of the malware to execute? What has occurred in that time?

Greater attention has been paid to challenge *(3) quick classification*, insofar as this problem also encompasses the need for lightweight detection. Some previous work has proposed hardware based detection for lightweight monitoring. Syadi et al. [15] use high performance counters (HPCs) as features to train ensemble learning algorithms and scored 0.94 AUC using a dataset of 100 malicious and 100 benign Linux software samples. Ozsoy et al. [21] use low-level architectural events to train a multilayer perceptron on the more widely used [25] (and attacked) Windows operating system. The model was able to detect 94% of malware with a false-positive rate of 7% using partial execution traces of 10,000 committed instructions. The hardware-based detection models, however, are less portable than software-based systems due to the ability of the same operating system to run on a variety of hardware configurations.

Both Sun et al. [23] and Yuan [22] propose two-stage models to address the need for lightweight computation. The first stage comprises a lightweight ML model such as a Random Forest to alert suspicious processes, the second being a deep learning model which is more accurate but more computationally intensive to run. Two-stage models, as Sun et al. [23] note, can get stuck in an infinite loop of analysis in which the first model flags a process as suspicious but the second model deems it benign and this labelling cycle continues repeatedly. Furthermore, if the first model is prone to false negatives, malware will never be passed to the second model for deeper analysis.

Challenge *(4) the impact of automated actions* has been discussed by Sun et al. [23]. The authors also propose the two-stage approach as a solution to this problem. The authors apply restrictions to the process whilst the deeper NN analysis takes place followed by the killing of malicious-labelled processes. The authors found that the delaying strategy impacted benignware more than malware and used this two-stage process to account for the irreversibility of the decision to kill a process. The authors did not assess the impact on the endpoint with respect to the time at which the correctly classified malware was terminated.

3. Methodology-Three Approaches

As noted above, supervised learning models average errors across the training set but in the case of real-time detection and process killing, a *single* false positive on a benign process amongst 300 true-negatives would cause disruption to the user. The time at which an malware is detected is also important, the earlier the better. Therefore, the supervised learning model needs to be adapted to take account of these new requirements.

Tackling this issue was attempted in three different ways and all three are reported here in the interests of reporting negative results as well as the one which performed the best. These were:

(1) Statistical methods to smooth the alert surface and filter out single false-positives

(2) Reinforcement learning, which is capable of incorporating the consequences of model actions into learning

(3) A regression model based on the feedback of a reinforcement learning model made possible by having the ground-truth labels

Figure 1 gives a high-level depiction of the three approaches tested in this paper.

3.1. Statistical Approach: Alert Filtering. It is expected that transitioning from a supervised learning model to a real-time model will see a rise in false-positives since one single alert means benign processes (and all child processes) are terminated, which effectively renders all future data points as false positives. Filtering the output of the models, just as the human brain filters out transient electrical impulses in order to separate background noise from relevant data [26], may be sufficient to make supervised models into suitable agents. This is attractive because supervised learning models are already known to perform well for malware detection, as confirmed by the previous paper and other related work [11, 20, 27, 28]. A disadvantage of this approach is that it introduces additional memory and computational requirements both in order to calculate the filtered results and to track current and historic scores; therefore, a model which integrates the expected consequences of an action into learning is also tested: reinforcement learning.

3.2. Reinforcement Learning: Q-Learning with Deep Q Networks. The proposed automated killing model may be better suited to a reinforcement learning strategy than to

Table 1: Real-time malware detection literature problems considered.

	Problem considered										
Ref.	(1) Signal separation	(2) Early detection	(3) Quick classification/ latency	(4) Impact of automated actions	Resource consumption	Real-time tested	Malware types	OS	# Samples	Features	Algorithm
[19]			X		X		General	Linux	200	HPCs	Boosted DT
[15]			X		X		General	Linux	200	HPCs	Boosted DT
[20]		X	X		X	X	General	Linux	798	API calls	MLP
[21]		X			X	X	General	Windows	1,554	Memory addresses, instructions	NN
[22]			X			X	General	Windows, linux	500	API calls	NN
[23]	X	X	X			X	General	Windows	9,992	API calls	RF + NN
[24]		X	X	X		X	Crypto ransomware	Windows	497	File data	Rules

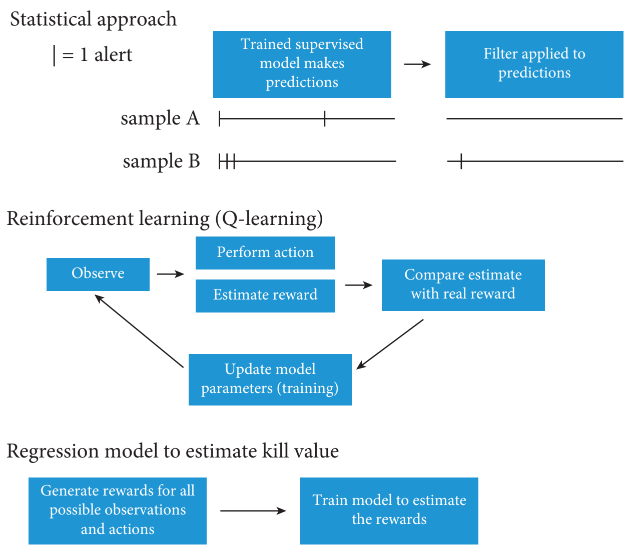
Statistical approach
\| = 1 alert
Trained supervised model makes predictions
Filter applied to predictions
sample A
sample B
Reinforcement learning (Q-learning)
Observe
Perform action
Estimate reward
Compare estimate with real reward
Update model parameters (training)
Regression model to estimate kill value
Generate rewards for all possible observations and actions
Train model to estimate the rewards

Figure 1: High-level depiction of three approaches taken.

supervised learning. Reinforcement learning uses rewards and penalties from the model's environment. The problem that this paper is seeking to solve is essentially a supervised learning problem, but one for which it is not possible to average predictions. There are no opportunities to classify the latter stages of a process if the agent kills the process, and this can be reflected by the reward mechanism of the reinforcement learning model (see Figure 1). Therefore, reinforcement learning seems like a good candidate for this problem space.

Two limitations of this approach are that (1) reinforcement learning models can struggle to converge on a balanced solution, and the models must learn to balance the exploration of new actions with the re-use of known high-reward actions; commonly known as the exploration-exploitation trade-off [29] (2) in these experiments, the reward is based on the malware/benignware label at the application level rather than being linked to the actual damage being caused; therefore, the signal is a proxy for what the model should be learning. This is used because, as discussed above, the damage caused by different malware is subjective.

For reinforcement learning, loss functions are replaced by reward functions which update the neural network weights to reinforce actions (in context) that lead to higher rewards and discourage actions (in context) that lead to lower rewards; these contexts and actions are known as state-action pairs. Typically, the reward is calculated from the perceived value of the new state that the action leads to, e.g., points scored in a game. Often this cannot be pre-labelled by a researcher since there are so many (maybe infinite) state-action pairs. However, in this case, all possible state-action pairs can be enumerated, which is the third approach tested (regression model, outlined in the next section).

The reinforcement model was still tested. Here the reward is $+N$ for a correct prediction, $-N$ for an incorrect prediction where N is the total number of processes impacted by the prediction. For e.g., if there is only one process in a process tree but 5 more will appear over the course of execution, a correct prediction gives a reward of +6, and incorrect prediction gives a reward of −6.

There are a number of reinforcement learning algorithms to choose from. This paper explores q-learning [30–33] to approximate the value or "quality" (q) of a given action in a given situation. Q-learning approximates q-tables, which are look-up tables of every state-action pair and their associated rewards. A state-action pair is a particular state in the environment coupled with a particular action, i.e., the machine metrics of the process at a given point in time with the action to leave the process running. When the number of state-action pairs becomes quite large, it is easier to approximate the value using an algorithm. Deep Q networks (DQN) are neural networks that implement q-learning and have been used in state-of-the-art reinforcement learning arcade game playing, see Mnih et al. [34]. A DQN was the reinforcement algorithm trialled here; although it did not perform well by comparison with the other methods, a different RL algorithm may perform better [35], but the results are still included in the interests of future work. The following paragraphs will explain some of the key features of the DQN.

The DQN tries out some actions; stores the states, actions, resulting states, and rewards in memory; and uses these to learn the expected rewards of each available action, with the highest expected reward being the one that is chosen. Neural networks are well-suited to this problem since their parameters can easily be updated, tree-based algorithms like random forests and decision trees can be adapted to this end but not as easily. Future rewards can be built into the reward function and are usually discounted according to a tuned parameter usually signified by γ.

In Mnih et al.'s [34] formulation, in order to address the exploration-exploitation trade off, DQNs either exploit a known action or explore a new one, with the chance of choosing exploration falling over time. When retraining the model based on new experiences, there is a risk that previous useful learned behaviours are lost; this problem is known as catastrophic forgetting [36]. Mnih et al.'s [34] DQNs use two tools to combat this problem. First, experience replay by which past state-action pairs are shuffled before being used for retraining so that the model does not catastrophically forget. Second, DQNs utilise a second network, which updates at infrequent intervals in order to stabilise the learning.

Q-learning may enable a model to learn when it is confident enough to kill a process, using the discounted future rewards. For example, choosing not to kill some malware at time t may have some benefit as it allows the model to see more behaviour at $t+1$ which gives the model greater confidence that the process is in fact malicious.

Q-learning approximates rewards from experience, but in this case, all rewards from state-action pairs can actually be pre-calculated. Since one of the actions will kill the process and thus end the "experience" of the DQN, it could be difficult for this model to gain enough experience. Thus pre-calculation of rewards may improve the breadth of experience of the model. For this reason, a regression model is proposed to predict the Q-value of a given action.

3.3. Regression Using Q-Values. Unlike classification problems, regression problems can predict a continuous value rather than discrete (or probabilistic) values relating to a set of output classes. Regression algorithms are proposed here to predict the q-value of killing a process. If this value is positive, the process is killed.

Q-values estimate the value of a particular action based on the "experience" of the agent. Since the optimal action for the agent is always known, it is possible to precompute the "(q-) value" of killing a process and train various ML models to learn this value. It would typically be quicker to train a regression model which tries to learn the value of killing a process than to train a DQN which explores the state-action space and calculates rewards between learning, since the interaction and calculation of rewards is no longer necessary. The regression approach can be used with any machine learning algorithm capable of learning a regression problem, regardless of whether it is capable of partial training.

There are two primary differences between this regression approach and the reinforcement learning DQN approach detailed in the previous section. Firstly, the datasets are likely to be the difference. Since the DQN generates training data through interacting with its environment, it may never see certain parts of the state-action space, e.g., if a particular process A is always killed during training before time $t*$, the model is not able to learn from the process A data after $t*$.

Secondly, *only* the expected value of killing is modelled by the regressor, whereas the DQN tries to predict the value of both killing and of not killing the process. This means that the equation used to model the value of process killing is only an approximation of the reward function used by the DQN.

The equation used to calculate the value of killing is positive for malware and negative for benignware; in both cases, it is scaled by the number of child processes impacted and in the case of malware, early detection increases the value of process killing (with an exponential decay). Let y be the true label of the process (0 = benign, 1 = malicious), N is the number of child processes, and t is the time in seconds at which the process is killed; then, the value of killing a process is:

$$(y*2-1)*(1+N)*\left(1+\left(y*\left(e^{-t}\right)\right)\right). \tag{1}$$

The equation above negatively scores the killing of benignware in proportion to the number of subprocesses and scores the killing of malware positively in proportion to the number of subprocesses. A bonus reward is scored for killing malware early, with an exponential decay over time.

4. Evaluation Methodology: Ransomware Detection

Previous, research has not addressed the extent to which damage is mitigated by process killing, since Sun et al. [23] presented the only previous work to test process killing and damage with and without process killing is not assessed. To this end, this paper uses ransomware as a proxy to detect malicious damage, inspired by Scaife et al.'s approach [24]. A brief overview of Scaife et al.'s damage measurement is outlined below:

Early detection is particularly useful for types of malware from which recovery is difficult and/or costly. Cryptographic ransomware encrypts user files and withholds the decryption key until a ransom is paid to the attackers. This type of attack is typically costly to remedy, even if the victim is able to carry out data recovery [37]. Scaife et al.'s work [24] on ransomware detection uses features from file system data, such as whether the contents appear to have been encrypted, and number of changes made to the file type. The authors were able to detect and block all of the 492 ransomware samples tested with less than 33% of user data being lost in each instance. Continella et al. [38] propose a self-healing system, which detects malware using file system machine activity (such as read/write file counts); the authors were able to detect all 305 ransomware samples tested, with a very low false-positive rate. These two approaches use features selected specifically for their ability to detect ransomware, but this requires knowledge of how the malware operates, whereas the approach taken here seeks to use features which can be used to detect malware *in general*. The key purpose of this final experiment (Section 6.5) is to show that our general model of malware detection is able to detect general types of malware as well as time-critical samples such as ransomware.

5. Experimental Setup

This section outlines the data capture process and dataset statistics.

5.1. Features. The same features as were used in previous work [11] are used here for process detection, with some additional features to measure process-specific data. Despite the popularity of API calls noted in Ref. [18], due to these findings and Sun et al.'s [23] difficulties hooking this data in real-time, these were not considered as features to train the model.

At the process-level, 26 machine metric features are collected; these were dictated by the attributes available using the Psutil [39] python library. It is also possible to include the "global" machine learning metrics that were used in the previous papers. Although global metrics will not provide process-level granularity, they may give muffled indications of the activity of a wider process tree. The 9 global metrics are: system-level CPU use, user-level CPU use, memory use, swap memory use, number of packets received and sent, number of bytes received and sent, and the total number of processes running.

The process-level machine activity metrics collected are: CPU use at the user level, CPU use at the system level, physical memory use, swap memory use, total memory use, number of child process, number of threads, maximum process ID from a child process, disk read, write and other I/O count, bytes read, written and used in other I/O processes, process priority, I/O process priority, number of command line arguments passed to process, number of handles being used by process, time since the process began, TCP packet count, UDP packet count, number of connections currently open, and 4 port statuses of those opened by the process (see Table 2).

5.1.1. Preprocessing. Feature normalisation is necessary for NNs to avoid over-weighting features with higher absolute values. The test, train, and validation sets (x) are all normalised by subtracting the mean (μ) and dividing by the standard deviation (σ) of each feature in the training set: $x - \mu/\sigma$. This sets the range of input values largely between -1 and 1 for all input features, avoiding the potential for some features to be weighted more important than others during training purely due to the scalar values of those features. This requires additional computational resources but is not necessary for all ML algorithms; this is another reason why the supervised RNN used in Ref. [11] may not be well-suited for real-time detection.

5.2. Data Capture. During data capture, this research sought to improve upon previous work and emulate real machine use to a greater extent than has previously been trialled. The implementation details of the VM, simultaneous process execution, and RL simulation are outlined below:

5.2.1. Environment: Machine Setup. The following experiments were conducted using a virtual machine (VM) running with Cuckoo Sandbox [40] for ease of collecting data and restarting between experiments and because the Cuckoo Sandbox emulates human interaction with programs to some extent to promote software activity. In order to emulate the capabilities of a typical machine, the modal hardware attributes of the top 10 “best seller” laptops according to a popular Internet vendor [41] were used, and these attributes were the basis of the VM configuration. This resulted in a VM with 4GB RAM, 128GB storage, and dual-core processing running Windows 7 64 bit. Windows 7 was the most prevalent computer operating system (OS) globally at the time of designing the experiment [25]. Although Windows 10 is now the most popular OS, the findings in this research should still be relevant.

5.2.2. Simultaneous Applications. In typical machine use, multiple applications run simultaneously. This is not reflected by behavioural malware analysis research in which samples are injected individually to a virtual machine for observation. The environment used for the following experiments launches multiple applications on the same machine at slightly staggered intervals as if a user were opening them. Each malware is launched with a small number (1–3) and a larger number (3–35) of applications. It was not possible to find up-to-date user data on the number of simultaneous applications running on a typical desktop, so here it was elected to launch up to 36 applications (35 benign + 1 malicious) at once, which is the largest number of simultaneous apps for real-time data collection to date. From the existing real-time analysis literature, only Sun et al. [23] run multiple applications at the same time, with a maximum of 5 running simultaneously.

Each application may in turn launch multiple processes, causing more than 35 processes to run at once; 95 is the largest number of simultaneous processes recorded; this excludes background OS processes.

5.2.3. Reinforcement Learning Simulation. For reinforcement learning, the DQN requires an observation of the resulting state following an action. To train the model, a simulated environment is created from the pre-collected training data whereby the impact of killing or not killing a process is returned as the next state. For process-level elements, this reduces all features to zero. A caveat here is that in reality, killing the process may not occur immediately and therefore memory, processing power, etc., may still be being consumed at the next data observation. For global metrics, the process-level values for the killed processes (includes child processes of the killed process) are subtracted from the global metrics. There is a risk again that this calculation may not correlate perfectly with what would be observed in a live machine environment.

In order to observe the model performance, a visualisation was developed to accompany the simulated environment. Figures 2 and 3 show screenshots of the environment visualisation for one malicious and one benign process.

5.3. Dataset. The dataset comprises 3,604 benign executables and 2,792 malicious applications (each containing at least one executable), with 2,877 for training and validation and 3,519 for testing. These dataset sizes are consistent with previous real-time detection dataset sizes (Das et al. [20] use 168 malicious, 370 benign; Sayadi et al. [15] use over 100 each benign and malicious; Ozsoy et al. [21] use 1,087 malicious and 467 benign; Sun et al. [23] use 9,115 malicious, 877 benign). With multiple samples running concurrently to simulate real endpoint use, there are 24K processes in the training set and 34K in the test set. Overall, there are 58K behavioural traces of processes in the training and testing datasets. The benign samples comprise files from VirusTotal [42], from free software websites (later verified as benign with VirusTotal), and from a fresh Microsoft Windows 7 installation. The malicious samples were collected from two different VirusShare [43] repositories.

In Pendelbury et al.’s analysis [13], the authors estimate that in the wild between 6% and 22% of applications are malicious, normalising to 10% for their experiments. Using this estimation of Android malware, a similar ratio was used in the test set in which 13.5% were malicious.

5.3.1. Malware Families. PUA = potentially unwanted application, RAT = remote access trojan.

This paper is not concerned with distinguishing particular malware families, but rather with identifying malware in general. However, a dataset consisting of just one malware

Table 2: 26 process-level features: 22 features + 4 port status values.

Category			
CPU use (%)	System level	User level	
Memory use (bytes)	Total	Physical (nonswapped)	Swap
Child processes	Count	Maximum process ID	Number of threads
I/O operation bytes on disk (bytes)	Read	Write	Nonread-write I/O operations
I/O operation count on disk	Read	Write	Nonread-write I/O operations
Priority	Process priority	I/O process priority	
Network # packets	TCP packet count	UDP packet count	
Network # bytes	# Bytes sent	# Bytes received	
Network other	Number of connections currently open	Statuses of the ports opened by the process (4 statuses)	
Miscellaneous	Number of command line arguments passed to process	Number of handles being used by process	

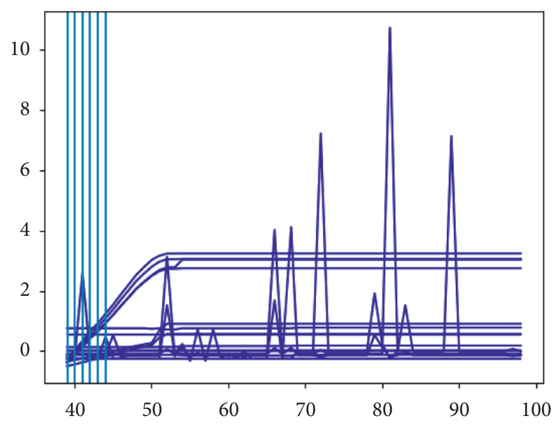
Figure 2: Benignware sample, normalised process-level metrics, 6 observations made without process being killed.

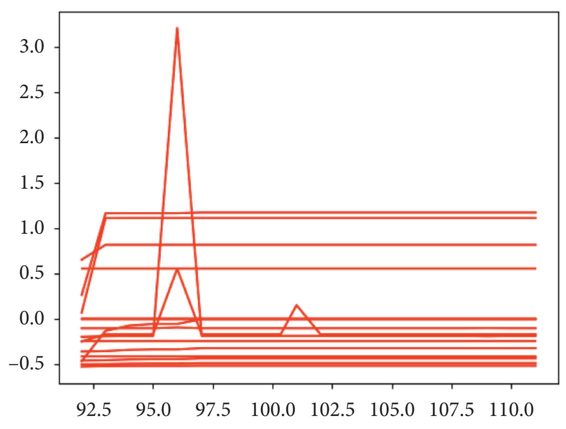
Figure 3: Malware sample, normalised process-level metrics, no observations made yet.

family would present an unrealistic and easier problem than is found in the real world. The malware families included in this dataset are reported in Table 3. The malware family labels are derived from the output of around 60 antivirus engines used by VirusTotal [42].

Ascribing family labels to malware is nontrivial since antivirus vendors do not follow standardised naming conventions and many malware families have multiple aliases. Sebastián et al. [44] have developed an open source tool, AVClass, to extract meaningful labels and correlate aliases between different antivirus outputs. AVClass was used to label the malware in this dataset. Sometimes there is no consensus amongst the antivirus' output or the sample is not recognised as a member of an existing family. AVClass also excludes malware that belongs to very broad classes of malware (e.g., "agent," "eldorado," and "artemis") as these are likely to comprise a wide range of behaviours and so may be applied as a default label in cases for which antivirus engines are unsure. In the dataset established in this research, 2,121 of the 2,792 samples were assigned to a malware family. Table 3 gives the number of samples in each family for which more than 10 instances were found in the dataset. 315 families were detected overall, with 27 families being represented more than 10 times. These better-represented families persist in the train and test sets, but the other families have little overlap. 104 of the 154 other families seen in the test set are not identified by AVClass as being in the training set.

5.3.2. Malicious Vs. Benign Behaviour. Statistical inspection of the training set reveals that benign applications have fewer sub-processes than malicious processes, with 1.17 processes in the average benign process tree and 2.33 processes in the average malicious process tree. Malware was also more likely to spawn processes outside of the process tree of the root process, often using the names of legitimate Windows processes. In some cases, malware launches legitimate applications, such as Microsoft Excel in order to carry out a macro-based exploit. Although Excel is not a malicious application in itself, it is malicious in this context, which is why malicious labels are assigned if a malware sample has caused that process to come into being. It is therefore possible to argue that some processes launched by malware are not malicious, because they do not individually cause harm to the endpoint or user, but without the malware they would not be running and so can be considered at least undesirable even if only in the interests of conserving computational resources.

Table 3: Malware families with more than 10 samples in the dataset. 315 families were represented in the dataset, with 27 having being represented more than 10 times. Basic description provided which does not cover the wide range of behaviours carried out by some malware families but is intended to indicate the range of behaviours in the top 27 families included in the dataset.

Malware family	# Train set	# Test set	Total	Description
Startsurf	66	273	339	Adware
Fareit	33	222	255	Spyware
Vigram	23	212	235	Adware
Winwrapper	78	8	86	PUA
Downloadguide	15	59	74	Adware
Gandcrab	5	54	59	Ransomware
Emotet	12	46	58	Credstealer
Chapak	4	37	41	Installer
Virut	30	2	32	Backdoor
Installmonster	12	18	30	Installer
Noon	8	22	30	Spyware
Gamarue	11	18	29	Backdoor
Razy	7	16	23	Crypto stealer
Zeroaccess	23	0	23	Rootkit
Soft32downloader	5	22	23	Installer
Appster	7	15	22	PUA
Prepscram	1	20	21	Installer
Zusy	2	19	21	Spyware
Darkkomet	17	1	18	RAT
Adposhel	4	14	16	Adware
Swrort	13	0	13	Backdoor
Slugin	13	0	13	Installer
Vobfus	11	2	13	Installer
Speedingupmypc	1	11	12	Adware
Relevantknowledge	5	6	11	Adware
Kuaizip	4	7	11	PUA
Bladabindi	7	4	11	Backdoor
Other (≤10 instances)	377	260	602	—
# Other families (≤10 instances)	184	154	288	—
Unknown	333	291	671	—
Total	**1,137**	**1,655**	**2,792**	—

5.3.3. Train-Test Split. The dataset is split in half with the malicious samples in the test set coming from the more recent VirusShare repository, and those in the training set from the earlier repository. This is to increase the chances of simulating a real deployment scenario in which the malware tested contains new functionality by comparison with those in the training set.

Ideally, the benignware should also be split by date across the training and test set; however, it is not a trivial task to calculate the date at which benignware was compiled. It is possible to extract the compile time from PE header, but it is possible for the PE author to manually input this date which had clearly happened in some instances where the compile date was 1970-01-01 or in one instance 1970-01-16. In the latter case (1970-01-16), the file is first mentioned online in 2016, perhaps indicating a typographic error [45]. Using Internet sources such as VirusTotal [42] can give an indication when software was first seen, but if the file is not very suspicious, i.e., from a reputable source, it may not have been uploaded until years after it was first seen "in the wild." Due to the difficulty in dating benignware in the dataset collected for this research, samples were assigned to the training or test set randomly.

For training, an equal number of benign and malicious processes are selected, so that the model does not bias towards one class. 10% of these are held out for validation. In most ML model evaluations, the validation set would be drawn from the same distribution as the test set. However, because it is important not to leak any information about the malware in the test set, since it is split by date, the validation set here is drawn from the training distribution.

5.3.4. Implementation Tools. Data collection used the Psutil [39] Python library to collect machine activity data for running processes and to kill those processes deemed malicious. The RNN and Random Forests were implemented using the Pytorch [46] and Scikit-Learn [47] Python libraries, respectively. The model runs with high priority and administrator rights to make sure the polling is maintained when compute resources are scarce.

6. Experimental Results

6.1. Supervised Learning for Process Killing. First, we demonstrate the unsuitability of a full-trace supervised learning malware detection model, which achieved more than 96% detection accuracy in Ref. [11]. The model used is a gated-recurrent unit recurrent neural network since this algorithm is designed to process time-series data. The hyperparameter configuration of this model was conducted using a random search of hyperparameters (see Table 4 in the Appendix for details.)

It is expected that supervised malware detection models will not adapt well to process-killing due to the averaging of loss metrics as described earlier. Initially, this is verified by using supervised learning models to kill processes that are deemed malicious. For supervised classification, the model makes a prediction every time a data measurement is taken from a process. This approach is compared with one taking average predictions across all measurements for a process and for a process tree as well as the result of process killing. The models with the highest validation accuracy for classification and killing are compared.

Figure 4 illustrates the difference in validation set and test set F1-score, true-positive rate, and false-positive rate for these 4 levels of classification: each measurement, each process, each process tree, and finally showing process killing; see Figure 5 for diagrammatic representation of these first 3 levels. Table 5 reports the F1, TPR, and TNR for classification (each measurement of each process) and for process killing.

The highest F1-score on the validation set is achieved by an RNN using process data only. When process killing is applied, there is a drop of less than 5 percentage points in the F1-score, but more than 15 percentage points are lost from the TNR.

On the unseen test set, the highest F1-score is achieved by an RNN using process data + global metrics, but the improvement over the process data + total number of processes is negligible. Overall, there is a reduction in F1-score from (97.44, 94.61) to (74.91, 77.66), highlighting the initial challenge of learning to classifying individual processes rather than entire applications, especially when accounting for concept drift. Despite the low accuracy, these initial results indicate that the model is discriminating some of the samples correctly and may form a baseline from which to improve.

The test set TNR and TPR for classification on the best-performing model (process data only) are 79.70 and 82.91, respectively, but when process killing is applied, although the F1-score drops by 10 percentage points, the TNR and TPR move in opposite directions with the TNR falling to 59.63 and TPR increasing to 90.24. This is not surprising since a single malicious classification results in a process being classed as malicious. This is true for the best-performing models using either of the two feature sets (see Figure 4).

6.2. Accuracy Vs. Resource Consumption.

Previous work on real-time detection has highlighted the requirement for a lightweight model (speed and computational resources). In the previous paper, RNNs were the best performing algorithm in classifying malware/benignware, but RNNs have many parameters and therefore may consume significant RAM and/or CPU. They also require preprocessing of the data to scale the values, which other ML algorithms such as tree-based algorithms do not. Whilst RAM and CPU should be minimised, taking model accuracy into account, inference duration is also an important metric.

Although the models in this paper have not been coded for performance and use common python libraries, comparing these metrics helps to decide whether certain models are vastly preferable to others with respect to computational resource consumption. The PyRAPL library [49] is used measure the CPU, RAM, and duration used by each model. This library uses Intel processor "Running Average Power Limit" (RAPL) metrics. Only data preprocessing and inference is measured as training may be conducted centrally in a resource-rich environment. Batch sizes of 1, 10, 100, and 1000 samples are tested with 26 and 37 features, respectively, since there are 26 process-level features and 37 when global metrics are included. Each model is run 100 times for each of the different batch sizes.

For the RNN, a "large" and a "small" model are included. The large models have the highest number of parameters tested in the random search (981 hidden neurons, 3 hidden layers, sequence length of 17) and the smallest (41 neurons, 1 hidden layer, sequence length of 13). These two RNN configurations are compared against other machine learning models which have been used for malware detection: Multi-Layer Perceptron (feed-forward neural network), Support Vector Machine, Naive Bayes Classifier, Decision Tree Classifier, Gradient Boosted Decision Tree Classifier (GBDTs), Random Forest, and AdaBoost.

26 features = process-level only, 37 features = machine and process level features

Table 6 reports the computational resource consumption and accuracy metrics together. Decision tree with 38 features is the lowest cost to run, RNN performs best at supervised learning classification on the validation set but only just outperforms the decision tree with 26 features, which is the best performing model at process killing on the validation set at 92.97 F1-score. The highest F1-score for process killing uses a Random Forest with 37 features, scoring 77.85 F1, which is 2 percentage points higher than the RF with 26 features (75.97). The models all perform at least 10 percentage points better on the validation set, indicating the importance of taking concept drift into account when validating models.

6.3. How to Solve a Problem like Process Killing?

From the results above, it is clear that supervised learning models see a significant drop in classification accuracy when processes are killed as the result of a malicious label. This confirmation of the initial hypothesis presented here justifies the need to examine alternative methods. In the interests of future work and negative result reporting, this paper reports all of the methods attempted and finds that simple statistical manipulations on the supervised learning models perform better than using alternative training methods. This section briefly describes the logic of each method and provides a textual summary of the results with a formula where appropriate. This is followed by a table of the numerical results for each method. In the following section, let P be a set of processes $\{p_0, p_1 \cdots p_P\}$ in a process tree, $t*$ be the time at which a prediction is made, let $\hat{y}_i$ be the prediction for process i at time $t*$ where a prediction equal to or greater than 1 classifies malware.

Table 4: Hyperparameter search space and the hyperparameters of the model giving the lowest mean false-positive and false-negative rates.

	Possible values	Process-level data	Process-level data + global metrics
Hyperparamter			
Hidden neurons	8–1024	253	193
Depth	[1–3]	2	2
Batch size	[64, 128, 256]	128	256
Epochs	1–200	89	161
Dropout rate	[0, 0.1, 0.2, 0.3, 0.4, 0.5]	0.1	0.1
Window size	2–30	16	6
Loss function		Binary cross-entropy	
Weight update rule		Adam [48]	
Recurrent unit		GRU cell	
Validation F1-score (∗100)	—	**97.43**	94.61

6.3.1. Mean Predictions. Reasoning: Taking the average prediction across the whole process will smooth out those process killing results.

Not tested. This was not attempted for two reasons: (1) Taking the mean at the end of the process means the damage is done. (2) This method can easily be manipulated by an attacker: 50 seconds of injected benign activity required 50 seconds of malicious activity to achieve a true positive

$$\hat{y}_i = \frac{1}{t*} \sum_{t=0}^{t*} \hat{y}_i^t. \tag{2}$$

6.3.2. Rolling Mean Predictions. Reasoning: Taking the average over a few measurements will eliminate those false positives that are caused by a single false positive over a subset of the execution trace. Window sizes of 2 to 5 are tested. Let w be the window size:

$$\hat{y}_i = \frac{1}{w} \sum_{t=t*-w}^{t*} \hat{y}_i^t. \tag{3}$$

Summary of results: A small but unilateral increase in F1-Score using a rolling window over 2 measurements on the validation set. Using a rolling window of size 2 on the test set saw a 10 to 20 percentage point increase in true negative rate (to a maximum of 80.77) with 3 percentage points lost from the true-positive rate. This was one of the most promising approaches.

6.3.3. Alert Threshold. Reasoning: Like the rolling mean, single false positives will be eliminated but unlike the rolling mean, the alerts are cumulative over the entire trace such that a single alert at the start and 30 seconds into the process will cause the process to be killed rather than requiring that both alerts are within a window of time. Between 2 and 5 minimum alerts are tested

$$\hat{y}_i = w - \sum_{t=0}^{t*} \hat{y}_i^t. \tag{4}$$

Summary of results: Again, a small increase across all models, with an optimal minimum number of alerts being 2 for maximum F1-score, competitive with the rolling mean approach.

6.3.4. Process-Tree Averaging. Reasoning: The data are labelled at the application level; therefore, the average predictions across the process tree should be considered for classification

$$\hat{y}_i = \frac{1}{P} \sum_{p=0}^{P} \widehat{y_i^{t*}}. \tag{5}$$

Summary of results: Negligible performance increase on validation and test set data (less than 1 percentage point). This is likely because few samples have more than one process executing simultaneously.

6.3.5. Process-Tree Training. Reasoning: The data are labelled at the application level; therefore, the sum of resources of each process tree should be classified at each measurement, not the individual processes.

Summary of results: Somewhat surprisingly, there was a slight reduction in classification accuracy when using process tree data. One explanation for this may be that the process tree creates noise around the differentiating characteristics that are visible at the process level.

6.3.6. DQN. Reasoning: Reinforcement learning is designed for state-action space learning. Both pre-training the model with a supervised learning approach and not pre-training the model were tested.

Summary of results: Poor performance, typically converging to either kill or not kill everything, of the few models that did not converge to a single dominant action; it does not distinguish malware or benignware well, indicating that it may not have learned anything. Reinforcement learning may help the problem of real-time malware detection and process killing, but this initial implementation of a DQN did not converge to a better or even competitive solution to supervised learning. Perhaps, better formulation of rewards (e.g., damage prevented) would help the agent learn.

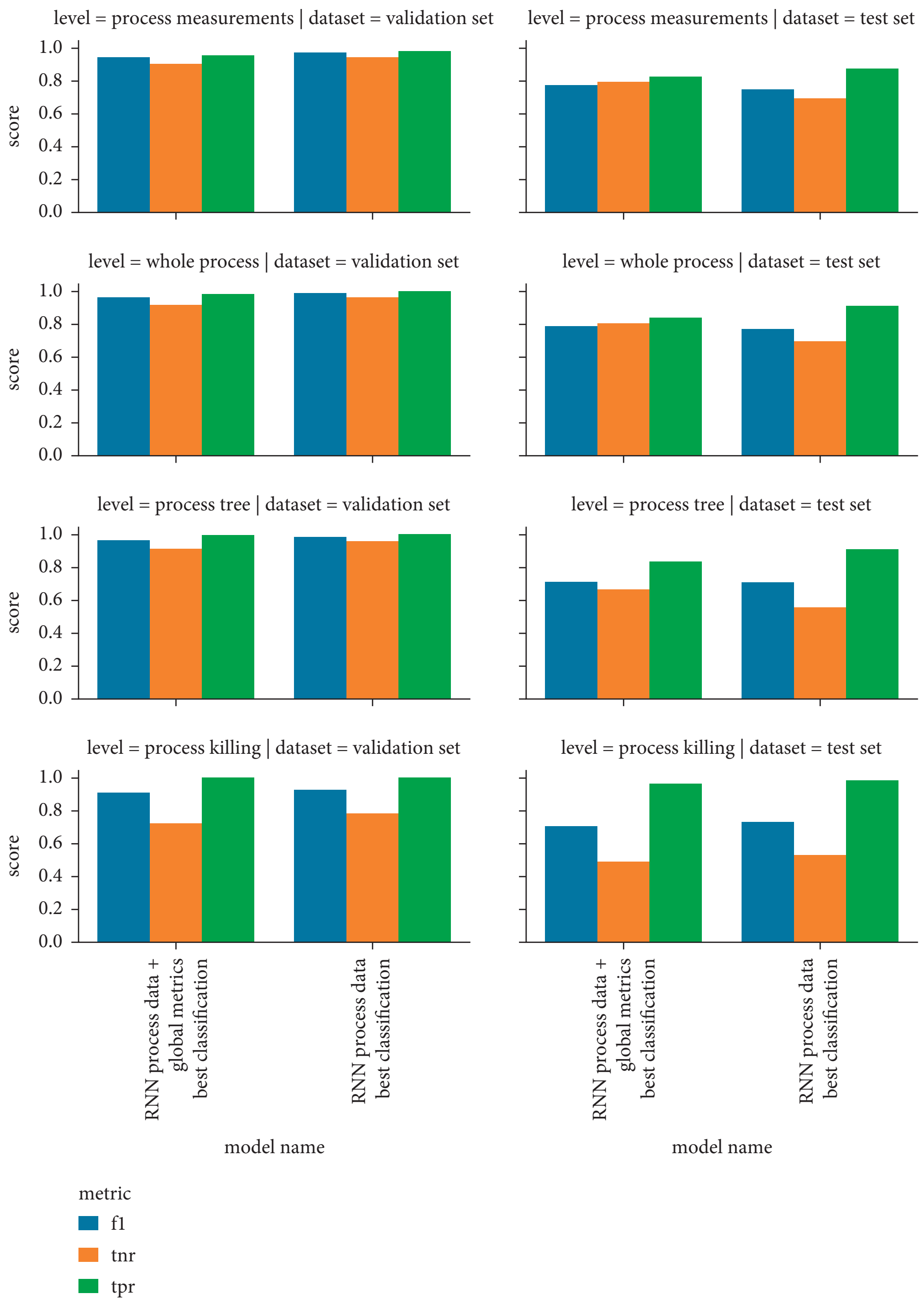

Figure 4: F1 scores, true positive rates (TPR), and true negative rates (TNR) for partial-trace detection (process measurements), full-trace detection (whole process), whole application (process tree), and with process-level measurements + process killing (process killing) for validation set (left column) and test set (right column).

6.3.7. Regression on Predicted Kill Value. Reasoning: Though the DQN explores and exploits different state-action pairs and their associated rewards, when the reward from each action is known in the first place and the training set is limited, as it is here, Q-learning can be framed as a regression problem in which the model tries to learn the return (rewards + future rewards), the training is faster and can be used by any regression-capable algorithm. Let N be the number of current and future child processes for p_i at $t*$

$$(y * 2 - 1) * (1 + N) * \left(1 + \left(y * \left(e^{-t*}\right)\right)\right). \qquad (6)$$

Summary of results: Improved performance on true negative rate, although not perceptible for the highest-scoring F1 models since F1-scores reward true positives more than true negatives, this metric can struggle to reflect a balance between the true-positive and true-negative rates. The highest true-negative rate models are all regression models.

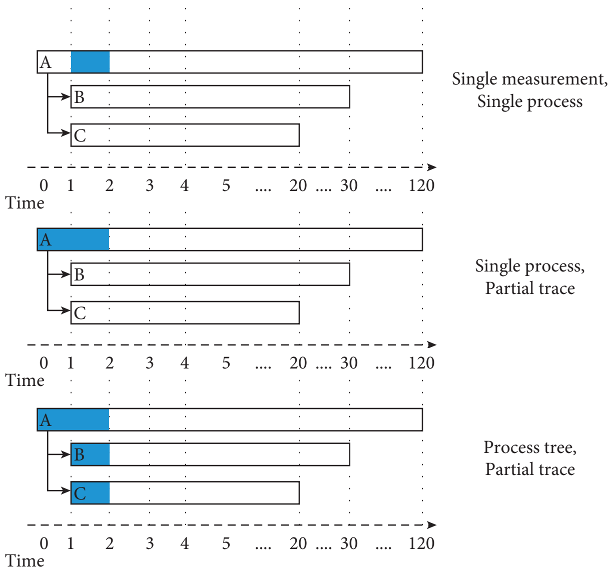
A
B
C
Single measurement, Single process
0 1 2 3 4 5 .... 20 .... 30 120
Time
Single process, Partial trace
Process tree, Partial trace

Figure 5: Three levels of data collection: each measurement, each process, each process tree.

Table 5: F1-score, true positive rate (TPR), and true negative rates (TNR) (all $*$ 100) on test and validation sets for classification and process killing.

Features	Metric	Classify	Dataset	Kill
Proc. Data	F1	**97.44**	Validation set	**91.20**
Proc. Data	tnr	94.72	Validation set	85.71
Proc. Data	tpr	98.64	Validation set	95.80
Proc. Data + glob.	F1	94.61	Validation set	87.69
Proc. Data + glob.	tnr	90.57	Validation set	77.31
Proc. Data + glob.	tpr	95.93	Validation set	95.80
Proc. Data	F1	74.91	Test set	**72.63**
Proc. Data	tnr	69.41	Test set	59.63
Proc. Data	tpr	87.52	Test set	91.82
Proc. Data + glob.	F1	**77.66**	Test set	71.83
Proc. Data + glob.	tnr	79.70	Test set	59.63
Proc. Data + glob.	tpr	82.91	Test set	90.24

Table 6: Average resource consumption over 100 iterations for a batch size of 100 vs. F1-scores on validation and test set for classification and process killing across 14 models..

Model	n features	Avg. cpu (μJ)	Avg. dram (W)	Avg. Duration (μs)	Val F1	Kill val F1	Test F1	Kill test F1
AdaBoost	26	127967.84	7981.51	6595.37	88.35	74.36	77.19	60.09
AdaBoost	37	125041.20	7142.93	6469.16	89.63	76.07	80.10	60.14
DT	26	3905.63	202.65	128.02	97.39	88.48	66.44	62.95
DT	37	**2113.67**	**134.29**	**106.65**	96.32	83.57	79.61	62.50
GBDT	26	8788.41	338.78	349.31	92.27	78.26	82.47	63.33
GBDT	37	11005.80	486.46	329.45	93.13	80.26	84.94	63.46
MLP	26	11044.88	645.14	461.04	82.84	70.18	41.62	57.65
MLP	37	12932.09	628.64	555.42	73.00	67.63	57.66	57.26
NB	26	6947.67	297.87	185.73	75.80	67.42	62.90	56.11
NB	37	5187.96	258.80	177.37	75.58	67.61	61.88	55.33
RF	26	238621.20	11052.84	8997.31	97.12	**92.97**	71.58	75.97
RF	37	236598.44	9967.63	8879.97	96.57	91.05	**85.55**	**77.85**
RNN	26	887664.31	48885.96	27869.30	**97.44**	90.70	74.91	73.08
RNN	37	312108.07	17120.90	10414.58	94.61	87.31	77.66	71.95
SVM	26	6630490.84	464082.07	282026.57	78.34	67.04	68.16	56.91
SVM	37	7792179.78	730786.06	429081.31	64.89	65.68	61.39	56.25

Table 7 lists the F1, TPR, and TNR on the validation and test set for each of the methods described above. The best-performing model on the test and validation sets is reported and the full results can be found in Appendix Table 8–10. Small improvements are made by some models on the validation F1-score, but the test set F1-score improves by 4 percentage points in the best instance.

In most cases, the models with the highest F1-score on the validation and test sets are not the same. The highest F1-score is 81.50 from an RF using a minimum alert threshold of 2 and both process-level and global process metrics.

6.4. Further Experiment: Favouring High TNR. Although the proposed model is motivated by the desire to prevent malware from executing, the best TNR reported amongst the models above is 81.50%. 20% of benign processes being killed would not be acceptable to a user. Whilst this research is a novel attempt at very early-stage real-time malware detection and process killing, one might consider the usability and prefer a model with a very high TNR, even if this is at the expense of the TPR.

Considering this, the AdaBoost regression algorithm achieves a 100% TNR with a 39.50% TPR on the validation set. The high FNR is retained in the test set standing at 97.92%, but the TPR drops even further to just 8.40%. The GBDT also uses regression to estimate the value of process killing and coupled with a minimum of 4 alerts performs well on the test set but does not stand out in the validation set, see Table 11.

Although less than 10% of the test set malicious processes is killed by the AdaBoost regressor, this model may be the most viable despite the low TPR. Future work may examine the precise behaviour and harm caused by malware that is/is not detected. To summarise results, the most-detected families were Ekstak (180), Mikey (80), Prepscram (53 processes), and Zusy (49 processes) of the 745 total samples.

6.5. Measuring Damage Prevention in Real Time. Although a high percentage of processes are correctly identified as malicious by the best performing model (RF with 2 alerts and 37 features), it may be that the model detects the malware after it has already caused damage to the endpoint. Therefore, instead of looking at the time at which the malware is correctly detected, a live test was carried out with ransomware to measure the percentage of files corrupted with and without the process killing model working. This real-time test also assesses whether malware can indeed be detected in the early stages of execution or whether the data recording, model inference, and process killing is too slow in practice to prevent damage.

Ransomware is the broad term given to malware that prevents access to user data (often by encrypting files) and holds the means for restoring the data (usually a decryption key) from the user until a ransom is paid. It is possible to quantify the damage caused by ransomware using the proportion of modified files as Scaife et al. [24] have done in developing a real-time ransomware (only) detection system. The damage of some malware types are more difficult to quantify owing to their dependence on factors outside the control of the malware. For example, the damage caused by spyware will depend on what information it is able to obtain, so it is difficult to quantify the benefit of killing spyware 5 seconds after execution compared with 5 minutes into execution. Ransomware offers a clear metric for the benefits of early detection and process killing.

Although the RF with a minimum of 2 alerts using both process and global data gave the highest F1-score on the test set (81.50), earlier experiments showed that RFs are not one of the most computationally efficient models by comparison with those tested. Therefore, a decision tree is trained on process-only data (26 features) in case the time-to-classification is important for damage reduction despite the lower F1-score. For this reason, the decision tree model is used in this test. The DT also has a very slightly higher TPR (see Table 12) so a higher damage prevention rate may be partially due to the model itself rather than just the fewer features being collected and model classification speed.

22 fast-acting ransomware files were identified from a separate VirusShare [43] repository which *(i)* do not require Internet connection and *(ii)* begin encrypting files within the first few seconds of execution. The former condition is set because the malicious server may no longer exist and for safety, it is not desirable to connect to it if it does exist. Some malware is able to cause significant damage in seconds, in which the timeframes are impossible for a human to see, process, react to, and alert in.

The 22 samples were executed for 30 seconds each without the process killing model and the number of files modified was recorded. The process was repeated with 4 process killing models: DT with min. 2 alerts and 26 features, RF with min. 2 alerts and 37 features, AdaBoost regressor with 26 features, and GDBT regressor with min. 4 alerts and 26 features.

It was necessary to run the killing model with administrator privileges and to write an exception for the Cuckoo sandbox agent process which enables the host machine to read data from the guest machine since the models killed this process. The need for this exception highlights that there are benign applications with malicious-like behaviours, perhaps especially those used for networking and security.

Figure 6 and Table 13 give the total number of corrupted files across the 22 samples. The damage prevention column is a proxy metric denoting how many files were not corrupted using a given process killing model by comparison with no model being in place. The 22 samples on average each corrupt 910 files within 30 seconds.

The DT model almost entirely eliminates any file corruption with only three being corrupted. The RF saves 92.68% of files. The ordinal ranking of "damage prevention" is the same as the TPR on the test set, but the relationship is not proportional. The same ordinal relationship indicates that the simulated impact of process killing on the collected test set was perhaps a reasonable approximation of measuring at least fast-acting ransomware damage, despite the TPR test set metrics being based on other malware families, too.

Table 7: Summary of the best process killing models by model training methodology. F1, TNR, and TPR for validation and test datasets (full results in Appendix Tables 8–10).

Methodology	Best dataset	Model		Val			Test		
			n features	F1	tnr	tpr	F1	tnr	tpr
Supervised learning	Val	RF	26	92.37	87.39	96.64	74.57	62.71	92.95
	Test	RF	37	89.68	83.19	94.96	76.43	67.19	92.52
Rolling mean	Val	RF (min: 2)	26	**93.22**	94.12	92.44	78.26	73.83	89.76
	Test	RF (min: 2)	37	92.70	94.96	90.76	80.77	78.88	89.38
Alert threshold	Val	DT (min: 2)	26	92.17	95.80	89.08	73.43	67.44	86.56
	Test	RF (min: 2)	37	91.30	94.96	88.24	**81.50**	81.53	87.97
Process tree averaging	Val	RF	26	92.74	88.24	96.64	74.79	64.04	92.20
	Test	RF	37	90.48	84.03	95.80	76.34	67.66	91.92
Process tree training	Val	RF	26	90.35	82.58	98.32	74.20	52.44	92.74
	Test	RF	26	90.35	82.58	98.32	74.20	52.44	92.74
Q-learning	Val	DQN	26	51.71	72.27	44.54	27.74	55.50	26.94
	Test	DQN	26	51.71	72.27	44.54	27.74	55.50	26.94
Regression	Val	RF	26	91.94	87.39	95.80	74.77	66.05	90.35
	Test	RF	26	91.94	87.39	95.80	74.77	66.05	90.35

Table 8: Summary of process killing models, validation and test set score metrics [Table 1 of 3].

Model		Val			Test	
	f1	tnr	tpr	f1	tnr	tpr
AdaBoostModel_glo_pro	77.58	55.46	91.60	67.04	49.80	88.67
AdaBoostModel_glo_pro mean process tree	78.01	55.46	92.44	66.75	50.48	87.59
AdaBoostModel_glo_pro process tree min alerts: 1	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 2	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 3	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro process tree min alerts: 4	78.87	55.46	94.12	62.29	34.03	90.35
AdaBoostModel_glo_pro rolling mean window: 2	79.22	70.59	84.87	69.57	60.88	84.88
AdaBoostModel_glo_pro rolling mean window: 3	79.37	72.27	84.03	69.59	61.53	84.39
AdaBoostModel_glo_pro rolling mean window: 4	80.67	80.67	80.67	68.44	67.80	77.34
AdaBoostModel_glo_pro sum alerts min: 2	80.66	78.15	82.35	69.35	66.58	79.89
AdaBoostModel_glo_pro sum alerts min: 3	81.20	83.19	79.83	67.83	70.92	73.88
AdaBoostModel_glo_pro sum alerts min: 4	80.87	84.87	78.15	65.92	73.32	69.00
AdaBoostModel_pro	75.34	47.06	92.44	65.64	45.79	88.89
AdaBoostModel_pro mean process tree	75.86	48.74	92.44	65.74	47.83	87.59
AdaBoostModel_pro process tree min alerts: 1	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 2	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 3	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro process tree min alerts: 4	75.68	45.38	94.12	60.31	26.46	91.17
AdaBoostModel_pro rolling mean window: 2	78.03	64.71	86.55	69.35	59.63	85.47
AdaBoostModel_pro rolling mean window: 3	77.99	67.23	84.87	68.91	59.20	84.99
AdaBoostModel_pro rolling mean window: 4	80.83	79.83	81.51	69.01	66.44	79.40
AdaBoostModel_pro sum alerts min: 2	81.12	75.63	84.87	67.91	61.06	81.68
AdaBoostModel_pro sum alerts min: 3	81.17	80.67	81.51	66.64	64.97	76.42
AdaBoostModel_pro sum alerts min: 4	79.66	80.67	78.99	64.25	68.12	70.14
AdaBoostModel_pro_tree	75.08	47.73	94.96	64.12	30.58	86.60
AdaBoostRegression_pro_process	56.63	100.00	39.50	15.06	97.92	8.40
DTModel_glo_pro	84.53	71.43	94.12	71.41	58.19	90.62
DTModel_glo_pro mean process tree	85.93	73.95	94.96	71.49	59.48	89.70
DTModel_glo_pro process tree min alerts: 1	84.64	70.59	94.96	65.59	42.42	91.27
DTModel_glo_pro process tree min alerts: 2	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro process tree min alerts: 3	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro process tree min alerts: 4	84.64	70.59	94.96	65.56	42.34	91.27
DTModel_glo_pro rolling mean window: 2	88.70	88.24	89.08	75.09	70.49	86.94
DTModel_glo_pro rolling mean window: 3	87.87	87.39	88.24	74.57	69.77	86.61
DTModel_glo_pro rolling mean window: 4	89.08	93.28	85.71	74.04	74.29	81.63
DTModel_glo_pro sum alerts min: 2	89.74	91.60	88.24	75.48	72.64	85.69
DTModel_glo_pro sum alerts min: 3	89.47	94.12	85.71	74.00	75.62	80.38

Table 8: Continued.

Model	Val			Test		
	f1	tnr	tpr	f1	tnr	tpr
DTModel_glo_pro sum alerts min: 4	88.39	94.96	83.19	70.19	77.30	72.63
DTModel_pro	89.76	82.35	95.80	71.53	56.54	92.25
DTModel_pro mean process tree	90.91	84.03	96.64	72.13	59.38	91.06
DTModel_pro process tree min alerts: 1	90.20	82.35	96.64	64.45	37.04	92.79
DTModel_pro process tree min alerts: 2	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro process tree min alerts: 3	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro process tree min alerts: 4	90.20	82.35	96.64	64.42	36.97	92.79
DTModel_pro rolling mean window: 2	93.16	94.96	91.60	73.82	66.19	88.40
DTModel_pro rolling mean window: 3	91.77	94.96	89.08	73.49	66.15	87.80
DTModel_pro rolling mean window: 4	90.75	95.80	86.55	72.05	69.38	82.38
DTModel_pro sum alerts min: 2	92.17	95.80	89.08	73.43	67.44	86.56
DTModel_pro sum alerts min: 3	90.75	95.80	86.55	71.53	69.63	81.25
DTModel_pro sum alerts min: 4	89.29	95.80	84.03	67.58	70.81	73.55
DTModel_pro_tree	85.93	73.48	97.48	70.40	43.02	91.57
DTRegression_pro_process	89.06	80.67	95.80	71.62	57.98	91.22
GBDTModel_glo_pro	80.44	63.87	91.60	72.62	59.73	91.71
GBDTModel_glo_pro mean process tree	80.88	63.87	92.44	72.76	60.63	91.22
GBDTModel_glo_pro process tree min alerts: 1	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 2	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 3	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro process tree min alerts: 4	81.75	63.87	94.12	66.32	43.28	92.14
GBDTModel_glo_pro rolling mean window: 2	85.12	83.19	86.55	76.06	71.50	87.80
GBDTModel_glo_pro rolling mean window: 3	84.52	84.03	84.87	75.87	71.82	87.15
GBDTModel_glo_pro rolling mean window: 4	84.12	86.55	82.35	75.25	76.69	81.57
GBDTModel_glo_pro sum alerts min: 2	84.87	84.87	84.87	76.46	75.65	84.66
GBDTModel_glo_pro sum alerts min: 3	85.22	89.08	82.35	74.12	78.77	77.78
GBDTModel_glo_pro sum alerts min: 4	84.44	90.76	79.83	71.99	81.10	72.30
GBDTModel_pro	80.73	62.18	93.28	71.31	58.09	90.51
GBDTModel_pro mean process tree	82.05	64.71	94.12	71.76	59.59	90.14
GBDTModel_pro process tree min alerts: 1	80.71	59.66	94.96	64.88	40.34	91.33
GBDTModel_pro process tree min alerts: 2	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro process tree min alerts: 3	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro process tree min alerts: 4	80.71	59.66	94.96	64.87	40.30	91.33
GBDTModel_pro rolling mean window: 2	84.68	79.83	88.24	75.08	71.60	85.91
GBDTModel_pro rolling mean window: 3	84.08	80.67	86.55	74.99	71.71	85.64
GBDTModel_pro rolling mean window: 4	84.39	84.87	84.03	73.91	76.05	79.84
GBDTModel_pro sum alerts min: 2	85.48	84.03	86.55	74.50	74.40	82.33
GBDTModel_pro sum alerts min: 3	85.11	86.55	84.03	72.35	76.77	76.59
GBDTModel_pro sum alerts min: 4	83.84	88.24	80.67	70.17	78.38	71.71
GBDTModel_pro_tree	79.02	59.09	94.96	71.08	46.68	90.51
GBDTRegression_pro_process	89.71	87.39	91.60	71.84	80.57	72.52
MLPModel_glo_pro	66.67	13.45	93.28	57.92	19.00	90.68
MLPModel_glo_pro mean process tree	67.48	16.81	93.28	59.79	25.74	90.51
MLPModel_glo_pro process tree min alerts: 1	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 2	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 3	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro process tree min alerts: 4	67.46	13.45	94.96	57.61	17.64	90.84
MLPModel_glo_pro rolling mean window: 2	67.96	28.57	88.24	58.73	32.20	84.17
MLPModel_glo_pro rolling mean window: 3	67.79	34.45	84.87	58.90	34.31	83.20
MLPModel_glo_pro rolling mean window: 4	68.75	41.18	83.19	58.32	40.55	78.16
MLPModel_glo_pro sum alerts min: 2	68.94	38.66	84.87	59.51	39.19	81.30
MLPModel_glo_pro sum alerts min: 3	69.04	45.38	81.51	58.47	43.17	76.80
MLPModel_glo_pro sum alerts min: 4	70.63	53.78	79.83	57.23	47.72	71.76

The DT demonstrates that this architecture is capable of preventing damage, but the TNR on the test set of the DT model is so low (66.19) that this model cannot be preferred to the RF (81.53 TNR), which still prevents over 90% of file damage.

The GBDT prevents some damage, and detects a comparable number of ransomware samples (1 in 5). The AdaBoost regressor detected 2 ransomware samples of the 22, and in these two cases more than 64% and 45% of files were saved, respectively; perhaps, with more execution time,

Table 9: Summary of process killing models, validation, and test set score metrics [Table 2 of 3].

Model	Val			Test		
	f1	tnr	tpr	f1	tnr	tpr
MLPModel_pro	71.43	56.30	79.83	57.54	52.53	69.38
MLPModel_pro mean process tree	72.18	57.14	80.67	57.06	53.53	67.97
MLPModel_pro process tree min alerts: 1	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 2	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 3	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro process tree min alerts: 4	72.32	54.62	82.35	57.41	49.41	71.06
MLPModel_pro rolling mean window: 2	71.77	66.39	74.79	48.88	63.18	50.35
MLPModel_pro rolling mean window: 3	72.65	68.91	74.79	49.23	63.71	50.57
MLPModel_pro rolling mean window: 4	72.34	73.95	71.43	46.40	67.05	45.26
MLPModel_pro sum alerts min: 2	73.86	72.27	74.79	47.14	66.40	46.50
MLPModel_pro sum alerts min: 3	73.68	78.99	70.59	45.27	68.12	43.36
MLPModel_pro sum alerts min: 4	73.30	82.35	68.07	44.48	69.63	41.73
MLPModel_pro_tree	71.38	31.82	97.48	64.36	21.07	92.52
MLPRegression_pro_process	38.89	53.78	35.29	54.83	48.73	67.05
MLPRegression_pro_process mean process tree	37.32	57.14	32.77	56.75	57.37	65.15
NBModel_glo_pro	67.25	9.24	96.64	55.07	10.36	89.49
NBModel_glo_pro mean process tree	67.25	9.24	96.64	55.62	12.69	89.38
NBModel_glo_pro process tree min alerts: 1	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 2	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 3	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro process tree min alerts: 4	67.25	9.24	96.64	55.00	10.18	89.43
NBModel_glo_pro rolling mean window: 2	67.69	19.33	92.44	55.20	15.10	87.05
NBModel_glo_pro rolling mean window: 3	67.73	26.05	89.08	55.32	17.32	86.02
NBModel_glo_pro rolling mean window: 4	67.76	31.09	86.55	54.28	21.08	81.68
NBModel_glo_pro sum alerts min: 2	68.17	27.73	89.08	55.70	19.40	85.64
NBModel_glo_pro sum alerts min: 3	67.99	31.93	86.55	54.42	22.27	81.30
NBModel_glo_pro sum alerts min: 4	68.03	36.97	84.03	51.32	25.39	73.44
NBModel_pro	67.06	8.40	96.64	55.60	7.03	92.63
NBModel_pro mean process tree	67.06	8.40	96.64	56.17	9.61	92.41
NBModel_pro process tree min alerts: 1	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 2	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 3	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro process tree min alerts: 4	67.06	8.40	96.64	55.56	6.78	92.68
NBModel_pro rolling mean window: 2	67.69	19.33	92.44	56.01	13.41	89.81
NBModel_pro rolling mean window: 3	67.52	25.21	89.08	56.18	15.81	88.78
NBModel_pro rolling mean window: 4	67.99	31.93	86.55	54.94	19.61	83.90
NBModel_pro sum alerts min: 2	68.61	29.41	89.08	56.52	18.14	88.13
NBModel_pro sum alerts min: 3	68.21	32.77	86.55	55.27	21.30	83.63
NBModel_pro sum alerts min: 4	67.81	37.82	83.19	52.25	24.42	75.77
NBModel_pro_tree	66.10	10.61	98.32	61.25	8.63	92.69
NBModel_pro_tree mean process tree	66.10	10.61	98.32	61.25	8.63	92.69
RFModel_glo_pro	89.68	83.19	94.96	76.43	67.19	92.52
RFModel_glo_pro mean process tree	90.48	84.03	95.80	76.34	67.66	91.92
RFModel_glo_pro process tree min alerts: 1	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 2	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 3	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro process tree min alerts: 4	90.91	84.03	96.64	69.45	50.56	92.95
RFModel_glo_pro rolling mean window: 2	92.70	94.96	90.76	80.77	78.88	89.38
RFModel_glo_pro rolling mean window: 3	91.30	94.96	88.24	80.19	78.67	88.51
RFModel_glo_pro rolling mean window: 4	90.27	95.80	85.71	79.86	82.86	83.69
RFModel_glo_pro sum alerts min: 2	91.30	94.96	88.24	81.50	81.53	87.97
RFModel_glo_pro sum alerts min: 3	90.27	95.80	85.71	79.99	84.01	82.76
RFModel_glo_pro sum alerts min: 4	88.79	95.80	83.19	76.11	85.37	75.01
RFModel_pro	92.37	87.39	96.64	74.57	62.71	92.95
RFModel_pro mean process tree	92.74	88.24	96.64	74.79	64.04	92.20
RFModel_pro process tree min alerts: 1	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 2	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 3	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro process tree min alerts: 4	92.74	88.24	96.64	68.75	48.23	93.39
RFModel_pro rolling mean window: 2	93.22	94.12	92.44	78.28	73.83	89.76

Table 9: Continued.

Model	Val			Test		
	f1	tnr	tpr	f1	tnr	tpr
RFModel_pro rolling mean window: 3	91.38	94.12	89.08	77.47	73.25	88.78
RFModel_pro rolling mean window: 4	89.96	94.12	86.55	77.08	77.70	83.85
RFModel_pro sum alerts min: 2	91.38	94.12	89.08	77.98	74.65	88.40
RFModel_pro sum alerts min: 3	89.96	94.12	86.55	77.05	77.52	83.96
RFModel_pro sum alerts min: 4	88.50	94.12	84.03	73.11	79.35	75.61
RFModel_pro_tree	90.35	82.58	98.32	74.20	52.44	92.74
RFRegression_pro_process	91.94	87.39	95.80	74.77	66.05	90.35
SVMModel_glo_pro	65.23	15.97	89.08	57.34	24.24	86.23
SVMModel_glo_pro mean process tree	65.23	15.97	89.08	58.11	27.39	85.91
SVMModel_glo_pro process tree min alerts: 1	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 2	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 3	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro process tree min alerts: 4	65.23	15.97	89.08	57.32	23.81	86.45
SVMModel_glo_pro rolling mean window: 2	65.15	26.05	84.03	57.98	33.52	81.84
SVMModel_glo_pro rolling mean window: 3	64.65	31.09	80.67	58.14	35.46	80.98
SVMModel_glo_pro rolling mean window: 4	64.31	38.66	76.47	56.76	40.37	75.34
SVMModel_glo_pro sum alerts min: 2	65.05	36.13	78.99	58.35	39.08	79.13
SVMModel_glo_pro sum alerts min: 3	64.75	42.02	75.63	57.05	43.24	74.15
SVMModel_glo_pro sum alerts min: 4	64.89	51.26	71.43	54.70	47.40	67.59
SVMModel_pro	66.47	5.88	96.64	56.92	10.33	93.71

Table 10: Summary of process killing models, validation, and test set score metrics [Table 3 of 3].

	Val			Test		
SVMModel_pro mean process tree	67.25	9.24	96.64	57.55	13.34	93.33
SVMModel_pro process tree min alerts: 1	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 2	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 3	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro process tree min alerts: 4	66.47	5.88	96.64	56.21	7.28	93.88
SVMModel_pro rolling mean window: 2	66.87	15.97	92.44	58.60	22.02	90.30
SVMModel_pro rolling mean window: 3	67.30	24.37	89.08	58.82	24.42	89.27
SVMModel_pro rolling mean window: 4	67.99	31.93	86.55	57.98	28.97	84.66
SVMModel_pro sum alerts min: 2	67.96	28.57	88.24	59.52	27.61	88.73
SVMModel_pro sum alerts min: 3	68.90	35.29	86.55	59.06	33.35	84.12
SVMModel_pro sum alerts min: 4	68.75	41.18	83.19	56.68	38.87	76.10
SVMModel_pro_tree	65.73	9.09	98.32	61.79	9.88	93.19
Dqn	51.71	72.27	44.54	27.74	55.50	26.94
random_search_glo_pro_RNN	87.69	77.31	95.80	71.83	59.63	90.24
random_search_glo_pro_RNN mean process tree	88.03	78.15	95.80	72.50	61.67	89.81
random_search_glo_pro_RNN_Regression	85.71	72.27	95.80	72.44	61.78	89.59
random_search_pro_RNN	91.20	85.71	95.80	72.63	59.63	91.82
random_search_pro_RNN mean process tree	91.20	85.71	95.80	73.03	60.92	91.49
random_search_pro_RNN_Regression	88.37	78.99	95.80	72.71	60.70	91.06
random_search_pro_RNN_tree	88.19	80.67	94.12	73.72	65.79	88.56

Table 11: Two models' F1-score, TNR, TPR for the validation and test set scoring the highest TNR on the validation and test sets.

Methodology	Model	n features	Val			Test		
			F1	tnr	tpr	F1	tnr	tpr
Regression	AdaBoost	26	56.63	100.00	39.50	15.06	97.92	8.40
Regression + 4 alerts	GBDT	26	85.91	95.80	77.31	68.50	94.98	56.04

the files would be detected but the key benefit of process killing is to stop damaging software like these ransomware samples and this algorithm actually saw more files encrypted than when no killing model was used; this is because there will be a slight variance in the ransomware behaviour and execution time each time it runs. The Random Forest is the most plausible model, balancing damage prevention and TNR; however, the delay in classification may be a result of

TABLE 12: Random Forest and Decision Tree each with a minimum requirement of two alerts ("malicious classifications") to kill a process. F1, TNR, and TPR reported on validation and test set.

Model	n features	Val			Test		
		F1	Tnr	tpr	F1	tnr	tpr
RF (alerts: 2)	37	91.30	94.96	88.24	**81.50**	81.53	87.97
DT (rolling mean: 2)	26	**93.16**	94.96	91.60	73.82	66.19	88.40

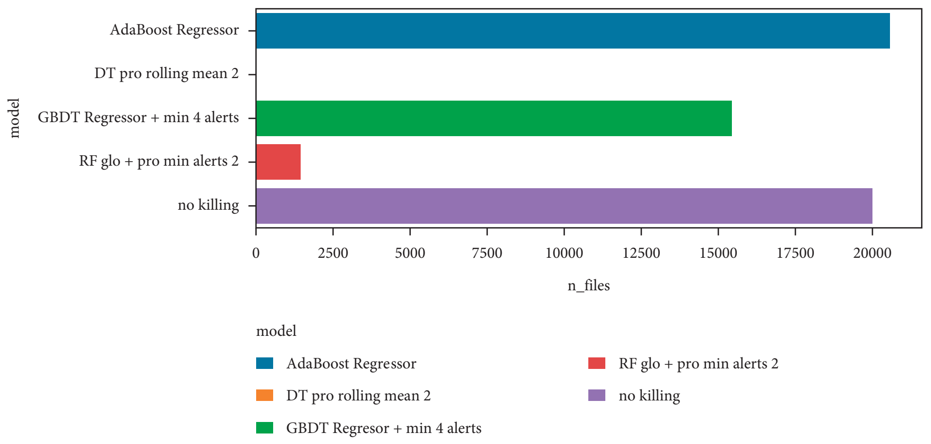

FIGURE 6: Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution.

TABLE 13: Total number of files corrupted by ransomware with no process killing and with three process killing models within the first 30 seconds of execution. Damage reduction is the percentage of files spared when no killing is implemented.

Model	Files damaged	Damage reduction	Detection rate (ransomware TPR)	Test set TPR
No killing	19,997	—	—	—
DT pro rolling mean 2	3	99.98%	100.00	88.40
RF glo + pro min alerts 2	1,464	92.68%	100.00	87.97
GBDT regressor + min 4 alerts	15,432	22.83%	22.07	56.04
AdaBoost regressor	20,578	0.00%	9.09	8.83

the requirement to collect more features and/or the real-time of the model itself.

7. Discussion: Measuring Execution Time in a Live Environment

Although algorithm execution duration was measured above, due to batch processing used by the models, the number of processes being classified can be increased by an order of magnitude with a negligible impact on execution time. The data collection and process killing both have linear, $O(n)$, complexity; where n is the number of processes; therefore it is expected that the number of processes impacts classification time. The RF with statistical filters has complexity $O(\text{nps})$ where p is the number of trees in the forest and s is the number of alerts considered by the filter; efficient library implementations of matrix operations means that the execution time does not scale linearly with n for the RF inference. Given this, a further experiment was carried out with the RF to measure in a live environment how long the data collection, model inference, and process killing takes as the number of processes increases. This was tested by executing more than 1000 processes in the virtual machine whilst the process killing RF runs.

Some processes demand more computational resources than others, and some malware in our test set locked pages in memory [50], which prevented the model from having sufficient resources to collect data, leading to tens of seconds during which no data were captured and many processes were launched. With better software engineering practices, the model may be more robust against this kind of malicious activity.

These differences in behaviour can cause the evaluation time to lag as demonstrated by the outlier points visible in Figure 7. The data show a broadly linear positive correlation between the number of processes (being monitored or killed) and the time taken for the data collection and process killing; this confirms the hypothesis that more processes equates to slower processing time. The slowest total processing time was 0.81 seconds (seen with both 17 and 40 simultaneous processes running), but the mean processing time is just under 0.3 seconds with 65 simultaneous processes, fitting comfortably within the 1-second goal time. Additional code optimisation could greatly improve on these

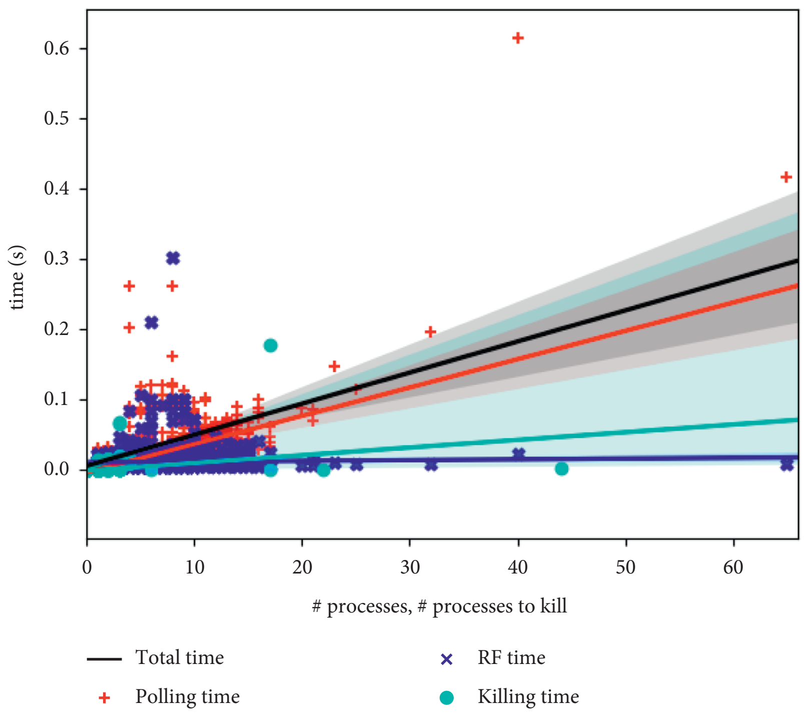

Figure 7: Mean time to collect data, analyse data with Random Forest, and kill varying numbers of processes.

initial results which indicate that the processing, even using standard libraries and a high-level programming language, can execute reasonably quickly.

8. Implications and Analysis

The experiments in this paper address a largely unexplored area of malware detection, by comparison with post-trace classification. Real-time processing and response has a number of benefits outlined above and the results presented here give tentative indications of the advantages and challenges of such an approach.

The initial experiments (Section 6.1) demonstrate that a high-accuracy RNN (as used in [11]) does not maintain high-accuracy when used in real-time with an automated response to classify individual processes rather than full application traces, since a single false positive classification of sequential data cannot be outweighed by later correct predictions.

The next set of experiments (Section 6.1) showed that whilst the RNN achieves one of the highest classification accuracies of a set of algorithms tested, it is not one of the best in terms of computational resource consumption or latency. However, a clear best-algorithm was not evident either since the low-resource consuming algorithms (like decision tree) did not always achieve high accuracy. Furthermore, all of the supervised learning algorithms were clearly unsuited to process killing with the highest F1 score from any algorithm being 77.85 on the test set compared with 85.55 for process-level classification alone. This 85.55 F1 score is lower than is seen in many dynamic malware detection research publications that use full-application behavioural traces, indicating the challenges of classification at the process level, where malware and benignware may share functionality.

Attempting to improve detection accuracy, three approaches were tested: statistical filtering, reinforcement learning, and a regression model estimating the utility (q-value) of killing a process. Statistical filters using rolling mean or alert thresholds were the only approach to improve on the supervised learning model F1 score. Reinforcement learning tended to kill processes too early and therefore not explore enough scenarios (and thus receive the requisite reinforcement) to allow benign processes to continue; this does not mean that future models could not improve upon this result. This may be supported by the success of the regression models in maintaining a high true-negative rate, given that these models ascribed a similar utility to killing processes as the reinforcement learning models.

The accuracy metrics tested thus far simply indicate whether a process was ever killed, but do not address whether damage was actually prevented by process killing. If damage was not prevented, there is little point to process killing and a database of alerts for analysis would be a better solution since the risk of killing benignware is eliminated. This is why the final set of experiments in Section 6.5 were conducted to test the detection models in real time and see if damage could be prevented by looking at the number file corrupted by ransomware before and after infection. Here, we found that it is possible to prevent 92% of files from being encrypted whilst maintaining a true negative rate of 82%. This result does not indicate that the system is ready for real-world deployment but that perhaps further model analysis probably including anomaly detection could raise the true negative rate to a usable point. This work also demonstrates the damage that certain malware can carry out in a short space of time and reinforces the need for further research in this area, since previous work has either focused solely on ransomware [24] or waited minutes to being classification [23], by which time it is too late.

9. Future Work

Real-time attack detection has wider applications than endpoint detection, as Alazab et al. [51] argue that Internet of Things networks in particular could benefit from real-time attack detection using heterogeneous data feed from different sensors combined using federated learning approaches.

However, some challenges remain to be solved; behavioural malware analysis research using machine learning regularly reports $>95\%$ classification accuracy. Although useful for analysts, behavioural detection should be deployed as part of endpoint defensive systems to leverage the full benefits of a detection model. Dynamic analysis is not typically used for endpoint protection, perhaps because it takes too long in data collection to deliver the quick verdicts required for good user experience. Real-time detection on the endpoint allows for observation of the full trace without the user having to wait. However, real-time detection also introduces the risk that malware will cause damage to the endpoint. This risk requires that processes detected as malicious are automatically killed as early as possible to avoid harm.

There are some key challenges to implementation, which have been outlined in this paper:

(i) The need for signal separation drives the use of individual processes and only partial traces can be used.

(ii) The significant drop in accuracy on the unseen test set, even without process killing demonstrates that additional features may be necessary to improve detection accuracy.

(iii) With the introduction of process killing, the poor performance of the models on either benignware classification (RF min 2 alerts: TNR 81% with an 88% TPR on the test set) or on malware classification (GBDT regressor min 4 alerts: 56% TPR with a 94% TNR on the test set) means that considerable further work is needed before very early stage real-time detection can be considered for real-world use.

(iv) Real-time detection using full execution traces of processes, however, may be viable. This is useful to handle VM-aware malware, which may only reveal its true behaviour in the target environment. Although the more complex approach using DQNs algorithms did not outperform the supervised models with some additional statistical thresholds, the regression models had better performance in correctly classifying benignware. Reinforcement learning could still be useful for real-time detection and automated cyber defense models, but the DQN in these experiments did not perform well.

(v) Despite the theoretical unsuitability of supervised learning models to state-action problems, these experiments demonstrate how powerful supervised learning can be for classification problems, even if the problem is not quite the one that the model is attempting to solve.

(vi) Future work may require a more comprehensive manual labelling effort at the process level and perhaps labelling sub-sections of processes as malicious or benign.

An additional consideration for real-time detection with automated actions is whether this introduces an additional denial-of-service vector using process injection for example to trigger process killing. This may also however indicate that an attacker is present and therefore aid the user.

10. Conclusions

This paper has built on previous work in real-time detection to address some of the key challenges: signal separation, detection with partial execution traces, and computational resource consumption with a focus on preventing harm to the user, since real-time detection introduces this risk.

Behavioural malware detection using virtual machines is a well-established research field yielding high detection accuracy in recent literature [3, 6, 11, 20]. However, as is shown here, fixed-time execution in a sandbox may not reveal malicious functionality. Real-time malware analysis addresses this issue but risks executing malware on the endpoint and requires detection to take place at the process level, which is more challenging as the definition of a malicious process can be unclear. These two reasons may account for the limited literature on real-time detection. Looking forward, real-time detection may become more popular if static data manipulation and VM-evasion continue to be used and the costs of malicious execution continue to rise. Real-time detection does not need to be an alternative to these approaches, but could hold complementary value as part of a defense-in-depth endpoint security.

To the best of our knowledge, previous real-time detection work has used up to 5 simultaneous applications, whereas other users may use far more. This paper has demonstrated that up to 35 simultaneous applications (and nearly 100 simultaneous processes) can be constantly monitored, where previous work [23] had tested a maximum of 5. Moreover, these results demonstrated that data collection presented a greater limiting factor than machine-learning algorithms, which can easily process 1000 samples with negligible impact on performance. This result is not too surprising since batch processing allows algorithms to achieve $O(1)$ complexity by comparison with $O(n)$ for data collection.

Automatic actions are necessary in response to detection if the goal is to prevent harm. Otherwise, this is equivalent to letting the malware fully execute and simply monitor its behaviour since human response times are unlikely to be quick enough for fast-acting malware. From a user perspective, the question is not "What percentage of malware was executed?" or "Was the malware detected in 5 or 10 minutes?" but "How much damage has been done?".

This paper found that by using simple statistical filters on top of supervised learning models, it was possible to prevent 92% of files from being corrupted by fast-acting ransomware thus reducing the requirements on the user or organisation to remediate the damage, since it was prevented in the first instance (the rest of the attack vector would remain a concern).

This approach does not achieve the detection accuracies of state-of-the art offline behavioural analysis models but, as stated in the introduction, these models typically use the full post-execution trace of malicious behaviour. Delaying classification until post-execution negates the principal advantages of real-time detection. However, the proposed model presents an initial step towards a fully automated endpoint protection model, which becomes increasingly necessary as adversaries become more and more motivated to evade offline automated detection tools.

Data Availability

Information on the data underpinning the results presented here, including how to access them, can be found in the Cardiff University data catalogue at 10.17035/d.2021.0148229014.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partly funded by the Engineering and Physical Sciences Research Council (EPSRC)-grant references EP/P510452/1 and EP/S035362/1. The research was also partly funded by Aureirbus Operations Ltd..

References

[1] U. Tatar, B. Nussbaum, Y. Gokce, and O. F. Keskin, "Digital force majeure: the mondelez case, insurance, and the (un) certainty of attribution in cyberattacks," *Business Horizons*, vol. 64, 2021.

[2] S. K. Sahay, A. Sharma, and H. Rathore, "Evolution of malware and its detection techniques," in *Information and Communication Technology for Sustainable Development*, pp. 139–150, Springer, Singapore, 2020.

[3] W. Huang and J. W. Stokes, "Mtnet: a multi-task neural network for dynamic malware classification," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment-Volume 9721, DIMVA 2016*, pp. 399–418, Springer-Verlag New York, Inc., New York, NY, USA, June 2016.

[4] W. Hu and Y. Tan, "Black-box attacks against rnn based malware detection algorithms," in *Proceedings of the Workshops at the mThirty-Second AAAI Conference on Artificial Intelligence*, Peking University, Beijing, China, February 2018.

[5] Y. Chen, Z. Shan, F. Liu et al., "A gene-inspired malware detection approach," in *Journal of Physics: Conference Series*vol. 1168, IOP Publishing, Article ID 062004, 2019.

[6] M. Ijaz, M. H. Durad, and M. Ismail, "Static and dynamic malware analysis using machine learning," in *Proceedings of the 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 687–691, IEEE, Islamabad, Pakistan, January 2019.

[7] I. You and K. Yim, "Malware obfuscation techniques: a brief survey," in *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pp. 297–300, IEEE, Fukuoka, Japan, November 2010.

[8] B. Kolosnjaji, A. Demontis, B. Biggio et al., "Adversarial malware binaries: evading deep learning for malware detection in executables," in *Proceedings of the 2018 26th European Signal Processing Conference (EUSIPCO)*, pp. 533–537, IEEE, Rome, Italy, September 2018.

[9] D. Carlin, P. O'Kane, and S. Sezer, "A cost analysis of machine learning using dynamic runtime opcodes for malware detection," *Computers & Security*, vol. 85, pp. 138–155, 2019.

[10] T. Shibahara, T. Yagi, M. Akiyama, D. Chiba, and T. Yada, "Efficient dynamic malware analysis based on network behavior using deep learning," in *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, Washington, DC, USA, December 2016.

[11] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, pp. 578–594, 2018.

[12] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20, IEEE, Fajardo, PR, USA, October 2015.

[13] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: eliminating experimental bias in malware classification across space and time," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19), {USENIX} Association*, pp. 729–746, Santa Clara, CA, USA, September 2019.

[14] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," 2016, https://arxiv.org/abs/1606.04435 arXiv preprint arXiv:1606.04435.

[15] H. Sayadi, N. Patel, S. M. Pd, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: a comprehensive analysis and classification," in *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, San Francisco, CA, USA, June 2018.

[16] N. Usman, S. Usman, F. Khan et al., "Intelligent dynamic malware detection using machine learning in ip reputation for forensics data analytics," *Future Generation Computer Systems*, vol. 118, pp. 124–141, 2021.

[17] P. Burnap, R. French, F. Turner, and K. Jones, "Malware classification using self organising feature maps and machine activity data," *Computers & Security*, vol. 73, pp. 399–410, 2018.

[18] M. Rhode, L. Tuson, P. Burnap, and K. Jones, "Lab to soc: robust features for dynamic malware detection," in *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track*, pp. 13–16, IEEE, Portland, OR, USA, June 2019.

[19] H. Sayadi, A. Houmansadr, S. Rafatirad, H. Homayoun, and P. D. Sai Manoj, "Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 212–215, ACM, Ischia, Italy, May 2018.

[20] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

[21] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: a framework for efficient online malware detection," in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 651–661, IEEE, Burlingame, CA, USA, February. 2015.

[22] X. Yuan, "Phd forum: deep learning-based real-time malware detection with multi-stage analysis," in *Proceedings of the 2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 1-2, IEEE, Hong Kong, China, May 2017.

[23] R. Sun, X. Yuan, P. He et al., "Learning fast and slow: PROPEDEUTICA for real-time malware detection," *CoRR*, vol. abs/1712, Article ID 01145, 2017.

[24] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 303–312, IEEE, Nara, Japan, June 2016.

[25] GlobalStats, "Market share of windows operating system versions," 2018, http://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide#monthly-201708-201709-bar.

[26] J. Benda, A. Longtin, and L. Maler, "Spike-frequency adaptation separates transient communication signals from background oscillations," *Journal of Neuroscience*, vol. 25, no. 9, pp. 2312–2321, 2005.

[27] H.-D. Huang, C.-S. Lee, H.-Y. Kao, Y.-L. Tsai, and J.-G. Chang, "Malware behavioral analysis system: Twman," in *Proceedings of the 2011 IEEE Symposium on Intelligent Agent (IA)*, pp. 1–8, IEEE, Paris, France, April 2011.

[28] T. Kim, B. Kang, and E. G. Im, "Runtime detection framework for android malware," *Mobile Information Systems*, vol. 2018, 2018.

[29] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, Vol. 135, MIT Press Cambridge, Cambridge, MA, USA, 1998.

[30] C. J. C. H. Watkins, *Learning from delayed rewards*, PhD thesis, King's University, London, UK, 1989.

[31] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[32] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," *Machine Learning Proceedings 1990*, vol. 1990, pp. 216–224, 1990.

[33] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[34] V. Mnih, K. Kavukcuoglu, D. Silver et al., "Playing atari with deep reinforcement learning," 2013, https://arxiv.org/abs/1312.5602 arXiv preprint arXiv:1312.5602.

[35] V. Mnih, A. P. Badia, M. Mirza et al., "Asynchronous methods for deep reinforcement learning," in *Proceedings of the International Conference on Machine Learning*, pp. 1928–1937, PMLR, New York City, NY, USA, June 2016.

[36] J. Kirkpatrick, R. Pascanu, N. Rabinowitz et al., "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[37] sophos, https://www.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/sophos-the-state-of-ransomware-2020-wp.pdf, 2020.

[38] A. Continella, A. Guagnelli, G. Zingaro et al., "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 336–347, ACM, Los Angeles, CF, USA, December 2016.

[39] P. S. Foundation, "Psutil python library," 2017.

[40] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "The cuckoo sandbox," 2012, https://cuckoosandbox.org/.

[41] Amazon.com, "Amazon laptops," 2018, https://www.amazon.com/Notebooks-Laptop-Computers/b?ie=UTF8&node=565108.

[42] B. Quintero, E. Martínez, V. Manuel Álvarezv, K. Hiramoto, J. Canto, and A. Bermúdez, "Virustotal," 2004, https://en.wikipedia.org/wiki/VirusTotal.

[43] VirusShare.com, "Virusshare.com," 2017, https://virusshare.com/.

[44] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: a tool for massive malware labeling," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 230–253, Springer, New York, NY, USA, September 2016.

[45] SoftAntenna, "Msys2 installer," 2019, https://www.softantenna.com/softwares/7115-msys2-installer.

[46] A. Paszke, S. Gross, S. Chintala et al., *Automatic Differentiation in Pytorch*NIPS-W, Long Beach, CA, USA, 2017.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort et al., "Scikit-learn: machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[48] D. P. Kingma and J. Ba, "Adam: 'a' method for stochastic optimization," *CoRR*, vol. abs/1412, p. 6980, 2014.

[49] C. Belgaid, A. d'Azémar, G. Fieni, and R. Rouvoy, "Pyrapl," Software version 0.2.3.1 https://pypi.org/project/pyRAPL/, 2019.

[50] M. Corporation, "Lock pages in memory," 2017, https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/lock-pages-in-memory.

[51] M. Alazab, S. P. Rm, M. Parimala, P. Reddy, T. R. Gadekallu, and Q.-V. Pham, "Federated learning for cybersecurity: concepts, challenges and future directions," *IEEE Transactions on Industrial Informatics*, 2021.