Universitat Oberta de Catalunya (UOC)

Máster Universitario en Ciencia de Datos (*Data Science*)

Master's Thesis

Area: Cybersecurity

Static PE antimalware evasion

Minimizing detection rate by using Reinforcement Learning

Author: Francisco Javier Gómez Gálvez

Tutor: Blas Torregrosa Garcia

Professor: Ferran Prados Carrasco

Barcelona, January 25, 2021

Copyright

Master's Thesis Information

Title:	Static PE antimalware evasion by using Reinforcement Learning
Author's name:	Francisco Javier Gómez Gálvez
Consultant's name:	Blas Torregrosa García
PRA Name:	Ferran Prados Carrasco
Delivery Date (mm/aaaa):	01/2020
Course:	Data Science
Master's Thesis' Area :	Data Science + Cybersecurity
Language:	English
Keywords:	malware, reinforcement learning, AV evasion, deep learning

Dedication

In dedication to my family and those friends who I consider to be part of it.

Abstract

Malware detection is a critical capability which is usually deployed in any production system as a first step to increase the infrastructure security. Due to this widespread security measure, and with the intention of carrying out the actions for which it has been designed, malware is constantly evolving in order to evade common detection techniques, ranging from simple changes aimed to evade signature-based detection to complex variations involving malware virtualization which are able to evade behavioural-based detection.

In this project, an experiment based on Reinforcement Learning is designed in order to improve the evasion capabilities of a given self-generated malware sample. Such design is carried out by defining the set of actions that can be taken in order to evade Static PE detection; an environment which evaluates the sample; a reward function that allows us to minimize the detection rate, and an agent which coordinates the entire process.

Tools used in the scope of this project are available for the general public, including those used for self-generating the samples as well as those used to emulate an environment with different antimalware solutions.

Keywords: reinforcement learning, malware, evasion, antimalware, deep learning, neural network

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The COVID-19 pandemic has forced society to rely on computing systems more than ever. In this sense, a relevant increase in cyberattacks has been observed[1] to target users and companies exploiting COVID-19 related fears to induce them to make mistakes which could lead to a security breach.

Depending on the attackers' modus operandi, there are different attack vectors[2] that can be used in order to compromise user endpoints or servers, but eventually the need of dropping malware or malicious tools in the target's system becomes clear. Due to the widespread use of antimalware capabilities in servers and endpoints, attackers tend to minimize the detection rates of malicious tools thus rendering a number of antimalware solutions useless as a detection capability. This is why during the last years, the practical application of Data Science[3] in security capabilities has allowed companies and individuals to stand a chance by switching from a signature-based to an anomaly-based detection strategy.

Unfortunately, as cybersecurity advances, so do the adversaries. There are already a number of papers showing how an attacker could use **Reinforcement Learning** in order to prepare against detection patterns and try to minimize the detection rate for a set of malware samples.

In this document, the feasibility of using Reinforcement Learning to evade Static PE Machine Learning detection models will be assessed to better understand the current state of the art from an attacker's perspective.

[1]https://www2.deloitte.com/ch/en/pages/risk/articles/covid-19-cyber-crime-working-from-home.html
[2]Most typical involve phishing or exploitation of a non-patched vulnerability
[3]Specifically, Machine Learning algoritms

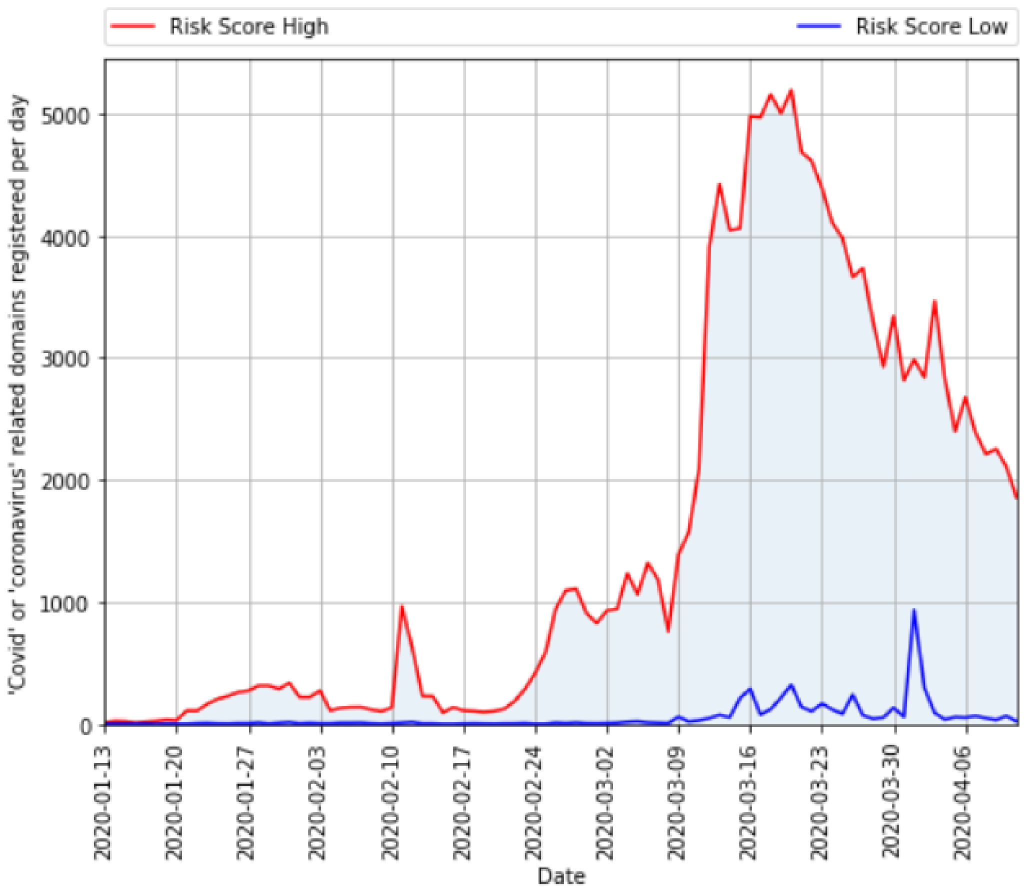


Figure 1.1: Rise of malicious domains during COVID-19. Source: Krebs On Security

1.2 Motivation

As attackers gain knowledge and develop more sophisticated techniques, cybersecurity needs to improve in order to effectively prevent, detect and respond against a potential attack. In order to properly defend against an adversary, a cybersecurity expert needs to know how an attacker's mind works, what are common TPPs[4] and carry out extensive Penetration Testing or Red Team exercises against the infrastructure to motivate a continuous improvement of the security landscape.

One of the issues that penetration testers face is the short lifetime of those tools used in an specific engagement. As a certain backdoor or malicious tool is first used, there is a chance that it will be reported as suspicious to some antimalware engines, and hence becoming extensively detected by several vendors hours or days after that.

Given that is possible[4] to carry out a number of operations over a binary file in a way that functionality is preserved while substantial changes are made to the underlying structure,

[4]Acronym for Techniques, Procedures an Protocols

our motivation will be to find an optimal way to industrialize such process and automate the generation of a clean sample from a well-known[5] sample.

1.3 Objectives

The main objective of this project is to take advantage of some existing Reinforcement Learning algorithms and try to decrease the detection rate for a known malware by means of modifications over the binary structure.

Ideally, in the end we should had built a black-box which can be fed with a malware sample in order to obtain a equally functional malware sample with a lesser detection rate than the original one.

1.4 Project scope

Due to cost and time constraints inherent to the nature of this project, the following scope limitations arise:

1. Malware samples will be limited to binary files, as plain-text scripts can be easily obfuscated to prevent detection
2. The target operating system will be Microsoft Windows, specifically Windows 10
3. Malware samples will be self-generated by using Kali Linux alongside with the Metasploit Framework
4. The detection rate will be obtained by making use of antimalware vendors' public API endpoints, which usually imposes restrictions in the requests rate.

Additionally, some specific techniques focused on binary modification might end up out of scope given the complexity of the implementation and they will only be discussed from a theoretical standpoint.

1.5 Hypothesis

Given that our interests lay in building a black-box system able to lower the detection rate of a sample, we first need to establish a series of hypothesis that help us to keep our project focused:

[5]already detected by a number of antimalware solutions

1. A sample will be considered malicious in the eyes of a vendor when their specific malware solution classifies it as something different that “harmless” or “undetected”
2. Binaries will always be be PE32 or PE64 executables
3. As functionality needs to be preserved in every iteration, the actions taken over the sample will avoid:
    - Altering the binary in a way that the operating system is unable to figure out its structure.
    - Altering the execution flow in a way that functionality is damaged

1.6 Practical Applications

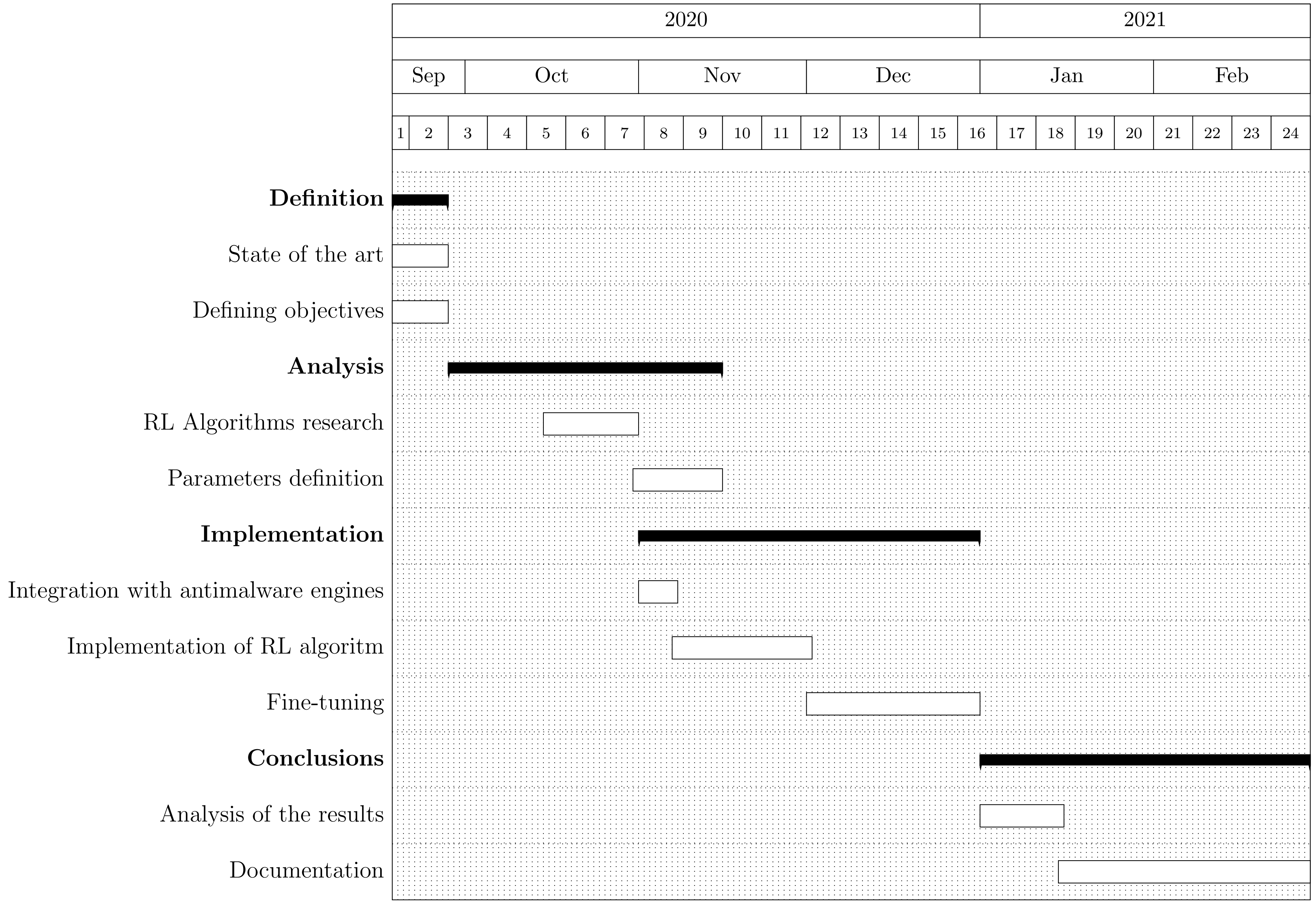
Some immediate practical applications arise from the current project:

- Generation of clean samples for Red Team exercises
- Amplification of a given malware dataset, resembling data augmentation techniques
- Support in improving antimalware capabilities by making use of GANs[6]

1.7 Planning

In order to set an appropriate calendar for the project, the following stages will be defined:

[6]Generative Adversarial Networks


2020
2021
Sep
Oct
Nov
Dec
Jan
Feb
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Definition
State of the art
Defining objectives
Analysis
RL Algorithms research
Parameters definition
Implementation
Integration with antimalware engines
Implementation of RL algoritm
Fine-tuning
Conclusions
Analysis of the results
Documentation

Chapter 2

State of the art

2.1 Previous studies

There are a number of different articles and resources dating from 2017 where machine learning is used as an useful approach to antimalware evasion. Depending on the knowledge that an attacker possesses regarding a specific antimalware solution there are different strategies to increase the evasion rate, ranging from white-box attacks where the antimalware architecture is well-known and we can try to minimize the gradient function to those in which the attacker has no knowledge about the antimalware inner workings and aims to lower the detection rate based on a score or good/bad indicator [2].

Focusing on the black box approach, one of the main requirements is to ensure that a binary subject to this process keeps its functionality after the modifications that are carried out. In [2] a set of seemingly innocuous actions are explored:

- Change existing sections names
- Create new unused sections
- Append bytes to the slack space in existing sections
- Append bytes to the overlay
- Break signature (if exists)
- Break header checksum
- Manipulate debug info (if exists)
- Pack/Unpack the file
- Create a new entry point which jumps to the original entry point
- Add an unused function to the import table

It is later in [1] where it is found that even the innocuous functions detailed previously can randomly break a PE file after several mutations. Looking to prevent that from happening, a model to test for the functionality of the resulting binaries is proposed in [3], ensuring that those mutations that cannot be properly executed in a virtual machine are discarded. In this scenario, it was shown that functional binaries after several mutations were able to decrease the detection rate by 80%.

Finally, in [4] a deep reinforcement learning model (DWEAF) which automatically chooses between different actions to minimize the detection rate (thus maximizing the reward) is proposed, improving the evasion rate of several malware samples over other frameworks (Gym-Malware).

In this project, we will take advantage of the previous studies, looking to lower the detection rate of a malware sample by means of several innocuous modifications while keeping functionality.

2.2 Theoretical framework

2.2.1 Introduction to RL

Reinforcement learning is defined as an area in of machine learning which focus on maximizing a *reward* provided by the *environment* after an *agent* takes an specific *action*.

In simple words, reinforcement learning help us to map certain actions with expected rewards, allowing us to choose the best course of action to benefit us in the context of a certain problem. A reinforcement learning problem has some differentiated elements that need to be carefully taken into account in any design:

- The **agent**, which is responsible of taking a specific action to alter the environment and gather the feedback in order to improve future predictions.
- The **environment**, which refers to anything that the agent interacts with. The environment could be seen as the outside world from the point of view of the agent. It is also the one who provides us feedback and allows us to measure changes and rewards.
- A set of **states**, which reflects the situation for the environment at a given time.
- A set of **actions**, often called the action space, which comprises the different changes that can be carried out over the environment.
- A **policy**, which defines how the agent should behave at a certain moment. This is the mapping between the states and actions, and can be seen as the core of the agent, enough to define behaviour and next actions to be taken.

- A **reward** signal, which is a value returned by the environment after an specific action has been taken and the environment has been altered. This reward signal would reflect how the environment perceives that specific action.
- A **value function**, which allow us to keep track of the overall reward for each one of the actions. This provide us with valuable information about the long run, allowing us to understand how expected rewards change over time or epochs and allowing us to have a better insight into the process.

It is equally interesting to talk about the **Markov property**, which states that a game or problem is a MDP[1] in case that we have enough information to play or proceed by only knowing our current state, without having information about previous states.

A formal approach for the aforementioned concepts, presented in [7] and [9] is:

Concept	**Description**
$\pi, s \to Pr(A\|s)$	A policy π is a mapping from states to the probabilistically best actions for those states
$\pi^* = \text{argmax}\,\{E(R\|\pi)\}$	An optimal policy π^* is such that produces the optimal reward based on our knowledge of the expected rewards for a set of policies.
$V_\pi : s \to E(R\|s, \pi)$	A value function V_π is a function that maps a state, s, to the expected reward in case we follow some policy π from the state s
$Q_\pi : (s\|a) \to E(R\|a, s, \pi)$	Q_π maps a pair (state,action) to the expected reward

Table 2.1: Fundamental concepts in RL

It is worth noting that there are a subset of RL problems which do not present all the elements that we described previously. Those are what we could call stateless RL problems, on which the outcome does not depend on a state, but only on the actions taken and the perceived reward. In this class, we can find the *k-armed bandit*, which will be out starting point for this work.

2.2.2 The k-armed bandit

The k-armed bandit takes its name from the slot machines (known as one-armed bandit), which can be considered a particular scenario for this problem where $k = 1$. In this situation, we are constantly faced with making a choice between a set of k different options. Once the choice has

[1]Markov decision process

been made, the *environment* provides feedback in the form of a *reward* which can be greater or lesser. Our objective is to find a way to maximize the reward over a time or steps.

If we denote the action select on time step t as A_t, and the reward returned in that time step time as R_t, we can define the expected reward given that action a is selected as:

$$q_*(a) = E[R_t|A_t = a] \tag{2.1}$$

As we cannot know the values associated to an action beforehand, we need to define the estimated value of action a at time step t as $Q_t(a)$ as:

$$Q_t(a) = \frac{\text{Sum of rewards for action } a \text{ when taken}}{\text{Number of times that } a \text{ was taken}}. \tag{2.2}$$

This allows us to average the measured rewards for every action along time, ensuring that we smooth any anomalies as the number of iterations grows. In this sense, it can be noted that, by the law of large numbers:

$$\lim_{t\to+\infty} Q_t(a) = q_*(a). \tag{2.3}$$

Given that we have a simple way to approach the expected reward, we can now focus on selecting a simple policy π that allow us to take the best actions based on the observed rewards.

2.2.2.1 The greedy approach

In this scenario, our policy will be based on always choosing the best action. This greedy action selection can be written as:

$$A_t = \text{argmax}_a Q_t(a) \tag{2.4}$$

where argmax_a makes reference to the action which throws the greatest average reward over time. In this scenario, we stick with the best action, so the most likely course of action is:

- The same action is repeated until the end of the algorithm (i.e. we have consumed all the time steps)
- The action is repeated until the estimated value of that action over time decreases in a way that it stops being the best action

The initialization of the reward vector needs to be taken carefully in this scenario, as we might end up in a situation where a reward for an action will never decrease enough to explore another actions, meaning that only one action from the action set will be chosen in execution time. For this reason, it is considered that the greedy approach is not necessarily the best in to approach an optimal solution, as it might converge too fast to a non-optimal solution while not exploring the rest of the choices.

2.2.2.2 The epsilon-greedy approach

As a way to motivate the exploration while keeping a greedy component that allow us to converge to an optimal solution, we introduce the ε-greedy approach. In this scenario, we make our model to behave greedily most of the time and choose a random action with an "ε" probability, ensuring that after some time steps it becomes unlikely to always choose the same action, so the algorithms forces itself to explore other alternatives.

It is also interesting to keep in mind that the choice of ε will condition our algorithm in terms of time steps needed to explore every action. It is trivial to show that the probability of keep choosing the best known action after t time steps drops exponentially:

$$\lim_{t\to+\infty} Pr(A_t = \text{argmax}_a Q_t(a)) = \lim_{t\to+\infty}(1-\varepsilon)^t = 0 \tag{2.5}$$

but at the same time, it is also complex to ascertain at what time step would the algorithm have gone over all the possible choices. This is due to the strong random component that comes with the ε-greedy model, and at this point we could establish the following analogy:

"Ascertaining after how many time steps we can consider all the "n" choices of an action set selected at least once *is comparable to* ascertaining after how many rolls of a n-sided die we can consider we have obtained every side at least once"

Based on empirical evidence[2], the die problem presents a discrete distribution $f[n]$ which can be approximated by a Rayleigh continuous distribution $g(x;\sigma)$ as follows:

$$Pr(a \le X \le b) = \sum_{n=a}^{b} f[n] \approx \int_a^b g(x;\sigma)dx = \int_a^b \frac{x}{\sigma^2} e^{\frac{-(x-n)^2}{2\sigma^2}} dx. \tag{2.6}$$

The exact parameters for the location (n) and scale (σ) of the distribution can be obtained respectively from the size of the action set and from a empirical fit of the distribution by using specific software (e.g. scipy)

Finally, considering that choosing $t \approx 3.717\sigma$ would ensure us a confidence of 99.9% for obtaining all the sides of the die after t time steps, we can generalise for our ε-greedy approach and derive that a potential number of time epochs to ensure that are actions are chosen can be

[2]Check Annex for evidence

approximated by:

$$t \approx \frac{3.717\sigma}{\varepsilon} \tag{2.7}$$

which is due to the algorithm only taking a random choice with a probability of ε, hence spending the rest of the time slot repeating a only action in a worst-case scenario. Note that for a totally random scenario ($\varepsilon = 1$) this becomes the original die problem of choosing random actions until we have gone over all possible options.

2.2.2.3 Contextual bandits

As we have seen in previous sections, the *k-armed bandit* problem is a basic example of reinforcement learning where actions are taken based on the reward provided by the environment. In this analogy our agent would be the player, the environment would be the slot machine, and our policy would be strategy we choose to take the best action (greedy or ε-greedy).

This classic problem is missing one of the fundamental elements that is usually present in any reinforcement learning problem: the state. The *k-armed bandit* is known as one stateless problem within the reinforcement learning field, as it does not need any kind of state or transition between states.

Contextual Bandits is a stateful version of the classic *k-armed bandit*, were there are different signals coming from the environment that allow us to search for a better policy by the association of signals to actions. This is also called an *associative task search*, as there is a trial-and-error learning in order to find the best action, and there is an association of those actions with situations in which they are the best possible choice.

The contextual bandits only lacks a proper policy able to map states with actions to be considered a full RL problem. Designing and implementing such approach will be the cornerstone of this project.

Chapter 3

Work development

3.1 Design

3.1.1 Design plan

As has been stated in previous sections, a number of elements need to be defined in the scope of our project. As our objective is to build a black-box system in which malware will be automatically modified based on RL techniques, we can begin to materialize such elements in a more specific way:

- The **agent** will be in charge of orchestrating the changes in the environment, including new samples generation and modification over existing malware samples.
- The **action space** will be defined by a set of actions that effectively modifies the binary while preserving its functionality.
- The **environment** needs to be a set of antimalware engines with the ability to classify the sample in, at least, detected or undetected for every antimalware engine. The **state** of the environment in a specific time step will be the action taken for that time step.
- The **reward** provided by the environment will be defined in a way that an undetected result makes the reward higher, and a detected result will make the reward lower.
- The **policy** will be chosen based on feasibility given the design calculations. The greedy, ε-greedy and deep learning approaches will be studied in the design phase.

The specifics of every element will be designed in the next section

3.1.2 Theoretical design

In order to meet the project timeline in an effective and efficient way, we will try to discard any unfeasible implementation in the design phase, ensuring that we reach a final implementation in the shortest time possible.

At this point, our aim will be to build up the foundations for our RL antimalware evasion project by focusing on the different elements

3.1.2.1 The environment

The definition of the environment is the cornerstone of the entire project, as it will be our "playground" around what the different actors in the RL problem will revolve.

Defining or choosing the right environment is inherently linked to tied to our main objective. The following are some valid environments ideas in an antimalware evasion project:

- A known **antimalware algorithm** that, in presence of a sample, provides us with a feedback that could range from a simple classification in terms of malicious or benign to a more complex risk scoring (white box approach, focused on evading a specific algorithm) **(Option 1)**
- A known **antimalware product** that, in presence of a sample, provides us with a classification between malicious or benign for such sample(black box approach, aimed at evading a specific vendor) **(Option 2)**
- A series of **different atimalware products** that, in presence of a sample, provides us with a number of products detecting the sample as malicious or benign (black box approach, aimed at evading a number of different solutions) **(Option 3)**

The three aforementioned environments could be built and our RL approach be implemented to achieve the defined objective. In order to choose our approach, the following constraints need to be taken into consideration:

1. Antimalware algorithms are usually proprietary, and public ones are usually deprecated.
2. Integration via API with a specific vendor depends on the features available for the product and the type of subscription (in our case, free), which adds complexity to our solution.
3. Our solution should work in most cases, and given that we do not have information or statistics about the relative deployment of different vendors by company typology, we should focus on evading as many different vendors as we can.

After careful examination of the scenario and the different constraints, we decide to choose option 3, given that:

- We do not have an interest in an specific algorithm or vendor
- We do not have enough time to test different products

- There are online services which allow for submitting a file and scanning it against a number of known antimalware solutions

Choosing the sandbox As part of our analysis, the following comparison[1] was carried out between 3 major online scanning solutions:

	VirusTotal	MetaDefender	Jotti's
Vendors	60+	30+	15
Public API	Yes	Yes	No
Distribution	Yes	Yes	Yes
Consistency	OK	KO	OK

The criteria used for the comparison is as follow:

1. **Vendors**: number of different antimalware engines supported
2. **Public API**: availability of a free interface for file scanning
3. **Distribution**: automatic sharing of samples with antimalware vendors
4. **Consistency**: ability to replicate previous results in a short time-span (before the distribution takes place) and lack of haphazard errors in experimental tests

From our analysis, VirusTotal seems the most feasible choice in order to have a big number of different antimalware engines to test against, a mature public API[2] and a good response after several manual tests.

Defining the reward Once we have chosen a specific service, we begin by inspecting the information provided by VirusTotal in the event of analysing a specific file. The following is a sample (only showing 1 antimalware for brevity) of a real response received by the API after uploading a malicious file

[1]Information gathered from official documentation for each platform
[2]Documentation available on https://developers.virustotal.com/reference

```
"data": {
    "attributes": {
        "date": 1605379112,
        "results": {
            "ALYac": {
                "category": "malicious",
                "engine_name": "ALYac",
                "engine_update": "20201114",
                "engine_version": "1.1.1.5",
                "method": "blacklist",
                "result": "Trojan.CryptZ.Gen"
            },
        ...
        },
        "stats": {
            "confirmed-timeout": 0,
            "failure": 0,
            "harmless": 0,
            "malicious": 53,
            "suspicious": 0,
            "timeout": 2,
            "type-unsupported": 4,
            "undetected": 17
        },
        "status": "completed"
    },
    "id": "MjM2MWU2YWM0ZmYxZGFkNGIyZDEzODIwMzI3YTYzMmU6MTYwNTM3OTExMg==",
    "type": "analysis"
},
"meta": {
    "file_info": {
        "md5": "2361e6ac4ff1dad4b2d13820327a632e",
        "name": "qunmfvn",
        "sha1": "57955edcf711a005ca3056e61f2e2a73b09c4502",
        "sha256": "0fb574ee1237d26c383c482a59f58781609d390985f430...",
        "size": 73728
    }
}
```

Listing 3.1: JSON returned from VirusTotal

There is a fair amount of useful information that can be extracted from the response showed in listing 3.1:

- An **unique identifier** for the file (SHA256)
- A summary of the different statuses for the file. In this sense
  - A *harmless* or *undetected* status will be considered a success from an evasion perspective
  - A *malicious* or *suspicious* status will be considered a failure from an evasion perspective
- The detailed information for every antimalware regarding the file, including:
  - Classification of the file (malicious, benign, timeout, ...)
  - Specific name of the threat

With this information we can define a signal reward function and also have a way to track changes between different mutations of our sample. This will allow us not only to explore the reward over time, but to obtain interesting data regarding how different vendors behave when facing a mutation of a known malware in every step.

We can define our signal reward function for a specific time step as follows:

$$\mathrm{R_n} = \begin{cases} \frac{n_{\text{undetected}}+n_{\text{harmless}}}{n_{\text{malicious}}+n_{\text{suspicious}}} \cdot 100 & \text{if } n_{\text{malicious}} + n_{\text{suspicious}} > 0 \\ 9999 & \text{other case} \end{cases}$$

A quick look at the reward function let us ascertain some things about its behaviour, which is the expected one for our scenario:

- it returns 0 in the event that the file is classified as malicious or suspicious by every antimalware
- it returns 9999 in case that the file is classified as harmless or undetected by every antimalware
- it increases as the number of successful results (undetected/harmless) increases
- it decreases as the number of unsuccessful results (malicious/suspicious) increases

It is equally important to note that we also need to keep track of the past rewards in order to weight them properly. The cumulative reward for a given action can be calculated as:

$$Q_{n+1} = Q_n + \frac{1}{n}\left(R_n - Q_n\right) \tag{3.1}$$

In our specific case, apart from taking the average reward as shown, later we will make use of the softmax function in order to translate our cumulative rewards to a probabilities vector which will give us the likelihood of best action for every action:

$$Pr\left\{A_t = a\right\} = \frac{e^{Q_t(a)}}{\sum_{b=1}^{k} e^{Q_t(b)}} = \pi_t(a) \tag{3.2}$$

Note that, in this scenario, the rewards only are used as a preference function. This is why they are not rewards any more, but auxiliary values which help our algorithm at balancing the action selection while looking for the best possible outcome.

Defining the states The different states for this environment can be a complicated concept to define, as there are several possibilities:

- Defining a binary state of “detected” or “undetected” based on a threshold we define for the total number of antimalware engines. This threshold would be defined based on a criteria of how many different antimalware engines we want to evade.
- Defining a binary state of “detected” or “undetected” based on a specific antimalware engine. We could focus on a well-known vendor and consider that our mutation is good enough when it evades a specific vendor.
- Defining the state based on the actions taken by the agent. In this situation, the state becomes dummy information as it is not generated by the environment, but by the agent.

In our scenario, the preference function is already taking into account the detection status, and as our design is based on gradient bandits we can afford to define the status based on the actions without the need of defining transition states.

So, our states will be numbered based on the action set that will be defined by the agent.

General overview The following is a high-level diagram of our environment. R_{t+1} and S_{t+1} refers respectively to the signal reward and state after the sample has been assessed by the different antimalware engines, and A_t is the action that modifies the sample prior examination.

A most detailed diagram will be built in the next sections as we design all the remaining elements of our project.

3.1.2.2 Designing the agent

In a RL problem, it is said that everything the agent interacts with should be a part of the environment. In this sense, the agent design will influence our previous environment design, so the full diagram will experience changes through this document.

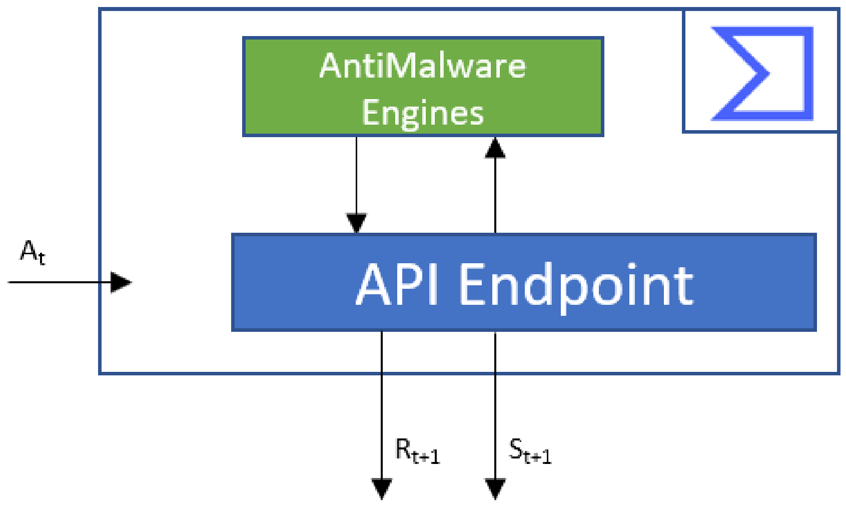


Figure 3.1: High-level overview of the designed environment

The following is a list of tasks and constraints to be taken into account:

- The agent should be able to alter the binary in a way that the resulting binary can be differentiated from the original one
- The agent should not break the functionality of the binary
- The agent should be able to understand the output of the environment (reward and state) in order to assess the results of taking different actions
- The agent should implement a policy in order to process the feedback from the environment and improve the decision-making process

In order to accomplish our objective, we will define what actions the agent can take and how it is going to learn from the environment.

Defining the action space After examination of previous researches[4], we come up with a list of actions that meet the criteria previously defined, allowing for appreciable alterations[3] in the binary while presumably keeping its functionality.

The following is a list of the actions[4] that our agent will be able to carry out over the environment:

1. Add random bytes at the end of the file.
2. Add random bytes at the end of every section

[3]The SHA-256 hash of the sample changes after applying the action, making it a virtually different mutation
[4]Refer to Annex for a detailed explanation of the PE stack

3. Add a new section filled with random bytes. Regarding the section types, those are randomly chosen from a predefined set: IDATA, BSS, TEXT, TLS_, RESOURCE, RELOCATION, UNKNOWN.
4. Rename every section by using random strings
5. Add a known library with a known function name. Both libraries and functions are chosen from a predefined set.
6. Add a known function to a currently existing library
7. Remove metadata from the file, including debug information, signature, datestamps and checksum.

There are a number of additional actions[5] that will not be considered in the scope of this work due to the high complexity that its implementation would imply.

Defining the policy Once the action space has been defined, we can theoretically assess the viability of using different policies in order to improve our action-making mechanism. As our approach is based on *contextual bandits*, we will revisit some of the possibilities we explored in previous sections.

Greedy policy Going after a greedy policy would limit the learning rate of the algorithm as there is a high chance of it getting stuck in an specific action. In order to reduce the likelihood of this, we should:

1. Carefully initialise the vector reward. Zeroing it would make the first action taken also the only action taken, as the average reward would never go under zero in the way it has been defined. In case we do not have an estimated value for initialising the reward vector, a redefinition of the signal reward function for considering negative values would help in a more dynamic action selection.
2. Increase the number of time steps to ensure that we go over all the possible actions

This option is not feasible because even if the predefined actions are harmless and are not supposed to break the binary functionality, this can actually be broken after undergoing a number of iterations[1] taking the defined actions. It is difficult to estimate a number of iterations needed based on the action space length, but we can never ensure that all the action set will be chosen at least once

[5]Changing the Original Entry Point (OEP) or encrypting and packing the stub of the payload

Epsilon-Greedy policy In order to improve our learning capabilities, the ε-greedy approach allows for a random inspection of all actions based on a predefined probability. As we discussed during the theoretical definition of this policy, we can estimate how many iterations would be needed to ensure that all actions are taken at least once.

For this scenario, an algorithm simulating a 7-side dice has been coded. This algorithm virtually rolls a dice until all its sides are shown at least once, and then register the number of rolls that were needed to reach that situation. Then, the resulting discrete histogram can be used to generate a density distribution that can be empirically approximate by means of a continuous Rayleigh distribution.

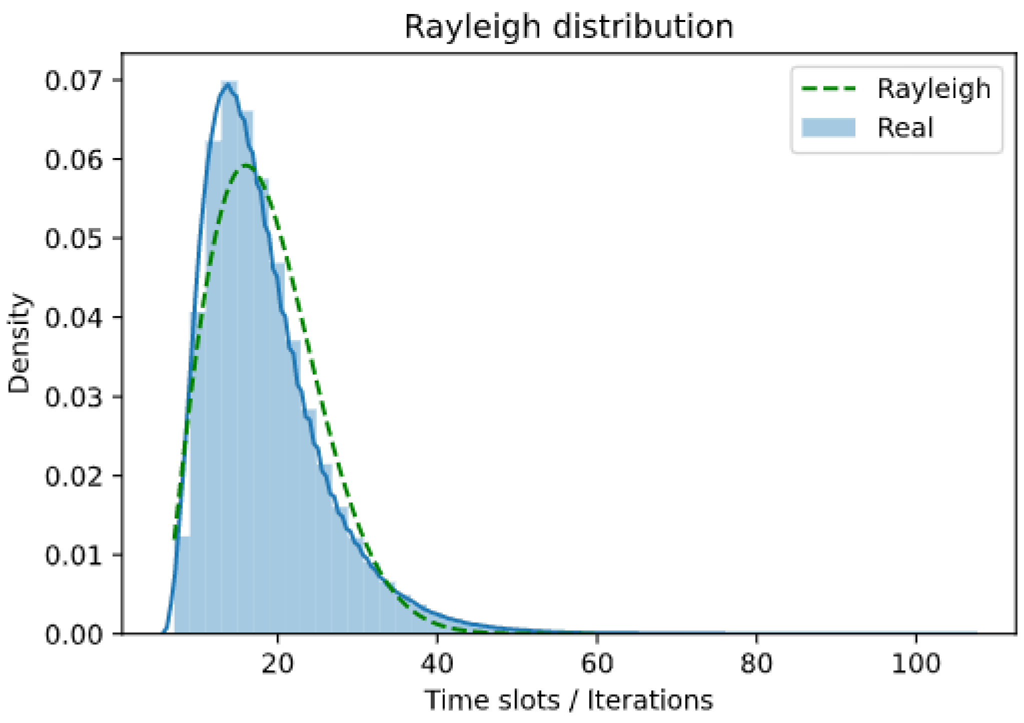


Figure 3.2: Approximated Rayleigh distribution after 1.000.000 iterations

As it is shown in Figure 3.2, our current real distribution can be approximated[6] by taking a Rayleigh distribution with loc = 5.7362 and $\sigma = 10.248355$. If we want to ensure that all sides of the dice (or all actions in our action set) are chosen at least once with a 99.99% confidence, we use the rayleigh property to get an estimate for a random choice scenario:

$$n_{\text{random}} \approx \frac{3.717}{\sigma} \approx 38 \text{ iterations},$$

and given that our random choices are only taken with a likelihood of $\varepsilon = 0.3$, we can write

[6]Scipy was used to approximate such parameters

$$n \approx \frac{n_{\text{random}}}{\varepsilon} \approx 127 \text{ iterations}$$

which are too many iterations to ensure that our binary does not get corrupted in the RL process. In summary, we could establish that depending on the ε chosen, we would need between 38 and 127 iterations to have enough confidence of taking every action at least once, but it is likely that the binary would break with that number of iterations.

Deep Learning approach It seems that a new approach that would allow our algorithm to effectively learn while exploring all the possible actions with the fewer number of iterations is needed. In this sense, we can take advantage of the Deep Learning concepts and try to build a neural network that automatically improves our decision-making process based on past rewards.

The strategy will be based on the use of the softmax function to convert our rewards into a probability vector (as shown in 3.2). This vector can then be used as part of a classification problem, where the output of this specific neural network would be a vector containing the probability for every action of being the best one at a determined time step.

Choosing a right architecture is the cornerstone when facing a Deep Learning problem. In this sense, opting for a FeedForward Neural Network (FNN) seems a plausible option. The context of our problem will also help us to close our initial design:

- The input layer shall be conformed by the inputs we have. This is our action space, so our input layer will have the same dimension that out action space (7 neurons)
- The output layer shall be conformed by the different actions and their associated probabilities. In this sense, the output layer shall have the same dimension that the input layer (7 neurons)
- The number of hidden layers will be kept as 1 given that our problem does not have a big number of dimensions or a high complexity in terms of data involved for the calculations.
- The number of neurons will be fixed at 100 in order to increase the prediction capabilities of the network, reduce the probability of falling into a local minimum and increase the learning rate of the network.

Figure 3.3 shows a generic FNN with 1 hidden layer, similar to what we are defining. In our scenario, $n = 7$ and we will choose $k = 100$.

Generating the malware One of the tasks that our agent will perform is to haphazardly generate the initial malware samples from a predefined set. In order to simplify our project,

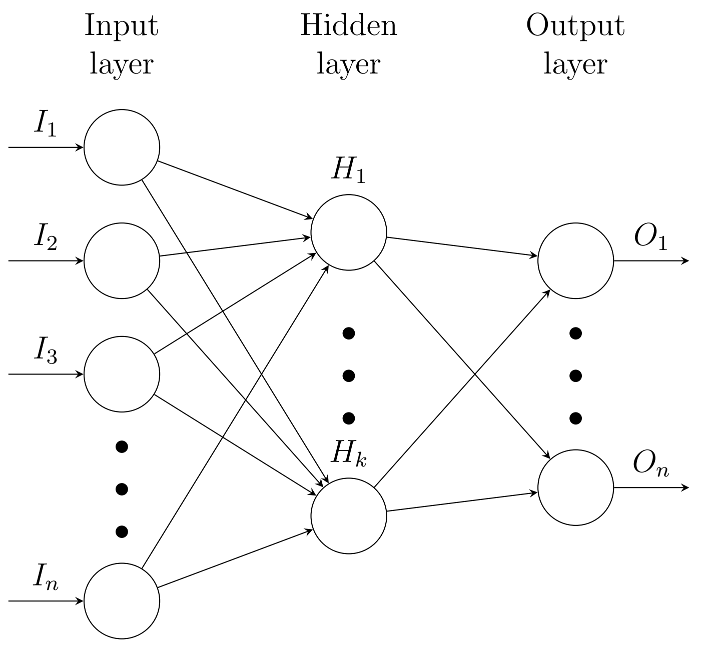


Figure 3.3: Feed-forward neural network with 1 hidden layer

we will focus on a set of widely used backdoors, shown in Table 3.1, where the variations are the existing different techniques for executing the same payloads.

Platform	**Architecture**	**Payload**	**Variations**
Windows	x86	Meterpreter	30
Windows	x64	Meterpreter	12
Windows	x86	Shell	11
Windows	x64	Shell	4
Windows	x86	VncInject	13
Windows	x64	VncInject	7
Windows	x86	Others	2
Windows	x64	Others	2

Table 3.1: Payloads used for the project

Table 3.2 shows the parameters we will use when generating our malware samples.

Parameter	**Value**
Host	127.0.0.1
Port	1234

Table 3.2: Parameters to be configured

These parameters have been chosen to ensure that, in case of accidentally running a sample outside the isolated environment, its effect are innocuous as it would listen or trigger a connection on the loopback interface, only locally addressable.

Finally, figures 3.4 and 3.5 show a high-level overview of the agent and the overall picture when it interacts with the environment.

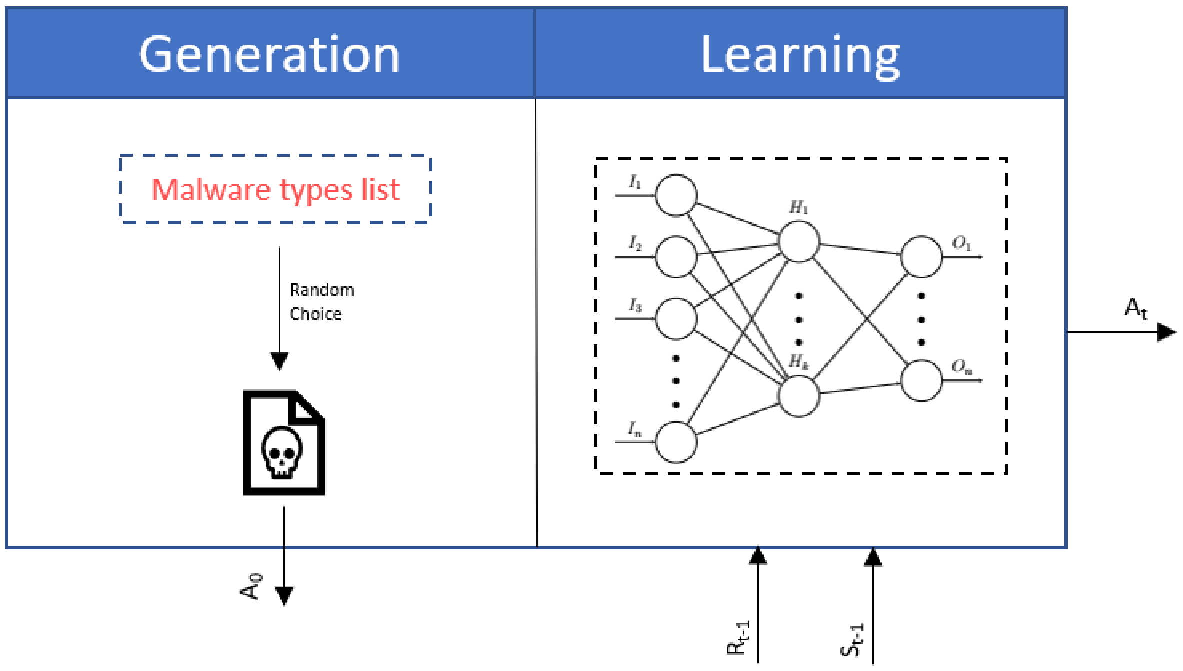


Figure 3.4: High-level overview of the designed Agent

Note that, as specified in 3.5, at a first moment we carry out a baseline definition. This is mainly sending the unprocessed sample to the antimalware engine in order to ascertain what is the detection rate prior to triggering our reinforcement learning process.

Once the foundations for our project have been laid, we can start to build a real implementation following the theoretical design that has just been defined.

3.2 Implementation

Once we are facing the implementation, some questions arise:

- What programming language is the most adequate for our task?
- What frameworks can be useful given our approach?
- How can we measure the success or failure of our system?

these and other questions are intended to be answered in this implementation phase. We will split this section between:

1. The implementation of the environment

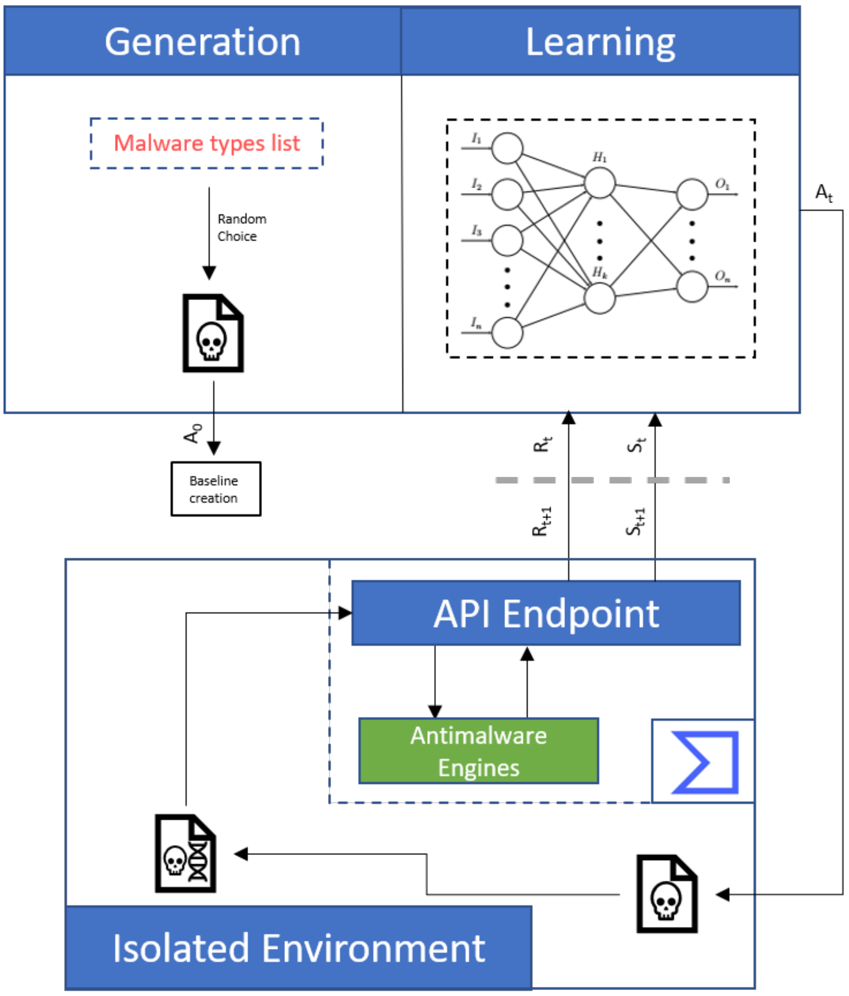


Figure 3.5: High-level overview of the system

2. The implementation of the agent
3. The implementation for the processing of the data gathered during the different iterations

After defining the aforementioned building blocks, we will be able to present conclusions and results.

As a cross requirement, the programming language that we will be using for the implementation will be *Python* due to its simplicity, flexibility and strong support received from the data science community through its packages and tutorials.

3.2.1 Implementing the environment

The environment is the cornerstone of our implementation, as we need to:

1. Implement an interface to communicate with our antimalware engine, including parsing the results and reporting back to the agent

2. Implement the actions that the agent can carry out to alter the environment
3. Implement the states and the rewards

3.2.1.1 API Communication

In order to implement the API communication, we will be using the *requests* package for python. This will allow for easy communication with HTTP(s) endpoints.

In order to effectively take advantage of the free capabilities of VirusTotal, the following actions are taken:

1. Create an user account
2. Request a personal API Key
3. Configure a header to use the API Key as a bearer token
4. Set up a request rate limit aligned with VirusTotal policies (i.e. 4 requests per minute)

Figure 3.6 shows the sequence diagram for our API Communication. It is worth noting that we need to implement a polling mechanisms to continuously check for the completion status of our analysis task, as we are dealing with a restful API without any kind of callback that allows for our environment to be notified based on events.

Once the results have been received, they are processed by means of the *JSON* package and translated into a *Pandas* dataframe, from which the following information is specially useful:

- Stats regarding how many antimalwares have detected the sample
- Detection information for the sample for each antimalware engine
- Specific information regarding the sample (size, sha256 hash, ...)

This provides us with valuable context information, as we can not only define the reward function, but also compare two different outputs from different iterations and map actions with specific evaded malware engines, correlate the sample size with detection rate, etc. This will be revisited later in this document.

The implementation of the parameters for our environment will be as follows:

- Our **reward function** will take the form we designed in 3.1.2.1
- Out **state** will be the last action carried out over the binary for each iteration

This information will be provided to the agent as a feedback that it can use to learn about the best course of action.

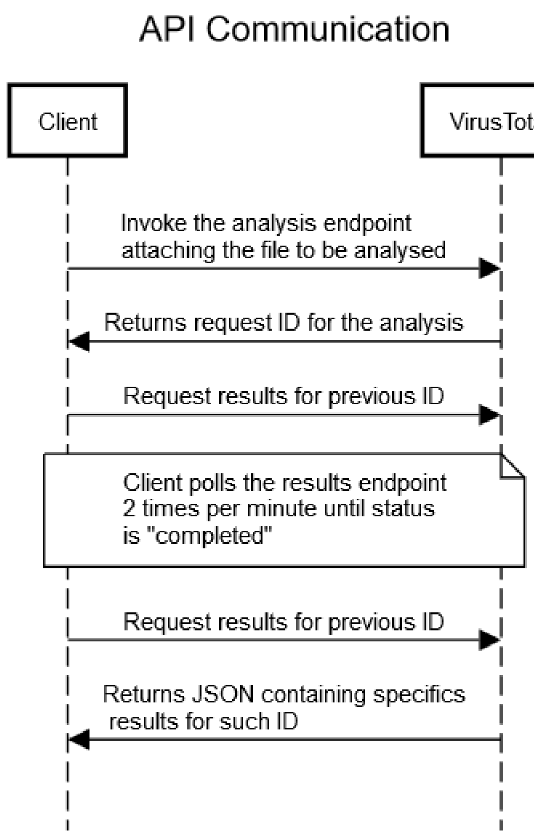

API Communication
Client
VirusTotal
Invoke the analysis endpoint attaching the file to be analysed
Returns request ID for the analysis
Request results for previous ID
Client polls the results endpoint 2 times per minute until status is "completed"
Request results for previous ID
Returns JSON containing specifics results for such ID

Figure 3.6: Sequence Diagram

3.2.1.2 Binary modifications

As has been stated in the design phase, we need to ensure that we can alter the binary without impacting functionality. In order to ease our task, we will rely upon the *LIEF*[8] framework for python, which allows us to interact with the binary file as an object that we can rebuild at the end of the process.

The PE Format Figure 3.7 shows a general overview of a PE File structure. As it was explained during the design phase, our approach will be to take advantage of this format and make modifications to our sample in a way that, at least, its SHA256 hash changes. Some examples are:

- Create new sections filled with random bytes
- Rename existing sections
- Add random bytes at the end of the sections
- Add new imports to the import table
- ...

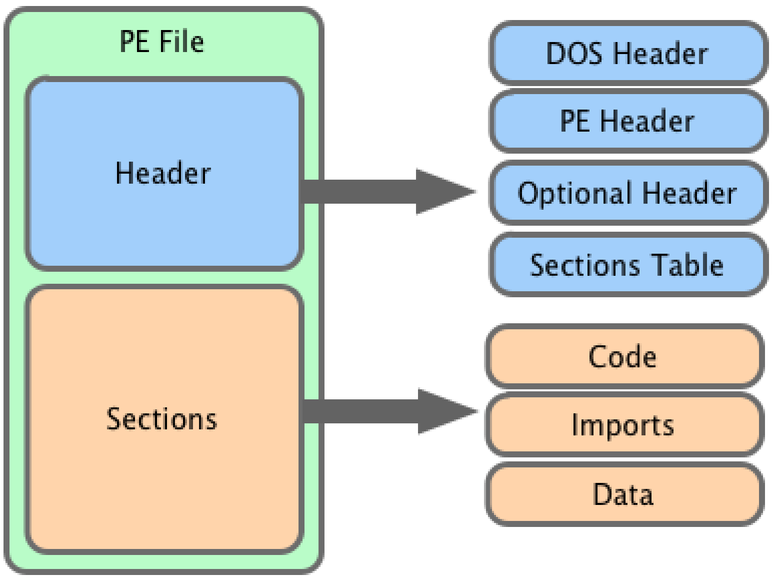


Figure 3.7: PE File structure

As no modification will be directly applied over the existing code, imports or data, the functionality of the binary should not be affected.

Additional information about the PE format can be found in Annex A.

Using LIEF *LIEF* will allow us to parse our binary file and work with an equivalent object which we can build later. Figure 3.8 shows a high-level overview of the LIEF architecture, while 3.9 shows a lower level overview of a modification over a ELF file

In our case, as stated before, we will be focusing on the PE Binary structure, as this is the one that Windows platform understand and can work with. Figure 3.3 shows the deployed setup in order to run LIEF.

Implementation Setup	
Architecture	x64
Operating System	Kali Linux 2020.3
Platform	Windows Subsystem for Linux (WSL)
Packages	Python 3.8 + LIEF

Table 3.3: Setup for testing

It is worth noting that, at the time of this writing, *LIEF* presented problems to be deployed over a Windows environment by using *pip*, that is why we moved to the WSL environment in order to test our implementation.

Every time we carry out the desired modifications over our file, a copy will be saved to be

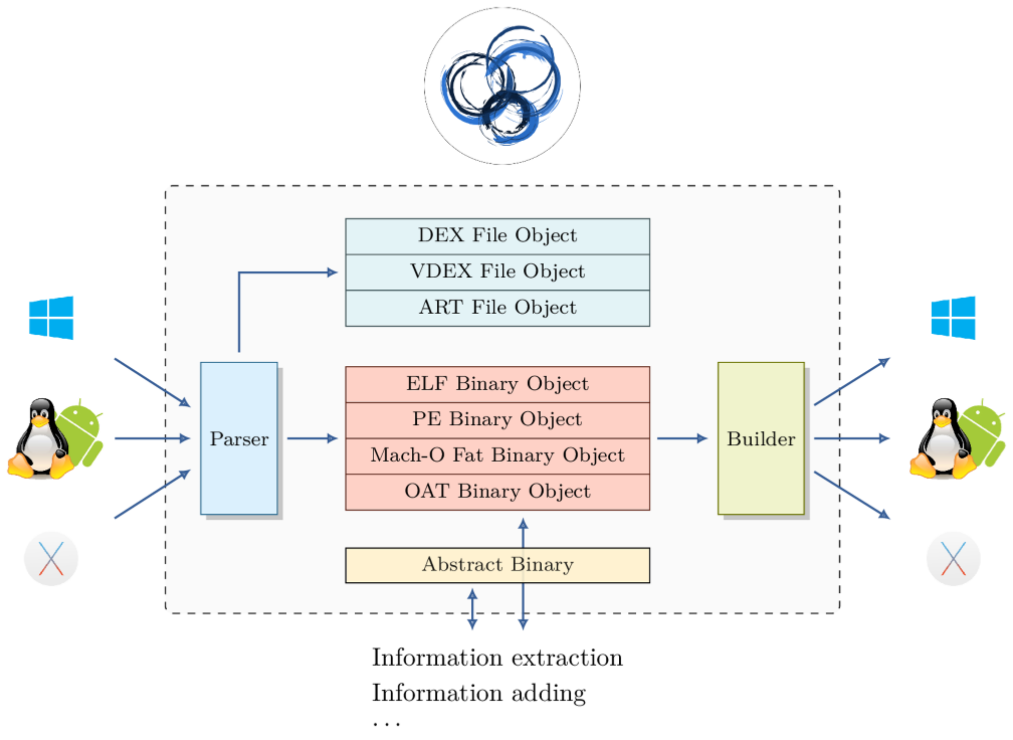


Figure 3.8: LIEF Architecture. Source: Quarkslabs

fed into the antimalware engine. Additionally, we will not carry out more than 1 modification per file, as every action involving a modification will be tested against the antimalware engine to analyse the reward and learn about the best course of action in the future.

There is a situation that is most common that we could have foreseen. As we have no automatic way of testing the files to ensure they are working correctly after every iteration, they might end up broken even if the actions are supposed to be innocuous from a theoretical standpoint. This was pointed out in previous researches [1] and explained through the invalid PE format that some malware samples are using, which when combined with out actions can effectively render our binary unstable or even corrupted.

While this is a setback which violates one of our initial assumptions, it is also of not particular significance in terms of evasion. This is because our initial approach was to evade static PE detection, and a broken binary still presents the static PE signatures that are analysed by the different antimalware engines. This means that even if our binary is not working, the detection rate for those antimalware engines relying on only static capabilities instead of dynamic ones should be unaltered, as no execution of such binary is needed to perform the static analysis.

Figure 3.10 shows a summary of the products / packages used in our environment's implementation.

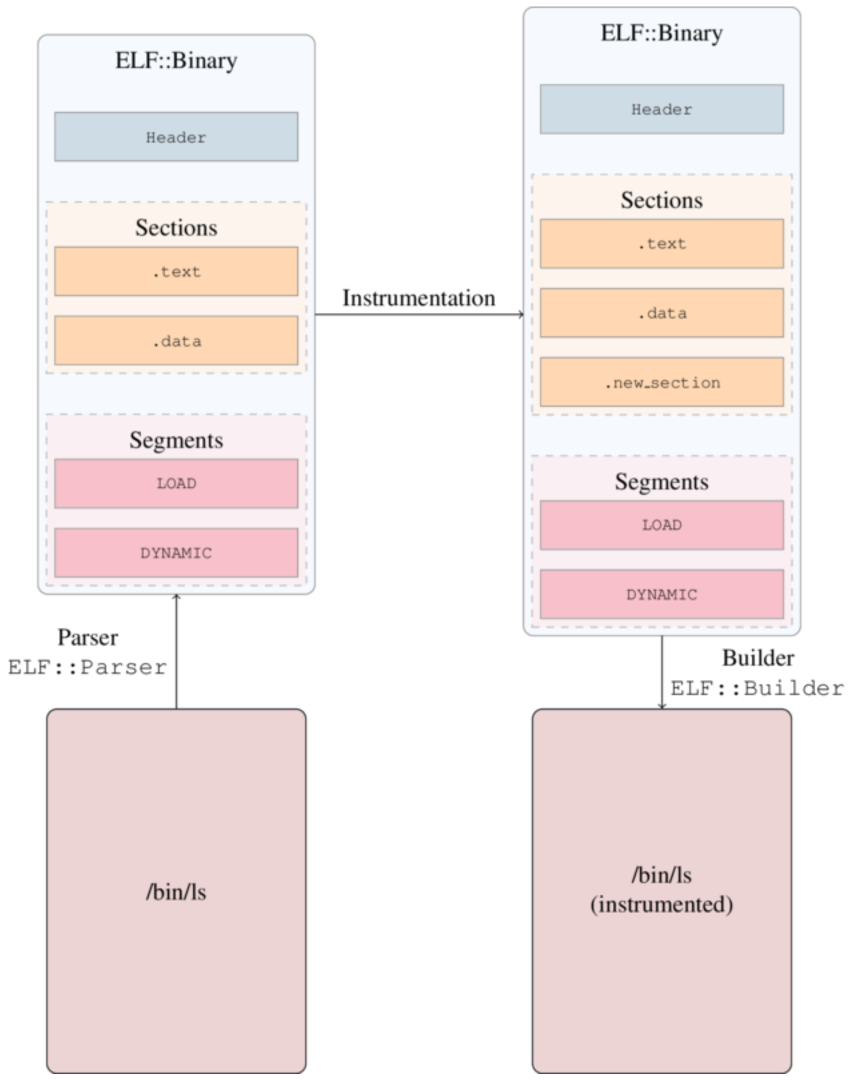


Figure 3.9: LIEF inner working for ELF files. Source: Quarkslabs

3.2.2 Implementing the agent

The agent has two main tasks to carry out:

1. Generate the initial malware sample
2. Gather the feedback from the environment, learn from it and take the best action to alter the environment

In order to simplify the implementation, we will make use of different packages from python and external tools that might assist us in this project.

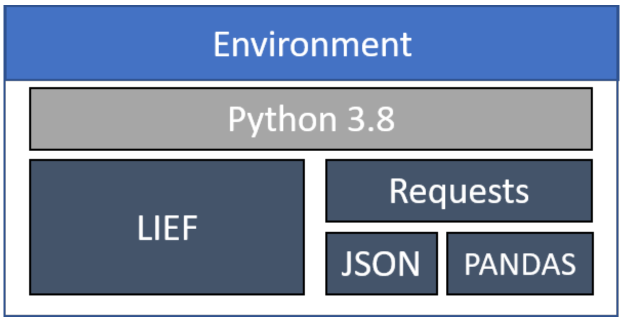


Figure 3.10: Implementation stack for the environment

3.2.2.1 Malware generation

Instead of sampling from an existing malware set, we will generate our own backdoors and sample from a set of predefined payloads referred in table 3.1. For this, we will make use of the *Metasploit* framework, specifically of the *msfvenom* tool that allows for the generation of different backdoors in different formats.

Our approach for the malware generation using *python* is as follows:

1. Define a list of payloads to be used
2. Choose a random payload from such list
3. Generate the sample by invoking *msfvenom*
4. Exclude the selected payload from the initial list
5. Provide the sample to the environment
6. Repeat the process

This process ensures that we have different payloads used in different runs. It is also of importance realising that some of the generated samples are based on predefined binary files which contain strings easily recognisable by any antimalware engine. This is not convenient, as the samples would be spotted independent on the changes we carry out. For this reason, we will always apply a small change over the generated samples, specifically replacing the word “PAYLOAD” for a random 6-characters string.

3.2.2.2 Feed-Forward Neural Network

In order to implement our neural network, we will make use of *PyTorch* as our framework to implement the design discussed in the previous section. In [9] we can study some basic implementations by using *PyTorch*. As for our problem, we define:

- The dimensions of the different layers:

  - $n = 7$ for the input layer
  - $k = 100$ for the hidden layer
  - $n = 7$ for the output layer

- The input function as the weighted sum.
- The activation function to be used as the rectifier, due to its advantages regarding the training of deep networks and the characteristic of not having a binary output.
- The loss function as the Mean Squared Error (MSE)
- The optimizer function to be used as Adam[7] as it combines a series of properties to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Additionally, the softmax function will be implemented to ensure that we always work with normalized values resembling a probability distribution. In each step, we will choose an action based on its probability value, and this values will be balanced in every iteration of the neural network based on the perceived rewards.

Finally, figure 3.11 shows a high-level overview of our implementation for the agent.

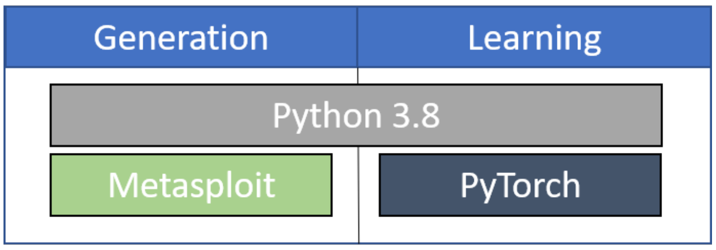


Figure 3.11: Agent's implementation stack

3.2.3 Data processing

Once our environment and agent have been built, it is equally important to have an idea on how the resulting data will be cleansed, processed and interpreted. A fair amount of information is available thanks to the API we are using, so we can gather for every mutation:

- State of detection state for every antimalware engine provided by VirusTotal
- Global stats of our detection rate

Having this information for every iteration, we can focus on the differences between two successive iterations or even the difference between the initial sample and the last mutation.

[7]Adaptive Moment Estimation

In order to focus on these differences, we will make use of the *DeepDiff* package for python, which allow us to compare different JSONs outputs and obtain the relevant differences.

In order to process those differences and generate relevant visualizations, we will make use of *Pandas*' dataframes and the*Seaborn* package for Python. Figure 3.12 shows a basic overview of this processing.

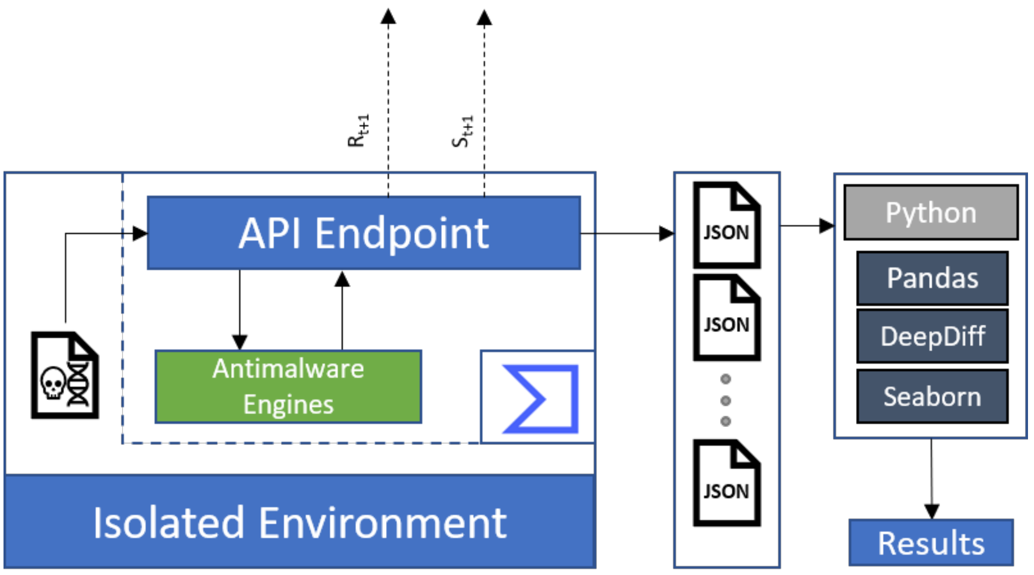


Figure 3.12: Data processing overview

Once our full system has been implemented[5] as shown in figure 3.13 we can move to analyse the results obtained.

3.3 Results and analysis

3.3.1 Scope and limitations

Due to the rate limit imposed by our antimalware engine API, the time constraint that our project is subject to and additional drawbacks found during the testing phase, our results will be limited to:

- 9 different malware samples randomly generated from the predefined dataset
- 20 mutations per sample

In our worst-case scenario, as no multi-threading is implemented in our code, we need to wait an average of 5 minutes until the sample is analysed[8]. After approximately 15 hours, we have 180 new mutations fully analysed by VirusTotal.

[8]This has been empirically observed, depending on the VirusTotal workload at the time of the tests

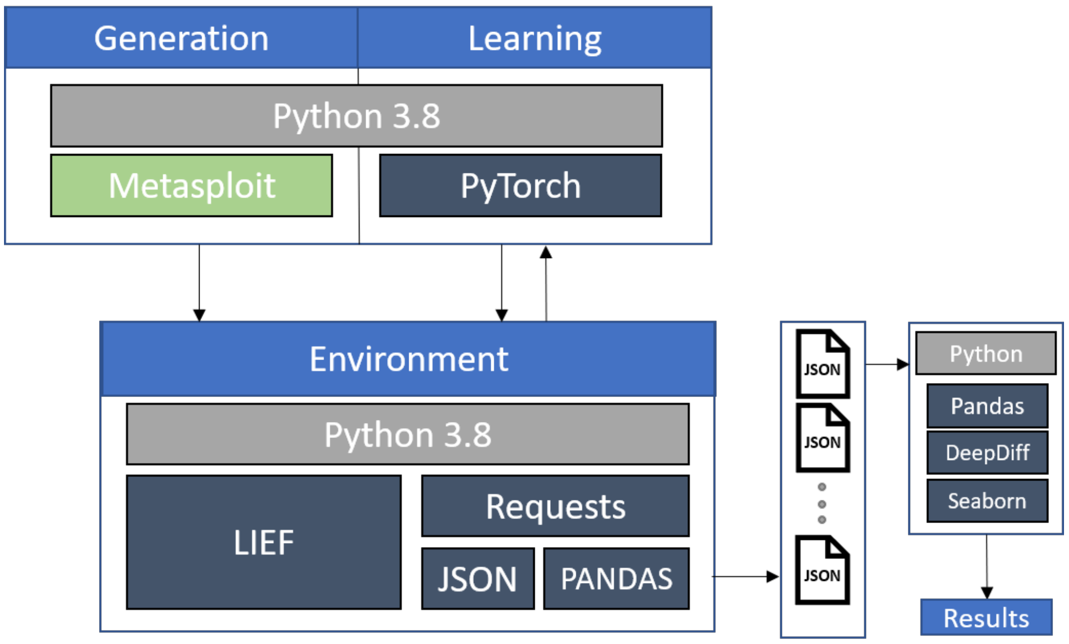


Figure 3.13: Full system stack overview

Another reason why we have limited our dataset is due to VirusTotal ToS[9]. Every mutation we send will be typically detected by at least one antimalware vendor, and this will trigger an automatic submission to those vendors tagging it as harmless or undetected. This means that every time our algorithm runs, we are generating noise to multiple vendors. Even if no policy is being violated by our use of the API, we will try to keep noise to a minimum.

3.3.2 Main results

A way to measure the success of our implementation is to graph the evolution of the reward signal for every sample and its resulting mutations. Figure 3.14 shows this trend of the reward function for the different payloads and mutations automatically generated by our neural network.

A growing trend for the different iterations in each one of the sample under analysis can be observed. In this sense, based on the way in which the reward function was defined, figure 3.15 shows the expected negative correlation between our reward and the number of antimalware vendors detecting our sample.

We can also appreciate there is difference between the slopes for the different payloads tested, some of them evading a greater number of antimalware engines by using the same iterations. Figure 3.16 shows how smaller samples tend to evade antimalware detection sooner than larger ones.

[9]Terms of Service, specifically the section regarding “Sample & Community Content Guidelines”

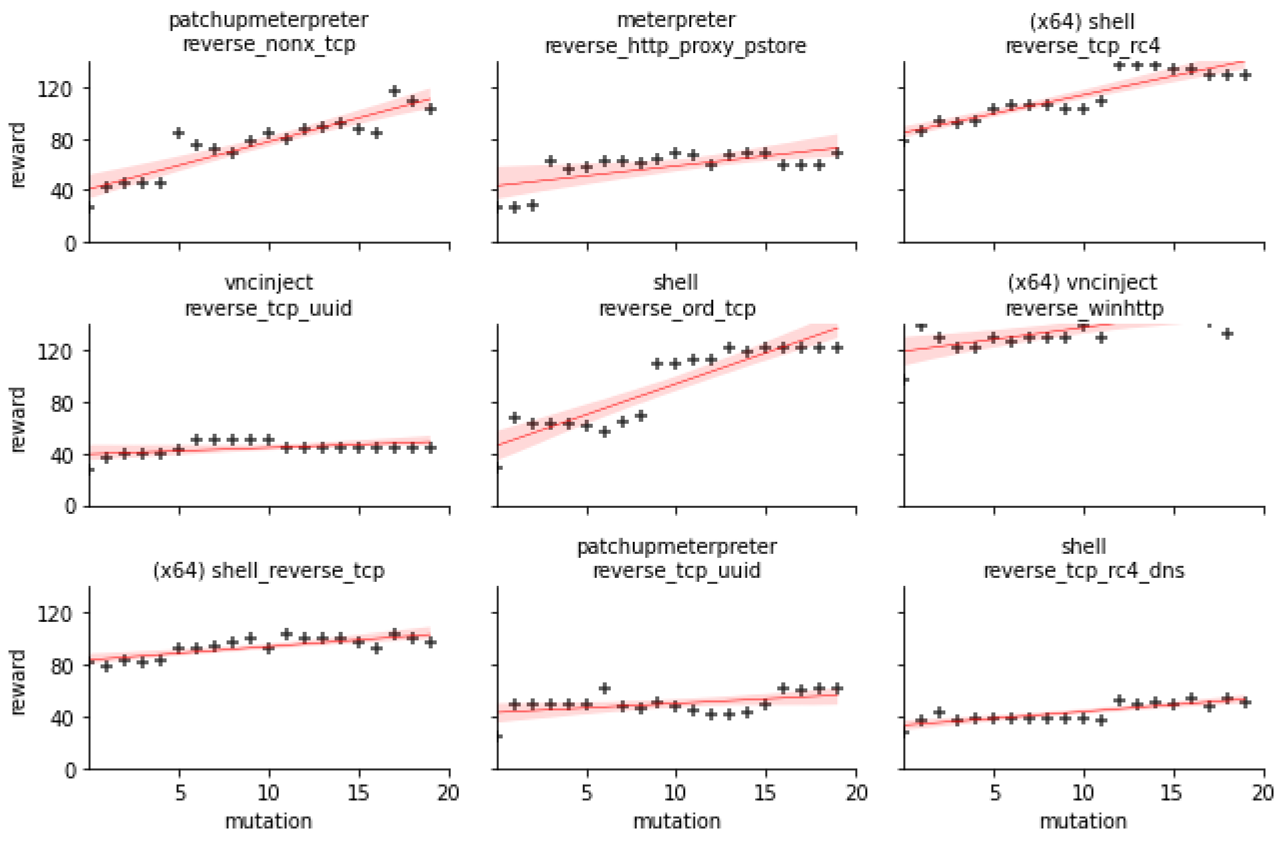


Figure 3.14: Reward signal for every payload and mutation

In a first moment, we can think that this is due to the different between our payloads being staged or inline payloads:

- **Inline payloads** are those that contains all the exploit, including shellcode and necessary logic to carry out the malicious actions it has been designed for. It works as an all-in-one, taking the form of a binary that can be executed in the target architecture in a stable way.
- **Staged payloads** are those that only contains a small amount of code necessary to communicate with a C2[10] Server to download (stage) the main payload. Examples of this are

As staged payloads are smaller than its counterparts, they also take advantage of a smaller signature in terms of malware detection. Inline payloads usually contains specific strings, traces of shellcodes and a more complex implementation than a stager payload, allowing antimalware vendors to tag different parts of such binary. From this, it is inevitably derived that our changes,

[10]Command and control, or C&C

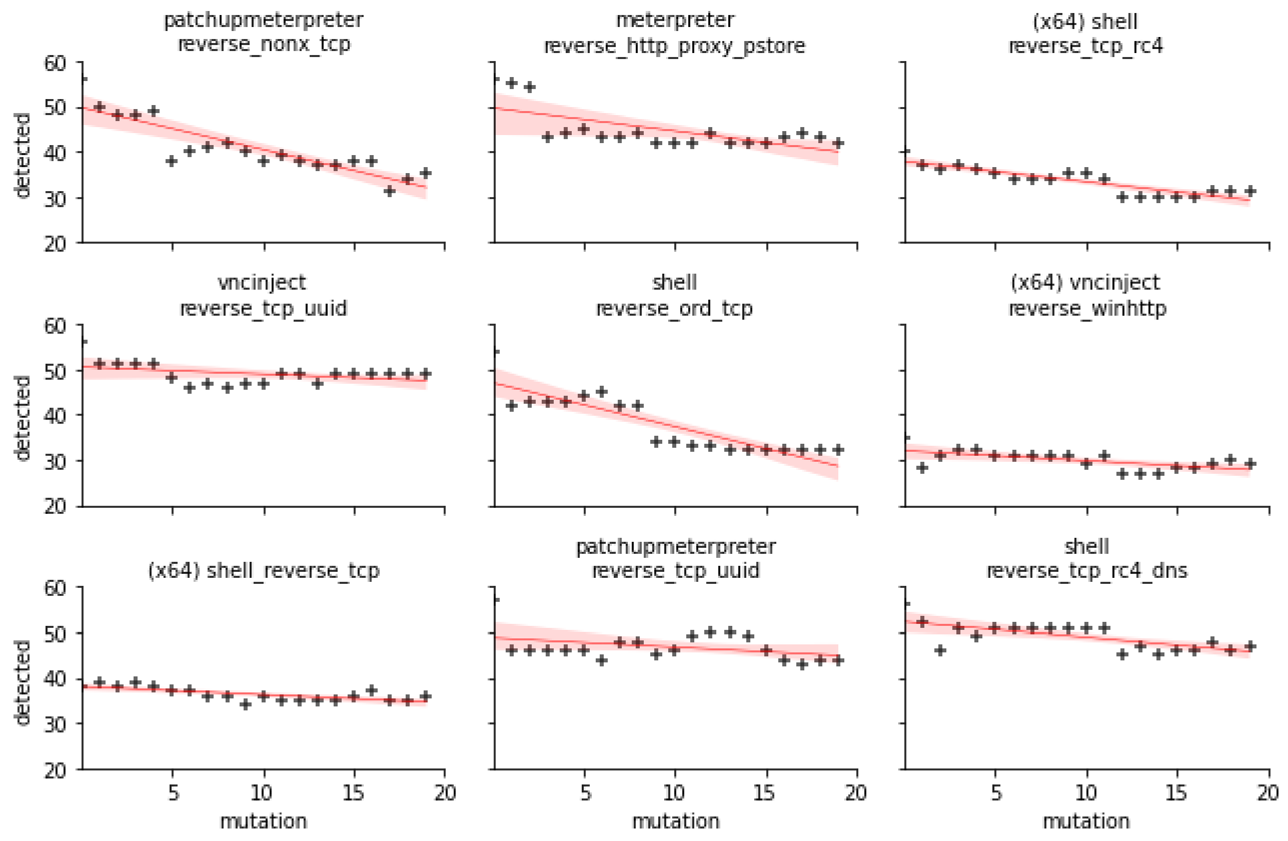


Figure 3.15: Number of antimalware vendors marking the samples as malicious/suspicious

predefined through our action space, would not have a huge impact over the inline payloads, as the antimalware vendors would be tagging specific strings or pieces of the actual code.

Despite that, after analysing the code for our payloads[11], we find that we only got staged payloads in our dataset, meaning that there is a more fundamental component to this difference, being it only a matter of the strings contained in the original binary.

Table 3.4 shows the improvement for every payload calculated as $\frac{\text{Final}-\text{Initial}}{\text{Initial}}$ alongside the number of strings present in the file, obtained by using the *strings* command in Linux. We can see how the initial detection rate is lesser for those binaries containing less predefined strings.

There are also notable improvements in some of our samples after undergoing our RL process. For example, the *Reverse Ordinal TCP* payload is widely detected in an initial stage, but it seems to evade a great number of antimalware engines.

Unfortunately, as it is depicted in the table 3.5, further tests show us that most of our binaries have broken after 20 iterations, with only 33.3% of them working correctly.

Despite further research needs to be made to ascertain the exact reasons why this happened, it seems plausible that LIEF is randomly breaking the binary file in the building phase. This

[11]Source code available at https://github.com/rapid7/metasploit-framework/blob/master/documentation/modules/pa

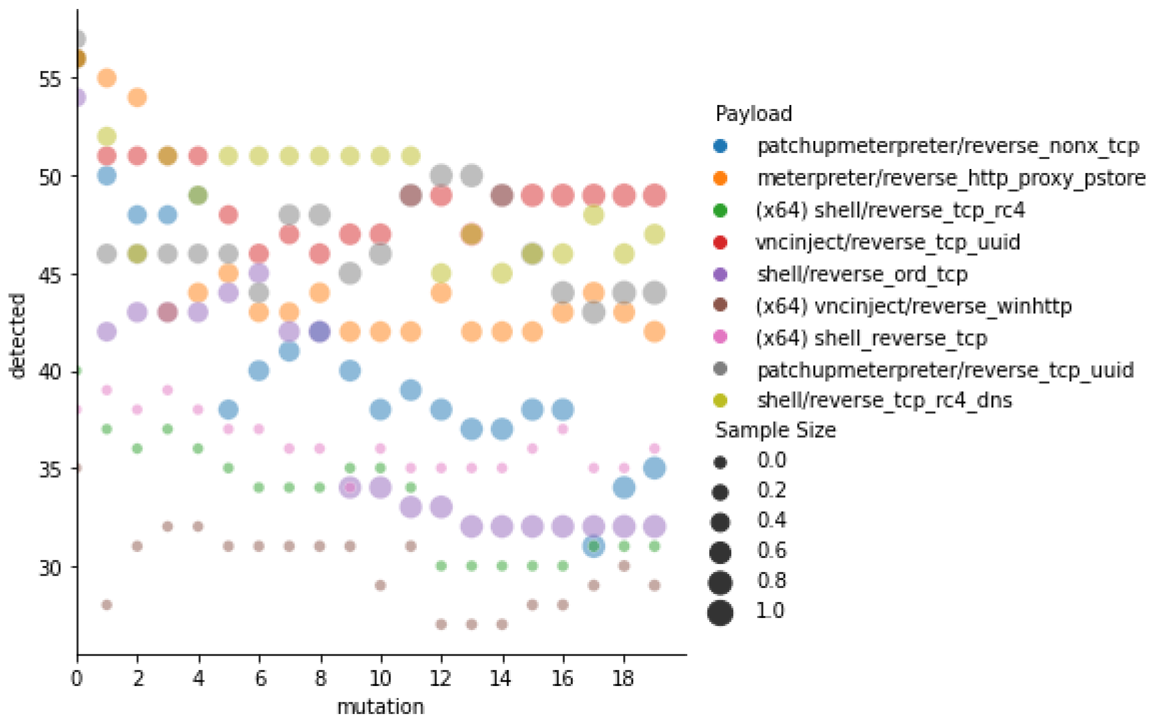


Figure 3.16: Number of antimalware vendors marking the samples by file size

was already pointed out in [1] and suspected to happen due to the mutations not respecting obfuscation tricks or less common uses of the PE format. Some examples are:

- Binaries using specific instructions in the import address table, forcing us to do a custom patching over the assembly code in the build phase.
- Binaries violating the PE standard to take advantage of Windows lazy parsing in order to avoid detection.

Apart from this inconvenience, we have still achieved an amount of evasion against static detection. It is important to note that a static analysis does not need to execute the file, just understand its structure and tag specific signatures. Antimalware engines throwing an undetected or failure result as a consequence of not being able to run the mutation are probably relying in dynamic detection capabilities rather than static ones.

3.3.3 Additional results

Apart from the perceived improvement in our evasion ratio, we can also take advantage of the information provided by the multiple antimalware engines in order to assess the effectiveness of the different vendors. Figure 3.17 shows a top 10 of vendors evaded after major changes to our sample (20 iterations) and only after a one iteration. As expected, most of them are not

Payload (Windows)	Initial	Final	Improvement	Strings
shell/reverse_ord_tcp	54	32	41%	698
patchupmeterpreter/reverse_nonx_tcp	56	35	38%	690
meterpreter/reverse_http_proxy_pstore	56	42	25%	706
x64/shell/reverse_tcp_rc4	40	31	23%	25
x64/vncinject/reverse_winhttp	35	29	17%	27
shell/reverse_tcp_rc4_dns	56	47	16%	700
patchupmeterpreter/reverse_tcp_uuid	57	49	14%	697
vncinject/reverse_tcp_uuid	56	49	13%	684
x64/shell_reverse_tcp	38	36	5%	22

Table 3.4: Initial and final number of detections for our payloads. Those marked in bold have been tested and verified to work properly.

widely-known vendors, but still we can see some popular ones (McAfee or MalwareBytes) being evaded after just 1 iteration for at least 2 of our samples.

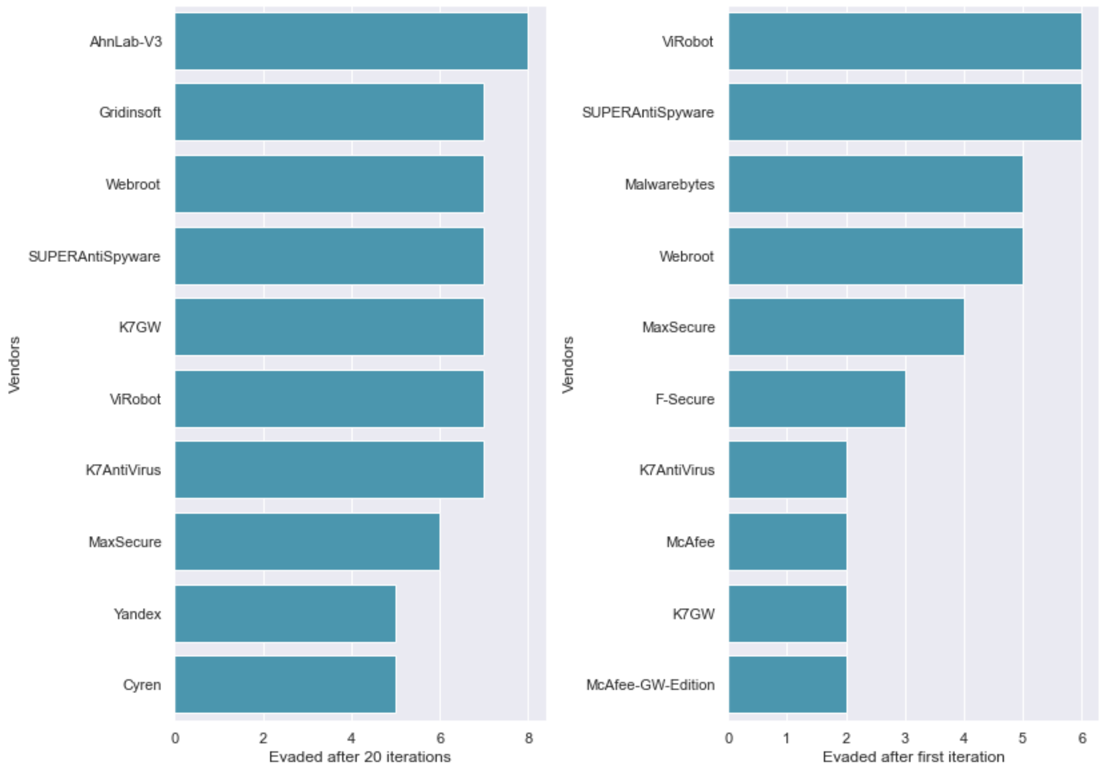


Figure 3.17: Top 10 of antimalware vendors evaded after 20 iterations (left) and after only 1 iteration (right)

Figure 3.18 shows the count of evasion for every vendor after 20 iterations. For this, we have compared differences in terms of detection between our original samples and our last mutations.

Payload	File	Details
(x64/shell) reverse tcp	jtvkjyk.exe	**OK** - Received reverse connection
(x64/shell) reverse tcp_rc4	wtzafio.exe	**Broken** - Invalid access to memory location
(x64/vncinject) reverse winhttp	gqviaoe.exe	**Broken** - Invalid access to memory location
(vncinject) reverse tcp (uuid)	djeavkm.exe	**Broken** - Invalid access to memory location
(patchupmeterpreter) reverse nonx tcp	obphfrt.exe	**Broken** - Invalid architecture
(patchupmeterpreter) reverse tcp uuid	djeavkm.exe	**Broken** - Invalid access to memory location
(shell) reverse ord tcp	uipecyo.exe	**Broken** - Missing dependency
(shell) reverse tcp rc4 (dns)	bqigswm.exe	**OK** - Received reverse connection
(meterpreter) reverse http proxy pstore	aybhclr.exe	**OK** - Received reverse connection

Table 3.5: Status of our final mutations

3.3.4 Conclusion

During the development of this project, a simple framework for antimalware evasion based on reinforcement learning has been designed and implemented. The results show how an existing malware sample can be automatically modified by assessing how the environment reacts to the mutations generated from such sample, learning from past events by means of a feed-forward neural network and taking actions intended to keep the binary functionality intact.

We have also shown how the reward function presents a positive slope as the iterations increase, which implies an equally increasing evasion rate, observed to improve between 5% and 41% . From this point, we moved to test the resulting mutations in order to assess their effectiveness in terms of functionality, and found that most of the samples were broken during the process, only a 33% of them keeping their original functionality. Further research is needed in that direction to ascertain if mutations can be fixed or need to be discarded and re-generated in a different way to keep the functionality.

Finally, we have made use of the information gathered for every mutation and performed a simple benchmark to assess the static detection capabilities of different antimalware vendors. We have found that wide-known vendors seem to be harder to evade, while small vendors can be bypassed in the very first iteration of our algorithm.

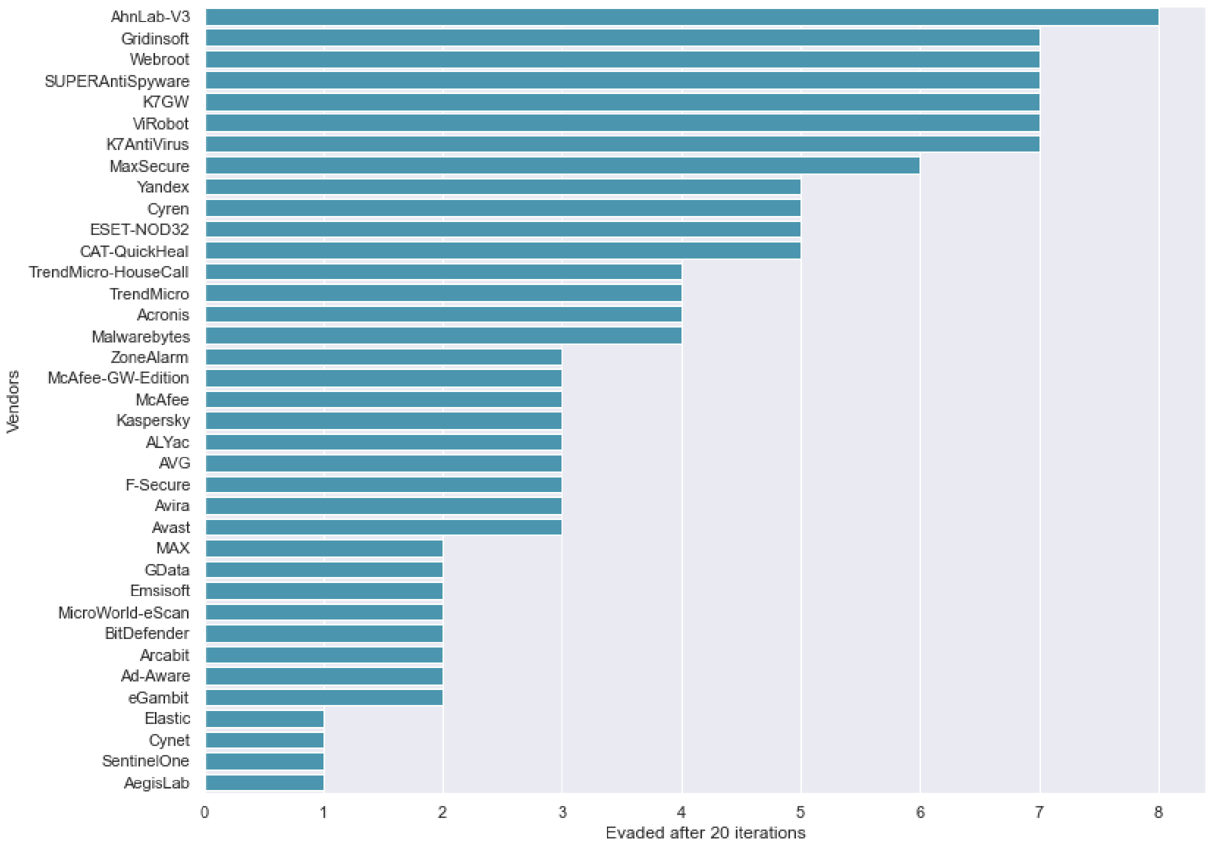


Figure 3.18: Total of antimalware vendors evaded after 20 iterations

3.3.5 Future lines of work

Given the time constraints for this project, some improvements need to be considered as part of the future lines of work. The following is a non-exhaustive list of potentially interesting points to be taken into account:

- Improve the agent to automatically check if a mutation is still functional, and apply a penalization over the reward function in case the resulting binary is broken, allowing to rollback the changes and take a different path.
- Explore further neural network models that might allow for a faster learning rate.
- Add new actions to the action space. Some examples are:
  - Add an unconditional JMP from a new EIP to the original one
  - Add packing capabilities
  - Add stub encryption

Appendices

Appendix A

Malware Generation and Manipulation

A.1 Metasploit Overview

Metasploit is described by Rapid7 as the world's most used penetration testing framework [1]. It is a complete suite that intends to cover the end-2-end process when planning a compromise job, as depicted in Figure A.1

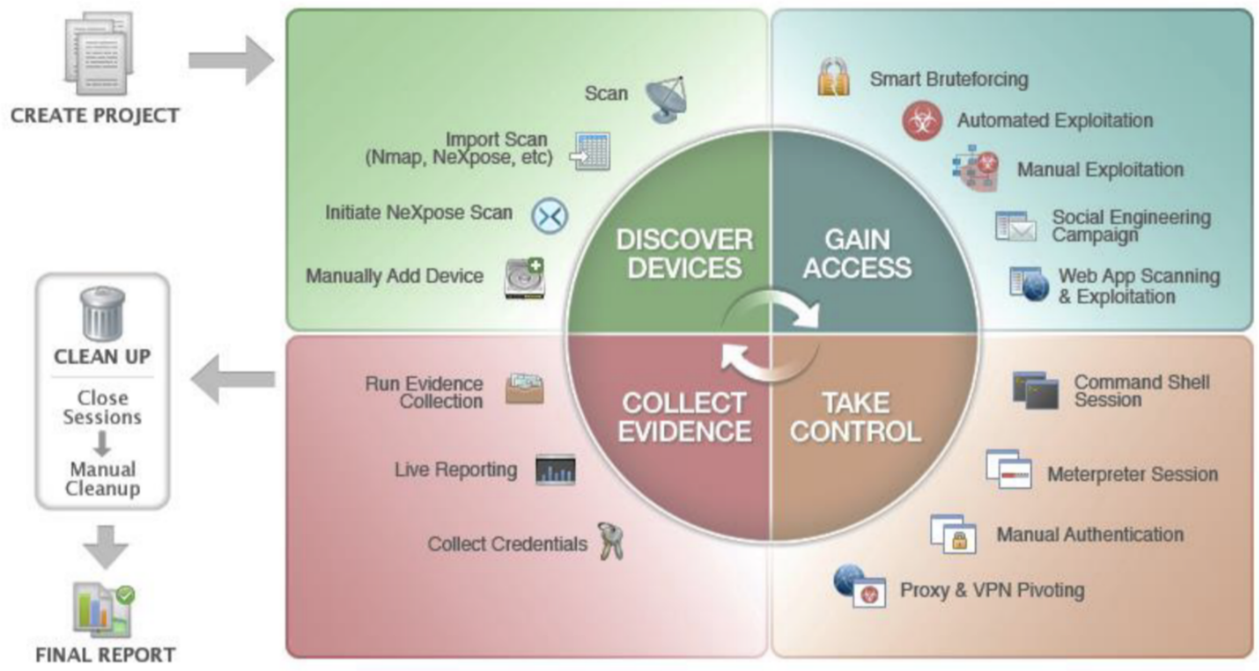


Figure A.1: Overview of MetaSploit Framework

As part of this process, the payload generation usually plays a fundamental role in the exploitation and post-exploitation phases. In order to ease this task, MetaSploit provides us

[1]https://www.metasploit.com/

with the *msfvenom* framework, which allows for a quick generation of 500+ payloads with different encodings and formats.

The following is an excerpt of the code used to invoke msfvenom, which returns the payload randomly chosen and the path for the generated sample.

```python
PAYLOADS = [ ... ]

def create_payload(used_payloads):
    output_file = f"/tmp/{get_random_string(4)}.exe"
    available_payloads = list(set(PAYLOADS) ^ set(used_payloads))
    payload = random.choice(available_payloads)

    # Payload generation

    args = ("msfvenom", "-p", payload,
    ↪  "LHOST=127.0.0.1","LPORT=1234","-f","exe","-o",output_file)
    popen = subprocess.Popen(args,stdout=subprocess.DEVNULL)
    popen.wait()

    # Subtle payload modification

    args = ("sed", "-i", f"s/PAYLOAD/{get_random_string(7)}/g", output_file)
    popen = subprocess.Popen(args, stdout=subprocess.PIPE)
    popen.wait()
    return payload, output_file
```

This code also made a slightly modification over the binary by using "*sed*" in order to remove suspicious strings that might be tagged by antimalware vendors.

Note that *msfvenom* provide us with additional powerful options, such as:

- Generating different formats, from raw shellcodes to office documents macros
- Encrypting the payload to evade detection
- Constraint the payload to a maximum length

but, for the sake of simplicity, we will be generating the payloads without using advanced options, as the higher the complexity of the binary, the higher the probability of such binary getting corrupted after a number of iterations by using *LIEF*

A.2 The PE format

As it is extensively explained in [6], the Windows PE format describes the structure of modern Windows program files such as .exe, .dll, and .sys files and defines the way they store data.

PE files contain x86 instructions, data such as images and text, and metadata that a program needs in order to run.

The PE format was originally designed to do the following:

- Tell Windows how to load a program into memory by describing with parts of a file should be loaded into memory, and at what location.
- Supply medio or resources that a running program may use in the course of its execution (e.g. strings, images or videos)
- Supply security data such as digital code signatures to ensure that code comes from a trusted source.

The PE format accomplishes all of this by leveraging the series of constructs shown in Figure A.2

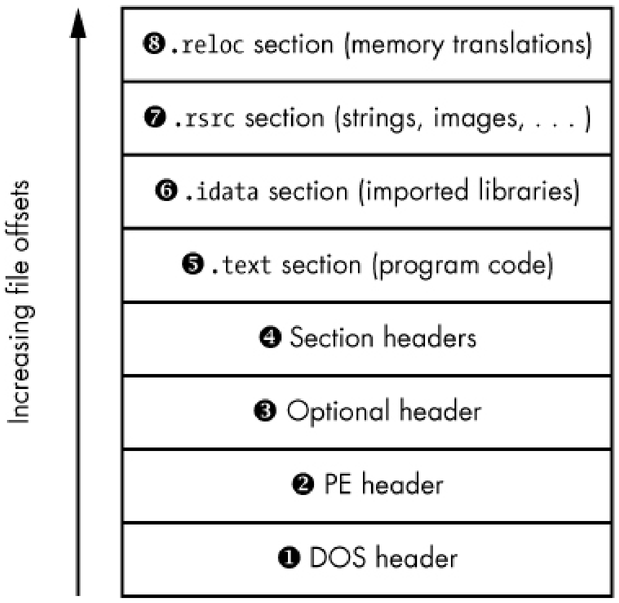

❽ .reloc section (memory translations)
❼ .rsrc section (strings, images, . . .)
❻ .idata section (imported libraries)
❺ .text section (program code)
❹ Section headers
❸ Optional header
❷ PE header
❶ DOS header
Increasing file offsets

Figure A.2: PE File

As the figure shows, the PE format includes a series of headers telling the operating system how to load the program into memory. A brief explanation for the different elements follows:

The PE Header The PE Header defines a program's general attributes such as binary code, images, compressed data, and other program attributes. It also tells us whether a program is designed for 32 or 64-bit systems. The PE header provides basic but useful contextual information to be used by an antimalware engine. Indeed, the header includes a timestamp field that can give away the time at which the malware author compiled the file, being this of special relevance when it comes to binary templates generated by *msfvenom* in our case.

The Optional Header The optional header defines the location of the program's entry point in the PE file, which refers to the first instruction the program runs once loaded, as well as the size of the data that Windows loads into memory, the program targets (e.g. Windows GUI or Windows command line), and other high-level details about the program. The information in this header is crucial, as a program's entry point tells where to begin the execution flow.

Section Headers Section headers describe the data sections contained within a PE file. A section in a PE file is a chunk of data that either will be mapped into memory when the operating system loads a program or will contain instructions about how the program should be loaded into memory. In other words, a section is a sequence of bytes on disk that will either become a contiguous string of bytes in memory or inform the operating system about some aspect of the loading process.

Section headers also tell Windows what permissions it should grant to sections, such as whether they should be readable, writable, or executable by the program when it's executing.

A number of sections, such as .text and .rsrc, are depicted in Figure A.2. These get mapped into memory when the PE file is executed. Other special sections, such as the .reloc section, aren't mapped into memory. The following is a brief description of the some of different sections types that can be found in a PE file:

The .text Section Each PE program contains at least one section of x86 code marked executable in its section header; these sections are almost always named .text

The .idata Section The .idata section contains the Import Address Table (IAT), which lists dynamically linked libraries and their functions. The IAT reveals the library calls a program makes, which in turn can betray the malware's high-level functionality.

The Data Sections The data sections in a PE file can include sections like .rsrc, .data, and .rdata, which store items such as mouse cursor images, button skins, audio, and other media used by a program.

The information in the .rsrc (resources) section can be vital to antimalware engines because by examining the printable character strings, graphical images, and other assets in a PE file, certain elements can be tagged and linked with the binary functionality.

The .reloc Section A PE binary's code is not position independent, which means it will not execute correctly if it's moved from its intended memory location to a new memory location. The .reloc section gets around this by allowing code to be moved without breaking. It tells the Windows operating system to translate memory addresses in a PE file's code if the code has been moved so that the code still runs correctly. These translations usually involve adding or subtracting an offset from a memory address.

Bibliography

[1] Hyrum S. Anderson, David Evans, Anant Kharkar, Phil Roth, and Bobby Filar. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv: 1801.108917v2*, 1 2018.

[2] Hyrum S. Anderson, Anant Kharkar, Phil Roth, and Bobby Filar. Evading machine learning malware detection. *Blach HAT USA*, 7 2017.

[3] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. Armed: How automatic malware modifications can evade static detection? *5th Internationalk Conference on Information Management*, 2019.

[4] Zhiyang Fang, Junfeng Wang, Boya Li, Siqi Wu, Yingie Zhou, and Haiying Huang. Evading anti-malware engines with deep reinforcement learning. *IEEE*, 4 2019.

[5] Francisco Javier Gómez Gálvez. Implementation for static PE evasion. `https://github.com/fgomezgalvez/StaticPEvasion`, 2020.

[6] Joshua Saxe and Hillary Sanders. *Malware Data Science - Attack Detection and Attribution.* No Starch Press, 2018.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning:An Introduction (2nd Edition).* The MIT Press, 2015.

[8] Romain Thomas. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/, April 2017.

[9] Alexander Zai and Brandon Brown. *Deep Reinforcement Learning in Action.* Manning, 2020.