Active Malware Analysis using Stochastic Games

Simon A. Williamson
School of Information Systems
Singapore Management University
Singapore
swilliamson@smu.edu.sg

Pradeep Varakantham
School of Information Systems
Singapore Management University
Singapore
pradeepv@smu.edu.sg

Debin Gao
School of Information Systems
Singapore Management University
Singapore
dbgao@smu.edu.sg

Ong Chen Hui
DSO National Laboratories
Singapore
ochenhui@dso.org.sg

ABSTRACT

Cyber security is increasingly important for defending computer systems from loss of privacy or unauthorised use. One important aspect is threat analysis — how does an attacker infiltrate a system and what do they want once they are inside. This paper considers the problem of Active Malware Analysis, where we learn about the human or software intruder by actively interacting with it with the goal of learning about its behaviours and intentions, whilst at the same time that intruder may be trying to avoid detection or showing those behaviours and intentions. This game-theoretic active learning is then used to obtain a behavioural clustering of malware, an important contribution for both understanding malware at a high level and more crucially, for the deployment of effective anti-malware defences. This paper makes the following contributions: *(i)* A formal definition of the game-theoretic active malware analysis problem; *(ii)* A fast algorithm for learning about a malware in the active analysis problem which utilises the concept of reducing entropy in the beliefs about the malware; *(iii)* A virtual machine based agent architecture for the implementation of the active malware analysis problem and *(iv)* A behaviour based clustering of malware behaviour which is shown to be more accurate than a similar clustering using only passive information about the malware.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security, Experimentation

Keywords

Malware Analysis, Stochastic Game

1. INTRODUCTION

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems – Innovative Applications Track (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

Cyber security has emerged as one of the most important problems in the modern internet age, with cyber attacks resulting in millions of pounds of damage to organisations and individuals. The rise of the internet has enabled the propagation of malicious software (malware), exposing home computer users and organisations alike to threats previously unimagined. Such threats include the stealing of users private data such as usernames, passwords and email contacts. These malware can also turn an unwitting computer users system into a tool for attacking the rest of the internet such as in Distributed Denial of Service attacks and the sending of spam. As a result, the problem then is of *Cyber Security* — how to defend a computer system against unauthorised use either by a piece of software or by a human attacker.

With this established, the aspect of cyber security considered in this paper is in threat analysis: Given a piece of malware or a human intruding on a system, can we learn about the behaviours and intentions of that intruder. This is also known as malware analysis. Here behaviours are taken to mean the software vulnerabilities exploited to gain access to some part of a system (such as opening a port or installing some keylogging software), and intentions are the goals of the attackers, do they want to steal some information or use the machine as part of a larger coordinated attack. This process has clear game-theoretic implications since intruders often want to mask their access in order that the attack can be used again. Consequently, we have the situation that we want to learn the maximum information possible about a malware on our computer system. Conversely that malware wants to avoid giving away information about it's behaviours and intentions. It is important to note that this is a central problem in cyber security, as the results of malware analysis are used to power virus detection.

Against this background, malware analysis is performed by a human security expert on each newly discovered malware. The security expert must use various tools to identify how the malware infiltrates a working system, how it propagates to other systems and what it does whilst on the infected system. This is a time consuming manual process during which the the expert will examine the malware binary, execute it, examine logs and make some trial interactions with the infected system. Due to the volume of threats that need to be analysed, several authors have proposed automated analysis techniques. Now, these techniques can generally be classified according to whether they are *static*, which means that the binary is analysed, or *dynamic*,

which means that the malware is executed and its effects monitored. Automated static analysis techniques include *Eureka* [6] which analyses the code and produces a control flow graph representing the malware logic. It does this by scanning for system calls and grouping them together and assigning functions (since several system calls are involved in a high level operation such as creating a file). However static analysis is becoming increasingly difficult as malware becomes more sophisticated at using techniques such as code obfuscation (where the binary is randomised but preserves the original logic) and polymorphic binaries (the code is mutated to change its identifiable features such as variable names). Consequently, dynamic analysis is an interesting avenue to consider further. Here the malware is actually executed and the results are analysed (which bypasses the problems with static analysis). Several automated analysis techniques have been proposed here including [1], [7] and [4] which gather execution traces (a list of the operations the malware performed) and then attempt to classify these traces. There are several means to classify the traces i.e. support vector machines or distance measures [2].

That said, we can identify a weakness with automated dynamic analysis techniques when compared with a human security expert. Specifically, all of these techniques are *passive*, meaning that the malware is executed and a log is generated. However, a human security expert would interact with an infected system by placing *honey* in several locations. Examples of this include creating fake Internet Explorer activity or sending emails. Malwares have unique responses to such activity which are missed by passive analysis. Consequently, we propose an automated technique based on *Active Dynamic Analysis*. This means that the system will interact with the malware, basing its next action on the response of the malware, with the aim of learning the maximum amount of information about the policy of that malware. There exist some steps in this direction such as [5] which acknowledges that some malware require an input and so they define a set of possible inputs to test. However, a weakness with this approach is that the input sets must be defined before execution and are not reactive to what the malware has done thus far. This makes this approach potentially slow since many unproductive paths may be explored. We will attempt to address this weakness by using software agents to *react* to what the malware has done and choose the next input. Table 1 summarises our classification of this space.

Table 1: Malware Analysis

	Dynamic	Static
Active	[5]	
Passive	[1], [7], [2], [4]	[6]

Finally, this paper makes the following contributions: *(i)* A formal definition of the game-theoretic active malware analysis problem; *(ii)* A fast algorithm for learning about a malware in the active analysis problem which utilises the concept of reducing entropy in the beliefs about the malware; *(iii)* A virtual machine based agent architecture for the implementation of the active malware analysis problem and *(iv)* A behaviour based clustering of malware behaviour which is shown to be more accurate than a similar clustering using only passive information about the malware.

2. MOTIVATING SCENARIOS

This section provides examples of active malware analysis on a single machine and on a network of machines. In the rest of this paper we will formalise these examples in terms of the *Active Malware Analysis Game*. We provide these illustrative examples to make concrete the process of active malware analysis and how it contrasts with simple passive analysis. Specifically, both of these examples will show that simply passively monitoring what a malware does is not guaranteed to find all aspects of that malwares behaviour and that many modern malwares only exhibit some actions in response to data or actions on the infected system, or even worse than that, may potentially try to evade analysis.

2.1 Malware acting on a Single Machine

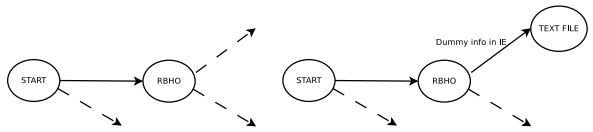
We first show the difference between passive and active analysis in the case of a single malware. Now, passive analysis:

*The malware **BZub.ji** is executed in a clean test environment. The subsequent trace is analysed by a technician and it is discovered that a browser helper object (RBHO) has been installed, which calls a new program placed in the Windows system directory. This program is analysed separately, but code obfuscation techniques render static analysis redundant. The program is executed but seems to be inert.*

By way of contrast, using active analysis techniques:

*The malware **BZub.ji** is executed in a clean test environment, alongside an analysis agent. The agent monitors that a browser helper object has been installed and that a secondary program has been installed. The agents model of existing malware indicates that these two activities indicate a modification of Internet Explorer has been used. The agent tests this aspect of it's model by executing a simulated interaction with Internet Explorer and it observes that a file has been created and is updated as it uses Internet Explorer. Simple textual matching shows that this file contains some of the honey that it used in IE. Consequently the agent has confirmed an aspect of the malware behaviour that passive analysis could only find with a human help.*

Figure 1 shows the difference in information received between the two types of analysis.



(a) Passive analysis (b) Active analysis

Figure 1: Analysis of the single malware scenario

2.2 Malware acting on a Network (Botnets)

The following example takes place in the context of a network of computer systems linked by some arbitrary topology. Assuming infiltration starts with a single machine, we first describe what we learn about the botnet using passive techniques and then contrast with active analysis:

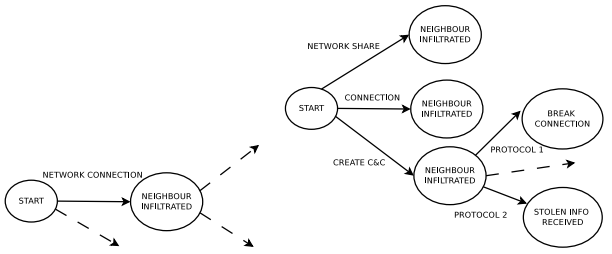
*The malware **Sinowal.aj** is executed on a single clean machine. Analysis of subsequent network traffic reveals that the infiltrated machine attempts to connect to a series of domains (presumably the command centre of the botnet). However no domains exist within the restricted network, so nothing further happens. Some connections to neighbours are recorded and the botnet grows larger by capturing these machines. The botnet takes no further action.*

This is contrasted with a more active analysis:

*The malware **Sinowal.aj** is executed on a single clean machine. The infiltrated machine attempts to connect to a*

series of domains, so a second machine on the network (with an analysis agent) poses as one of these domains and a connection is formed. This machine is now posing as the command centre of the botnet. After some trial and error with the command protocol employed by the botnet, the machine is able to successfully communicate with the bot. Fake activity on the infiltrated machine causes it to connect to other machines on the network via shares and it is found that the malware can spread through these actions. Dummy information placed on these machines is also found to be harvested and sent to the command machine. Note that ***honey*** *can take the form of fake private information such as passwords or fake user activity such as opening a network share.*

Again Figure 2 shows the difference in information received between the two types of analysis.



(a) Passive analysis (b) Active analysis

Figure 2: Analysis of the botnet scenario

As we can see from these two examples the basic notion of active analysis remains the same, namely that more information about a malware can be found by interacting with an infected system (be it a single machine or network) than by just watching what a malware does. However the only difference between these two systems is the types of monitoring required (file system/ registry changes on a single machine verses network traffic in the botnet scenario) and the types of actions that can be taken (dummy Internet explorer activity versus connecting to network shares or sending emails). After establishing some basic notation we generalise these scenarios into the Active Malware Analysis Game.

3. BACKGROUND

This section describes the necessary background for the use of agent techniques in malware analysis. We first present a model of multi-agent interactions, Stochastic Games which can be used to model the game-theoretic interactions we presented in the previous section. Then we consider multi-agent learning within those games, the RMax algorithm which is capable of learning about the policies of unknown agents.

3.1 Stochastic Games

$\{N, S, A, \{R_j\}_{j \leq N}, T\}$ describes a discounted stochastic game:

- N is the set of players.
- S is the state space.
- $A = A_1 \times A_2 \times \cdots \times A_N$, represents the full set of actions, with A_j representing the set of actions for agent j.
- $R_j : S \times A \to \mathbb{R}$ is the reward function for agent j.
- $T : S \times A \times S' \to \mathbb{R}_{[0,1]}$ is the transition function.

Algorithm 1 The RMax algorithm.

```
1:  while true do                                          ▷ For each time step
2:      s ← CurrentState                                   ▷ Get current state
3:      a ← Action(s)                  ▷ Get best action for s using model
4:      r, s' ← Execute(a, s)          ▷ Reward and resulting state
5:      if Times played a in s ≤ Threshold then
6:          Update av(r, s, a)
7:          Update av(s', s, a)
8:          if Times played a in s = Threshold then
9:              R(s, a) = av(r, s, a)                      ▷ Set model
10:             T(s, a, s') = av(s', s, a)                 ▷ Set model
```

The game is played as follows. At the first stage the game is in an initial state $s^1 \in S$. At stage m the players are informed of the past history $\left(s^1, a^1, s^2, \ldots, a^{m-1}, s^m\right)$, where s^t is the state of the game at timestep t and a^t is the action combination the players played at that state. Every player j chooses, independently of the others from its policy, π_i receives a stage payoff $R_j\left(s^m, a^m\right)$, where $a^m = \left(a_j^m\right)_{j \in N}$, and the game moves to a new state s_{m+1} according to the transition probability $T\left(s^m, a^m, s^{m+1}\right)$. Less formally, a stochastic game consists of a finite set of stage games between two or more agents. In each of these stage games, the agents can choose from a set of possibly unique actions, and depending on the choice made by all agents, are assigned a reward. Further to this, again depending on the actions chosen and the original stage game, a transition will occur to a new stage game, with possibly different actions and rewards.

The goal is to compute policies, $\pi_i : G \to A_i$ for all agents i at every time step t, such that no agent has an incentive to deviate. G indicates the history of observed states and actions of other agents. Put simply, this policy is a function which maps the history of the game to an action for agent i. The optimal policy returns the best action for that player.

3.2 The RMax Algorithm

RMAX [3] is an algorithm for learning an appropriate policy in a stochastic game. It assumes that the opponent is an initially unknown part of the environment, so it is suitable for single agent problems with an unknown underlying transition and reward function or multi-agent problems, such as ours, with an unknown opponent. The algorithm starts off with an optimistic model which assumes the maximum possible reward for all possible state actions. The learning procedure then proceeds by computing an optimal policy for this model and as states and actions become *known* (according to a polynomial threshold) updating the model and re-computing the policy. This algorithm is guaranteed to learn the policy for the agent in polynomial time. Figure 1 gives the algorithm in detail. The updates at lines 6 and 7 are the mean results of the previous trials. This purpose of using the average is to capture what happens when that action is taken both in the presence of a stochastic world model (which may result in different outcomes for the same action) or an opponent with an unknown policy (who may change which action she plays in the same state).

With this established, the next section builds on stochastic games in order to represent the unique characteristics of malware analysis on a computer system. Then in Section 4, we utilise RMax in the construction of a learning algorithm for active malware analysis which exploits those unique characteristics of the problem to learn quickly.

4. ACTIVE MALWARE ANALYSIS GAME

In this section we present the *Active Malware Analysis Game* between an analysis agent, n_1 and a malicious agent (mal-

ware) n_2. This formalisation captures, amongst others, the two scenarios presented in Section 2. The game specifies the interactions between a malware, who is trying to infiltrate a system, and an analyser who wants to learn about that malware. In turn the malware may be trying to avoid such learning. The game is defined as follows:

- The infiltrated system is represented as a weighted graph where V is the set of vertices, E is the set of edges connecting the vertices. Each vertex is a state of core components of the system (ex: important flags in the registry) and edges represent transitions in the state of the computer that the malware can induce.
- The malware, n_2, may change several components of the infiltrated system. This represents a path through the graph past those vertices indicating the affected components. Thus the effects of the malware are represented by its location on the graph $v^m \in V$.
- The strategy space of the malware n_2 is the next change it can take from its current location, given as the neighbours of that location $a_2 = v^m \cup neighbour(v^m)$
- The analysis agent, n_1, may place within the system honey (simulated user activity). The set of locations of places honey can be placed is described by the subset of leaf nodes of the graph $V^H \subset V$.
- The strategy space of n_1, $a_1 = V^H$ means it can move the honey h from its current location, v^h, to a new location (or leave it where it is), $v^h \in V^H$. That is, the agent can create new simulated user activity and remove other activity. We use location on the graph to represent some fake user activity or data in place, with the preceding vertices representing state changes before this information is introduced.
- The global state space is given as the location of the honey, v^h, and of the malware, v^m, $S = V \times V^H$.
- Agent n_2 has a fixed, possibly stochastic, but unknown policy $\pi : S \rightarrow a_2$ which gives the probability of moving to a connected vertex given the location of the agent and the distribution of the honey on the graph.
- There is a reward function for agent n_1 associated with learning the policy and preferences of agent n_2. We will consider this further in the next section.
- We assume that the malware always starts out on a clean system at the start vertex $v^0 = s^1$.

If we refer back to the description of stochastic games, we can see that the graph represents a computer operating system and that vertices represent the different physical states that the operating system may be in (whether a certain registry variable is set or a type of file exists). Further to this, edges in the graph represent actions that the malware can take to change the condition of the operating system and exploit weaknesses (the transition function). Honey locations are the action space of the analysis agent. Consequently, it can be seen that a path from the start vertex to a honey represents a behaviour of the malware and we are interested in learning which behaviours a malware exhibits. We can see this in the Figures 1 and 2 where paths along the graph are an accumulation of the changes the malware has made, and that some edges are only taken in response to a particular honey (in these figures we represent this as the labels on the edges for simplicity).

Finally, it is clear that this is an instance of learning an opponents unknown policy in a stochastic game — both the analysis agent and the malware (be it software or a human agent) take sequential decisions and the reward function is linked because the analysis agent wants to learn the malware policy whilst that agent may want to hide its own policy.

5. LEARNING IN THE ACTIVE MALWARE ANALYSIS GAME

This section describes the learning algorithm we employ within the malware analysis game. Specifically, we describe several variants of the RMax algorithm that we later test in the empirical section. For these variants we compare their respective learning rates and finally describe how we represent learnt information for a user interested in generating a signature for the new malware.

5.1 Learning using Entropy Reduction

RMax assumes the optimal policy will be learnt with polynomial time. Now, in a real application such as ours, this is not feasible since placing a piece of dummy information on a computer system can take seconds. As a result, we are not interested in obtaining the eventual optimal policy, but in learning the most possible about the malware in very few timesteps (≤ 20). Consequently, we incorporate this notion into the reward function. However, we still want to employ RMax so that we can guarantee that the analysis agent is exploiting its knowledge of the malware effectively, whilst at the same time learning as much as possible. Given this, we define an information-centric utility function for the agent which can be used within RMax, and optimised in-order to learn about the malware as fast as possible.

Now, the malware policy π is defined as the distribution over the possible edges towards the honey (V^H) given the current location of the honey v^h and the malware position v^m. From the starting position v^0 to each of the honey locations v^H there is a path p_i which is defined as the list of edges $e_i \in E$ the malware will take to that honey $e_0, e_1, \ldots, e_n$. The malware then, must select a path $p_i \in P^H$ from the set of paths based on the location of the honey and its policy. The aim is to learn this policy which describes the probability of taking path p_i based on the location of the honey s, $Pr(p_i|s = v^H)$ for all possible honey locations.

$$Pr(p_i|s = v^H) = \prod_{e_n \in p_i} Pr(e_n|s = v^H) \quad (1)$$

where $Pr(e_n|s = v^H)$ is the probability of taking edge e_n.

We need a utility function which rewards the agent for learning this function online (since the initial value is an uninformative distribution). Consequently, we maximise the negative of the entropy of this function, where π_i is the current estimate of the malware policy:

$$U(\pi_i) = -[\sum_{s \in V^H} \sum_{p_i \in P^H} Pr(p_i|s) \log(Pr(p_i|s))] \quad (2)$$

With this established, the **MYOPIC** algorithm uses this utility function for 1-step lookahead action selection. At each timestep t the agent selects the action which maximises $U(\pi_t)$ and then updates π_t to a new policy π_{t+1} by updating the probabilities of taking an edge. Specifically the malwares *historical frequency* of playing edge e_n in s is defined as:

$$Pr(e_n|s = v_j^H) = \sigma_{e_n,s}^t = \frac{1}{t} \sum_{\tau=0}^{t-1} I\{e_n^\tau = s\},$$

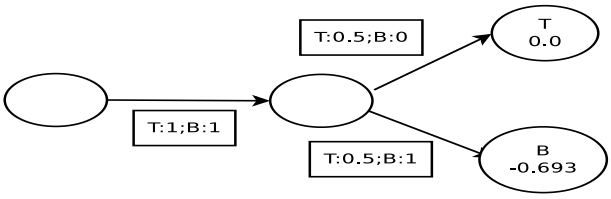


Figure 3: Myopic example.

where $I\{e'_n = e^\tau_n\}$ is an indicator function equal to one if e'_n is the action played by the malware at time τ, and zero otherwise. This algorithm works as follows: The example in Figure 3 shows a simple policy space for a dummy malware which starts in the first node attempts to move to the nearest honey, which can be at T or B. Previous trials have shown that the malware will steal information at B if it exists, but as yet we do not know what happens when information is at T, except that the malware moves to the second node in either case. Here we can see that if the honey is placed at B again, in this case (and assuming the malware tries to steal it), no new information is gained so the total entropy in the policy remains at -0.693. However, if the honey is placed at T then the entire policy can be learnt and entropy goes to zero, so *MYOPIC* would choose this action.

It should be noted that we abandon the optimality guarantees given by RMAX, however an optimal polynomial solution is not appropriate in our problem. Further to this, after presenting some benchmarks, the next section shows that this entropy reduction algorithm is guaranteed to be at least as fast as a random walk, and in general faster.

5.2 Benchmarks

PASSIVE: This algorithm does not take any action in response to the malware and represents the learning performed by dynamic, passive malware analysis such as [1]. This approach assumes that the malware will reveal its policy without interaction from the analysis agent. This allows us to benchmark our results against passive automated.

RANDOM: Selects a uniform distribution random action.

RMAX_OPTIMAL: Now, the aim of the problem is to learn the behaviour of the malware whilst she is changing the underlying system. As a result, this seems like a straightforward application of the RMax algorithm to the Active Malware Analysis Game. This is in contrast to using single agent learning algorithms which potentially ignore the problem that the malware may be trying to hide its policy from the analysis agent. However, this is potentially slow.

5.3 Exploration Rates

Now we define the exploration rates for our algorithms and justify the information-centric reward measure.

The single step expected entropy reduction $E[H]$ in the belief of the malware policy π_b is defined as:

$$E[H(\pi_b, v^m)] = \sum_{p \in P(v^m)} \sum_{v^h \in V^H} \pi(p, s) A(\pi_b, v^m, v^h) \\ *[H(b(\pi_b, p, v^h)) - H(\pi_b)]$$

where p is a path from the set of all paths from the malware position v^m to the honey locations and $\pi(p, v^h)$ is the true malware policy and is the probability of taking path p in state v^h. $A(\pi_b, v^m, v^h)$ is the action selection function and is the probability of selecting honey location v^h for the current belief and malware position. Finally, $b(\pi, p, v^h)$ is the belief revision function giving a new belief π'_b when the malware takes path p in state v^h for prior belief π_b.

A random action selection policy is defined as follows:

$$A(\pi_b, v^m, v^h) = \frac{1}{|V^H|} \tag{3}$$

whilst the **MYOPIC** action selection gives:

$$A(.) = \begin{cases} 1 & v^h = h' \wedge argmax_{h'}[H(b(\pi_b, p, h')) - H(\pi_b)] \\ 0 & \text{otherwise} \end{cases}$$

Since **RANDOM** gives the expected entropy over all possible choices of $v^h \in V^H$ then this must include the v^h that would be chosen by **MYOPIC**. This means that we can decompose the expression for the expected entropy using **RANDOM** in terms of the expected entropy for **MYOPIC** and the expected entropy over all states not including that one chosen in **MYOPIC**. Now, we define J as the expected entropy using **MYOPIC**:

$$J = argmax_{v^h} \sum_{p \in P(v^m)} \sum_{v^h \in V^H} \pi(p, v^h)[H(b(\pi_b, p, v^h)) - H(\pi_b)] \tag{4}$$

where J_v is the v^h chosen in J which maximises the expression. Then remembering the choice of J_v, the expected entropy, using **RANDOM**, over the remaining set is:

$$\frac{|V^H| - 1}{|V^H|} \sum_{p \in P(v^m)} \sum_{v^h \in V^H_{-J_v}} \pi(p, v^h)[H(b(\pi_b, p, v^h)) - H(\pi_b)] \tag{5}$$

Now, let K represent the maximum entropy in the remaining set, $K = \sum_{p \in P(v^m)} \sum_{v^h \in V^H_{-J_s}} \pi(p, v^h)[H(b(\pi_b, p, v^h)) - H(\pi_b)]$ then the expected entropy for **RANDOM** is *at most* as large as:

$$\frac{1}{|V^H|} J + \frac{|V^H| - 1}{|V^H|} K \tag{6}$$

Putting all of this together,

$$RANDOM \leq MYOPIC$$
$$\frac{1}{|V^H|} J + \frac{|V^H| - 1}{|V^H|} K \leq J$$
$$K \leq J$$

Which means that as long as there is an v^h which is larger than all others, **MYOPIC** will reduce the entropy more quickly. If not, then it will do the same as **RANDOM**.

These expressions can be extended to a finite horizon n:

$$E[H_n(\pi_b, v^m)] = \sum_{p \in P(v^m)} \sum_{v^h \in V^H} \pi(p, v^h) A(\pi_b, v^m, s)[IH(\pi_b, p, v^h) \\ + E[H_{n-1}(b(\pi_b, p, v^h), d(v^m, p))]]$$

where $IH(\pi_b, p, v^h) = H(b(\pi_b, p, v^h)) - H(\pi_b)$ and $d(v^m, p)$ is the malware transition from v^m along p to a new m'.

5.4 Learning over Multiple Malware

The techniques defined thus far are adequate for learning about a single malware, however they do not answer our larger research question: how similar is a new malware to an existing family of malwares? This is important in the context of using the information to power anti-virus defences. Specifically, by indicating a malwares similarity to existing

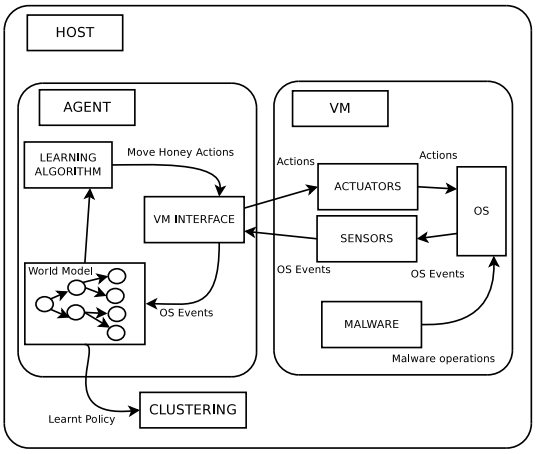


Figure 4: Active Malware Analysis Framework.

malwares the process of generating a signature is simplified. This is the goal of all automated techniques. To address this, we will use the standard K-Means clustering approach which maintains a set of k families of malware together with a representative mean malware policy. The goal then is to learn the policy of a new malware sample, and then assign it to an existing family or even create a new one, using a simple distance metric.

The distance metric takes the learnt policy and computes the distance for each transition (in all states). This is then summed to give a measure over the entire policy:

$$distance(\pi_i, \pi_j) = \sum_{v^h \in S} \sum_{p \in P(v_0)} |\pi_i(p, v^h) - \pi_j((p, v^h)| \quad (7)$$

6. ARCHITECTURE

This section describes the implementation of the Active Malware Analysis Game. The overall architecture is given in Figure 4. Since we are interacting with real examples of malware, we must run that malware binary (MALWARE) on a virtual machine (VM), and in order that the VM can be reset by the analysis agent (AGENT) must be located on the host system (HOST). Now, the agent makes use of several sensors to detect the state of the VM operating system (OS). Also, in order to interact with the malware, our agent needs access to a suite of actuators on the VM. Consequently we require an interface between the agent on the host and the sensors (SENSORS) and actuators (ACTUATORS) on the VM (VM INTERFACE). Next we give specific details of the actuators and sensors deployed in our empirical analysis.

6.1 Sensors

The Active Malware Analysis Game depicts the operating system state as a graph with vertices representing states for core components and edges are transitions between those states. Now, in order to detect the current state of the operating system and when such transitions occur (at the behest of the malware), we require a suite of sensors. Each sensor is designed to monitor one specific component such as whether a process has been registered to autorun when the operating system is started. For example this requires monitoring changes to the registry for the key: $\backslash Software\ \backslash Microsoft \backslash Windows \backslash CurrentVersion \backslash Run$. Similarly for other aspects such as browser helper objects, hidden services, parameters, and file system changes. By starting with the analysis performed by security experts on previous malware, we can generate a comprehensive set of such sensors, and the graph construction between them can be automated.

6.2 Actuators

The Active Malware Analysis Game allows the analysis agent to take actions in the operating system which the malware may or may not respond to. These actions allow the agent to move *honey* around the system. The purpose is to learn how the malware changes the operating system (using the sensors) in response to all of the possible honeys that might be deployed by the analysis agent. Consequently, our architecture requires a suite of honey actions. These include placing dummy sensitive information in several key locations or performing some simulated user activity on the operating system. One example of this includes opening Internet Explorer, going to a website and entering a username and password in fields denoted as such. Other examples include creating dummy configuration files for common programs or an address book of email address amongst others. In a similar fashion to the sensors, we start with types of honey identified by security experts to create a comprehensive suite.

As a final note, it can be seen that the architecture is easily expanded with new actuators and sensors should these be deemed necessary. The analysis agent will continue to learn as before with these new components and no change is required in the underlying algorithm.

7. EXPERIMENTS

In this section we demonstrate the utility of our active analysis framework by clustering a dataset of several previously analysed malwares. We show that the automatic clustering is accurate with regards to a human generated clustering and that it outperforms clustering performed using passive dynamic analysis. Further to this, we demonstrate that our entropy reduction learning algorithm is more useful in this malware analysis scenario than an RMax based algorithm. First we describe the experimental scenario, and then show the clustering performance over several algorithms.

7.1 Experimental Configuration

We experiment with a dataset of 50 malwares drawn from several families. These families are given in Table 2 All of these malware have previously been analysed manually and assigned to a cluster (both based on their static and dynamic properties). We allow each of our 4 learning algorithms ($RMAX\_OPTIMAL$, $MYOPIC$, $RANDOM$ and $PASSIVE$) to interact with each malware for 20 timesteps in a clean virtual machine. This is repeated 30 times and the average learnt policy is used in the clustering phase. Following this, we initialise the K-MEANS algorithm with 10 random means in the policy space and allow the clustering algorithm to run for 1000 iterations. This is repeated 30 times for the average clustering. Table 3 summarises the

Table 2: Experimental Malware

Zlob	Hooker	KeyLogger	LdPinch
PdPinch	QQPass	Sinowal	AdvanceKeyLogger
BZub	Luzia	VB	

types of behaviours and intentions we consider in this example (although the real set is larger):

Now, the interesting thing to note about this set of malware is that some parts of their behaviour are conducive to

Table 3: Experimental Behaviours and Intentions

FSys/Rservice	Install service
Address	Emails
FSys/RBHO	Install BHO
Text	File contents
RAuto	Autorun
IE/Keylog	Private data from websites
RFile	Register file location
IE/Keylog/Cache	Private data from history

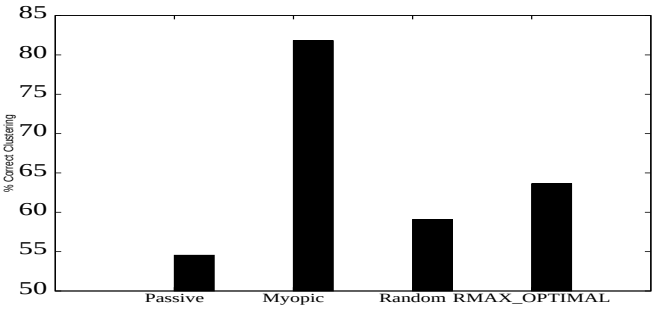


Figure 5: Cluster Identification Rate.

passive analysis and some are more appropriate for active analysis. Specifically, many of the malware will install themselves in the system in various ways such as hidden services or injecting DLLs into other processes for example. Parts of this behaviour is identifiable by passive analysis because it always happens when the malware executed. However some parts are not, such as when a keylogger writes a file in response to entering a username or inferring that an already running process has been captured by the malware.

7.2 Clustering

We first compare the clustering obtained by our various algorithms with an ideal clustering identified by malware analysis experts. As Figure 5 shows, the algorithm $MYOPIC$ is significantly more accurate when identifying clusters than $PASSIVE$, with a correct identification rate of 81% verses 54%. This is because $MYOPIC$ can identify a far larger part of the malware policy and consequently obtains a more informative clustering. Also, both $RMAX\_OPTIMAL$ and $RANDOM$ also outperform $PASSIVE$ because they all do active learning. However, the learning time is severely constrained and they do not learn as fast as $MYOPIC$ so consequently they are not as effective as that algorithm. We will show these results in more detail next to explain the improvement in performance.

Moving on, when we compare the features learnt in the malware policies we can see the impact on clustering to see why $MYOPIC$ performs much better than $PASSIVE$. Specifically, Figures 6, 8, 10 and 9 show the clustering for the algorithms $MYOPIC$, $PASSIVE$, $RMAX\_OPTIMAL$ and $RANDOM$ respectively. Further to this, Figure 7 shows the clustering done by a security expert. The x-axis shows the feature space for the possible mechanisms employed by the malware to infiltrate the system. The y-axis shows the possible locations of sensitive data that a malware might steal from. Both of these are in our restricted scenario. Each figure shows boxes for each identified cluster located in the space of mechanisms and intentions. The size of the box indicates the relative proportion of the corpus of sample mal-

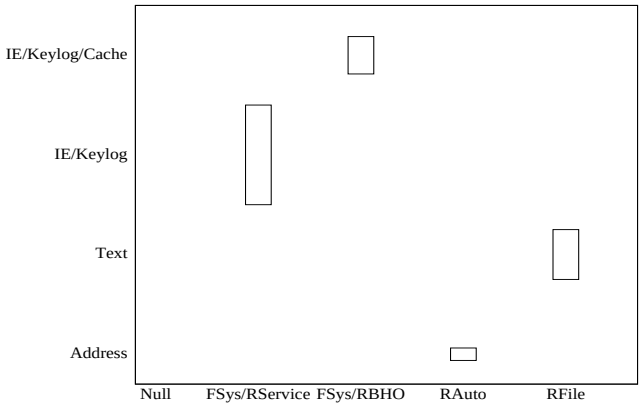


Figure 6: Feature extraction using $MYOPIC$.

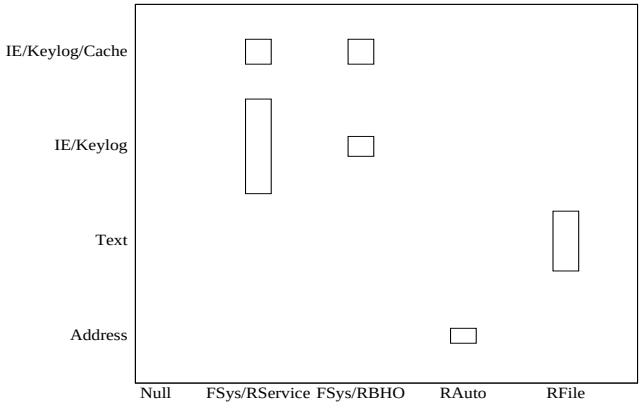


Figure 7: Expert classification.

wares in this particular cluster.

With this established, in Figure 6 we can see that a large grouping of malwares installs a DLL as a system service and proceeds to keylog the users actions. The next smallest family registers an executable and steals data from text files. An even smaller cluster registers a browser helper object and uses Internet Explorer to steal information. Finally a small group of malwares installs an autorun entry and raids the address book of Outlook.

As we can see in Figure 7, the clustering by $MYOPIC$ is very close to the one done by an expert. However the only divergence is in the large clusters which perform information stealing using keylogging and the cache at the same time. In some cases, $MYOPIC$ fails to differentiate between information stolen from the cache and from keylogging. This is because the malware in this case is able to perform both actions at the same time which breaks some of the underlying assumptions of a stochastic game. Further to this, sometimes there are some delays in placing the honey and when the malware reacts (perhaps because of errors in the malware or when it does polling) However, despite these physical limitations, as we will see next this clustering is still highly accurate compared to other approaches.

Specifically, the clustering from $MYOPIC$ should be contrasted with Figure 8 which has not learnt as much detailed information and so the clusterings are much courser. Here the algorithm typically can learn about how the malware is

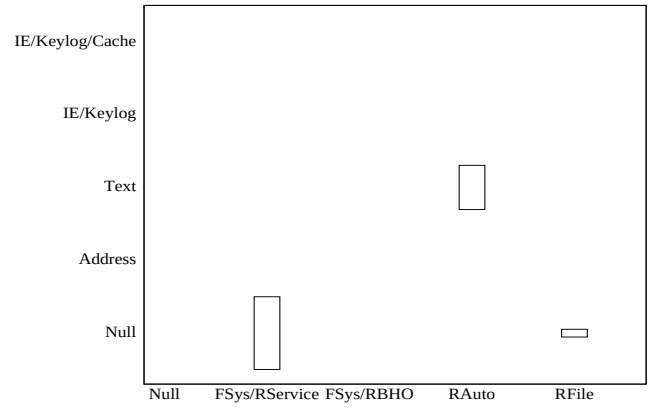


Figure 8: Feature extraction using $PASSIVE$.

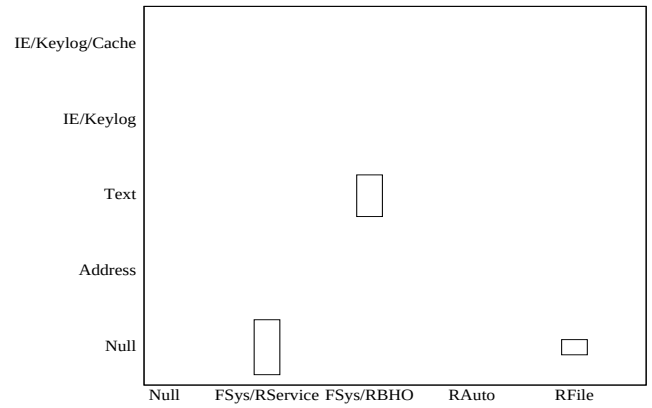


Figure 9: Feature extraction using $RANDOM$.

installed (a browser helper object or hidden service) but cannot find out about the intentions. An exception is the cluster of malwares that steals from text files - some of these files are created by the system and so are present even if a user does not create them. This is because, whilst the analysis is dynamic (meaning the malware is executed), it is *passive*, meaning that we do not interact with the malware as a security expert tasked with analysis would. An illustrative example is the cluster of malwares which register a file: In Figure 6 we also see that this cluster does some keylogging, however in Figure 8 this information is missing and the clustering puts most of these malwares with other groups. As a result, automated analysis is limited in its usefulness unless it is *active* because many variants of malware require some user interaction to exhibit their full suite of behaviours.

Finally, we should consider what happens with active analysis using a slower learning algorithm - Figures 10 and 9. Here we can see that $RANDOM$ is effectively the same as $PASSIVE$ in the clustering it performs because it does not learn the important part of the policy space in the short time allotted. This highlights the importance of learning quickly in this domain. $RMAX\_OPTIMAL$ is better, and does in fact learn one of the clusters which requires active analysis (the cluster installing a keylogger in Internet Explorer), however even it does not learn the complete set of intentions for this cluster because it misses that this type of malware family also searches the cache.

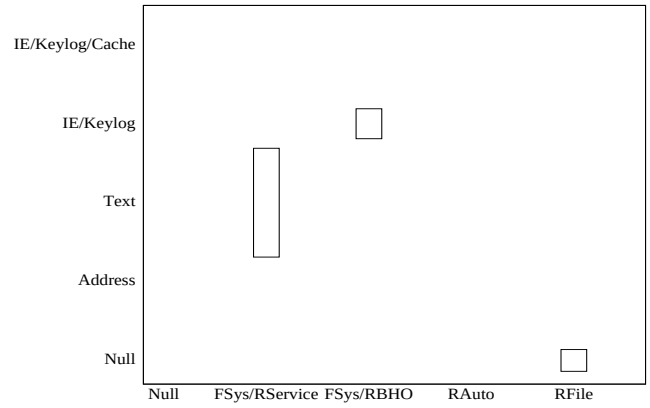


Figure 10: Using $RMAX\_OPTIMAL$.

8. CONCLUSION

This paper has introduced the application of Automated Active Malware Analysis using stochastic games and multi-agent learning. We defined a game capturing the active malware analysis problem. Following this, we developed an extension of RMax based on reducing entropy for learning quickly in the constrained time horizon of such games. We showed theoretically that this extension is faster than standard techniques. Finally, we presented a comprehensive empirical demonstration of our deployed framework for active malware analysis. We learnt the policies of 50 malwares and achieved a clustering very close to the one proposed by human security experts.

In future work, we intend to extend the application framework to the issue of learning about botnet malware. The game remains the same as in this paper, but a new architecture must be developed to monitor networks of systems, rather than the single system implemented in this paper. We also intend to extend the theoretical justification for the entropy reduction based algorithm, showing that it is faster than any other heuristic in this game.

9. REFERENCES

[1] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74320-0_10.

[2] Ulrich Bayer, Paolo M. Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. 2009.

[3] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3:213–231, March 2003.

[4] Christopher Kruegel, Engin Kirda, Ulrich Bayer, and Andreas Moser. Dynamic analysis of malicious code. *Journal in Computer Virology*, 1 2006.

[5] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 231 –245, may 2007.

[6] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 481–500. Springer Berlin Heidelberg, 2008.

[7] GÃl'rard Wagener, Radu State, and Alexandre Dulaunoy. Malware behaviour analysis. *Journal in Computer Virology*, 4:279–287, 2008. 10.1007/s11416-007-0074-9.