ASPIRE: Automated Security Policy Implementation Using Reinforcement Learning

Yoni Birman, Shaked Hindi, Gilad Katz, Asaf Shabtai
Department of Software and Information Systems Engineering
Ben-Gurion University of the Negev

ABSTRACT
Malware detection is an ever-present challenge for all organizational gatekeepers. Organizations often deploy numerous different malware detection tools, and then combine their output to produce a final classification for an inspected file. This approach has two significant drawbacks. First, it requires large amounts of computing resources and time since every incoming file needs to be analyzed by all detectors. Secondly, it is difficult to accurately and dynamically enforce a predefined security policy that complies with the needs of each organization (e.g., how tolerant is the organization to false negatives and false positives). In this study we propose ASPIRE, a reinforcement learning (RL)-based method for malware detection. Our approach receives the organizational policy – defined solely by the perceived costs of correct/incorrect classifications and of computing resources – and then dynamically assigns detection tools and sets the detection threshold for each inspected file. We demonstrate the effectiveness and robustness of our approach by conducting an extensive evaluation on multiple organizational policies. ASPIRE performed well in all scenarios, even achieving near-optimal accuracy of 96.21% (compared to an optimum of 96.86%) at approximately 20% of the running time of this baseline.

CCS CONCEPTS
-Information systems → Expert systems; Data mining; Data analytics; Security and privacy → Intrusion/anomaly detection and malware mitigation; Malware and its mitigation; Computing methodologies → Artificial intelligence; Machine learning;

KEYWORDS
malware detection, deep reinforcement learning, behavior-based malware detection, machine learning, portable executable, artificial intelligence

1 INTRODUCTION
Malware detection is an ever-present problem for organizations, often with significant consequences [2]. Specifically, Portable Executable (PE) files are one of the most significant platforms for malware to spread. PEs are common in the Windows operating systems, and are used by executables and dynamic link libraries (DLLs), among others. The PE format is essentially a data structure which holds all the necessary information for the Windows loader to execute the wrapped code.

Malware constantly evolve as attackers try to evade detection solutions, the most common of which being the anti-virus. Anti-virus solutions mostly perform static analysis of the software’s binary to detect pre-defined signatures, a trait that renders them ineffective in recognizing new malware even if similar functionality has been recorded. Moreover, obfuscation techniques such as polymorphism and metamorphism [33] further exacerbate the problem.

The need to deal with the continuously evolving threats led to significant developments in the malware detection field in recent years. Instead of searching for pre-defined signatures within the executable file, new approaches attempt to analyze the behaviour of the portable executable (PE) file. These methods often rely on statistical analysis and machine learning (ML) as their decision making mechanism, and can generally be thought of as belonging to one of two families: static analysis and dynamic analysis [18]. In this study we focus on the static analysis techniques.

Static analysis techniques [29] employ an in-depth look at the file, without performing any execution. Solutions implementing static analysis can be either signature-based or statistics-based. Signature-based detection is the more widely used approach [6] because of its simplicity, relative speed and its effectiveness against known malware. Despite these advantages, signature-based detection has three major drawbacks: it requires frequent updates of its signature database, it cannot detect unknown (i.e., zero-day) malware [6], and it is vulnerable to obfuscation techniques [33].

Statistics-based detection mainly involves the extraction of features from the executable, followed by training of a machine learning classifier. The extracted features are varied and may include executable file format descriptions [19], code descriptions [23], binary data statistics [17], text strings [5] and information extracted using code emulation or similar methods [33]. This approach is considered more effective than its signature-based counterpart in detecting previously unknown malware – mostly due to its use of machine learning (ML) [3, 5, 7, 10, 23] – but tends to be less accurate overall [20]. For this reason, organizations often deploy an ensemble of multiple behavioural and statistic detectors, and then combine their scores to produce a final classification. This process of producing this classification can be achieved through simple heuristics (e.g., averaging) or by more advanced ML algorithms [12].

Despite its effectiveness, the ensemble approach has two significant shortcomings. First, using an ensemble requires that organizations run all participating detection tools prior to classifying a file. This practice is needed both in order to make scoring consistent and because most ML algorithms (like those often used to reach the final ensemble decision) require a fixed-size feature set. Running all detectors is time and resource intensive and is often not necessary for clear-cut cases. This practice results in “wasted” computing resources. Moreover, the introduction or removal of a detector often requires that the entire ML model be retrained, a fact that limits flexibility and the organization’s ability to respond to new threats.

ABSTRACT
Malware detection is an ever-present challenge for all organizational gatekeepers. Organizations often deploy numerous different malware detection tools, and then combine their output to produce a final classification for an inspected file. This approach has two significant drawbacks. First, it requires large amounts of computing resources and time since every incoming file needs to be analyzed by all detectors. Secondly, it is difficult to accurately and dynamically enforce a predefined security policy that complies with the needs of each organization (e.g., how tolerant is the organization to false negatives and false positives). In this study we propose ASPIRE, a reinforcement learning (RL)-based method for malware detection. Our approach receives the organizational policy – defined solely by the perceived costs of correct/incorrect classifications and of computing resources – and then dynamically assigns detection tools and sets the detection threshold for each inspected file. We demonstrate the effectiveness and robustness of our approach by conducting an extensive evaluation on multiple organizational policies. ASPIRE performed well in all scenarios, even achieving near-optimal accuracy of 96.21% (compared to an optimum of 96.86%) at approximately 20% of the running time of this baseline.

CCS CONCEPTS
-Information systems → Expert systems; Data mining; Data analytics; Security and privacy → Intrusion/anomaly detection and malware mitigation; Malware and its mitigation; Computing methodologies → Artificial intelligence; Machine learning;

KEYWORDS
malware detection, deep reinforcement learning, behavior-based malware detection, machine learning, portable executable, artificial intelligence

1 INTRODUCTION
Malware detection is an ever-present problem for organizations, often with significant consequences [2]. Specifically, Portable Executable (PE) files are one of the most significant platforms for malware to spread. PEs are common in the Windows operating systems, and are used by executables and dynamic link libraries (DLLs), among others. The PE format is essentially a data structure which holds all the necessary information for the Windows loader to execute the wrapped code.

Malware constantly evolve as attackers try to evade detection solutions, the most common of which being the anti-virus. Anti-virus solutions mostly perform static analysis of the software’s binary to detect pre-defined signatures, a trait that renders them ineffective in recognizing new malware even if similar functionality has been recorded. Moreover, obfuscation techniques such as polymorphism and metamorphism [33] further exacerbate the problem.

The need to deal with the continuously evolving threats led to significant developments in the malware detection field in recent years. Instead of searching for pre-defined signatures within the executable file, new approaches attempt to analyze the behaviour of the portable executable (PE) file. These methods often rely on statistical analysis and machine learning (ML) as their decision making mechanism, and can generally be thought of as belonging to one of two families: static analysis and dynamic analysis [18]. In this study we focus on the static analysis techniques.

Static analysis techniques [29] employ an in-depth look at the file, without performing any execution. Solutions implementing static analysis can be either signature-based or statistics-based. Signature-based detection is the more widely used approach [6] because of its simplicity, relative speed and its effectiveness against known malware. Despite these advantages, signature-based detection has three major drawbacks: it requires frequent updates of its signature database, it cannot detect unknown (i.e., zero-day) malware [6], and it is vulnerable to obfuscation techniques [33].

Statistics-based detection mainly involves the extraction of features from the executable, followed by training of a machine learning classifier. The extracted features are varied and may include executable file format descriptions [19], code descriptions [23], binary data statistics [17], text strings [5] and information extracted using code emulation or similar methods [33]. This approach is considered more effective than its signature-based counterpart in detecting previously unknown malware – mostly due to its use of machine learning (ML) [3, 5, 7, 10, 23] – but tends to be less accurate overall [20]. For this reason, organizations often deploy an ensemble of multiple behavioural and statistic detectors, and then combine their scores to produce a final classification. This process of producing this classification can be achieved through simple heuristics (e.g., averaging) or by more advanced ML algorithms [12].

Despite its effectiveness, the ensemble approach has two significant shortcomings. First, using an ensemble requires that organizations run all participating detection tools prior to classifying a file. This practice is needed both in order to make scoring consistent and because most ML algorithms (like those often used to reach the final ensemble decision) require a fixed-size feature set. Running all detectors is time and resource intensive and is often not necessary for clear-cut cases. This practice results in “wasted” computing resources. Moreover, the introduction or removal of a detector often requires that the entire ML model be retrained, a fact that limits flexibility and the organization’s ability to respond to new threats.
The second shortcoming of the ensemble approach is the difficulty of implementing the organizational security policy. When using ML-based solutions for malware detection, the only "tool" available for organizations to set their policy is the final confidence score: files above a certain score are blocked, while the rest are allowed in. Under this setting it is difficult to define the cost of a false-negative compared to that of a false-positive or to quantify the cost of running additional detectors. In addition to being hard to define, such security policies are also hard to refine: minor changes to the confidence score threshold may result in large fluctuations in performance (e.g., significantly raising the number of false-alarms).

In this study we propose ASPIRE, a reinforcement learning-based framework for managing a malware detection platform consisting of multiple malware detection tools. For each file, our approach sequentially queries various detectors, deciding after each step whether to further analyze the file or produce a final classification. ASPIRE's decision-making process is governed by a pre-defined reward function that awards points for correct classifications and applies penalties for misclassification and heavy use of computing resources.

Our approach has two advantages over existing ensemble-based solutions. First, it is highly efficient, since easy-to-classify files are likely to only require the use of less-powerful (i.e. efficient) classifiers. We can therefore maintain near-optimal performance at a fraction of the computing cost. Secondly, organizations can clearly and deliberately define and refine their security policy. We achieve this goal by enabling practitioners to explicitly define the costs to each element of the detection process: correct/incorrect classification and resource usage.

Our contributions in this study are threefold:

- we present a reinforcement learning-based approach for ensemble-based malware detection. Our approach was able to achieve near-optimal accuracy of 96.21% (compared to an optimum of 96.86%) at approximately 20% of the running time of this baseline.
- we conduct an extensive analysis of multiple security policies, designed to simulate the needs and goals of a different organizational types. In addition to demonstrating the robustness of our approach, we analyze the effect of various policy preferences on detection accuracy and resource use.
- we release the dataset used in our evaluation for general use. In addition to the files themselves, we release for each file the confidence scores and meta-data of each of the malware detectors used in our experiments.

2 BACKGROUND AND MOTIVATION

In this section we provide a general overview of deep reinforcement learning (DRL) algorithms and their advantages. We then elaborate on our motivation in applying them to field of malware detection.

2.1 Deep Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning that addresses decision making in complex scenarios, possibly when only partial information is available. The ability of RL algorithms to explore the large solution spaces and devise highly efficient policies to address them (especially when coupled with deep learning) was shown to be highly effective in areas such as robotics and control problems [21], genetic algorithms [26], and achieving super-human performance in complex games [25].

RL tasks normally consist of both an agent and an environment. The agent interacts with the environment \( E \) in a sequence of actions and rewards. At each time-step \( t \), the agent selects an action \( a_t \) from \( A = \{a_1, a_2, ..., a_k\} \) that both modifies the state of the environment and also incurs a reward \( r_t \). Reward can be either positive or negative\(^1\). For each given task (in our case, the classification of a single file), the goal of the agent is to interact with the environment in a way that maximizes future rewards \( R_t = \sum_T r_t \) where \( T \) is the index of the final action (i.e., classification decision).

A frequent approach for selecting the action to be taken at each state is the action-value function \( Q(s, a) \) [27]. The function approximates the expected returns should we take action \( a \) at state \( s \). While the methods are varied, RL algorithms which use Q-functions aim to discover (or closely approximate) the optimal action-value function \( Q^* \) which is defined as \( Q^*(s, a) = \max_{\pi} \mathbb{E}[R|s_t = s, a_t = a, \pi] \) where \( \pi \) is the policy mapping states to actions [27]. Since estimating \( Q \) for every possible state-action combination is highly impractical [14], it is common to use an approximator \( Q(s, a; \theta) \approx Q^*(s, a) \) where \( \theta \) represents the parameters of the approximator. Deep reinforcement learning (DRL) algorithm perform this approximation using neural nets, with \( \theta \) being the parameters of the network.

While RL algorithms strive to maximize the reward based on their current knowledge about the world (i.e., exploitation), it is important to also encourage the exploration of other additional states. Many methods for maintaining this exploration/exploitation balance have been offered in the literature, including importance sampling [22], \( \epsilon \)-greedy sampling [30] and Monte-Carlo Tree search [24]. In this study, we use \( \epsilon \)-greedy sampling.

**Actor-critic algorithms for reinforcement learning.** Two common problems in the application of DRL algorithms is the long time they need to converge due to high variance (i.e., fluctuations) in gradient values, and the need to deal with action sequences with a cumulative reward of zero (zero reward equals zero gradients, hence no parameter updates). These challenges can be addressed by using actor-critic methods, consisting of a critic neural net that estimates the Q-function and an actor neural net that updates the policy according to the critic. The use two separate networks have been shown to reduce variance and accelerate model convergence during training. In our experiments we use the actor-critic with experience replay (ACER) algorithm [32]. Experience replay [13] is a method for re-introducing the model to previously seen samples in order to prevent catastrophic forgetting (i.e., forgetting previously learned scenarios while tackling new ones).

2.2 Motivation

The ever-evolving threat of malware creates an incentive for organizations to diversify their detection capabilities. As a result, organizations often install multiple solutions [11] and run them all for every incoming file. This approach is both costly – in computing resources, processing time, and even the cost of electricity – and often unnecessary since most files can be easily classified.

\(^1\)Please note that for convenience we use the term "cost" to describe negative rewards
A logical solution to this problem would be using a small number of detectors for clear-cut cases and a larger ensemble for difficult-to-analyze files. This approach, however, is challenging to implement for two reasons. The first challenge is assigning the right set of detectors for each file. Ideally, we would like this set to be large enough to be accurate but also as small as possible so it is computationally-efficient. Striking this balance is by no means a trivial task, especially when a large number of detectors is available. The second challenge is the fact that different organizations have different preferences when faced with the need to balance detection accuracy, error-tolerance, and the cost of computing resources. Using these preferences to guide detector selection is an open and difficult problem.

To the best of our knowledge, every existing ensemble solution requires running all detectors prior to producing a classification. This requirement is a result of the supervised learning algorithm (e.g., SVM, Random Forest) often used for this purpose. As a result, not only are existing solution unable to address the first challenge we mention, they are also extremely constrained in addressing the second.

Even after setting aside the issue of computational cost (which is moot due to the use of all detectors for each file), striking the right balance between different types of classification errors – false-positive (FP) and false negative (FN) – remains a challenge. Usually, the only “tool” available for managing this trade-off is the confidence threshold, a value in the range of [0, 1] designating the level of certainty by the classifier of the file being malicious. Aside from being a blunt instrument (small changes in this value can cause large fluctuations in detection performance), recent studies [8] suggest that the confidence score is not as reliable an indicator as commonly assumed.

The use of reinforcement learning offers an elegant solution to both problems. First, this type of algorithms enables practitioners to assign clear numeric values to each classification outcome, as well as to quantify the cost of computing resources. These values reflect the priorities of the organization, and can be easily adapted and refined as needed. Secondly, once these values have been set, the reinforcement learning algorithm automatically attempts to define a policy (i.e., strategy) that maximizes them. This policy is likely to reflect organizational priorities much more closely than the use of a confidence threshold. Finally, since reinforcement learning algorithms are designed to operate based on partial knowledge, there is no need to run all detectors in advance; the algorithm interactively selects a single detector, evaluates its performance and then determines whether the benefit of using additional detectors is likely to be worth their computational cost. Moreover, the selection of detectors is dynamic, with different detector combinations used for different scenarios.

3 ASPIRE: AUTOMATED SECURITY POLICY IMPLEMENTATION USING REINFORCEMENT LEARNING

In this research we present ASPIRE, an automated security policy implementation using reinforcement learning. The goal of our approach is to automatically learn a security policy that best fits organizational requirements. More specifically, we train a deep neural network to dynamically determine when sufficient information exists to classify a given file and when more analysis is needed. The policy produced by our approach is shaped based on the values (i.e., rewards and costs) assigned to correct and incorrect file classifications, as well as to the use of computing resources. We introduce a RL framework that explores the efficacy of various detector combinations and continuously performs cost-benefit analysis to select optimal detector combinations.

The main challenge in selecting detector combinations can be modelled as an exploration/exploitation problem. While the cost (i.e., computing resources) of using a detector can be very closely approximated in advance, its benefit (i.e., the usefulness of the analysis) can only be known in retrospect. RL algorithms perform well in scenarios with high uncertainty where only partial information is available, a fact that makes them highly suitable for the task at hand. ASPIRE’s architecture, describing the interaction between the agent and the environment, is presented in Figure 1.

![Figure 1: ASPIRE high-level architecture](image)

We next present the state and action-spaces used by our approach and describe the cost/reward structure used in our experiments and the rationale of setting different security policies for different types of organizations.

**States.** The states that make up our environment consist of all possible score combinations by the participating detectors. More specifically, for a malware detection environment consisting of \( K \) detectors, each possible state will be represented by a vector \( V = \{v_1, v_2, ..., v_K\} \), with the value of \( v_x \) being set by

\[
    v_x = \begin{cases} 
    0,1 & \text{if detector } x \text{ has been applied} \\
    -1 & \text{otherwise} 
    \end{cases} 
\]  

(1)

Therefore, the initial state for each incoming file is a vector consisting entirely of -1 values. As various detectors are chosen to analyze the files, entries in the vector are populated with the confidence scores they provide. All scores are normalized to a [0, 1] range, where a confidence value of 1 indicates full certainty of the file being a malware and 0 indicates full certainty in its being benign. An example of a possible state vector can be seen in Figure 2.
When designing the reward function for the analysis runtime, we consider the number of available detectors in the environment. For an environment consisting of \( K \) detectors, the number of actions will be \( K + 2 \): one action for the activation of each detector, and two additional actions called “malicious” and “benign”. Each of the two latter actions produces a classification decision for the analyzed file, while also terminating the analysis process.

Rewards. The rewards need to be designed so that they reflect the organizational security policy, namely the tolerance for errors in the detection process and the cost of using computing resources:

- **Detection errors.** We need to consider two types of errors: false-positives (FP), which is the flagging of a benign file as malicious (i.e., a “false alarm”), and false-negative (FN), which is the flagging of a malicious file as benign. In addition to the negative rewards incurred by misclassification, it is also possible to provide positive reward for cases where the algorithm was correct. We elaborate on this further in Section 5 and present the various scoring schemes used in our evaluation.

- **Computing resources.** In this study we chose the time required to run a detector as the approximated cost of its activation. In addition to being a close approximator of other types of resources use (e.g., CPU, memory), running time is a clear indicator of an organization’s ability to process large volumes of incoming files. To put it simply, reducing the average time required to process a file enables organizations to process more files with less hardware.

When designing the reward function for the analysis runtime, we needed to address the large difference in this measure between various detectors. As shown in Table 2 in Section 4.2, average running times can vary by orders of magnitude (from 0.7s to 44.29s, depending on the detector). In order to mitigate these differences and encourage the use of the more “expensive” (but also more accurate) detectors, we define the cost function of the computing time as follows

\[
C(t) = \begin{cases} 
  t & \text{if } 0 \leq t \leq 1 \\
  \min \{1 + \log_2(t), 6\} & \text{if } t > 1
\end{cases}
\]

It is important to note that while we only consider running time as the computing resource whose cost needs to be taken into account, our approach can be easily adapted to include additional resources such as memory usage, CPU runtime, cloud computing costs and even electricity. As such, ASPIRE enables organizations to easily and automatically integrate all relevant costs into their decision making process, something that has not been possible before with other ML-based approaches.

\section{Dataset Malware Detection Analysis}

Our dataset consists of 24,737 PE files, equally divided between malicious and benign. While we were unable to determine the creation time of each file, all files were collected from the repositories of the network security department of a large organization in October 2018. We analyze each file using four different malware detectors, and make both the file corpus and the classification scores publicly available.\(^2\) In the remainder of this section we first describe the detectors used in our experiments and then analyze their performance – both in absolute terms and in relation to each other.

\subsection{The Detectors}

Our selection of detectors was guided by three objectives:

- **Off-the-shelf software.** The ability to use malware detection solution without any special adaptation demonstrates that our approach is generic and easily applicable.

- **Proven detection capabilities.** By using detectors that are also in use in real-world organizations we ensure the validity of our experiments.

- **Run-time variance.** Since the goal of our experiments is to demonstrate ASPIRE’s ability to perform cost-effective detection (with running time being our chosen cost metric), using detection solutions that vary in their resource requirements was deemed preferable. Moreover, such variance is consistent with real-world detection pipelines that combine multiple detector “families” [11].

Following the above-mentioned objectives, we selected four detectors to be included in our dataset: `pifile`, `byte3g`, `opcode2g`, and `manalyze`.

- **pifile.** This detector uses seven features extracted from the PE header: `DebugSize`, `ImageVersion`, `IatRVA`, `ExportSize`, `ResourceSize`, `VirtualSize2`, and `NumberOfSections`, all presented in [19]. Using those features, we trained a Decision Tree classifier to produce the classification.

- **byte3g.** This detector uses features extracted from the raw binaries of the PE file [17]. First, it constructs trigrams (3-grams) of bytes. Secondly, it computes the trigrams term-frequencies (TF), which are the raw counts of each trigram in the entire file. Thirdly, we calculate the document-frequencies (DF), which represent the rarity of a trigram in the entire dataset. Lastly, since the amount of features can be substantial (up to 256\(^3\)), we use the top 300 DF-valued features for classification. Using the selected features, we trained a Random Forest classifier with 100 trees.

- **opcode2g.** This detector uses features based on the disassembly of the PE file [16]. First, it disassembles the file and extract the opcode of each instruction. Secondly, it generates bigrams (2-grams) representation of the opcodes. Thirdly, both the TF and DF values are computed for each bigram. Lastly, as done for byte3g, we

\(^2\) Links to all materials will be provided pending acceptance.
select the 300 features with the highest DF values. Using the selected features, we trained a Random Forest classifier with 100 trees.

**manalyze.** This detector is based on open-source heuristic scanning tool named Manalyze\(^3\). This detector offers multiple types of static analysis capabilities for PE files, each implemented in a separate “plugin”. In our version we included the following capabilities: packed executables detection, ClamAV and YARA signatures, detection of suspicious import combinations, detection of cryptographic algorithms, and the verification of authenticode signatures. Each plugin returns one of three values: benign, possibly malicious, and malicious. Since Manalyze does not offer an out-of-the-box method for combining the plugin scores, we trained a Decision Tree classifier with the plugins’ scores as features.

### 4.2 Detectors Performance Analysis

In this section we analyze and compare the performance of the various detectors. We explore the effectiveness of various detector combinations and explain why the selection of only a subset of possible detectors is likely to produce near-optimal performance at a much lower computational cost.

**Overall detector performance.** We begin by analyzing the upper bound on the detection capability of our four detectors. In Table 1 we present a breakdown of all files in our dataset as a function of the number of times they were incorrectly classified by the various detectors. All detectors were trained and tested using 10-fold cross-validation, and we present an average of the results. We define Incorrect classification as a confidence threshold above 0.5 for a benign file or one that is equal or smaller than 0.5 for a malicious file.

| # Misclassification | # Files | % of Files |
|---------------------|---------|------------|
| 0                   | 18062   | 73.02      |
| 1                   | 5149    | 20.81      |
| 2                   | 969     | 3.92       |
| 3                   | 397     | 1.60       |
| 4                   | 160     | 0.65       |

The results in Table 1 show that approximately 73% of all files are classified correctly by all detectors, while only 0.65% (160 files) are not detectable by any method. We derive two conclusions from this analysis: a) Approximately ~26.5% of the files in our dataset potentially require that we use multiple detectors to achieve correct classification; b) only a small percentage of files (1.6%) is correctly classified by a single classifier, which means that applying all four detectors for a given file is hardly ever required. We argue that these conclusions support our hypothesis that a cost-effective approach for using only a subset of possible detectors.

**Absolute and relative detector performance.** Our goal in this analysis is first to present the performance (i.e., detection rate) of each detector, and then determine whether any classifier is dominated by another (thus making it redundant, unless it is more computationally efficient). We begin our analysis by presenting the absolute performance of each detector. As can be seen in Table 2, the accuracy of the detectors ranges between 82.88%–95.5%, with the more computationally-expensive detectors generally achieving the better performance.

| Detector     | Accuracy (%) | TPR    | FPR    | Mean Time (sec) |
|--------------|--------------|--------|--------|-----------------|
| manalyze     | 82.88        | 0.844  | 0.186  | 0.75            |
| pefile       | 90.59        | 0.902  | 0.090  | 0.70            |
| byte3g       | 94.89        | 0.937  | 0.039  | 3.99            |
| opcode2g     | 95.50        | 0.951  | 0.041  | 42.99           |

Next we attempted to determine whether any detector is dominated by another. For each detector, we analyzed the files it misclassified in order to determine whether they would be correctly classified by another detector. The results of this analysis, presented in Table 3, show that no detector is being dominated. Moreover, the large variance in the detection rates of other detectors for misclassified files further suggests that an intelligent selection of detector subsets – where the detectors complement each other – can yield high detection accuracy.

**Table 2: The performance of the participating detectors.** We present overall accuracy, the true-positive (malware detection) rate and the false-positive (misclassification of benign files) rate. In addition, we present the mean running time of each detector, calculated over all files in the dataset. The running times were measured on machines utilizing the same specifications, detailed in Section 5.1.

`. Table 3: Complementary detection performance. For the detectors presented in each row, we show the detection accuracy of the other detectors on the files it misclassified.

| Detector     | Accuracy (%) | TPR    | FPR    | Mean Time (sec) |
|--------------|--------------|--------|--------|-----------------|
| manalyze     | -            | 82.96% | 90.09% | 91.01%          |
| pefile       | 68.96%       | -      | 73.43% | 78.93%          |
| byte3g       | 66.24%       | 50.32% | -      | 60.69%          |
| opcode2g     | 65.71%       | 55.90% | 55.99% | -               |

**Detectors confidence score distributions.** Next we analyze the confidence score distribution of the various detectors. Our goal in this analysis is to determine whether the detectors are capable of nuanced analysis; we hypothesize that detectors which produce multiple values on the [0,1] scale (rather than only 0s and 1s) might enable our DRL approach to devise more nuanced policies for selecting detector combinations.

The results of our analysis are presented in Figure 3. While it is clear that all detectors assign either 0s or 1s to the majority of files, a large number of files (particularly for the less-expensive,
Table 4: The running time and performance of all possible malware detector combinations

| #  | Detector Combination                           | Aggregation Method | Mean Accuracy (%) | Mean Time (sec) | FP (%) | FN (%) |
|----|-----------------------------------------------|--------------------|-------------------|-----------------|--------|--------|
| 1  | manalyze, pe/file, byte3g, opcode2g           | stacking (RF)      | 96.86             | 49.73           | 1.52   | 1.62   |
| 2  | manalyze, byte3g, opcode2g                    | majority           | 96.71             | 49.03           | 1.45   | 1.84   |
| 3  | manalyze, pe/file, byte3g, opcode2g           | majority           | 96.65             | 49.73           | 1.40   | 1.95   |
| 4  | byte3g, opcode2g                              | majority           | 96.37             | 48.28           | 1.65   | 1.98   |
| 5  | pe/file, byte3g, opcode2g                     | majority           | 96.30             | 48.98           | 1.61   | 2.09   |
| 6  | manalyze, pe/file, byte3g                     | majority           | 95.98             | 45.74           | 1.77   | 2.25   |
| 7  | manalyze, pe/file, byte3g                     | majority           | 95.62             | 5.44            | 1.95   | 2.43   |
| 8  | byte3g, opcode2g                              | majority           | 95.57             | 48.28           | 3.23   | 1.20   |
| 9  | manalyze, opcode2g                            | majority           | 95.56             | 45.04           | 2.12   | 2.32   |
| 10 | opcode2g                                     | none               | 95.50             | 44.29           | 2.07   | 2.43   |
| 11 | pe/file, opcode2g                             | majority           | 95.44             | 44.99           | 2.06   | 2.49   |
| 12 | manalyze, pe/file, byte3g, opcode2g           | stacking (DT)      | 95.16             | 49.73           | 2.48   | 2.36   |
| 13 | manalyze, byte3g                              | majority           | 95.15             | 4.74            | 2.43   | 2.43   |
| 14 | byte3g                                       | none               | 94.89             | 3.99            | 1.96   | 3.15   |
| 15 | pe/file, byte3g                               | majority           | 94.85             | 4.69            | 2.32   | 2.83   |
| 16 | pe/file, opcode2g                             | or                 | 92.99             | 44.99           | 5.81   | 1.19   |
| 17 | pe/file, byte3g                               | or                 | 92.83             | 4.69            | 5.55   | 1.63   |
| 18 | pe/file, byte3g, opcode2g                     | or                 | 92.62             | 48.98           | 6.58   | 0.80   |
| 19 | manalyze, pe/file                             | majority           | 92.40             | 1.45            | 3.47   | 4.14   |
| 20 | pe/file                                      | none               | 90.60             | 0.70            | 4.52   | 4.88   |
| 21 | manalyze, opcode2g                            | or                 | 88.67             | 45.04           | 10.56  | 0.77   |
| 22 | manalyze, byte3g                              | or                 | 88.58             | 4.74            | 10.51  | 0.91   |
| 23 | manalyze, byte3g, opcode2g                    | or                 | 88.13             | 49.03           | 11.40  | 0.47   |
| 24 | manalyze, pe/file, opcode2g                   | or                 | 86.31             | 45.74           | 13.27  | 0.42   |
| 25 | manalyze, pe/file, byte3g                     | or                 | 86.28             | 5.44            | 13.13  | 0.60   |
| 26 | manalyze, pe/file                             | or                 | 86.23             | 1.45            | 12.36  | 1.41   |
| 27 | manalyze, pe/file, byte3g, opcode2g           | or                 | 85.82             | 49.73           | 13.88  | 0.30   |
| 28 | manalyze                                     | none               | 82.88             | 0.75            | 9.32   | 7.80   |

less-accurate detectors) receives intermediary values. We therefore conclude that the classifications produced by the detectors are sufficiently diverse to support a nuanced DRL-policy. The efficacy of this policy will be evaluated by our experiments, presented in Section 5.3.

Detectors combinations performance and time consumption. Finally, we provide a comprehensive analysis on the performance and time consumption for all possible detector combinations, presented in Table 4. To evaluate the performance of each combination, we aggregated the confidence score using three different methods, presented in [12]. The first method, or, classifies a file as malicious if any of the participating detectors classifies it as such (yields a score of 0.5 and above). This method mostly improves the sensitivity, but at the cost of higher false-positives percentage (benign files classified as malicious). The second method, majority, uses voting to classify the files. The third method, stacking, combines the classification confidence scores by training a ML model, with the scores provided as its features. In our evaluation, we used two types of classifiers – Decision Tree (DT) and Random Forest (RF) – and evaluated each using 10-fold cross-validation.

Interestingly, our analysis shows that in the case of majority, the optimal performance is not achieved by combining all classifiers, but rather only three of them. Furthermore, some detector combinations (manalyze, pe/file, byte3g) outperform other detector sets while also being more computationally efficient. The results further support our claim that an intelligent selection of detector combinations is highly important.

It should be noted that for each file, the times were measured in an isolated computer process on a dedicated machine to prevent other processes interruptions. In addition, the machines executing the detectors were identical utilizing the same hardware and firmware specifications.

5 EVALUATION

To the best of our knowledge, this study represents the first attempt to craft a security policy by performing a cost-benefit analysis that takes into account the resources required to use various detectors. In this section we evaluate the performance of our proposed approach in several scenarios and demonstrate its effectiveness. Moreover, we show that simple adjustments to our algorithm’s reward function (which reflects the organization’s priorities) leads to significant changes in the detection strategy. We argue that this approach is more effective (and intuitive) than existing approaches.
The remainder of this section is organized as follows: we begin by describing the environment used for running our experiments. Next, we describe our experimental setup and evaluated scenarios. Finally, we present the results of our evaluation and offer an analysis.

5.1 The Evaluation Environment

We used three VMware ESXi servers, each containing two processing units (CPUs). Each server had a total of 32 cores, 512GB of RAM and 100TB of SSD disk space. Two servers were used to run the environment and its detectors, while the remaining server housed our DRL agent. In our experiments, we deployed two detectors in each server. This deployment setting can easily be extended to include additional detectors or replicated to increase the throughput of existing ones. Our main goal in setting up the environment was to demonstrate a large scale implementation which is both scalable and flexible, thus ensuring its relevance to real-world scenarios. Figure 4 presents our infrastructure structure in detail.

Both the agent processes and the detectors run on virtual machines with the Ubuntu 18.04 LTS operating system. Each machine has 4 CPU cores, 16GB of RAM and 100GB of SSD storage. The agent uses a management service that allows both the training and execution of the DRL algorithm, using different tuning parameters. Upon the arrival of a file for analysis, the agent stores it in a dedicated storage space, which is also accessible to all detectors running in the environment. The agent also utilizes an external storage to store file and detector-based features, all logging information, and the analysis output. All this information is later indexed and consumed by an analytics engine. The agent communicates with the environment over HTTP protocol.

5.2 Experimental Setup

The following settings were used in all our experiments:

- We used 10-fold cross validation in all experiments, with label ratios maintained for each fold. The results presented in this study are the averages of all runs.
- We implemented the framework using Python v3.6. More specifically, we used the ChainerRL\(^1\) deep reinforcement library to create and train the agent, while the environment was implemented using the OpenAI Gym [4].
- Both the policy network and the action-value network consist of the following architecture: input layer of size 4 (the state vector’s size), a single hidden layer of size 20 and an output layer of size 6 (the size of the action space – four detectors and the two possible classifications). All layers except for the output used the ReLU activation function, while the output layer used softmax.
- We set our initial learning rate to \(7e^{-4}\), with exponential decay rate of 0.99 and a fuzz factor (epsilon) of \(1e - 2\). Our chosen optimizer was RMSprop [28]. In all experiments, our model trained until convergence.
- We set the size of the replay buffer to 5000. We start using it in the training process after 10,000 episodes.
- In order to discourage the agent from querying the same detector twice (which is an obvious waste of resources, since no new information is gained), we define such actions to incur a very large cost of \(-10,000\). The same “fine” applies to attempts to classify a file without using even a single detector.

---

\(^1\)https://github.com/chainer/chainerrl

Figure 3: The distribution of the files in our dataset based on confidence score assigned to them by each detector.

Figure 4: Experimental Setup Infrastructure Architecture
5.3 Experimental Results

We hypothesize that our proposed ASPIRE approach has two major strengths: a) it can produce near-optimal performance at reduced computational cost; and b) the use of rewards enables us to easily define and tune our security policies by assigning a “personalized” set of detectors for each file.

To test the robustness of our approach, as well as its ability to generalize, we define five use-cases with varying emphasis on correct/incorrect file classifications and computational cost. The rewards composition of each use-case is presented in Table 7, along with its overall accuracy and mean running time. It is important to note that the computational cost of using a detector is never calculated independently, but rather as a function of correct/incorrect file classification. Additionally, the computational costs of the malware detectors were defined based on the average execution time of the files we used for training. This practice enabled the algorithm to converge faster. Our experiments show that this type of setting outperforms other approaches for considering computational cost, as it strongly ties the invested resources to the classification outcome. Next we describe our five use-cases and their rationale.

**Experiment 1.** In this experiment we set both the reward for correct classification and the cost of incorrect classification to be equal to the cost of the running time. On one hand, this setting “encourages” ASPIRE to invest more time analyzing incoming files and also provides higher rewards for the correct classification of more challenging files. On the other hand, the detector is discouraged from selecting detector configurations that are likely to reduce its accuracy for a given file. Additionally, our approach is not likely to be inclined to pour additional resources into difficult-to-classify cases where the investment of more time is unlikely to provide additional information.

**Experiment 2.** This setting of this experiment is similar to that of experiment 1, except for the fact that the cost of incorrect classifications is 10x higher than the reward for correct ones. We hypothesized that this setting will cause the algorithm to be more risk-averse and invest additional resources in the classification of challenging files.

Please note that experiments 1 & 2 are not designed to assign high priority to resource efficiency, but instead focus on accuracy. The remaining experimental settings are designed to give greater preference to the efficient use of resources.

**Experiments 3-5.** In this set of experiments we explore policies where the rewards assigned to correct classification are fixed while the cost of incorrect classification depend on the amount of computing resources spent to reach the classification decision. We explore three variants of this approach, where the cost of incorrect classification remains the same but the rewards for correct classifications are larger by one and two orders of magnitude (1, 10, and 100).

This set of experiments has two main goals: first, since only the cost of an incorrect classification is time-dependent, we expect experiments 3-5 to be more efficiency-oriented. Our aim is to determine the size of this improvement and its effect on the accuracy of our approach. Secondly, we are interested in exploring the effect of varying reward/cost ratios on the policy generated by ASPIRE. Since we explore scenarios in which the reward for correct classifications is either significantly smaller or larger than the cost of incorrect ones, our expectation was to obtain better understanding of ASPIRE’s decision mechanism.

**Results.** A summary of the results is presented in Table 5 while a detailed breakdown of the detector combinations used by each of our generated DRL policies can be found in Table 6. We present a detailed comparison of the results obtained by our various experiments is shown in Tables 7-10.

**Table 5: The cost/reward setup of our experiments. The function C(t) is presented in Equation 2.**

| Exp. # | TP | TN | FP | FN | Accuracy (%) | Mean Time (sec) |
|-------|----|----|----|----|--------------|----------------|
| 1     | C(t) | C(t) | -C(t) | -C(t) | 98.81 | 49.63 |
| 2     | C(t) | C(t) | -10C(t) | -10C(t) | 98.82 | 49.58 |
| 3     | 1 | 1 | -C(t) | -C(t) | 96.21 | 49.63 |
| 4     | 10 | 10 | -C(t) | -C(t) | 95.42 | 3.68 |
| 5     | 100 | 100 | -C(t) | -C(t) | 91.22 | 0.73 |

**Table 6: Distribution of detector combination choices made by the agent for each of our experimental policies.**

| Exp. # | Acc. (%) | Action sequences | Time (sec) Files (%) |
|-------|----------|------------------|---------------------|
| 1     | 96.81    | byte3g,opcode2g,manalyze,pefile | 49.73 86.82 |
|       |          | byte3g,opcode2g,manalyze | 49.03 8.40 |
|       |          | byte3g,opcode2g,pefile | 48.98 4.54 |
|       |          | byte3g,opcode2g | 48.28 0.24 |
| 2     | 96.79    | opcode2g,manalyze,pefile,byte3g | 49.73 80.11 |
|       |          | opcode2g,pefile,byte3g | 48.98 19.89 |
| 3     | 96.21    | byte3g,pefile,opcode2g | 48.98 12.67 |
|       |          | byte3g,pefile | 4.69 2.15 |
|       |          | byte3g,pefile,opcode2g,manalyze | 49.73 80.11 |
| 4     | 95.42    | manalyze,byte3g,pefile | 5.44 50.77 |
|       |          | manalyze,pefile | 1.45 22.49 |
|       |          | manalyze | 0.75 16.89 |
|       |          | manalyze,byte3g | 4.74 16.89 |
| 5     | 91.22    | pefile,manalyze | 0.70 96.17 |
|       |          | pefile | 1.45 3.83 |

Overall, the results show that ASPIRE is capable of generating highly effective detection policies. The policies generated in experiments 1-2 outperformed all the methods presented in the baseline except for the top-performing one, which is a combination of all classifiers and the Random Forest algorithm. While this baseline method marginally outperforms our approach (98.86% to 96.81% and 96.79% for experiments 1 and 2 respectively), it is also slightly more computationally expensive (49.74 seconds on average compared with 49.63 and 49.58 for experiments 1 and 2 respectively).
Our experiments clearly demonstrate that security policy can be very effectively managed through the use of different cost/reward combinations. Moreover, it is clear that the use of DRL offers much greater flexibility in the shaping of the security policy than the simple tweaking of the confidence threshold (the only available method for most ML-based detection algorithms).

These results are as we expected, since the policies we defined for experiments 1 and 2 were geared towards accuracy rather than efficiency.

The policies generated by experiments 3-5 are more interesting, as they each achieve a different accuracy/efficiency balance. Moreover, each of the three policies was able to reach accuracy results that are equal or better to those of the corresponding baselines at a much lower cost. The policy generated by experiment 3 reached an accuracy of 96.21% with a mean time of 10.5 seconds, compared with its closest baseline “neighbor” which achieved an accuracy of 96.3% in a mean time of of 48.28 seconds (almost five time longer). Similarly, the policy produced by experiment 4 achieved the same accuracy as its baseline counterpart (pe/ile,opcode2g) while requiring only 3.68 seconds on average compared with the baseline’s 45.28 seconds ~ a 2% improvement. The policy generated by experiment 5 requires 0.728 seconds per file on average, which is comparable to time required by the baseline method “pe/ile”. Our approach, however, achieves higher accuracy (91.22% vs 90.6%).

**Analysis.** Our experiments clearly demonstrate that security policy can be very effectively managed through the use of different cost/reward combinations. Moreover, it is clear that the use of DRL offers much greater flexibility in the shaping of the security policy than the simple tweaking of the confidence threshold (the only available method for most ML-based detection algorithms).

When analyzing the behavior (i.e., the detector selection strategy) of our policies, we find that they behaved just as we could have expected. The policies generated by experiments 1 and 2 explicitly favored performance over efficiency, as the reward for correct classification was also time-dependent. As a result, they achieve very high accuracy but only a marginal improvement in efficiency. For experiments 3-5, the varying fixed cost that we assigned to the correct classifications played a deciding role in creating the policy. In experiment 3, the relative cost of a mistake was often much larger than reward for a correct classification. Therefore, the generated policy is cautious, achieving relatively high accuracy (but at impressive efficiency). In experiment 5, the cost of an incorrect classification is relatively marginal, a fact that motivates the generated policy to prioritize speed over accuracy. The policy generated by experiment 4 offers the middle ground, reaching a slightly reduced accuracy compared with experiment 3, but managing to do so in about 33% of the running time.

Finally, we consider it important to elaborate on the major strength of our approach: the ability to craft a “personalized” set of detectors for each file. In Figure 5 we show the overall distributions of detector combinations (represented by their running time), chosen by the policies of all experiments. The differences among the different policies are clear, showing the clear connection between rewards and policies. Experiment 4 offers an excellent use case for this connection, as its policy utilizes multiple detector combinations...
of varying costs. This diversity helps to explain ASPIRE’s ability to achieve high accuracy at much smaller computational cost. It is important to stress again that the detector combinations are not chosen in advance. Instead, they are chosen iteratively, with the confidence score of the already-applied detectors used to guide the next step chosen by the policy.

6 RELATED WORK

6.1 Malware Detection Techniques for PEs

Portable executable files can be represented in multiple ways, a fact that has contributed to the large number of approaches proposed for its analysis. The most common (and simple) approach of representing a PE file is by calculating its hash value [9]. This method is frequently used by anti-virus engines to “mark” and identify malware, as both computing and retrieving hashes is fast and efficient.

Additional studies propose representing PEs using their binary data. Wan et al. [17], for example, suggest using a dictionary of byte n-grams (sequences of n bytes) for malware classification. The authors examined different n-grams sizes ranging from three to six, as well as three feature selection methods. They experimented with four types of models: artificial neural network (ANN), decision tree (DT), naïve bayes (NB) and support vector machine (SVM). The decision tree algorithm achieved the best accuracy of 94.3% with less than 4% of false-positives.

Another type of features is generated using the disassembly of a PE file and extracting opcode n-grams. The use of opcode n-grams to classify malware was suggested by [16]. The authors examined different sizes of n-grams ranging from three to six, as well as three feature selection methods. To classify the files, they used several models such as ANN, DT, Boosted DT, NB and Boosted NB. The best results achieved by the DT and the Boosted DT models, with more than 93% accuracy, less than 4% false-positives and less than 17% false-negatives.

Lastly, the PE format (i.e., metadata) can be used to represent the PE file [1, 5, 19]. The format of PE files has a well-defined structure, which includes information necessary to the execution process, as well as some additional data (such as versioning info and creation date). In [19], the authors used seven features extracted from the PE headers to classify malicious files: DebugSize, ImageVersion, IatRVA, ExportSize, ResourceSize, VirtualSize2, and NumberOfSections. The study presents the results of multiple machine learning algorithms used for classifying the PEs: IBK, Random Forest, J48, J48 Graft, Ridor and PART. The evaluation results show similar performance for all classifiers, reaching an accuracy of up-to 98.56% and a false-positive rate as lower as 5.68%.

6.2 Reinforcement Learning in Security Domains

Reinforcement learning is used in the security domains mainly for adversarial learning and malware detection. In the field of adversarial learning, RL can be successfully used to modify malware files as to better avoid detection [1]. This goal was achieved by attacking static analysis detector while equipping the agent with a set of malicious functionality-preserving operations.

In the malware detection domain, Silver et al. [3] presented a proof of concept for an adaptive rule-based malware detection framework. The proposed framework employs a learning classifier systems combined with a rule-based expert system. The VirusTotal online malware detection service served as the PE file malware classifier, using multiple static PE file feature for detection. A reinforcement learning algorithm was then used to determine weather a PE is malicious.

In their paper, Mohammadkhani and Esmaeilpour [15] used RL for classifying different malware types using a set of features commonly used by anti virus software. A similar example in the same domain was presented by [31] for optimizing malware detection on mobile devices. The authors used reinforcement learning to control the offloading rate of application traces to the security server, an optimization that is critical for mobile devices. The proposed solution consisted of a deep Q-network coupled with a deep convolutional neural network.

7 CONCLUSIONS AND FUTURE WORK

In this research we have presented ASPIRE, a RL-based approach for malware detection. Our approach dynamically and iteratively assigns various detectors to each file, constantly performing cost-benefit analysis to determine whether the use of a given detector is “worth” the expected reduction in classification uncertainty. The entire process is governed by the organizational policy, which sets
the rewards/costs of correct and incorrect classifications and also defines the cost of computational resources.

When compared to existing ensemble-based solution, our approach has two main advantages. First, it is highly efficient, since easy-to-classify files are likely to require the use of less-powerful classifiers, a fact that gives us the ability to maintain near-optimal performance at a fraction of the computing cost. As a result, it is possible to analyze a much larger number of files without increasing hardware capacity. Secondly, organizations can clearly and easily define and refine their security policy by explicitly setting the costs of each element of the detection process: correct/incorrect classification and resource use. Since the value of each outcome is clearly quantified, organizations can easily experiment with different values and fine-tune the performance of their models to the desired outcome.

In future work, we intend to explore several directions. First, we would like to increase the number of detectors and integrate a dynamic analysis component in our environment. The use of dynamic analysis involves multiple challenges (for example, setting up the required environments and their analysis) and is therefore a challenging field of research. Secondly, we would like to explore the use of our approach in a transfer learning setting, where a model trained on set of detectors is used as shortest the required training period for other configurations.

REFERENCES

[1] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. 2017. Evading machine learning malware detection. Black Hat (2017).

[2] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel JG Van Etten, Michael Levi, Tyler Moore, and Stefan Savage. 2013. Measuring the Cost of Cybercrime. In The Economics of Information Security and Privacy. Springer, 265–300.

[3] Jonathan J Blount, Daniel R Tauritz, and Samuel A Mulder. 2011. Adaptive Rule-Based Malware Detection Employing Learning Classifier Systems: A Proof of Concept. In 2011 IEEE 55th Annual Computer Software and Applications Conference Workshops. IEEE, 110–113.

[4] Greg Brockman, Vicki Chen, Ludwig Pettersson, Joannes Vermorel, Mehryar Mohri. 2018. OpenAI Gym. arXiv:arXiv:1801.01973

[5] Young Han Choi, Byoung Jin Han, Bynong Chul Bae, Hyung Geun Oh, and Ki Wook Sohn. 2012. Toward extracting malware features for classification using static and dynamic analysis. In 2012 8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC). IEEE, 126–129.

[6] Anuha Damdaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. Journal of Computer Virology and Hacking Techniques 13, 1 (2017), 1–12.

[7] Mojtaba Eskandari, Zeinab Khorshidpour, and Sattar Hashemi. 2013. HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection. Journal of Computer Virology and Hacking Techniques 9, 2 (2013), 77–93.

[8] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In international conference on machine learning. 1050–1059.

[9] Kent Griffin, Scott Schneider, Xin Hu, and Tri-Cker Chieu. 2009. Automatic Generation of String Signatures for Malware Detection. In International workshop on recent advances in intrusion detection. Springer, 101–120.

[10] Jieqiong Hou, Minhui Xue, and Haifeng Qian. 2017. Unleash the Power for period for other configurations. Secondly, we would like to explore the desired outcome.

[11] Nwokedi Idika and Aditya P Mathur. 2007. A survey of malware detection techniques. Purdue University 48 (2007).

[12] Khaled N Khawasmeh, Meltem Oztsoy, Caleb Donovick, Nael Abo-Ghazaleh, and Dmitry Ponomaryov. 2015. Ensemble Learning for Low-level Hardware-supported Malware Detection. In International Symposium on Recent Advances in Intrusion Detection. Springer, 3–25.

[13] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning 8, 3-4 (1992), 295–321.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. Nature 518, 7540 (2015), 529.

[15] Sepideh Mohammadkhani and Mansour Esmaeilpour. 2018. A new method for behavioural-based malware detection using reinforcement learning. International Journal of Data Mining, Modelling and Management 10, 4 (2018), 314–330.

[16] Robert Moskovich, Clint Feher, Nir Trachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. 2008. Unknown Malcode Detection Using OPCODE Representation. In European Conference on Intelligence and Security Informatics. Springer, 204–215.

[17] Robert Moskovich, Dima Stoppel, Clint Feher, Nir Nissim, and Yuval Elovici. 2008. Unknown malcode detection via text categorization and the imbalance problem. In 2008 IEEE International Conference on Intelligence and Security Informatics. IEEE, 156–161.

[18] Radu S Firsoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. 2015. Analysis of malware behavior: Type classification using machine learning. In 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA). IEEE, 1–7.

[19] Karthik Raman et al. 2012. Selecting Features to Classify Malware. InfoSec Southwest 2012 (2012).

[20] Konrad Rieck, Philipp Trinitis, Carsten Willems, and Thorsten Holz. 2011. Automatic Analysis of Malware Behavior Using Machine Learning. Journal of Computer Security 19, 4 (2011), 639–668.

[21] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In International Conference on Machine Learning. 1809–1807.

[22] Christian Robert Shelton. 2001. Importance Sampling for Reinforcement Learning with Multiple Objectives. (2001).

[23] PV Shijo and A. Salim. 2015. Integrated Static and Dynamic Analysis for Malware Detection. Procedia Computer Science 46 (2015), 804–811.

[24] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneer selvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. nature 529, 7587 (2016), 484.

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. Nature 550, 7676 (2017), 354.

[26] Felipe Petrostich-Vuch, Vashidh Madhavan, Edordo Conti, Joen Lemhan, Kenneth O Stanely, and Jeff Clune. 2017. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 (2017).

[27] Richard S Sutton and Andrew G Barto. 2018. Reinforcement Learning: An Introduction (2nd ed.). MIT press Cambridge.

[28] Tijmen Tieleman and Geoffroy Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSER: Neural networks for machine learning 4, 2 (2012), 26–31.

[29] Scott Tweedwell and Mian Zhou. 2009. A heuristic approach for detection of obfuscated malware. In 2009 IEEE International Conference on Intelligence and Security Informatics. IEEE, 291–299.

[30] Joannes Vermorel and Mehrayeh Mohri. 2005. Multi-armed Bandit Algorithms and Empirical Evaluation. In European Conference on Machine Learning. Springer, 437–448.

[31] Xiaoqyue Wan, Geyi Sheng, Zanda Li, Liang Xiao, and Xiaojiang Du. 2017. Reinforcement Learning Based Mobile Offloading for Cloud-Based Malware Detection. In GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, 1–6.

[32] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. 2016. Sample Efficient Actor-Critic with Experience Replay. arXiv:1611.02244 (2016).

[33] Ilus You and Kangbin Tim. 2010. Malware Obfuscation Techniques: A Brief Survey. In 2010 International Conference on Broadband, Wireless Computing, Communication and Applications. IEEE, 297–300.