Reinforcement Learning-aided Dynamic Analysis of Evasive Malware

MATTEO BUNINO

INTERNSHIP AT HUAWEI - AI4SEC
MSC IN DATA SCIENCE AND ENGINEERING

MSc Thesis, September 2021 - February 2022

Supervisors from PoliTO:
PROF. FRANCESCO VACCARINO
francesco.vaccarino@polito.it

Supervisor from Huawei:
DR DANIELE SGANDURRA
daniele.sgandurra@huawei.com

Scientifically supporting AI4Sec team members:
DR NEDIM SRNDIC
ADRIAN CHIRITA

Abstract

As cyberspace becomes more and more complex, malware authors strive to take advantage of the growing number of vulnerabilities. This requires security researchers to invest more effort in developing automated malware analysis tools, able to cope with the increasing pace of suspicious binaries. The Achilles heel of automated malware analysis tools is evasive malware, which may put countless strategies in place to impede analysis. For instance, malware trying to detect that they are under observation in an analysis environment and, as a consequence, conceal their malicious behavior by performing only innocuous operations. Evasive malware can hinder analysis by either performing static code obfuscation (e.g. via packers, crypters) or dynamic evasion (e.g. sandbox and debugger evasion).

The goal of this thesis is to explore the application of Reinforcement Learning (RL) to dynamic analysis, to reduce the burden of exhaustive exploration of conditional paths. To this end, we develop a model that is capable of noticing new evasive schemes via RL. State-of-the-art approaches generally identify evasions through *fingerprinting*, which is not effective in identifying slight mutations of evasion schemes. In contrast, this work employs a language model to abstract out the syntax of binary code, while preserving its semantics. Furthermore, to improve over state-of-the-art solutions, the presented solution considers both evasion schemes and the malicious nature of regions of code protected by evasive conditions, to better distinguish true evasive behaviors from false positives. As a consequence, this method is easily extended to guide the search of hidden malicious capabilities in a suspicious binary.

However, RL with function approximation, compared to supervised learning, is subject to additional sources of instability during training since the assumption of i.i.d. samples is often violated. In addition, as in many real-world RL applications, the analysis is complicated by the presence of sparse rewards and non-Markov states. To take this into account, we conduct extensive experiments devoted to attain convergence to a good policy, which includes the comparison of three improvements over the vanilla DQN model. The results show how the RL agents can learn good policies, while acting on multiple binaries at the same time, being able to correctly retrieve hidden malicious capabilities with an improvement over a random exploration policy.

Abstract

Comme le cyber espace devient de plus en plus complexe, les auteurs de logiciels malveillants s'efforcent de tirer parti du nombre croissant de vulnérabilités. Cela oblige les chercheurs en sécurité à investir des efforts dans le développement d'outils d'analyse automatisés des logiciels malveillants, capables de faire face au rythme croissant des binaires suspects. Le talon d'Achille des outils d'analyse automatisée des logiciels malveillants est un logiciel malveillant évasif, qui peut mettre en place des stratégies innombrables pour empêcher l'analyse. Par exemple en détectant ce qu'il est sous observation dans un environnement d'analyse et, par conséquent, en masquant son comportement malveillant en exécutant des opérations inoffensives. Les logiciels malveillants évasifs peuvent entraver l'analyse en effectuant soit une offuscation de code statique (par exemple, des packers, des crypteurs) ou une évasion dynamique (par exemple, une évasion de sandbox et de débogueur).

L'objectif de cette thèse est d'explorer l'application de l'apprentissage par renforcement (RL) à l'analyse dynamique, visant à réduire le fardeau de l'exploration exhaustive des branchements conditionnels, et à développer un modèle capable de remarquer de nouveaux schémas d'évasion en utilisant la puissance du RL. Les approches de pointe sur l'état de l'art identifient généralement les évasions par *fingerprinting*, ce qui n'est pas efficace pour identifier de légères mutations dans les schémas d'évasion. Par contre, ce travail utilise un modèle de langage pour faire abstraction de la syntaxe du code binaire, tout en préservant sa sémantique.

Cependant, l'usage de l'approximation de fonction sur RL, en comparant avec l'apprentissage supervisé, est affecté par des sources supplémentaires d'instabilité pendant l'entraînement car l'hypothèse d'échantillons i.i.d est souvent manqué. En plus, comme dans de nombreuses applications RL dans le monde réel, l'analyse est compliquée par la présence de récompenses éparses et d'états non-Markov. Pour prendre cela en compte, nous faisons des expériences approfondies dédiés sur la convergence vers une bonne stratégie, qui inclue trois améliorations comparées avec le modèle DQN. Les résultats montrent comment les agents RL peuvent apprendre des bonnes stratégies, tout en agissant sur plusieurs binaires en même temps, étant capables de récupérer correctement les capacités malveillantes cachées avec une amélioration comparée sur une stratégie d'exploration aléatoire.

Contents

Acknowledgements

This work would not have been possible without the support of my colleagues in the AI4Sec team, which provided my with valuable advice on the heterogeneous topic of machine learning applied to cyber security. In particular, I am grateful to Dr. Daniele Sgandurra, Haigang Zhang, Dr. Nedim Srndic and Adrian Chirita, who supported me with many brainstorming processes, helping me to iteratively refine my ideas, converging to the solution presented in this thesis. In addition, I would thank the other members of the AI4Sec team, including Dr. Claas Grohnfeldt, Michele Russo and Dr. Tu Nguyen, who helped me with useful tips for my project and gave me precious advice for my future.

From my side, I brought to the team the knowledge I acquired during my Master's studies, in particular taking part to two research-oriented semester projects supervised by Prof. Pietro Michiardi. These projects considerably improved my experience on dealing with challenging research topics, which proved to be critical for the outcome of this work. I wish to acknowledge Prof. Marios Kountouris, who supervised me during this internship and provided me with useful suggestions.

However, research is not only fueled by knowledge and good intentions, but also from well-being. In AI4Sec I found a friendly environment, which helped me to grow professionally and, above all, personally. Furthermore, I am deeply grateful to my family, including my girlfriend Elena, my parents Ezio and Tiziana, and my brothers Giacomo and Alessandro, who supported me during this challenging period, allowing me to express my full potential. A special thank to my grandma, Elvina, who indirectly fueled most of my successes with her abundant and delicious meals.

Chapter 1

Introduction

This work is the result of a six months internship done in the *AI4Sec* team of the Huawei's research center located in Munich. AI4Sec is a research team focused on the application of Machine Learning (ML) methods to cyber security, with the goal of accelerating the transition to intelligent and automated defences against cyber threats.

As the number and the complexity of modern malware samples increase, security researchers agree on the fact that one of the most promising directions to tackle this problem is to develop automated analysis tools capable of promptly blocking a malicious sample before it could create any harm. In an ideal world an automatic analyzer should produce reliable results, with low resources requirements (e.g. memory) in a short time. With state-of-the-art analysis techniques, only one or two properties are possible at the same time, suggesting that there is still room for improvement. Machine learning is likely the tool that will allow to perform a further leap forward towards superior automated analysis tools. In fact, recent advantages in this field suggest as ML models can sometimes surpass human-level performances on some tasks, like games. Some of these algorithms have been implemented resorting to Reinforcement Learning (RL), such as Alpha Go Zero [Sil+17], Mu Zero [Sch+20] and Agent57 [Bad+20a]. Encouraged by these promising results, AI4Sec decided to explore the application of RL to the analysis of evasive malware, starting with this internship. The ultimate goal is to replace as much as possible the human intervention in the tedious and expensive analysis of malware. However, a major flaw of automated malware analysis is their susceptibility to evasive malware, which can put countless strategies in place to hinder analysis. Evasive malware can deceive both static and dynamic analysis techniques, but the dynamic evasions are the most difficult to tackle. For this reason, the research presented in this thesis has been focused on finding an effective application of RL to dynamic evasive malware, proving that an RL agent can be successfully trained to find hidden malicious behaviors (a.k.a. capabilities) in evasive malware.

Originally, traditional malware analysis techniques relied on analyzing the static code of suspicious binary samples. Static analysis addresses the disassembled instructions from a sample, representing them at different abstraction levels, such as Control-flow Graphs, which enrich the semantics extracted by the analysis tools. Given the strong adversarial nature of the security domain, malware authors began to adopt countermeasures to static analysis which involve the obfuscation of static binary code, such as compression and encryption. These procedures can be complicated *at will* by malware authors, blocking any attempt of static analysis, in practice. However, the statically obfuscated code cannot be run, thus the original binary code is restored at runtime by an ad-hoc *deobfuscation routine*, created by the malware author. Once the routine has completed, the original malicious code can be executed.

Security researchers have proposed a large number of solutions to react to static obfuscations via dynamic analysis which, in contrast with static analysis, executes the code and gathers behavioral information at runtime. Although static code obfuscations can be tackled by dynamically executing the suspicious samples, dynamic analysis is not a silver bullet as it can only access to the instructions that are executed, introducing a trade-off among exploration of alternative execution paths and computational cost. In addition, malware authors have adapted their malicious samples to counter dynamic analysis as well. Evasive malware looks for evidence of dynamic analysis tools in the environment in which it is run, disguising its malicious behavior before it can be noticed by the analysis tools. As a consequence, dynamic analysis introduces another trade-off, among *transparency*, namely how much the analysis environment can be made similar to a victim environment, and *visibility*, proportional to how much information can be extracted from the analyzed sample. More invasive analysis approaches, such as Dynamic Binary Instrumentation (DBI), allow to extract fine-grained details on the low-level actions performed by a binary, however this introduces clear evidence of the analysis environment, which can be recognized and exploited by evasive malware. On the other hand, *bare-metal* analysis methods allow for high transparency, while being able to access only superficially the sample under analysis (e.g. analyzing data flows during execution). The ultimate challenge of modern malware analysis is to develop analysis techniques that are stealthier against increasingly sophisticate evasive malware, while providing rich information to reliably discriminate malware from goodware.

The class of malware tackled in this work is the *Trigger-based* evasive malware category, which can implement numerous strategies to evade from a dynamic analysis environment, including checking for some *artifacts*[1] left by the analysis environment or any piece of evidence that it is running on the predefined victim host, and not somewhere else. The information

[1]An artifact is an unwanted trace that testifies the presence of some analysis tool in the execution environment. For example, CPU signature and computing time overheads caused by a virtualized environment.

acquired by the environment is then checked by a condition, which is called *trigger condition*. When the trigger condition is satisfied, the hidden malicious behavior is executed, otherwise the malware will disguise its real behavior. As a consequence, it is sufficient to explore the conditional graph to find dormant malicious behaviors in trigger-based evasive malware. The conditional graph is the graph which nodes are the conditions and the edges are the two possible execution paths starting from that condition: one is executed when the condition is true and the other when the condition is false.

Example 1.1. *Trigger-based evasive malware.*
The popular WannaCry ransomware checks if the domain `www.ifferfsodp9ifjaposdfjhgosurijfaewrwergwea[.]test`, supposed to be inexistent, is reachable. The authors of this malware were aware that some analysis environments are made more *transparent* by simulating the access to the network, regardless if a domain exists or not. The authors assumed that when that domain is reachable, there is a high chance that the sample is run in an analysis environment, thus an evasion is performed. Often, the evasion consists in exiting the execution or performing some innocuous operations.

When `www.ifferfsodp9ifjaposdfjhgosurijfaewrwergwea[.]test` was registered, it became reachable by all the running instances of the malware, acting as a kill switch and stopping the attack.

State-of-the-art solutions aim at identifying evasive malware by comparing their behavior with some well-known patterns, which roughly is a rule-based approach and it can be hindered by performing some mutations in the evasion schemes. Furthermore, also goodware programs may sometimes perform some legitimate operations which can be confused as evasive by analysis tools, as explained by [Maf+21] and [Gal+22]. For instance, evasive malware can check for the size of the screen, to detect whether it is run in an analysis server where no screen is present. However, many goodware samples that use a graphical user interface may perform the same class of checks. Analyzing the checks made by a suspicious binary is not enough, but we should also consider the possible outcomes of a condition.

To solve the aforementioned problems, this work proposes to address binary code by representing it with an abstraction that preserves its semantics, while being robust to slight variations in the syntax of code. Furthermore, the false positive rate of the conditions identified as evasive can be reduced by considering also the nature (i.e. goodware vs malicious) of the code *triggered* by a suspicious condition, other than just checking the nature of a condition. To this end, this work provides an alternative to current automatic dynamic analysis techniques, based on RL, with the goal of training a RL agent to explore the conditional graph looking for hidden malicious capabilities. The agent is jointly trained to recognize the evasive conditions, hijacking the execution to counter the evasion, and to learn the features of malicious capabilities. In fact, the agent is provided with a positive reward whenever it forces the execution of a protected malicious capability.

In practice, this is implemented by equipping the RL agent with a debugger. A considerable contribution of this work consists of the implementation of a tool to allow a RL agent to interact with a debugger to dynamically interact with the analyzed binary. The interpreter I have developed allows to aggregate the information the debugger collects by stepping instruction-by-instruction on the target binary. Then, the data collected by the interpreter is pre-processed and sent to the agent as states. In addition, also the response of the agent (the action) is handled by the interpreter, which implements it as low-level debugger step operations (e.g. step through, step into).

The task of discovering hidden malicious capabilities has been formalized as a Markov Decision Problem (MDP), which has been extended to the domain of function approximation since the state space is continuous and to leverage the generalization properties offered by function approximation methods. In fact, a key improvement over the state of the art is the ability of the agent to learn a good policy that generalizes across different evasion schemes and malicious behaviors. However, as many other real-world applications of RL, I had to address different problems typical of RL approaches, including unstable off-policy learning, non-Markov states and low credit assignment. Solving a credit assignment problem means determining the contribution of each action to a final outcome, which becomes difficult to solve when the reward function is sparse. In fact, a reward is given only when a capability is found and at the end of an episode, proportionally to the number of capabilities found. This problem can be partially solved by extending the discrete reward functions to be continuous, providing the agent with some clue about its "distance" from a malicious capability. However, the results section explains how this approach may be deceiving for the agent, making learning more unstable.

The experiments have been carried out on a synthetic dataset of evasive samples, which generation has been inspired by the structure of common evasive malware, in terms of evasive checks and malicious behaviors put in place. Extensive experimentation has been performed with different combinations of binary code embedding methods, including TfIdf and DO2VecC [Che17], reward functions including discrete and continuous reward, and RL algorithms such as DQN [Mni+13], double DQN [vHGS15], prioritized DQN [Sch+16] and DQN(λ) [DA19]. The results have been compared and an ablation study has been performed, highlighting the contribute of each component.
The contribution of this work is threefold and can be summarized as:

- The study of the malware analysis domain and a comparative analysis of state-of-the-art analysis techniques, with the identification of a promising new research direction aimed at applying RL to trigger-based evasive malware.
- The development of the first method, to the best of my knowledge, to find hidden malicious capabilities in trigger-based evasive malware via RL, employing a debugger to dynamically interact with the analyzed binary. This method proved to be effective, making the agent able to recognize the right path to reach the hidden malicious capabilities, improving over the random exploration policy. The learned state-action values can be used to guide the exploration of unseen samples, reducing the cost of *exhaustive exploration*, which is often unfeasible.
- The implementation of a Python tool that makes it possible for the RL agent to interact with the GDB debugger, aggregating the information as states and implementing the high-level actions taken by the agent as low-level debugger operations. Furthermore, an additional collection of Python tools has been developed for various tasks, including the generation of a synthetic dataset of binaries emulating evasive malware, the setup of a simulated victim environment in which the evasive binaries are run, and the utilities to perform code block embedding resorting to Natural Language Processing (NLP) and training of the RL models.

This thesis is organized as the following. Chapter 2 provides an introduction to the malware analysis techniques, comparing static and dynamic analysis, underlying their benefits and drawbacks, explaining what are the possible evasions techniques that can be put in place by malware to disguise malicious behavior. This chapter also motivates the chosen research direction, focused on applying RL to trigger-based evasive malware. Then, I justify why this direction is more promising than applying RL to other malware analysis tasks, such as static code deobfuscation, API invocation deobfuscation, and fuzzing. Chapter 3 performs a comparative analysis of different block embedding methods, taken from the NLP domain, justifying the choice of TfIdf and Doc2VecC for this domain of application. Then, this chapter delineates the theory of RL, starting from the general MDP formalization until the definition of off-policy learning with function approximation methods. Eventually, this chapter describes the proposed formalization of the RL problem when applied to the task of identifying hidden malicious capabilities in evasive malware, explaining which algorithms have been implemented taking inspiration from the literature. Chapter 4 explains the experimental setting in which the experiments have been carried out and give more details about the best hyperparamters resulting from extensive search. Furthermore, this chapter outlines some interesting directions for future work, to improve the proposed solution. Chapter 5 summarizes the work and provides a general discussion of the method. Appendix A gives a detailed explanation of the implementation of the Python GDB debugger interpreter, which is one of the main contributions of this work.

Chapter 2

Evasive Malware Analysis

This chapter introduces binary analysis techniques employed by malware analysis researchers to identify whether a suspicious sample is capable of any malicious behavior. Binary analysis techniques mainly divide into static and dynamic approached, which have their own pros and cons and in general there is not a clear dominance of the one on the other. Their effectiveness is mostly task-dependent. Nevertheless, malware authors have evolved their strategies to keep up the pace of new analysis methods, devising more and more sophisticate evasive techniques that considerably increase the difficulty of analyzing modern malware samples. Evasive malware is a generic definition that involves both static and dynamic analysis tools, leveraging their weaknesses. For instance, malware performing static evasions breaks the assumptions on which static code analyzers are based, namely that the binary code is accessible and can be analyzed all at once.

The second part of this chapter compares different applications of Reinforcement Learning (RL) to four analyzed cases studies of evasive malware, including static code deobfuscation, API invocation deobfuscation, emulator evasion (e.g. virtual machine), trigger-based malware.

Eventually, I observe that tackling trigger-based evasive malware can be well defined in an RL setting, and motivate why a debugger would be the best tool to perform this task, by providing a comparative analysis with other binary analysis tools. In fact, trigger-based behavior identification is are easier to be framed as a RL problem, because the states, actions and rewards do not require advanced domain expert knowledge, giving the RL agent the possibility to learn the best policy from data.

is the most promising direction, among the explored ones, and I provide a solution

2.1 Analysis of Malware Analysis Domain

It is not straightforward to find a sensible application of Reinforcement Learning (RL) without incurring in a process of trial-and-error while exploring the malware analysis domain. In this section I will give an high-level description of different malware analysis techniques I have studied, looking for a useful application of RL. At first, I will describe the background of malware analysis techniques in Section 2.1.1. Then I will explain three main static binary obfuscation techniques in Section 2.1.2, whereas in Section 2.1.3 is explained how some malware is able to evade dynamic analysis tools, for instance by behaving innocently when an analysis environment is detected.

The Malware Analysis Project in which I am involved, aims to explore the application of RL to counter evasive techniques frequently employed by malware to bypass analysis tools. This defines mainly two directions of research which include addressing static evasions (e.g. binary code obfuscation) or dynamic evasions (e.g. sandbox / VM / debugger elusions).

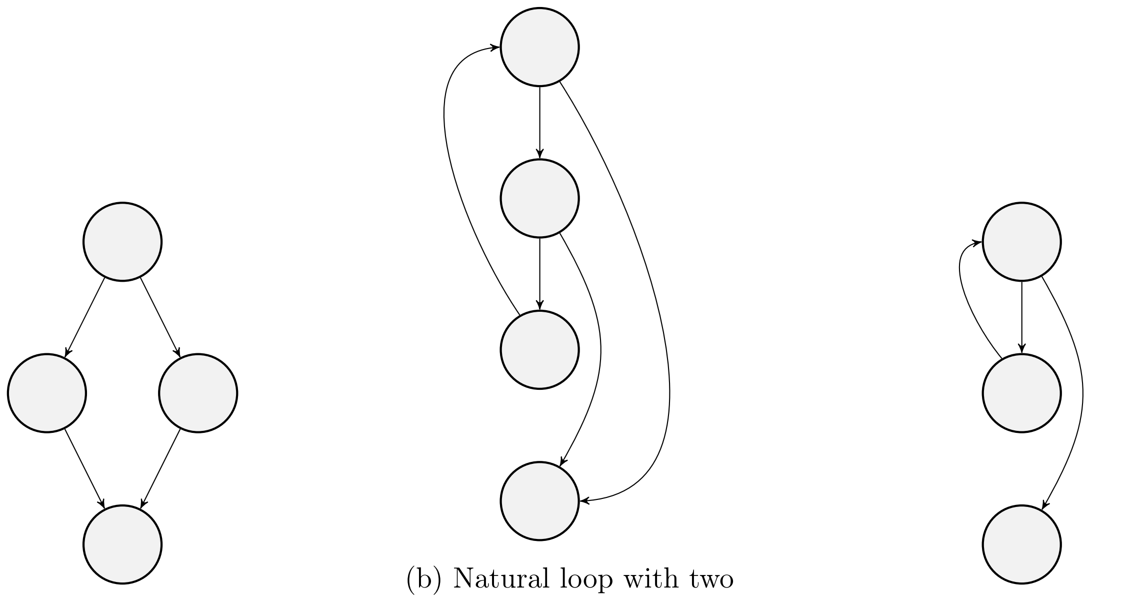
2.1.1 Background on Malware Analysis Techniques

This section introduces some key concepts from the binary analysis domain. There are two main ways of analyzing a malware: *statically* or *dynamically*. Each approach has its own advantages and disadvantages, which are described and compared below.

Static Analysis

Static analysis does not require to execute the code and assumes to have access to the disassembled binary code of a sample all at once. Two important techniques of static analysis that I have studied are Control-flow graph (CFG) analysis and Symbolic Execution. The former analyzes the conditional branches graph looking for some patterns typical of malware behavior, while the latter represents the variables as symbols and determines which values of these symbols causes a part of the program to execute, satisfying a set of conditions.

Control-flow Graph analysis extracts and analyses information from the structure of the code. In particular, a CFG is a graph whose nodes represent *basic blocks* and its directed edges represents jump instructions. A basic block is a continuous (straight-line) sequence of instructions with no branches except that at the end. A basic block is delimited by jumps or jump targets. Figure 2.1 depicts some examples of Control-flow Graphs for If-then-else CFG, a natural loop with two exists and the CFG for a while loop. The quality of the outcome of the analysis based on CFGs can be negatively influenced when the CFG is hard to extract due to its complexity or when it is obfuscated. Malware authors often take advantage of these



(a) If - then - else. (b) Natural loop with two exits. (c) While loop.

Figure 2.1: Three simple Control-flow Graphs.

weaknesses of CFG analysis. On the other hand, the advantage of this approach is that it does not need the binary to be run and can collect information also for the regions of code that are not executed.

Compared to dynamic analysis, static analysis generally requires less computational power to extract the same amount of information from a sample (e.g. percentage of code analyzed), however it can be easily fooled by making the malicious code hardly accessible, namely obfuscating it (e.g. binary code encryption / compression).

Dynamic analysis

The adoption of more and more sophisticate schemes of static binary obfuscation [Uga+15], such as packers, crypters, API invocation obfuscation and control-flow graph obfuscation has pushed security analysts to move to the more expensive dynamic analysis approaches, which solve some of the problems of static obfuscation. The importance of analyzing a sample at runtime is underlined in [Com+10], where the authors argue that integrating binary code analysis with data generated by the binary is key for a precise detection of malware behavior. For instance, if a suspicious binary sends many packets containing SMTP keywords to port 25 it is easily identifiable as a spammer, whereas it would be far harder to conclude the same by just analyzing its binary code.

A dynamic analysis environment is composed of two main components: a *runtime envi-*

ronment and an *analysis component*. The runtime environment comprises the hardware and the software that makes it possible to execute the sample under analysis, whereas the analysis component consist of all the instrumentations used to gather and analyse information from the sample. To integrate the analysis tools with the execution environment, dynamic analysis systems resort to some type of emulation or virtualization for the runtime environment. This inevitably introduces some evidence of the presence of the analysis component, reducing the *transparency* of the runtime environment. The less an analysis runtime environment is distinguishable from a normal machine, the more it is said to be *transparent*. To make the environment more transparent some works [KVK14; KVK11] propose to separate the analysis instrumentation from the execution environment. However this is subject to a trade-off: *visibility* (how much information we can gather from the execution of the sample) versus *transparency* (how well we can hide the analysis tools from the malware).

Dynamic analysis include three main approaches: black-box, debugger and Dynamic Binary Instrumentation (DBI). The main methodologies of these approaches are summarised in the following:

- **Black-box** dynamic analysis executes a binary to gather behavioral information, such as the called APIs, system calls trace and data-flow analysis. Black-box dynamic analysis does not inspect the content of a binary (e.g. code). A very popular black-box technique is *fuzzing*, which aims to explore the input space of a program to reproduce a target behavior.

- **Debuggers** are powerful tools employed by researchers and analysts to complement or replace the analysis done with automatic methods, especially with advanced malware. A debugger is more powerful than black-box dynamic analysis because it can access to the disassembled binary code and can dynamically interact with the binary under analysis.

- **DBI**. DBI is a powerful technique that enables security analysts to perform instrumentation of a target binary with the possibility of extracting information with high granularity. This consists in inserting hooks within the binary code to some user-defined analysis functions within the binary code. When a hook is reached, the hooked function is called to perform any sort of analysis or logging of the current state of execution. In some cases, DBI allows analysts to collect richer information with respect to debuggers. Recently this techinque became more popular in the research community due to its flexibility. Intel PIN is a popular DBI tool in the research community[1].

[1] https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

Previous research showed that dynamic techniques have a low code coverage, namely every time the binary is executed only the behavior of the executed code is revealed. This is a major drawback, compared to static binary analysis techniques that provide a global perspective on the whole binary sample. Some proposed solutions partially solve the problem by increasing the number of paths that are dynamically explored. This is achieved by providing different inputs to cause the execution to follow alternative paths like [Bru+08] or fuzzing [Man+19], or by forcing alternative paths. The drawback of tools that explore multiple execution paths suffer from the *path explosion* problem, as described in [Com+10].

2.1.2 Static Binary Obfuscation

The aim of this section is to present state-of-the-art static obfuscation techniques and to expose the complexity that comes form their diversity. At first, I will present the main binary code obfuscation strategies that hinder the comprehension of the code for a reverse engineer or a static analysis tool, then the class of API obfuscation techniques is introduced, eventually I describe a third class of subtle obfuscation tactics that dynamically hijack the static control-flow graph (CFG), being able to bypass static analysis tools.

Static Obfuscation of Binary Code

The following ones are a subset of the most popular **code obfuscation** techniques, discussed in [Men+21; Che+18; Van+21].

- **Packing** (or compression): likely the most popular category of obfuscators. The original malware code is compressed and restored at run time by a specific *unpacking routine* before being run. The process of packing and unpacking is made intentionally confusing, to hinder the task of a reverse engineer.
- **Encryption**: code is encrypted and the decryption key is hidden in the binary or computed at run time.
- **Mixed Boolean-Arithmetic coding (MBA)**: increase syntactic complexity of opcode while preserving its semantics. Informally, a given code block can be made far more garbled while still producing the same outputs for some inputs. This considerably slows down the work of a reverse engineer that has to spend hours to understand the code.
- **Virtualization**: translates a code portion into bytecode together with a custom virtual machine. Execution of the obfuscated code can be divided in 3 steps: (1) fetch the next

bytecode instruction to execute, (2) decode the bytecode and find the corresponding handler, (3) and finally execute the handler. Prevents CFG[2] analysis.

- **Junk code**: mix data and code together.
- **Path-based obfuscation**: takes advantage of the path explosion problem to thwart symbolic execution, massively adding additional feasible paths through dedicated encodings.

Further analysis is carried out in the works [Uga+15; RM13], explaining in more details the ways how an obfuscator can usually be implemented.

It is quite clear that there is a little possibility to generalize across different packing schemes. Some authors explored the possibility of designing an *universal unpacker* strategy, such as [Che+18; Roy+06; KPY07; MCJ07]. Nevertheless, these strategies are based on heuristics which are efficient but may be vulnerable to slight modifications of the obfuscation protocol.

API Invocation Obfuscation

When a binary program imports system APIs or other external functions, it dereferences them from a lookup table called Import Address Table (IAT), which at loading time is filled in resorting to the metadata written in a similar data structure, called Import Name Table (INT). As malware's APIs provide rich information about malicious behavior, one common anti-analysis strategy is API obfuscation, which removes the metadata of imported APIs from malware's PE header and complicates API name resolution from API callsites. At run time, the needed APIs are called relying on non-standard IATs that have to be reconstructed by an *ad-hoc* routine, before those functions are called. API obfuscation hinders both static analysis (by hiding IAT and INT) and dynamic analysis. In fact, a reverse engineer may place a hook instruction[3] at the beginning of each API and easily understand which API was about to run. As the APIs' entry point is obfuscated, there are no guarantees to hook them. Furthermore, when the API calls are obfuscated, the binary cannot even be executed in some scenarios, impeding dynamic analysis. The interested reader can find additional information in the following important works which address API obfuscation [Kaw+13; KIM18; Che+21].

Static Control-flow Graph Obfuscation

Among the numerous obfuscations schemes presented in [RM13] which can be performed on various parts of a binary (e.g. constants, function boundaries), it is important to mention the

[2]CFG: Control-flow Graph.
[3]An example of hook instruction is a break-point in a debugger.

control-flow obfuscations.

These obfuscations play a central role as they hinder static analysis, for instance the control-flow graph creation. This is achieved by malware creators by either adding paths that are never executed at runtime or by dynamically hijacking the static CFG, namely inserting control-flow transfers impossible to recognize from a *static analyzer*. The most popular obfuscations presented in [RM13; Men+21] are:

- **Indirect jumps (and calls)**: jump to an address which is computed at runtime.
- **Non-returning calls**: jump to a function that tampers with its return address to jump somewhere in the code. Never return to the caller function.
- **Exception-based control transfer**: instantiate at runtime an exception handler and then cause an exception (e.g. divide by zero) to jump to the handler instead of going on with the execution.
- **Opaque predicates**: obfuscate control flow by creating artificial conditions in programs. The conditions are traditionally tautologies and dynamic runs of the code will follow a unique path.

Dynamically, these are not a problem and the *true* control-flow graph can be created.

2.1.3 Dynamic Evasive Malware

When security researchers started proposing dynamic analysis techniques to counter statically obfuscated malware, due to the strong adversarial nature of the security domain, malware authors began to adapt their samples to evade dynamic analyzers in increasingly sophisticated ways. The recent works of [Maf+21; Gal+22] perform a longitudinal study of evasive techniques employed by malware, providing a rich and updated taxonomy of evasions. According to the referenced works, evasive malware can be divided in the following categories:

Anti debugger. Malware samples can put in place various techniques to hinder the analysis performed using debuggers. Anti-debugger techniques are carried out by looking for some hints that a debugger is attached to itself (the malicious binary), or is running on the same system, resorting to some API calls or analyzing the memory. For instance, in Windows the API `isDebuggerPresent` determines whether the calling binary is being debugged, whereas in Linux calling `ptrace` on the same process returns -1 if a debugger is attached to the calling process, achieving the same goal. As explained in [Gal+22], a debugger can be countered by examining how some exceptions are handled. A malicious sample may raise some exceptions and catch them using custom handlers defined for this purpose. However, a debugger will catch these exceptions before, preventing the custom handler from being executed. The malware perform a form of evasion if it detects that the handler was not executed or it can hide the malicious behavior in the handler itself.

Virtual machine checks. Emulation and virtualization are among the most popular setups for dynamic analysis. As for any other analysis environment, these techniques are not perfectly *transparent*, namely they introduce some evidence of their presence, which can be identified and exploited by evasive malware to conceal their real behavior. Researchers and security analysts have invested a considerable amount of time in finding these traces and providing solutions to improve transparency. However it has been shown that it is extremely hard to create a perfectly transparent virtual machine by [Gar+07]. Malware authors can gather information in several ways like checking the MAC address, checking the CPU metadata, computing time discrepancy of the execution of a block of instructions in a bare-metal machine versus in a virtualized environment.

Anti instrumentation. Dynamic Binary Instrumentation (DBI) is a major trend in recent analysis approached analyzing evasive malware, as shown by the works of [Pol+17; Gal+22; Maf+21; DEl+20]. This techinque resorts to a Just In Time (JIT) compiler that injects some analysis code in the original binary sample, including hooks to some analysis routines. When an analysis routine is called, it collects information about the sample execution. Evasive malware can compare the program counter between instrumented and normal execution in order to track the presence of the analysis tool. In other words, they can detect an anomalous number of instructions in a code block.

Timing. One approach of timing evasion is based on the knowledge that every analysis tool employing some sort of interaction with the analyzed sample (e.g. emulation) introduces some non-negligible time overhead, as argued in [Gar+07]. Malware authors can rely on this precious information and compare at runtime the discrepancy between the expected execution time and the actual one.

Another approach of timing evasion is motivated by the fact that most of automatic dynamic analysis tools run the analysis for a certain amount of time. A recent work of [Kuc+21] shows how most malware samples expose their entire behavior within two minutes of execution. A malware author can take advantage by the fact that most dynamic analysis tools may run the samples for few minutes, given the considerable cost of this analysis. Evasive malware could stale or disguise its malicious behavior for a sufficiently long time to bypass the dynamic analysis limited runtime budget.

Environment. There are some form of checks on the execution environment that evasive malware can perform in order to identify the presence of instrumentation tools. The works of [Gal+22; Coz+18] explains how some evasive malware check for the existence of file paths that contain suspicious keywords that may suggest the presence of an analysis tool (e.g. "vmware"). Similarly on Windows some evasive samples access keys inside the registry. Other checks can be performed on the file system, list of running processes, installed drivers, presence of monitor and other devices (e.g., printer, webcam, headset).

Human presence. Some malware samples counter analysis checking if they are run in an automated dynamic analysis environment. This can be done by looking for specific evidence of human presence such as the browsing history, the content of some filesystem locations (i.e. music folder), mouse movements as showed in the work of [CS20].

These are the most popular evasion techniques of interest for my work. The reader is referred to the works [Gal+22; Maf+21] and to the previous surveys of [Afi+19; BY17] for a deeper exploration of evasive techniques in malware.

Analyzing evasive malware

Security researchers agree that evasive malware checks for the nature of the environment in which is run relying on *fingerprinting* techniques, looking for some evidence left by analysis tools [DEl+20; Pal+09; LKM11; RKK07; Mir+17; Gal+22; Maf+21]. Once the analysis environment is discovered, evasive malware halts its execution or conceals its malicious behavior, performing some *seemingly-goodware* task.

To analyze evasive malware, researchers have proposed different approaches that can be grouped in two main classes, depending whether they are aimed at spotting evasive behavior or contrast it.

The approaches belonging to the first class perform automatic identification of evasive samples, for instance by comparing their behavior in different runtime environments [KVK14; KVK11]. After a sample has been detected as potentially evasive, it is sent to a more advanced (and expensive) analysis, often performed manually.

On the other hand, some approaches try to mitigate the evasive behaviors by making the runtime environment more transparent, trying to hide the analysis tools, or by providing the sample with tampered information regarding the environment (e.g. force the API `isDebuggerPresent` to return always 0). In a recent research trend the analysis of evasive malware is mostly performed resorting to Dynamic Binary Instrumentation (DBI) [DEl+19; Pol+17; DEl+20]. A main drawback of known approaches to tackle evasive samples is that there are no guarantees that a solution can detect novel malware evasive behavior, since this is an undecidable problem as stated by [Gal+22].

The recent work of [Gal+22] explains how the number of evasive samples has increased over the last 10 year and how evasive techniques have shifted in this time frame. They state that in 2010 about 70% of malware samples employed evasive techniques and their adoption has slightly increased in the last decade by about 12%. As expected, they observed that techniques have adapted to the new defences put in place by security researches, proving that malware authors are able to keep up with the advances made in the security domain. [Maf+21] performed a large-scale investigation on more than 180K samples, noticing that more than 40% of the samples collected in 2020 employed at least one evasive technique, while they were about 30% in 2016. Furthermore, they observed that in the last years the number of VM evasions has decreased, probably due to the spreading of virtualized applications that malware authors want to be able to infect. On the other hand, the number of anti-debugger evasions has increased.

Trigger-based Behavior in Malware Samples

The works of [PBM17; Bru+08] provide the definitions of trigger-based behavior and trigger condition which are the following:

Definition 2.1. *Trigger-based behavior.*
A trigger-based behavior is a behavior which is triggered depending on some event that depends on the external environment, rather than something that can be computed internally by the binary.

Definition 2.2. *Trigger condition.*
A trigger condition is usually performed on some variable that was (indirectly) influenced by some interaction with the external environment. When a trigger condition is verified, the code containing the (hidden) trigger-based behavior is executed.

Evasive malware with trigger-based behavior will perform a malicious action only when a set of trigger conditions are met, as shown in Listing A.2 with an example. In this example, the malware performs four checks on the environment in which it is run and only when all of them are satisfied the malicious action is performed. This class of malware can be considered as subclass of evasive malware as the evasive behavior is put in place specifically after a *condition* is met. On this assumption are based the works of [Bru+08; TIR+20]. In general, anti-analysis techniques employed by evasive malware are broader and can be implemented in other ways, for instance evading a debugger through exception handling, as explained in Section 2.1.3. The reader is referenced to [Gal+22; Maf+21] for a more detailed explanation.

A popular example of trigger-based malware is the one implementing Command and Control (C&C). A C&C malware infects a victim host and then waits for further instruction sent by the malware author from a remote location, usually over the network. An example of C&C malware are *backdoors* that allow the attacker to access the victim host and perform unauthorized operations. Malware implementing this functionality typically express some malicious behavior

```
char curr_time[20];
// First trigger condition
if (! is_debugger_attached()){
    // Loop until the second trigger condition is met
    do{
        get_curr_time(curr_time);
    }while(strcmp(curr_time, "11:59") != 0);
    // Third trigger condition
    if(strcmp(gethostname(), "CHOOKOO") == 0){
        // Check last evasive condition: C4
        if (are_private_files_present()){
            // Perform the malicious action
            steal_data();
        }
    }
}
else{
    // A debugger is analyzing me! Evade
    exit(0);
}
```

Listing 2.1: Example of evasive malware that checks a set of evasive conditions.

only when explicitly triggered from an external source, The malicious actions performed by a malware implementing this functionality include: data theft (e.g. sensitive data), shutdown or frequent reboot of the infected host to disrupt the services provided by the target host, download other malware.

2.2 Reinforcement Learning for Anti-Evasion Solutions

In the previous section I have explained the theoretical background also including the references to the works that addressed evasive malware with heuristics. In this section I will analyze the possibility to apply RL to address the aforementioned issues. I will compare the existing heuristic solutions in the state of the art with a potential RL implementation. In Section 2.2.1 and Section 2.2.2 I analyze the feasibility to tackle static binary code and API obfuscations with RL, respectively. Section 2.2.3 analyzes from a RL perspective the case of dynamic evasive malware, whereas in Section 2.2.4 the promising direction of trigger-based malware behavior is explored in details.

2.2.1 Binary Code De-obfuscation

At the beginning I devoted my time to learn how binary code obfuscators work, with the aim of developing an unpacking algorithm based on Reinforcement Learning. However, code obfuscation schemes may be rather different from each other, which makes it very difficult to find a common ground on which a machine learning model could generalize across. Obfuscators implement subtle tactics based on small details and require to be designed by expert security researchers. As of today, the development of code obfuscators and deobfuscators require an expensive human-in-the-loop approach. Our goal is to design a model able to *learn* and *generalize*, able to overcome this bottleneck.

After an intensive study of static obfuscation schemes, I attained the conclusion that it is not easy to train a RL model on a set of obfuscated binaries, hoping it to be able to counter future obfuscation schemes, because this may require the definition of new states, actions and rewards. In other words, the RL agent is strictly bound to the packing schemes it sees at training time with no guarantee of success on unseen ones. Furthermore, training an agent to solve new obfuscation schemes intrinsically requires to update the definition of states/actions/rewards, process which depends on domain experts supervision, which is not much different from directly looking for an heuristic to revert the obfuscation. There is no advantage of using RL, unless we can prove that this method is more efficient than an heuristic.

Indeed agent actions and states can be defined at a very low level, imitating debugger step actions (e.g. step-into, step-through), and reward can be given at the end of unpacking (success/fail), but I believe this approach shifts too much the complexity burden to the policy: obtaining good performances may be just a features engineering exercise with little scientific relevance. In fact, new obfuscation schemes may act on different parts of the binary files that were never exploited by previous obfuscators. As a consequence, a generic RL deobfuscator should consider the whole binary, which would increase the dimensionality of the problem, thus its complexity. Given my previous considerations, I decided to keep exploring the malware analysis literature to find another malware analysis task to solve with RL.

2.2.2 API Invocation De-obfuscation

Current literature addresses API obfuscation with various approaches. I performed an in-depth study of two works that employ *code tainting* [Kaw+13; KIM18] which is a dynamic approach and [Che+21], an hybrid approach (static and dynamic). All these authors provide solutions able to cope with a large amount of API obfuscations with few drawbacks.

Although in this setting it is easier to define the agent goal (i.e. positive reward when an API is correctly identified), it may easily degenerate to a supervised learning problem that has to identify which API call among a set of potential candidates is the real target, whereas the others are just there to hinder the work of a reverse engineer. In the work of [Che+21] it is proved that most API obfuscation techniques can be reverted without any need for learning or generalization.

2.2.3 Emulator Evasion

Given its inner adversarial nature, emulator evasion could be easily framed as a RL problem where an agent plays the role of a sandbox that aims to become more transparent to the malware. The other agent plays the role of the malware which in turn aims at understanding whether it is run in a simulated environment or not, hiding or not its malicious behavior. In the work of [CS20], a visualization of this framework is provided: the sandbox's agent has to imitate human-like interactions (e.g. mouse movement speed, patterns, clicks), while the malware performs a Reverse Turing Test to uncover the real nature of the environment in which it is run.

However, my team leader suggested me to explore alternative applications as many techniques to make sandboxes transparent are already well known, the problem is their implementation cost. An interesting work that could be extended with RL is the one of [Ahm+21], which aims at finding the smallest subset of countermeasures to reduce their cost, while making a sandbox transparent for a specific malware. This direction is left for future work.

2.2.4 Trigger-based Behavior

The most promising direction is addressing the issue of evasive malware that put in practice their malicious behavior only when specific conditions are met, whereas behaving as goodware otherwise. For instance, these samples can sneakily put in place simple conditions that are checked at runtime. If the malware is executed in a dynamic analysis engine, the condition would drift the execution flow away from the code implementing a malicious behavior. As a consequence, the analyzer would not be able to notice the noxious content of the analyzed binary.

Example 2.1. *Trigger-based malware example from* [Bru+08]

Let's consider a malware worm similar to MyDoom. In this example, the `ddos` action will only be activated if the call from `GetLocalTime` returns 10:06 11/9. Thus, the `ddos` action is a trigger-based behavior which will only be triggered at this specific time.

An RL agent could act on the binary code to force the execution of alternative conditional paths until the malicious behavior is reached and is properly recognized by an external malware detector. I could also relax the strong assumption that the examined binary is indeed a malware, thus stopping the exploration of alternative conditional paths when a sufficient fraction of the overall code has been explored, even if the malware detector never detected some malicious behavior. The RL agent could explore the conditional graph by tampering with the binary code instructions that control the directions taken by the execution flow. These instructions are the Conditional Jump Instructions (CJIs) and the control flow can be hijacked in a condition by interfering with its natural functioning. Section 2.2.4 further explains how this can be carried out.

An alternative approach could be fuzzing, which explores the input space of a sample in order to trigger some hidden behavior and treats the binary as a black box. It has been implemented with RL by [BGS18]. Nevertheless, I am convinced that the former solution is better than fuzzing, for the following reasons:

- Taking decisions by considering also the binary code may lead to better decisions, with respect to fuzzing that treats the binary as a black box.
- The former idea can be further extended to other binary analysis tasks that involve interaction with the binary code, that fuzzing could not address by design.
- Fuzzing is more vulnerable to time hardening (e.g. sleep(10), infinite loop), whereas an agent that operates on code may be trained to bypass them.

Conditional Jump Instructions (CJIs)

The RL agent interacts with the debugged binary, by interfering with the execution of the conditional jump instructions. These jump instructions are assembly instructions that have a syntax of the form `CJI <addr>` to perform a jump to the address location if the logical expression evaluated over the flags register is true. Usually these instructions are preceded by a comparison operation among two values, which also updates the content of the flags register. Some examples of CJIs are: `je` (jump if equal), `jne` (jump if not equal), `jle` (jump if lower or equal). The Intel instruction set includes 32 CJI.

Vulnerability	Analysis method		
	Static	Black-box dynamic	Debugger
Static CFG obfuscation (§2.1.2)	X		
Time hardening		X	
Anti-debugger/sandbox checks		X	X
Static code obfuscation (§2.1.2)	X		
Branch cardinality explosion	X		X
API obfuscation (§2.1.2)	X	X	X

Table 2.1: Analysis methods vulnerabilities to malware countermeasures.

2.3 RL-Aided Evasive Malware Analysis

In this section I present the result of the analysis carried out in the previous sections that began with the research of state-of-the-art literature of solutions to counter evasive malware until the identification of a specific analysis technique to apply to a precise problem, namely discovering hidden malicious behavior in a binary where an RL agent acts as a debugger in order to identify the conditions that trigger that behavior.

Project goal. The previous analysis underlines the increasing importance of developing new defensive solutions that should be able to cope with an increasing number of samples. A promising research direction is to address evasive samples in an automated dynamic analysis setting. The goal of this work is to develop an alternative dynamic analysis approach based on machine learning, with the aim of achieving good performances in the detection of evasive malware, while reducing the exploration burden, typical of dynamic analysis. Most state-of-the-art solutions counter evasive malware with hard-coded heuristics, which are a double-edged sword: they are efficient and effective until a (potentially simple) mitiga tion is discovered, which makes them useless. The goal of my research is to find a valuable solution based on learning, rather than heuristic, with the aim of being more robust to slight perturbations.

2.3.1 Chosen Analysis Technique

After studying which are the most popular binary analysis methods, I have summarized a comparative analysis of their drawbacks in Table 2.1. The methods analyzed are static analysis, black-box dynamic analysis and debuggers which have been introduced in Section 2.1.1.

With *time hardening* I refer to all those techniques put in place to slow down dynamic analysis, while not affecting static analysis. For instance, a malware can set up a timer and perform a malicious action after τ minutes of execution, which would require to keep busy a dynamic analyzer for at least that amount of time.

Branch cardinality explosion is a technique employed by malware creators to hinder Control-flow graph analysis by adding numerous useless conditional paths that are never visited at runtime. As a consequence static analysis is considerably slowed down or sometimes made unfeasible. Since a debugger can potentially tamper with the regular execution flow in order to explore alternative conditional paths, I consider that also debuggers, in principle, can be affected by this technique.

To sum up, taking into account the features of static and dynamic analysis and how much they are vulnerable to evasive malware countermeasures summarized in Table 2.1, I conclude that debuggers are the most interesting research direction in which I to set my work. There is an interesting work of [Pen+14] that develop a tool which puts in place a similar strategy of exploring the execution graph by forcing alternative conditional paths. Since this dynamic analysis tool was specifically developed for this task, it could be integrated with an RL agent to potentially provide superior performances, compared to a general purpose debugger. This possibility is left for future work.

Given the analysis of Section 2.2.4, I have decided to design an RL agent that acts as a debugger, with the goal of finding hidden malicious behavior in a binary by acting on the conditional paths. This choice is further motivated in the following section.

2.3.2 Chosen Family of Evasive Malware

The scope of my work is to address all the dynamic evasion techniques that are checked by a condition. Of course this does not cover the whole evasions zoo, as some of them are implemented to hinder analysis before they can be noticed, for instance the debugger evasion through exception handling (see Section 2.1.3). However, inventing new evasions that do not involve checking a condition requires malware creators greater research and implementation efforts, which they may not be willing to perform. It is well known that malware authors mostly resort to code-reuse approaches, with minor research effort, as described in [Maf+21; Com+10]. This is also valid for evasion techniques. [Gal+22] describes how the prevalence of an evasion techinque swiftly increases after it is discovered and published by security analysts. Nevertheless, once it has been discovered, a mitigation is usually proposed alongside. Therefore, in principle, the malware samples that may succeed to evade the analysis technique proposed in this work, can be restricted to the ones implementing evasions that are both unknown by the security community, and are implemented without checking a condition.

Example 2.2. *Different debugger evasion techniques.*
As explained in [Maf+21], there are different debugger evasion techniques that can be put in place by malware authors. In Windows there are some APIs that can be used to infer the presence of a debugger, like `IsDebuggerPresent` and `CheckRemoteDebuggerPresent`. They can be checked with a condition and the malware author could exit the execution when these conditions are met.

Nevertheless, there are some more sophisticated evasive approaches like using `NtSetInformationThread` with `ThreadHideFromDebugger` as a parameter. This hides from the debugger a thread that may intentionally perform some malicious actions. Since the analysis tool, in this case the debugger, is blind to this event, it cannot automatically implement a mitigation at runtime.

State of the art study of evasive malware strongly relies on Dynamic Binary Instrumentation (DBI), as explained in [Pol+17], [DEl+20], [Gal+22] and [Maf+21]. The most used tool is the Intel PIN. However, for the proof-of-concept developed in this project, I use a debugger given its reduced engineering effort, compared to DBI.

In the works of [Bru+08; PBM17] the analysis of trigger-based behavior is carried out resorting to *concolic execution*, which is a mix of *symbolic* execution (static code analysis) and *concrete* execution (dynamic approach). Symbolic execution represents variables as symbols and when a conditional branch is encountered, symbolic execution splits in two instances: in one instance the symbols are constrained to interval of values that satisfy the condition, and vice versa for the other instance. The goal of symbolic execution is to associate a set of symbols' values to each execution path. This is achieved resorting to a solver, which solves the path conditions and generates concrete input values for the symbols. However, symbolic execution has some limitations including conditional paths explosion, conditions that cannot be solved statically as they require information computed at runtime, constrained computational resources that in some cases limit a thorough exploration of the conditional paths, and the dependence on external code such as libraries. These limitations ca be partially addressed by equipping a symbolic executor with a concrete execution tool, able to tackle some unsolvable conditions and bypass the need to execute symbolically also external code.

Alternatively, the work of [Com+10] trades code coverage for reduced computational cost. They propose a solution to avoid path explosion that consists in executing a sample in a sandbox and collecting behavioral information that includes API calls and data-flow analysis. The binary code responsible for a security related behavior, called *genotype*, is then identified and collected in a database. Dormant behavior is then recognized in test samples in a rule-based fashion, looking for matches in the genotypes database. This approach does not take into account anti-sandbox evasion techniques during the dynamic analysis phase and overlooks the considerable percentage of samples using static code obfuscation, which would hinder the matching of genotype code in the target sample. However it provides interesting insight on the importance of considering data-flow analysis when learning malware behavior.

Identifying Malicious Trigger-based Behavior

In the works of [Bru+08; PBM17; TIR+20], the search for trigger-based behavior is performed starting from some *candidate* locations. [Bru+08] and [TIR+20] identify some specific API calls, system events, network events and other custom-defined events as candidates, whereas [PBM17] more broadly defines a candidate as any input from the runtime environment. After a candidate has been identified, mixed symbolic and concrete execution are performed to determine whether the candidate is responsible for triggering some behavior.

As a result of my research on trigger-based evasive malware, I conclude that an trigger-based evasive malware sample is characterized by the combination of the following factors:

1. The sample performs some suspicious operations, such as collecting specific information on the runtime environment (e.g., `isDebuggerPresent`, `getDateTime`). This constitutes a *candidate* interaction.

2. The outcome of this operation intervenes in a trigger condition.

3. The trigger condition, when satisfied, leads to a block of code that exposes malware behavior, taking inspiration from [Com+10].

As shown in [Gal+22] and [Maf+21], there is an overlap among the information that goodware and evasive malware samples gather from the runtime environment. This is justified in some cases where goodware put in place evasive techniques to protect intellectual property. Goodware may be confused with evasive malware when solely taking into account the information it collects form the environment in which it is run. For instance, some goodware samples are legitimately interested in performing CPU fingerprinting, checking for human presence (e.g. getting the mouse location), or monitoring the list of running processes. This underlines how taking into account only candidate interactions may lead to false positives. Analyzing also the behavior triggered by the candidate interaction would provide richer information to better discriminate trigger-based evasive malware from goodware. To this end, my proposed solution extends previous approaches by including additional information from the behavior of binary code, in order to identify with more accuracy trigger conditions that conceal malicious behavior.

Chapter 3

Reinforcement Learning Problem Definition

This chapter addresses the application of Reinforcement Learning (RL) to problem of finding hidden malicious capability in evasive malware binaries. At first I explain how the prevalence of RL solutions in the cyber security domain is rather low, suggesting a potential intrinsic difficulty of applying this technique, However, this also opens the possibility to advance the current state of the art.

The task of finding dormant malicious behavior is carried out by the RL agent by exploring the conditional graph and forcing the execution to take alternative conditional paths, when needed. To provide the agent with enough information regarding the code is it exploring, Section 3.2 provides a description concerning how the disassembled assembly instructions can be represented in a numerical format (i.e. embedding). This task can be achieved with different strategies, each one with its own benefits and drawbacks. Thus, a comparative analysis of different embedding methods is provided to justify the chosen embedding algorithm.

The RL problem is at first formalized starting from a more abstract definition, the Markov Decision Problem (MDP), which is further extended to sample methods such as Monte Carlo and Temporal-Differences, which allow to approximate a MDP when the environment dynamics are not known, but can be simulated by interacting with the environment. These method can be applied to the case of large or continuous states spaces resorting to function approximation and learning can be improved using different heuristics available in literature, depending on the task to solve. In the last part of this chapter I provide a definition of an RL task aimed at solving the problem of evasive malware, which may degenerate to an *hard-exploration* problem. To improve learning, I take inspiration from works in the state of the art that previously addressed similar ill-conditioned problems.

3.1 Reinforcement Learning for Cyber Security

In this section I will give a broad overview of current implementations of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) in the cyber security domain. Despite the goals of the Malware Analysis Project are focused on a specific task, researching the state-of-the-art literature helped me to gather useful insights on how Markov Decision Processes (MDPs) are implemented to solve cyber security tasks.

Current approaches in the malware analysis setting often employ an RL agent as an adversary, training it against a specific detector. This is useful to find some blind spots of analysis tools. However, to the best of my knowledge there is no study in literature which proves that the improved defence systems are actually better to counter human adversaries.

The survey of [NR20] gives a broad overview of some directions that can be followed, including DRL for cyber-physical systems, intrusion detection systems like [Xu10] and the broader DRL-based game theory for cyber security. However, the proposed algorithms do not directly tackle the task of malware analysis. In the works of [Fan+19; And+18; ZLC20] an agent is trained to perform some modifications to malware samples to evade a static malware detector, for instance MalConv of [Raf+17]. The authors argue that this approach may underline some blind spots of malware detectors, with the goal of improving them.

The scarce literature in this field may suggest an intrinsic difficulty of developing valuable RL solutions, but at the same time it is a promising research direction as important breakthroughs may be achieved.

3.2 Code block embedding

The goal of the RL agent is to find hidden malicious behavior in a suspicious binary, by exploring alternative conditional paths at runtime. To this end, the agent is equipped with a debugger. which collects information about the analyzed binary by stepping instruction-by-instruction, and every time a condition is met, the agent can decide to hijack it, forcing the execution flow to take an alternative path. The RL agent receives as state the embedding of the current code block, delimited by the previous condition and the current one. Some additional features can be added to the state vector, but this is out of the scope of this discussion. An example of code block of disassembled binary instructions (opcodes) is shown in Listing A.1.

Since both opcodes and code blocks are categorical data, they have to be transformed in some numerical representation before being fed to a machine learning model, since it can only work with numbers. The process of transforming categorical data to numerical data can be achieved by means of *embedding*, which in this case consists of finding a representation for code

```
0x000000000000118d <main+68>:    je     0x11ac <main+99>
0x000000000000118f <main+70>:    movl   $0x0,-0xc(%rbp)
0x0000000000001196 <main+77>:    lea    0xe6d(%rip),%rdi
0x000000000000119d <main+84>:    call   0x1050 <puts@plt>
0x00000000000011a2 <main+89>:    addl   $0x1,-0xc(%rbp)
0x00000000000011a6 <main+93>:    cmpl   $0x2,-0xc(%rbp)
0x00000000000011aa <main+97>:    jle    0x1196 <main+77>
```

Listing 3.1: The *current* condition is at the address `<main+97>`. The instructions block that constitutes the current state is defined by the range `[<main+68>, <main+97>]`.

blocks as vectors of real numbers that preserve as much semantic information as possible about the embedded block. Making a parallelism with natural language processing (NLP), opcodes are equivalent to words and code blocks are equivalent to paragraphs or documents. Therefore, code block embedding can be performed by applying document embedding algorithms developed for NLP tasks. Finding a meaningful embedding for code blocks means finding an embedding space, such that semantically related blocks of code share some geometric properties. For instance, embeddings of similar basic blocks should be close each others whereas different ones should be far apart.

Subsequently, I will explain how different NLP techniques can be applied in this domain to address the problem of code block embedding.

3.2.1 Word embedding

This section gives an overview of the most popular word embedding algorithms in the NLP field like Word2Vec, GloVe, ELMo and BERT. This introduction lays the groundwork for explaining the intuition behind state-of-the-art document embedding models, as most of them rely on word embeddings. These methods are based on the *distributional hypothesis*, which is derived from the semantic theory of language and states that words that occur in the same context tend to share similar meanings. This concept can be extended to assembly opcodes as their order defines a specific behavior, and different permutations of the same opcodes can generate different outcomes.

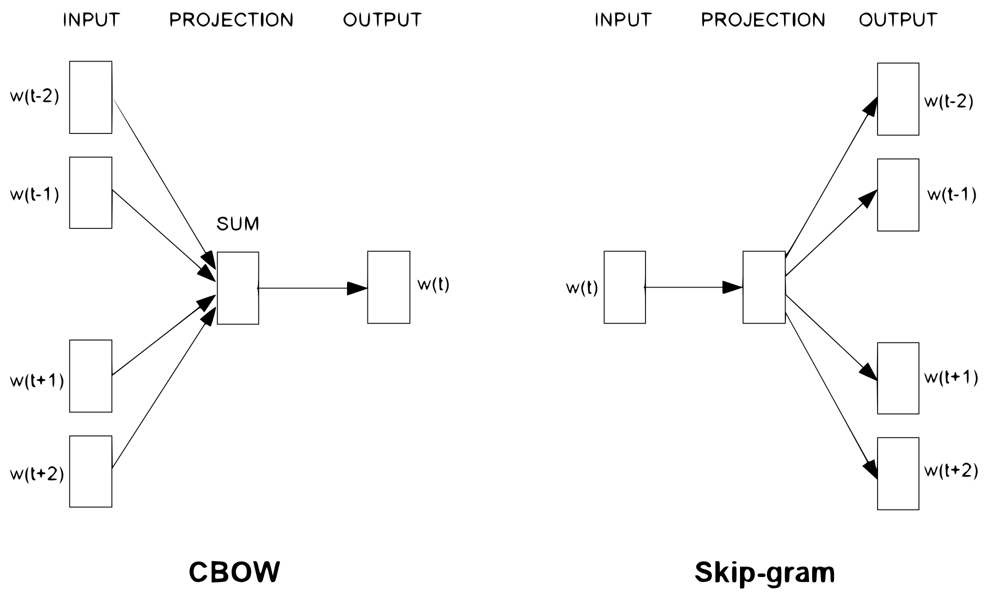


Figure 3.1: Alternatives of solving the self-supervised task of Word2Vec. The CBOW architecture predicts a target word given its contextual words, whereas the Skip-gram model predicts the contextual words given a target. Picture credits: [Mik+13].

Word2Vec

Wod2Vec [Mik+13] is a machine learning approach that learns an embedding space for words by solving a self-supervised task. Self-supervised learning can be placed between unsupervised and supervised learning, as it optimizes a supervised learning objective where the labels are inferred by the structure of data, without requiring human labeling. The self-supervised task solved by Word2Vec is based on finding the embedding vector that better includes the relations among contextual words, namely the words that appear within a sliding window on the text. In practice, this is achieved by either predicting a target word starting from the embeddings of its neighbouring words or, vice versa, the model can be trained to predict the contextual words given the embedding of the target word. The former is called *contextual bag of words (CBOW)* model, whereas the latter is the *Skip-gram* method. Both methods are depicted in Figure 3.1.

Word2Vec is implemented with an encoder-decoder structure obtained by stacking two fully connected layers. The encoder projects the *one-hot* encoding of each word to a lower-dimensional embedding space and the decoder maps the embedding vector to a vector space of the same dimension of the one-hot encoding. Empirical evidence suggests that Skip-gram produces better representations for rare words, whereas the CBOW method generates slightly better embeddings for frequent words and is faster to train. An interesting (and desirable) feature of the embedding space generated by these methods is that contextual ($\approx$ similar) words are associated with embedding vectors close each others, whereas semantically unrelated words are be embedded far apart. Word2Vec is a simple architecture which provides a valuable baseline, even though modern NLP has shifted to more powerful approaches as explained in the next section.

Other models

Given the broad zoo of NLP models, I thereafter mention only a sub-sample of the most interesting ones, comparing them with Word2Vec.

GloVe [PSM14] is a direct competitor of Word2Vec as it also provides a lightweight solution to perform word embedding. Extensive experimentation presented in many works in literature evidence how Word2Vec and GloVe usually attain similar performances, and the dominance of one over the other is task-dependent. The main difference of GloVe with respect to Word2Vec is the self-supervised task it solves. In fact, Word2Vec only takes into account contextual words, whereas GloVe generates word embeddings considering the *global* context of each word. This is mathematically represented as the probability of co-occurrence of two words in the same document.

ELMo [Pet+18] solves a major flaw of the methods presented so far, which associate each word with the same embedding, regardless of the context. This is a sub-optimal approach, since natural language is ambiguous and allows *polysemy*, namely the same word can have different meanings depending on the context. ELMo solves this problem by generating for the same word an *ad-hoc* embedding for each context. In practice this is implemented by stacking two bi-directional recurrent neural networks (LSTM). Recurrent architectures suffer from long-term memory degradation and require complex training procedures that slow down learning. Nevertheless, ELMo considerably improved state-of-the-art performances on numerous NLP tasks, with respect to its predecessors.

BERT [Dev+19] further improves the ELMo architecture by putting in place two major improvements inherited by transformer models introduced by [Vas+17]. First, word embeddings are processed in parallel, rather than sequentially, how recurrent architectures do. Second, it employs the *attention* mechanism, which follows the same intuition used by ELMo which consists of generating different embeddings for a word depending on its context. BERT is pretrained in a similar way as Word2Vec CBOW, namely predicting a subset of masked words in a document. This language model has a considerably large amount of parameters which vastly outnumber the parameters used by Wor2Vec or GloVe models. As a result, it requires a large amount of training data and computational resources to produce the remarkable results presented in its paper. Since the first version of Word2Vec in 2013, language models have grown in number of parameters, which made them better at solving various NLP tasks, at the cost of requiring larger and larger training datasets, entailing more computational cost.

3.2.2 Document embedding

Finding a proper embedding vector for a word is a well-defined task, regardless if that vector is independent from the context (Word2Vec, GloVe) or not (ELMo and BERT). In fact, all the word embedding approaches presented in the previous section are trained by solving a self-supervised task that depends on either the ordering of words (Word2Vec, ELMo) or their co-occurrence in a document (GloVe, BERT), which is a sensible formulation under the *distributional hypothesis*. Nevertheless, the concepts of *context* and *ordering* are not easily transferable to documents. A document is a sorted collection of words and changing their ordering would alter its meaning. Not the same is true for a *corpus*, namely a collection of documents.

Tf-Idf is a popular baseline for non-parametric document embedding. It associates each document with a vector having size equal to the number of unique words in the corpus of documents. For instance, if there are 100K unique words, each document will be represented with a 100K-dimensional vector. Each component represents a word and its value is is the sum of two quantities: the term frequency (Tf) and the inverse document frequency (Idf). The term frequency is bounded in $[0, 1]$ and represents the proportion of the occurrences of a word in a document, thus it is higher for frequent terms within a document. The Idf term is inversely proportional to the prevalence of a word across the documents of the corpus. The more a word appears in different documents, the lower is its inverse document frequency. For short, Tf-Idf gives larger weight to words that are common in a document, while being rare in the corpus, because these are the ones which more likely represent the meaning of the document. A major drawback of Tf-Idf is the large dimensionality the embedding vectors it produces, which grows with the vocabulary size. However, this problem can be solved by resorting to some unsupervised dimensionality approach like Principal Component Analysis (PCA), which finds a linear transformation to a lower-dimensional vector space by minimizing the L2 norm of the reconstruction error.

Doc2Vec [LM14] is a technique developed by the authors of Word2Vec which represents documents with one-hot encoded vectors and learns a meaningful vector representation for them. However, this strategy implies that at test time a new representation has to be learned for unseen documents, worsening its inference time. In addition, the number of parameters of the model depends on the dataset size. An interesting improvement of this early work is **Doc2VecC** [Che17], which improves Doc2Vec in two ways. First, it trains a model similar

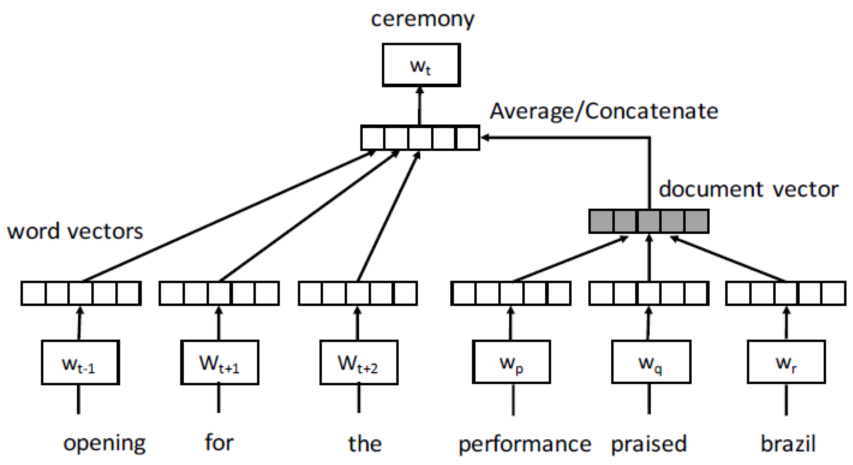


Figure 3.2: Doc2VecC architecture where two input streams contribute to the prediction of the target word: one comes from the embeddings of words sampled in the local context of the target word, whereas the other is the average of a subset of words sampled from the whole document (global context). Credits: [Che17].

to Word2Vec CBOW in which a second input stream is added besides the one dedicated to contextual words. Doc2VecC predicts a target word using the embeddings of some words sampled from its *local* context, and the average of the embeddings of other words sampled from the *global* context of the target word. The global context is the whole document. Second, it introduces a data-dependent regularization by defining a corruption model. The authors argue that downsampling a portion of terms from the documents favors rare or informative words, while reducing the importance of common uninformative terms. Furthermore, this improves training performances. As a result, the word embeddings produces by this model can be used to represent a document as the average of the embedding vectors of the words that compose that document, while still preserving its semantics. The structure of Doc2VecC model is shown in Figure 3.2.

Other approaches propose to generate an document embedding vector starting from the word embeddings of the terms belonging to that document. This approach involves dealing with an input size that depends on the number of terms in a document. Most machine learning models struggle to handle input data of variable size. The simplest approach would be summing or averaging the word embeddings obtained with some method presented in the previous section, however some works like [ALM17] proposed a way to learn the coefficients of a linear combination of word embeddings to preserve as much semantics as possible. Other works propose to use recurrent architectures, such as RNN and LSTM, which have the drawback of long-term memory degradation and slow training procedures. Furthermore, the authors of Doc2VecC compare it to some recurrent architectures, proving its superior performances. Transformer-based models like BERT, produce document embeddings that can be used by downstream machine learning tasks and are currently the state of the art for NLP tasks.

3.2.3 Chosen techniques

In this work, NLP methods for document embedding are applied to sequences of opcodes instructions extracted from dynamically disassembled binary code. Opcode instructions (e.g. `mov, add, jle, push, ret`) are treated as words and the blocks in which they are arranged can be considered as documents.

Given the real-time application and the small-sized dataset available, employing a large model (i.e. with many trainable parameters) would introduce a non-negligible time overhead every time a new code block has to be converted to its embedding and it would be subject to the risk of overfitting. Furthermore, this work is mostly dedicated to delivering a proof of concept to test whether an RL agent may be capable of successfully finding hidden malicious capabilities on some simplistic evasive samples. Assembly language is far less ambiguous than natural language, preventing phenomena like polysemy, which considerably reduces the need of complex language models to represent opcodes. Therefore, I exclude large models like BERT or ELMo, which application is left for future work.

Tf-Idf has no learnable parameters and if used together with a dimensionality reduction like PCA, the number of learnable parameters is less than the ones used by Word2Vec or GloVe methods. The assembly instruction set has a cardinality of few thousands instructions, which is considerably less than the vocabulary size of a natural language (e.g. English has hundreds of thousands words). This reduces the dimension of document embeddings produced by Tf-Idf methods applied to opcodes, compared to the NLP domain. However few thousands of dimensions require the embeddings vectors to be projected to a lower dimensional subspace to prevent the *curse of dimensionality.* This is obtained by resorting to PCA which consists of multiplying the document vector $\mathbf{x} \in \mathbb{R}^F$ produced by Tf-Idf of size F by a projection matrix $W \in \mathbb{R}^{F \times E}$ of size $F \times E$ ($F \gg E$) obtaining $\mathbf{e} = W^T\mathbf{x} \in \mathbb{R}^E$, with a computational cost of $\mathcal{O}(FE)$.

Doc2VecC is an embedding model capable of generating meaningful document embeddings, while requiring a low computational cost, which translates in low inference time. These two features make this model the most preferable for this real-time task, compared to the other solutions presented above. In fact, it generates regularized word embeddings that can be safely averaged while preserving the semantics of a document. Moreover, the computational cost required to generate document embeddings, starting from pretrained word embeddings, is reduced.

Since Tf-Idf is a classic and reliable baseline for NLP tasks, and Doc2VecC has some nice features, as presented above, both embedding methods will be compared in the task of code block embedding in the experiments.

3.3 Reinforcement Learning research

This section presents the theoretical formulation of the Markov Decision Problem, followed by its extensions to real-world domains where the environment dynamics are not known analytically, but can be simulated by interacting with the environment. These methods can be further extended with function approximation in the case of large or continuous states spaces, while learning speed can be improved by learning from a model of the environment, when available. Eventually, the notions previously introduced are applied to the task of revealing dormant malicious behavior in malware samples, taking into account the pitfalls of this domain of application and relating them with available solutions in the state-of-the-art literature.

3.3.1 Markov Decision Problem

The goal of this work is to train an RL agent to find hidden malicious behavior in evasive malware, by exploring the conditional path of a suspicious binary. This task is formalized taking inspiration from the Markov Decision Process (MDP), which is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p(s', r|s, a))$, which represents respectively the sets of states, actions, rewards and the *environment dynamics* $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \times \mathcal{R} \to [0, 1]$. At each time step, the agent receives the current state of the environment $s \in \mathcal{S}$ and takes an action $a \in \mathcal{A}$ according to a policy $\pi(a|s)$. As a result, the environment performs a *transition* to state $s' \in \mathcal{S}$, which generates a reward $r \in \mathcal{R}$. An MDP is finite if the sets $\mathcal{S}, \mathcal{A}, \mathcal{R}$ have a finite number of elements. The environment dynamics are enough to fully describe the environment and are defined as

$$p(s', r \mid s, a) \doteq \mathbb{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)$$

where $S_t, R_t, S_{t-1}, A_{t-1}$ are capitalized because they represent random variables.

In the ideal case, the environment dynamics are known and can be used to analytically describe the environment. For instance, it is possible to derive the *state-transition probability*

$$p(s' \mid s, a) \doteq \mathbb{P}(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{3.1}$$

A set of subsequent transitions defines a *trajectory*:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots S_T$$

where S_T is the terminal state and if it exists and can be always reached with a finite set of transitions, the RL task is said to be *episodic*. In an episodic task, each episode is independent form the others. An example of episodic tasks are the games like Chess and Go, where each player will eventually either win or lose and at the board will be reset at the beginning of each new game. Conversely, if a terminal state is never reached, the RL problem is a *continuing* task. An example is a thermostat that controls the heating system of a house, in order to keep the temperature stable in a fixed range.

Equation 3.1 implicitly states that only the immediately preceding state S_{t-1} and action A_{t-1} are needed to fully characterize the transition probability to state S_t. In other words, S_t is said to be *conditional independent* to the history preceding S_{t-1} and A_{t-1}, given S_{t-1} and A_{t-1}:

$$\mathbb{P}(S_t \mid S_{t-1}, A_{t-1}) = \mathbb{P}(S_t \mid S_0, A_0, S_1, A_1, \dots S_{t-1}, A_{t-1})$$

Therefore, in MDP the states are assumed to satisfy the *Markov property*, which means that each state stores enough information about all the past interactions of the agent with the environment.

The goal of the agent is to find a policy $\pi(a|s) : \mathcal{S} \times \mathcal{A} \to [0, 1]$, defining the probability of taking action a when in state s, with the goal of maximizing the *return*. The return represents the total future reward starting from state s and taking action a and is defined as:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T \tag{3.2}$$

$$= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{3.3}$$

$$= R_{t+1} + \gamma G_{t+1} \quad \text{(recursive definition)} \tag{3.4}$$

where $\gamma \in [0, 1]$ is the *discount factor* and has a double interpretation. From a mathematical point of view, it guarantees that the sum in Equation 3.2 is always finite, even in case of a continuing task where $T = \infty$. If $\gamma \in [0, 1)$:

$$\sum_{k=0}^{\infty} \gamma^k R_k \leq \sum_{k=0}^{\infty} \gamma^k R_{max} = R_{max} \sum_{k=0}^{\infty} \gamma^k = R_{max} \frac{1}{1-\gamma} < \infty$$

On the other hand, the discount factor gives an exponentially decaying weight to future rewards. As $\gamma \to 0$, the learned policy will privilege short-term rather than long-term rewards, which in some applications may produce a short-sighted agent.

The expected return starting from state s under a policy π is defined by the *state value function*:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \mid S_t = s \right], \forall\, s \in \mathcal{S} \tag{3.5}$$

A similar formulation is given for the *state-action value function*, which defines the expected return under a policy π, starting from state s and taking action a:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \mid S_t = s, A_t = a\right], \forall\ s \in \mathcal{S}, a \in \mathcal{A} \tag{3.6}$$

When the environment dynamics $p(s', r \mid s, a)$ are known, the above functions can be extended to derive the Bellman equations for $v_\pi(s)$ and $q_\pi(s,a)$:

$$\begin{align} v_\pi(s) &= \mathbb{E}_{a\sim\pi(a|s)} \left[\sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]\right], \forall\ s \in \mathcal{S} \tag{3.7}\\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \tag{3.8}\end{align}$$

$$\begin{align} q_\pi(s,a) &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a'|s) q_\pi(s', a')\right], \forall\ s \in \mathcal{S}, a \in \mathcal{A} \tag{3.9}\\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \tag{3.10}\end{align}$$

According to these equations, the value of a state (or state-action couple) can be recursively obtained by the value of the next states. Value functions define a partial ordering among policies, such that $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)\ \forall\ s \in \mathcal{S}$. Thus, for the optimal policy π^* we have that $v_{\pi^*}(s) \geq v_\pi(s)\ \forall\ \pi$. To simplify the notation, from now on $v_{\pi^*}(s) = v_*(s)$.

The form of the optimal value function under the optimal policy is defined by the *Bellman optimality equation*, which substitutes the expectation with the max operator

$$\begin{align} v_*(s) &\doteq \max_\pi v_\pi \tag{3.11}\\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right], \forall\ s \in \mathcal{S} \tag{3.12}\end{align}$$

which again has a recursive formulation. When the MPD is finite and the environment dynamics are known, this equation defines a linear system of $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns. Once $v_*(s)$ is known, the optimal policy $\pi^*(a|s)$ can be easily obtained by attributing non-zero probability to the actions that maximize the Bellman optimality condition, for each state $s \in \mathcal{S}$. If all the probability mass is always given to an action, the policy is deterministic and is said to be *greedy*. A greedy optimal policy always exists.

Similarly, the optimality equation can be defined also for the state-action value function:

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right], \forall\ s \in \mathcal{S}, a \in \mathcal{A} \tag{3.13}$$

In this case, obtaining the optimal greedy policy is even easier with respect to the case of $v_*(s)$:

$$\pi^*(s) = \arg\max_a q_*(s, a) \tag{3.14}$$

This approach trades computational complexity for memory usage. In fact, determining the optimal action for each state just needs a sweep of the action values for that state, thus requiring to store a value for each state-action pair.

The Bellman optimality equation can be solved iteratively, resorting to *dynamic programming*. This approach includes two alternatives: *policy iteration* and *value iteration*. The first alternates steps of policy evaluation and policy improvement, whereas the latter directly solves the Bellman optimality equation for state values. However, dynamic programming applications are restricted to finite MDP problems (with a limited number of states) where the environment dynamics $p(s', r \mid s, a)$ are known. Since this is out of the scope of this work, the reader is referred to [SB18] for a detailed explanation of these methods.

3.3.2 Extension of MDP

In real-world applications, it is rather uncommon to know the environment dynamics, which makes impossible to solve analytically the Bellman optimality equation and compute the optimal policy. However, the value functions can be *learned* simulating the dynamics by directly interacting with the environment.

Monte Carlo Methods

Monte Carlo (MC) methods approximate the value function, which represents the expected return, by taking the average of the returns produced by many trajectories starting from each state or state-action pair. Given a trajectory, the return for each state or state-action pair is obtained by unrolling the trajectory from the terminal state, backward propagating the discounted cumulative reward. Given a policy π, the corresponding value function is computed as shown in Algorithm 1.

Algorithm 1 MC for estimating $V \approx v_\pi$

```
Require: Policy $\pi$
  $Returns(s) \leftarrow$ an empty list for all $s \in \mathcal{S}$
  loop for each episode
      Generate episode trajectory under $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots S_{T-1}, A_{T-1}, R_T$
      $G \leftarrow 0$
      for $t = T-1, T-2, \dots 0$ do                ▷ Unrolling the trajectory from the end.
          $G \leftarrow R_{t+1} + \gamma G$
          Append $G$ to $Returns(S_t)$
          $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

However, if π is a greedy policy, namely a policy that always chooses the same action in each state, the trajectories generated by π will cover only a subset of state-action pairs, causing a poor estimate of the value function. Therefore, it is important to *maintain exploration*. This can be achieved for instance with *exploring starts*, which initializes each episode from a state-action couple sampled uniformly among all the possible pairs, or with *ϵ-greedy policies*. An ϵ-greedy policy is defined as:

$$\pi_\epsilon(a|S_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}|, & \text{if } a = A^* \\ \epsilon/|\mathcal{A}|, & \text{otherwise} \end{cases}$$

where A^* is the greedy action and $\epsilon \in [0, 1]$. The larger ϵ, the grater the entropy of π, thus the more π has an exploratory tendency. Since π is used to interact with the environment by generating a trajectory, ϵ introduces a trade-off among *exploitation*, namely taking the action that is expected to maximize the return, and *exploration*, which allows to collect more information on the environment and produce a better estimate of the overall value function.

Algorithm 1 can be extended to approximate the state-action value function $Q(s, a) \approx q_\pi(s, a)$. This brings a key advantage: as shown by Equation 3.14, the current policy π can be improved by taking the greedy action that maximizes the value function, without the need of knowing the environment dynamics $p(s', r|s, a)$. This process can be repeated until π approximates the optimal policy π^*. The process of improving the current policy is called *control*. The MC control to improve an ϵ-greedy policy is shown in Algorithm 2.

Algorithm 2 MC control to estimate $\pi \approx \pi^*$ with ϵ-greedy policies

```
Require: $\epsilon$-greedy policy $\pi$
  $Returns(s, a) \leftarrow$ an empty list for all $s \in \mathcal{S}, a \in \mathcal{A}$
  loop for each episode
      Generate episode trajectory under $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots S_{T-1}, A_{T-1}, R_T$
      $G \leftarrow 0$
      for $t = T-1, T-2, \dots 0$ do                ▷ Unrolling the trajectory from the end.
          $G \leftarrow R_{t+1} + \gamma G$
          Append $G$ to $Returns(S_t, A_t)$
          $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
          $A^* \leftarrow \arg\max_a Q(S_t, a)$                ▷ Improve if possible.
          for all $a \in \mathcal{A}(S_t)$ do
              $\pi(a|S_t) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)|, & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)|, & \text{otherwise} \end{cases}$
```

Although ϵ-greedy polices allow to better estimate the value function, compared to greedy policies, approximating the optimal policy π^* with an ϵ-greedy policy may lead to a sub-optimal estimate, as π^* is not guaranteed to have such form, for any ϵ. There is however a way to get the best from exploration and exploitation by using two different policies: a *behavior policy* (which can be totally random) is used to generate the episodes trajectories, guaranteeing a good degree of exploration; whereas the action values are learned for a *target policy*, without constraints on its form, to better approximate the optimal policy. The method of using a policy to interact with the environment and another to learn the state-action value function ($Q(s, a)$ for instance) is called *off-policy learning*, which is different from the the *on-policy* case, where the same policy is used for both tasks, as it is done in Algorithm 2.

MC estimation of value functions has three main defects: (1) it is an *offline* method, because the value functions are updated only at the end of an episode, requiring many simulations to reach reliable estimates; (2) it is subject to relatively high variance compared to other methods, since it estimates the expected return as the (weighted) sum of the rewards obtained in future steps, which are random variables; (3) its offline nature makes it impossible to apply it to continuing tasks, namely the ones that never reach a terminal state.

Temporal-Difference Methods

An alternative to MC method is Temporal-Difference (TD) learning, which takes inspiration from Bellman equations in order to obtain sample methods, like MC, based on the interaction with the environment, while providing the advantages of *online* learning. TD(0) methods define the return with *bootstrapping*, namely approximating future rewards with the value function estimate of the successor state:

$$\begin{aligned} G_t &= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \quad \text{(Monte Carlo)} \qquad (3.15)\\ &= R_{t+1} + \gamma V(S_{t+1}) \quad \text{(Temporal-Difference)} \qquad (3.16) \end{aligned}$$

This formulation allows to obtain a class of methods that learn online, because the return at time t is known as soon as R_{t+1} is available, thus the value function can be updated at each interaction with the environment. This allows TD(0) methods to be applicable also to continuing tasks. TD(0) estimates of the value function suffer from a lower variance with respect to MC, due to the truncated chain of rewards needed to compute the return, but this comes to the cost of higher bias. In fact, at the beginning of training, the estimate of $V(S_{t+1}$ is highly inaccurate. However, the estimates are statistically consistent and the bias vanishes upon convergence. Given a trajectory of transitions, the value function for state S_t under a policy π is iteratively approximated with the following update rule

$$\begin{aligned} V_{k+1}(S_t) &= (1-\alpha_k)\, V_k(S_t) + \alpha_k\, G_t \qquad (3.17)\\ &= V_k(S_t) + \alpha_k\, [G_t - V_k(S_t)] \qquad (3.18)\\ &= V_k(S_t) + \alpha_k\, [R_{t+1} + \gamma V_k(S_{t+1}) - V_k(S_t)] \qquad (3.19)\\ &= V_k(S_t) + \alpha_k\, \delta_t \qquad (3.20) \end{aligned}$$

where α_k is the step-size of the update. TD convergence is guaranteed for any decreasing step-size α_k that satisfies the Robbins-Monro conditions:

$$\sum_{n=0}^{\infty} a_n = \infty \quad \text{and} \quad \sum_{n=0}^{\infty} a_n^2 < \infty \qquad (3.21)$$

δ_t is called *TD error*. The TD error represents the difference between the previous value of $V(S_t)$ and the new estimate of the return G_t.

Similarly to MC methods, TD learning defines control methods to find an approximation of the optimal policy, which can be obtained with either on-policy or off-policy learning. On-policy control for TD(0) methods is achieved with the *Sarsa* algorithm, whereas it's off-policy version is the popular *Q-learning* algorithm. In both methods, for each state an action can be sampled following an ϵ-greedy strategy:

$$A_t = \begin{cases} \arg\max_a Q(S_t, a), & \text{with probability } 1-\epsilon \\ a \in \mathcal{A}(S_t) & \text{with probability } \epsilon/|\mathcal{A}(S_t)| \end{cases} \qquad (3.22)$$

Sarsa update rule is:

$$Q_{k+1}(S_t, A_t) = Q_k(S_t, A_t) + \alpha_k \left[R_{t+1} + \gamma Q_k(S_{t+1}, A_{t+1}) - Q_k(S_t, A_t)\right]$$

whereas the update rule for Q-learning is:

$$Q_{k+1}(S_t, A_t) = Q_k(S_t, A_t) + \alpha_k \left[R_{t+1} + \gamma \max_a Q_k(S_{t+1}, a) - Q_k(S_t, A_t)\right]$$

the optimal greedy policy can be obtained by applying Equation 3.14 upon convergence of the method.

Although TD(0) methods solve the problem of (slow) offline learning typical of Monte Carlo methods, they also introduce some new problems that did not affect MC methods. First they are more susceptible to the violation of the Markov property (non-Markov states). Second they may suffer from the problem of slow *credit assignment* whenever the rewards are sparse in an episode's trajectory, namely when most of the rewards are zero and only some of them give the agent some useful feedback on how to improve the current policy. Both these problems are caused by the fact that TD(0) methods update their value estimates depending only on the value estimates of next state.

The credit assignment problem (CAP) consists of determining the contribute of each action to a given outcome. In RL, the term credit is a synonym of *value*. In fact, an RL agent tries to solve the CAP by associating the right expected return to each state or state-action pair, in order to improve the expected cumulative reward in the long run.

To reduce the impact of the aforementioned problems, it is possible to extend TD(0) methods by updating the value estimates considering the next n rewards, before bootstrapping to correct the estimate. This approach creates a bridge between MC and TD(0) methods and allows to define two new classes of TD methods: TD(n) and TD(λ). TD(n) defines the return from state S_t as:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{k+n-1}(S_{t+n}) \tag{3.23}$$

and the resulting update rule for the estimated value function becomes:

$$V_{k+n}(S_t) = V_{k+n-1}(S_t) + \alpha_k \left[G_{t:t+n} - V_{k+n-1}(S_t)\right], \; 0 \leq t < T \tag{3.24}$$

where T is the length of the episode. The value function is updated T times as with TD(0) and this method allows for online learning, however, the first update is made after the first n transitions have been performed. TD(n) benefits from the *error reduction property* which defines the following upper bound on the worst case estimation error:

$$\max_s \Big| \mathbb{E}_\pi[G_{t:t+n} \mid S_t = s] - v_\pi(s) \Big| \leq \gamma^n \max_s \Big| V_{k+n-1}(S_t) - v_\pi(s) \Big| \tag{3.25}$$

for any $n \geq 1$. This upper bound is controlled by n. The larger n, the smaller the estimation error. Another advantage of TD(n) methods is their ability to better propagate sparse rewards backward in time, to the preceding states in the trajectory.

Function Approximation

The methods exposed so far are called *tabular*, because they assume a *finite* MDP, thus the value functions can be stored as vectors or matrices. Despite being effective for small problems, tabular methods do not scale to problems with a large number (millions) of states and are not applicable in the case of continuous states or actions space. Furthermore, the value functions are learned *independently* for each state (state-action pair), without *generalization* across similar states (state-action pairs).

Function approximation shifts the task of learning the values for each state or state-action pair to learning a parametrized version of the value functions that minimizes a given objective. The parametric state value function is $\hat{v}(s;\mathbf{w})$ with parameter $\mathbf{w}$. Analogously, for state-action values $\hat{q}(s,a;\mathbf{w})$. The objective to minimize is the Mean Squared Error (MSE) in the approximation of $v_\pi(s)$ by $\hat{v}(s;\mathbf{w})$:

$$\overline{VE}(\mathbf{w}) \doteq \sum_s \mu(s)\Big[v_\pi(s) - \hat{v}(s;\mathbf{w})\Big]^2 \tag{3.26}$$

where $\mu(s)$ is the proportion of times state s was visited.

The objective of Equation 3.26 can be optimized resorting to Stochastic Gradient Descent (SGD) method to iteratively update the parameters $\mathbf{w}$ in the direction of steepest decrease of $\overline{VE}(\mathbf{w})$. The resulting SGD update rule is:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k - \frac{1}{2}\alpha_k \nabla_{\mathbf{w}}\Big[v_\pi(s) - \hat{v}(s;\mathbf{w})\Big]^2 \tag{3.27}$$

$$= \mathbf{w}_k + \alpha_k\Big[v_\pi(s) - \hat{v}(s;\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w}) \tag{3.28}$$

if α_k decreases over time and respects the conditions of Equation 3.21, SGD is guaranteed to converge to a local minimum of $\overline{VE}(\mathbf{w})$. Since $v_\pi(s)$ is not known in practice, it is substituted with an unbiased estimator of it, U_t, such that $\mathbb{E}[U_t|S_t = s] = v_\pi(s)$. Under this assumption, the following update rule converges with the same guarantees of 3.28:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k\Big[U_t - \hat{v}(s;\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w}) \tag{3.29}$$

Function approximation can be applied to Monte Carlo methods by setting $U_t = G_t$, which is an unbiased estimate of $v_\pi(s)$. On the other hand, TD methods require an additional discussion because their extension with function approximation is not as straightforward as for

MC methods. TD methods compute G_t by considering the sequence of successive rewards up to step n, approximating the next ones with bootstrapping:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{k+n-1}(S_{t+n})$$

for any $n \geq 1$. Due to bootstrapping, $G_{t:t+n}$ is no more an unbiased estimate of $v_\pi(s)$ because also the target U_t of Equation 3.29 depends on the current value of the parameters $\mathbf{w}$. The step form Equation 3.27 to Equation 3.28 relied on the fact that U_t does not depend on the learnable parameters $\mathbf{w}$. As a consequence, when substituting U_t with the TD target in Equation 3.29, the update rule is called *semi-gradient* method, since it is not using the real gradient information.

The above equations can be easily extended to the case of learning the parameters of $\hat{q}(s, a; \mathbf{w})$. This allows to find the approximate optimal policy by solving a control task with the state-action value function.

Function approximation methods are often preferable over tabular methods even for low dimensional MDP problems, given their ability to *generalize* across similar states or state-action values. In fact, function approximation methods learn a parametric function that will associate similar states to similar values, without the typical tendency of tabular methods to estimate the value of each state independently.

TD(λ) methods

As previously explained, TD(0) methods can be extended to TD(n) to take the best from both TD and Monte Carlo worlds. However TD(n) has two drawbacks: learning delay and memory requirements, which both grow with n. In fact, the value function is updated only after the first n rewards have been observed and throughout exploration a memory of the last n rewards is needed.

TD(λ) methods solve these problems by updating the value function at each step, while propagating the information (reward) of future transitions to previous states. Each method is characterized by the way it defines the expected return and TD(λ) provides a new formulation defining the return as an exponentially decaying weighted average of TD(n) return for different n, which is called λ-return:

$$G_t^\lambda \doteq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{3.30}$$

$$= (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \quad \text{(Episodic case)} \tag{3.31}$$

where $\lambda \in [0,1]$ allows to continuously interpolate among Monte Carlo formulation of the return ($\lambda = 1$) and the return defined by one-step TD (when $\lambda = 0$). $G_{t:t+n}$ is the same quantity defined in Equation 3.23.

TD(λ) methods are used jointly with function approximation and the update rule for the parameters results from substituting U_t in Equation 3.29 with G_t^λ:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \Big[G_t^\lambda - \hat{v}(s; \mathbf{w}) \Big] \nabla_{\mathbf{w}} \hat{v}(s; \mathbf{w}) \tag{3.32}$$

However, this formulation is not enough to tackle the problems of TD(n), namely delayed learning and memory requirements proportional to n. In fact, G_t^λ depends on TD(n) returns. Equation 3.32 defines the *offline λ-return algorithm*, which in practice can be truncated up to some $n \ll T$, while usually performing better than TD(n), fixed n.

TD(λ) achieves online learning and constant memory requirements by employing *eligibility traces*, which work as a short-term memory on the updates of the value function. Eligibility traces allow to improve the estimate of the value function in previous states by propagating backwards the information (reward) encountered in successor states. Differently from the λ-return, which is said to have a *forward* view because the value of a state can be computed only after the future rewards have been observed, eligibility traces have a *backward* view because they refine the values of past states with future steps. TD(λ) update with eligibility trace $\mathbf{z}$ is:

$$\mathbf{z}_{k+1} = \gamma \lambda \mathbf{z}_k + \nabla_{\mathbf{w}} \hat{v}(S_t; \mathbf{w}) \tag{3.33}$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \, \delta_k \, \mathbf{z}_{k+1} \tag{3.34}$$

where δ_k is the TD error $R_{t+1} + \gamma \hat{v}_k(S_{t+1}; \mathbf{w}) - \hat{v}_k(S_t; \mathbf{w})$ already introduced in Equation 3.20 for TD(0) methods. As explained in the book of [SB18], for small step sizes α_k TD(λ) approximates well the offline λ-return method. TD(λ) can be applied to any parametrization of $\hat{v}(S_t; \mathbf{w})$. Nevertheless, when using *linear* function approximation this method can be further improved to achieve performances at least as good as λ-return algorithm, while preserving its desirable online properties. This improvement is called *true online TD(λ)* and the interested reader can find its derivation in the book of [SB18].

Planning

In some situations the interaction with the environment entails some sort of delay that would drastically reduce the speed of learning in terms of time. Moreover, it makes sense to re-use past knowledge acquired by interacting with the environment to further improve the estimated value function. *Planning* introduces a way to exploit previous transitions information to improve the values estimates in a sort of *offline* fashion, between two agent-environment interactions.

If the environment is slow to respond, planning introduces just a slight overhead, since it can be performed while waiting for the environment response.

Planning relies on a *model* of the environment, which is generated processing the past transitions, and it is different from *learning* in that it updates the values estimates depending on a model of the environment, rather than relying on the environment itself as *model-free* RL methods do. When the model represents faithfully the environment, planning can considerably speed-up convergence. Usually planning and learning are employed in tandem: the outcome of interactions with the environment are used to improve the values estimates (learning) and the model, whereas the model further refines the values estimates before another interaction with the environment is performed.

Dyna-Q is an example of interleaved learning and planning. The planning is based on a deterministic model of the environment that stores a buffer of past transitions $(S_t, A_t) \rightarrow (S_{t+1}, R_{t+1})$. Dyna-Q implements Q-learning for both planning and learning. If the environment is not static, the same state-action pair can generate a different transition, without affecting the method. Dyna-Q can be extended to Dyna-Q+ to foster exploration of states that the agent has not explored since a long time. Dyna-Q+ adds to the reward of a transition stored by the model, an *intrinsic reward*, proportional to the time elapsed since the last time that transition was performed in the environment. The full Dyna-Q+ method is shown in Algorithm 3.

Algorithm 3 Tabular Dyna-Q+

```
Require: $Q(s,a)$ and $Model(s,a)$
  loop forever
      $S_t \leftarrow$ current state.
      $A_t \leftarrow \epsilon$-greedy$(Q, S)$                                    ▷ Sample action.
      $(S_{t+1}, R_{t+1}) \leftarrow Environment(S_t, A_t)$          ▷ Agent-environment interaction.
      $Q_{k+1}(S_t, A_t) = Q_k(S_t, A_t) + \alpha_k \left[R_{t+1} + \gamma \max_a Q_k(S_{t+1}, a) - Q_k(S_t, A_t)\right]$
      $Model(S_t, A_t) \leftarrow (S_{t+1}, R_{t+1}, t)$                  ▷ Update model.
      loop$N$ times                                                   ▷ Perform $n$ planning steps.
          $(S, A) \leftarrow$ sample uniformly from model
          $(S', R) \leftarrow Model(S, A)$                               ▷ Get transition from model.
          $R \leftarrow R + k\sqrt{\tau}$                                      ▷ Intrinsic reward for small $k$.
          ($\tau$ is the time since this transition was previously performed in the environment).
          $Q(S, A) = Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$          ▷ Plannig.
```

Dyna-Q+ can be further improved by noticing that during planning each (S, A) pair is sampled uniformly from the model. This can be improved by providing each transition with a priority proportional to its TD error. When sampling from the model during planning, the priority would privilege updates to the values estimates that are likely less accurate. This approach is called *prioritized sweeping*.

Experience Replay (ER) was initially introduced by [Lin92] and consists of maintaining a buffer that stores transitions resulted from past agent-environment interactions. ER is not properly a model-based approach, but it mimics its intuition (e.g. Dyna). This method became popular with the work of [Mni+13] as it brings two main advantages when applied with Deep Reinforcement Learning (DRL) for function approximation: it improves sample efficiency and decorrelates the samples, reducing the bias of SGD gradients. Between two interactions with the environment, a batch of transitions is uniformly sampled from the replay memory and the parameters $\mathbf{w}$ of $\hat{q}(s, a; \mathbf{w})$ are updated to reduce the approximation error for $q_\pi(s, a)$. Since every transition stored in the replay memory can be reused multiple times, ER improves sample efficiency of DRL methods.

3.3.3 Revealing Dormant Malicious Behavior Through Reinforcement Learning

The solution propose in this work to tackle trigger-based evasive malware is a Reinforcement Learning-based approach that employs a debugger to explore a binary sample under analysis. Since I assume the trigger-based behavior to be triggered by a condition, the Reinforcement Learning (RL) agent is trained to explore the conditional paths forcing some branches to be executed, when necessary. The goal of this framework is to find all the quiescent malicious *capabilities* in the analyzed binary, if present. Malware capabilities can be considered as the building blocks of the whole behavior of a malware, performing some simpler tasks (e.g., sending spam, encode data using XOR, create process). The output of the RL agent is the list of all conditions in which to hijack the natural execution flow, in order to reveal all the hidden malicious capabilities. After a binary is patched following the instructions produced by the RL tool it can be sent to a malware analysis engine, in order to be further analyzed. The goal of the solution proposed in this work is to identify and mitigate all the trigger-based evasion attempts a malware sample may put in place, improving the information that a downstream malware analysis task can extract.

Markov Decision Process Formalization

The RL problem is formalized according to the Markov Decision Process (MDP). This entails the definition of *environment, actions space, states space* and *reward function.* These aspects, together with a description of the agent goals, are presented in this section.

This scenario violates the Markov property for states, like in most of the real-world problems that are solved with RL, because a code block does not bring enough information to characterize all the preceding code blocks. As a result, the couple (S_{t-1}, A_{t-1}) is not enough to fully characterize previous history. In other words,

$$\mathbb{P}(S_t \mid S_{t-1}, A_{t-1}) \neq \mathbb{P}(S_t \mid S_0, A_0, S_1, A_1, \dots S_{t-1}, A_{t-1})$$

To solve this problem the theory allows to reformulate the MDP as a Partially Observable Markov Decision Problem (POMDP), where at each time step the environment transitions to state $S_t \sim \mathcal{S}$, but the agent receives an *observation* $O_t \sim \mathcal{O}(S_t)$ of it, rather than the complete information describing the state. POMDP and MDP can be unified by introducing the approximate state $\hat{S}_t = f(O_0, \dots O_t)$, where f performs an arbitrary transformation of the previous history. In practice, usually just a subset of the past history is sufficient. Furthermore, to reduce the impact of the violation of the Markov property, additional information about the history of actions taken is included in the observation returned by the environment.

Environment
When interacting with the environment, composed of the malware sample and the debugger, I make the following assumptions:

- The Control-flow Graph (CFG) is not known *a priori* as it may be obfuscated or hard to compute (e.g., indirect jumps/calls). It is built dynamically, while debugging the code.
- When in a code block, the RL-based debugger executes its code and creates a feature representation for that code block. The feature representation has to convey as much information as possible about the potential *malicious nature* of the block (e.g., static features, behavioral features).
- The feature representation constitutes part of the state fed to the agent and is generated as ϕ(CB), where CB stands for code block and ϕ is a function that maps a code block to the features space.

Agent's Goal
The agent acts only on Conditional Jump Instructions (CJIs). Tampering with these instructions enables a debugger (the agent) to force the execution flow to follow an alternative conditional path. The goals of the agent are the following:

- Edit the CJIs to force the binary to reveal its malicious behavior.
- Be conservative: edit as little code as possible to reveal the hidden behavior.

Episodic Formulation

Debugging can be framed as a game episode which terminates when one of these mutually exclusive conditions is met:

- The debugged binary reaches the `exit()` instruction.
- The agent explicitly terminates debugging.
- The debugging crashes.
- The debugging budget is reached (total number of actions taken). We have to take into account the Turing halting problem: the execution of a binary is never guaranteed to come to an end. Therefore, the agent has a limited budget of actions before the debugging process is killed.

Actions

For each step of the interaction with the debugger, the agent is in a condition (e.g. If-Then-Else) and it has to decide whether to hijack the control-flow in that condition or not. Metaphorically, the interaction of the agent with the debugger binary can be seen as the agent walking in a maze and looking for some hidden malicious capabilities before exiting from it. A malicious capability can be considered as a flag in a *capture the flag* competition. From now on, the terms "flag" and "capability" will be used interchangeably. The actions that I have designed to enable the agent to explore the binary's control-flow graph are:

- **Hijack** the control-flow in the current condition. `CJI <addr>` checks some logical expressions concerning the bits of the flags register. When the expression is `true`, the jump will be performed. When the agent aims to hijack the control flow, it will tamper with the flags register in order to reverse the outcome of the logical expression checked by the CJI.
- **Jump back** in the code block before the previous condition.
- **Terminate debugging**. This action enables the agent to quit the exploration as soon as all the flags have been encountered, without using further computational power.
- **Do nothing**. Let the current condition follow its natural control-flow.

Figure 3.3 represents the steps the agent can take. Going right means taking the true branch, whereas left the false branch. The true branch is taken either if the condition autonomously

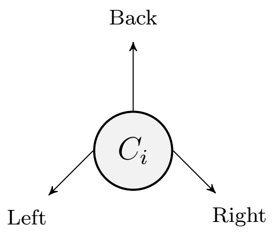


Figure 3.3: For each encountered condition, the agent can take one among three directions.

took that branch or the agent hijacked the control-flow to force that direction. A similar reasoning holds for the false branch.

States

Each observation returned by the environment includes the following elements: an embedding of the current code block instructions obtained resorting to NLP strategies adapted to disassembled binary code (opcodes); a binary scalar $\in \{0, 1\}$ that indicates whether the current condition is going to perform a jump (1) or not (0); the total number of iterations (steps) performed so far, normalized with respect to the actions budget in order to bound it in the interval $[0, 1]$; the proportion of *hijack* and *jump back* actions in two additional components, both normalized with respect to the budget of total actions.

I deem important to include in each observation the proportion of actions taken, to help the agent to better infer the reward function. In fact, a small negative reward is given when some "expensive" actions are taken, like *jump back*, which requires a non-negligible computational overhead. Furthermore, I want the agent to find hidden capabilities with the least number of interactions and hijacks. Also in this case, including these quantities in the observation provides the agent with more information about the "rules of the game", as it was done for TD-Gammon and other popular game-playing RL algorithms.

Eventually, according to the POMDP framework that extends MDP to settings with non-Markov states, a state is obtained as the concatenation of the last n observations. For instance, Alpha Go (Zero) uses the last $n = 8$ configurations of the board.

Rewards

Consistently with the analysis performed in Section 2.3.2, to accurately distinguish legitimate from malicious trigger-based behavior, each state includes features that carry behavior information. The goal is to make the agent learn to recognize hints of malicious behavior, deciding which conditional paths to explore accordingly. The ultimate goal of the RL agent is to find the quickest path on the conditional graph to executes all the hidden malicious capabilities.

To speed up convergence and encourage the agent to look for malicious content when exploring the binary, I give the agent a reward every time it steps in a region of code containing malicious capabilities. The presence of a capability that may suggest malware behavior is detected by an oracle. A good candidate for an oracle is the recent work of [Dow+21], which proposes a model for malicious capability recognition called *DeepReflect*.

At the end of a trainig episode a larger final reward is given consistently to the proportion of malicious capabilities found throughout the episode, with respect to the total number of malware capabilities hidden in the sample. The final reward is formalized as:

$$R_F = \beta \left(\# \text{ Found capabilities} - \frac{1}{2} \# \text{ Total capabilities} \right) \tag{3.35}$$

therefore, when all the flags are found, the final reward is $\frac{\beta}{2}$, when no flag is found it is $-\frac{\beta}{2}$. If some rewards are found, the terminal reward lays in the $[-\frac{\beta}{2}, \frac{\beta}{2}]$ interval.

The total number of capabilities is known because the dataset samples are synthetically generated by a script that I implemented. In general, when working with real binary samples, I assume that they have been previously analyzed by security experts that were able to find the great majority of hidden capabilities.

A reward, together with a new state, is given to the agent as the result of its interaction with the environment. The reward function is designed to encourage the agent to find the hidden flags with the least number of hijackings. A default negative small reward is given to the agent to prevent it from being trapped in loops, speeding up convergence to the path that leads to the next flag. The agent receives a large negative reward when it is not able to find any hidden flag and the debugged binary's execution terminated or crashed.

The definition of the reward function is an highly sensitive step as it can drastically influence the outcome of training. The easiest form of reward function is one that defines some scenarios and gives a reward when some condition is satisfied. This format of rewards is well understandable by humans but it also slows learning because it is hard for the agent to *generalize*. When a reward is given only for few specific transitions, it results sparse overall. This formulation causes a slow credit assignment, namely a slow propagation of the reward signal to previous transitions in a trajectory. Moreover, sparse and deceptive rewards make the task become an *hard-exploration* problem, which is one of the hardest class of tasks to be solved with RL. An example of discrete reward function is the following

- 0 for not hijacking (default reward).
- -1 when hijacking or jumping back.
- +1 for finding a malicious capability.
- -10 if the agent terminated debugging but no flag was found.
- +10 for finding all the flags.

- -10 for crash during debugging
- -10 for budget quota exceeded (`maxiter` reached).
- $\beta = 20$ for the final reward.

A better alternative to discrete reward function is a continuous one. In my experiments I employ the following continuous reward function:

$$\psi(s_i, a_i) = \xi(s_i; \gamma) + \alpha_T(s_i) + \alpha_H(s_i; h) + \alpha_B(s_i; b) + \rho(s_i) \tag{3.36}$$

where:

- $\xi(s_i; \gamma) = \max\{\exp(-\gamma\, ||\varepsilon(s_i) - f_1||_2^2), \ldots, \exp(-\gamma\, ||\varepsilon(s_i) - f_N||_2^2)\}$
  where $\varepsilon(s_i)$ extracts the code block embedding of the last observation, and $f_1, \ldots f_N$ are the embeddings of the malicious blocks. This functions gives the agent a clue about the location of malicious blocks in the block embedding subspace. As $\gamma \to \infty$, the contribute of this term to the overall reward becomes similar to a sum of Dirac's deltas $\xi(s_i; \gamma) \to \sum_i \delta_{f_i}(s_j)$, approximating the discrete reward.

- $\alpha_T(s_i), \alpha_H(s_i; h), \alpha_B(s_i; b)$ give a negative reward proportional to the total number of performed actions, hijacks and jumps back respectively. They all have the same form, for instance $\alpha_H(s_i; h) = -\frac{1}{h} \frac{\#\text{ hijack}}{\#\text{ actions}}$. h is a weighting term to control the reward.

- $\rho(s_i) \propto 1/\tau$ is an *intrinsic reward* term that fosters exploration of rarely-visited states. This quantity is bounded in the interval $(0, 1]$, whereas τ is the time elapsed since the last visit of state s_i. Intrinsic reward provide a more sensible exploration incentive with respect to ϵ-geedy approaches, as the exploration encouragement is given only where it is necessary, namely in the rarely-encountered states where the agent should invest more time exploring.

Furthermore, at the end of each episode an additional reward is given proportionally to the number of malicious capabilities found as described in Equation 3.35.

Planned experiments

In the last part of my internship I will perform a set of experiments of different RL algorithms trained on the synthetic dataset of evasive binaries I have generated. As a result of a literature research about modern RL approaches to solve real-world problems, I am planning to use the following models: vanilla DQN plus some improvements of it and DQN(λ), which implements TD(λ) methods.

DQN is an architecture trained with semi-gradient Q-Learning, employing a deep artificial neural network (ANN) for nonlinear function approximation. It was initially introduced in the work of [Mni+13] to solve 49 different Atari games without modifying the architecture, the hyperparameters values, or employing specific features for different tasks. The network was reset before being trained on a different task, learning a different policy for each game.

To improve the stability of the method, DQN employs Experience Replay (ER) to improve sample efficiency, to compensate for the need of large dataset to train deep learning models. Since training is carried out also on *old* interactions with the environment, is mandatory to use an off-policy method like Q-learning, treating old transitions as generated by a behavior policy. ER also has the benefit of decorrelating the samples when sampling a batch of transitions from the memory, which reduced the bias that correlated examples introduce in SGD gradients.

The second strategy to improve stability consists of using an auxiliary network, called *target* network, to compute the term $\gamma \max_a \tilde{q}(S_{t+1}, a; \mathbf{w}_k)$ of the TD error $\delta_k = R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a; \mathbf{w}_k) - \hat{q}(S_t, A_t; \mathbf{w}_k)$. The TD error is used in the update rule of Equation 3.29. The target network is identical to the primary one, and the up-to-date parameters of the primary are copied in the target network every K training iterations.

The last trick used by the authors to guarantee stability is to clip the TD error δ_k in the interval $[-1, 1]$.

DQN proved that an RL agent can attain human-level capabilities in playing some video games without much prior knowledge. However it also achieved poor performances on other games, underlying how this architecture was only an early work. In particular, DQN produced scarce results for games like *Montezuma's Revenge* which belongs to the class of *Hard-exploration* problems, because they requires the agent to learn complex strategies with sparse or deceptive feedback (rewards).

Some remarkable improvements of DQN are **Double DQN (DDQN)** [vHGS15], which extends DQN with *double Q-learning* algorithm, known for its ability to counter the *maximization bias* typical of methods that use the **max** operator, such as Q-learning. Another improvement to DQN is **Prioritized DQN (PDQN)** that samples transitions from the relay buffer with a probability proportional to their TD error, since they are likely contribute more to the learning of the parametrized value function. A further improvement was **Dueling DQN**, introduced by [Wan+16] that improves the training efficiency of DQN as its vanilla version, only updates the weights that contribute to the value of the action that has been taken, whereas Dueling DQN learns at the same time the value $\hat{v}(s; \mathbf{w})$ of a state and the advantage for each action $\hat{a}(s, a; \mathbf{w}) = \hat{q}(s, a; \mathbf{w}) - \hat{v}(s; \mathbf{w})$. More recent works like *Never Give Up* [Bad+20b] and *Agent57* [Bad+20a] further improve the original DQN architecture with more advanced strategies.

DQN(λ) is an alternative improvement of DQN explored by [DA19], who propose to shift from the TD(0) one-step (bootstrapped) return $G_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ to the λ-return of Equation 3.31. As previously explained, this would require the method to learn offline as the λ-return can only be computed at the end of the episode. The authors of DQN(λ) propose a recursive formulation for the λ-return that makes it possible to compute the return for each state by looking only at the return of the successor state. A key passage of the proposed architecture relied on *refreshing* the λ-returns of preceding states every time the value function is updated for a state. This would not be scalable on a large replay memory, so the authors restrict the cost by refreshing only a subset of transitions' returns contained in a cache (a subset of the replay memory). The network parameters are then trained with minibatches of transitions sampled from the cache.

DQN(λ) employs a prioritized memory and proves of being capable of achieving better performances than vanilla DQN on the Atari benchmark dataset. As discussed before, the λ-return improves the problem of slow credit assignment, facilitating the propagation of the reward signal.

Chapter 4

Experimental Setting

In this chapter is presented in details the experimental setting, starting form the description of the method used to generate a synthetic dataset of evasive binary samples. This dataset is composed of binaries made of independent building blocks, which are randomly sampled and connected in different ways to produce diverse samples, starting from the same fundamental blocks. Then, this chapter gives a detailed explanation of the methods used in the experiments, including code block embeddings and variations to the original DQN model [Mni+13]. DQN implements Q-learning with function approximation, where a Deep Neural Network is used to approximate a non-linear state-action values function. Eventually, this chapter shows the results obtained by the RL algorithms chosen to address the task of finding hidden malicious capabilities in evasive malware. An evaluation of the best model on unseen samples and an ablation study are carried out to complete the analysis.

4.1 Synthetic Dataset Creation

There are two main reasons that led to the creation of a synthetic dataset to train the RL agents to find hidden malicious capabilities in binary samples. The first one is the intrinsic difficulty in the security setting to access datasets due to reduced availability, non-disclosure due to strategic interest of security companies, potential misuse of malware samples by unauthorized people that may release malware (intentionally or not), and creating security issues.

The second reason is due to a methodological choice of exploring this new research direction step-by-step, testing the assumptions and the algorithms in an easier setting at first, in order to collect useful information to improve the RL models and refine the training pipeline. The knowledge gathered by training a proof of concept on a simpler dataset can pave the way for future work aimed at addressing real-world binary samples.

The synthetic dataset I have generated is composed of evasive binary samples, each one organized as a graph of building blocks. There are three classes of building blocks: common, evasive and flag.

- **Evasive** blocks gather some information from the environment and subsequently check a trigger condition. If the condition is not satisfied, the sample will perform some sort of evasion including exiting the execution or performing some trivial operation to disguise the malicious behavior.
- **Common** blocks implement trivial operations such as data structures operations.
- **Flag** blocks are the ones implementing some malware capability. In practice they do not put in place a real malicious behavior, but they just emulate it, being safe to be execute in any environment.

In addition to the synthetic dataset I have implemented a script to setup a simulated victim environment, including file system, configuration files and network. The file system includes a tree of directories, a location containing private documents and other files containing fake content. The configuration location emulates the `/etc` Linux directory, which is the equivalent for the Windows registry. In this location there are some simulated configuration files with textual fields, the `passwd` file, which in Linux is used to store users' passwords, and other files emulating the artifacts created by security products installed on the system. The network environment is emulated by running an HTTP server in a directory containing some HTML page samples, and an FTP server in another location. Both servers are accessed by the samples using network protocols to simulate real network traffic with remote locations. The FTP server emulates a remote filesystem to which malware can upload private files or from which download malicious files.

I have developed a Python script to generate a dataset of evasive binary samples that receives as input the following parameters:

- Dataset size: number of samples to produce.
- Samples length: number of building blocks composing each sample.
- Number of hidden flags.
- Proportion of evasive blocks with respect to the total number of blocks.
- Graph structure: tree or cyclic graph.
- Probability of creating loops in the graph, if it is a cyclic graph.

- Blocks sampling method: whether to sample the building blocks with replacement or not from the blocks pool.

At first a graph of building blocks is randomly generated for each sample. The graph is checked against some quality measures and discarded if it does not respect them. In fact, I consider a graph to be *well-formed*, when at least one flag block have to be present and the number of evasive blocks has to be within a tolerance interval centered on the value of the argument passed to the script. The graph can be a tree or a cyclic graph, which is obtained by adding some *backward edges* to an existing tree. A backward edge connects a node with an ancestor, creating a loop in the graph. The number of backward edges is controlled by an argument passed to the script that defined the probability of creating a backward edge between two nodes. For the loop to be implemented in source code, one of the node has to be the ancestor of the other. In other words, to create a backward edge between two nodes a path must exist between the two on the original tree. An example of backward edges is shown in Figure 4.1,

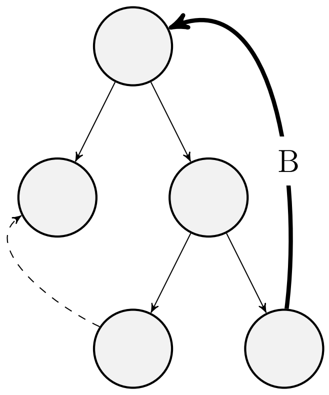
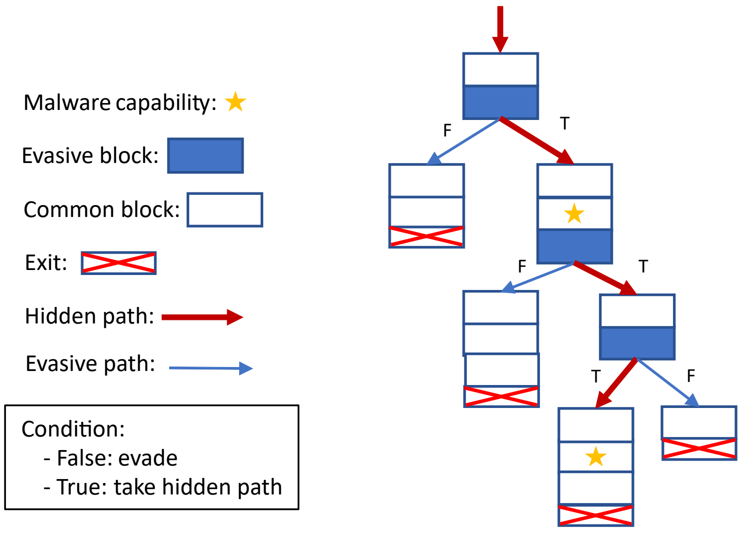
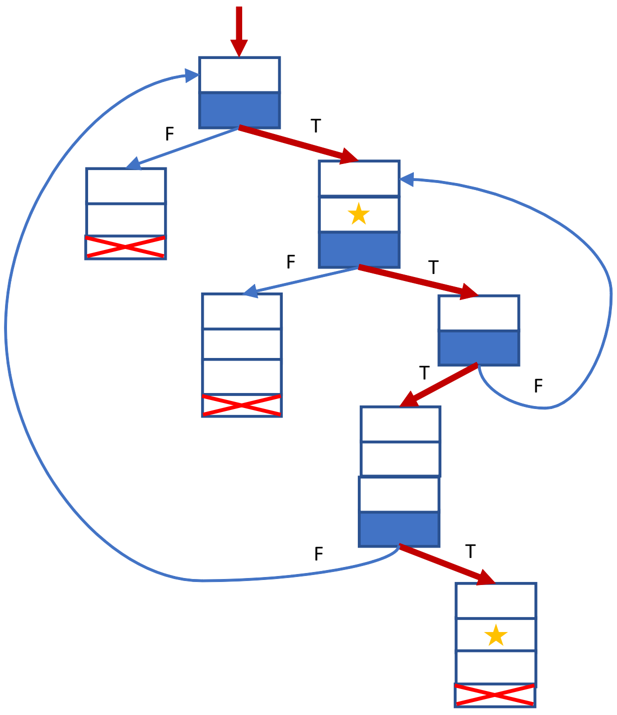


Figure 4.1: Example of a backward edge (in bold). However, not any edge can be added. For instance the dashed edge is not allowed.

An example of tree- and graph-structured blocks organization is given in Figure 4.2. In each graph of blocks exists a *hidden path*, which is the conditional path that makes it possible to execute all the malicious capabilities (flags). The edges that do not belong to the hidden path represent conditional branches executed as the result of some evasion. Some examples of hidden path and evasive edges are showed in Figure 4.2.



(a) Tree-shaped blocks graph.



(b) Cyclic block graph obtained by adding backward edges.

Figure 4.2: Example of tree- and graph-shaped block graphs representing the highest level of abstraction of the generated evasive samples.

The blocks graph of each sample is then converted to C source code arranging the source of each building block according to the conditions graph. The blocks graph can be a tree of nested if-then-else constructs or a cyclic graph containing also do-while and while-do loops. The next step involves the compilation of each sample's source with GCC compiler in a Linux environment. Eventually, each sample is automatically executed by a diagnosis script that checks for runtime errors. The binary samples resulting from this operation can be used to train and evaluate the RL agents.

4.1.1 Building Blocks Implementation

The binary samples are intended to run in a Linux environment, therefore the evasive blocks emulate the evasions performed by Linux malware, for instance looking for artifacts left by analysis tool in a simulated Linux environment. Below there is a list of all the building blocks I have implemented in C source code that will be randomly arranged to compose the binary samples of my synthetic dataset. Before implementing the blocks, I have collected information form different sources of knowledge on the security domain coming from both my colleagues, which are security researchers, and on the web. I have taken inspiration from popular malwares, such as WannaCry and Ryuk ransomware. The rational behind my design choice is to represent, to the best of my ability, real-world evasive malware, producing a valuable proof-of-work. If successful, this work could be improved and implemented in a security product. However, due to the time constrains of this internship, I had to restrict the set of blocks I planned to implement, keeping only the most meaningful for a proof-of-concept.

Evasive blocks

- Check for network. Some analysis systems may prevent a sample under analysis to connect to the network. The evasive condition checks if the HTTP server is reachable.
- Check for the existence of a specific web page on the Web (HTTP) server. This block is inspired to C&C approaches. For instance the WEBC2[1] malware family retrieves commands from web pages created by the malware author and performs malicious actions accordingly.
- Try to reach an domain that does not exists. If success, evade. Some analysis systems make the analyzed samples believe that the network is reachable by returning success every time the sample tries to connect to a domain. To counter this, the popular WannaCry ransomware checked the DNS for a domain that did not exist. Receiving a positive

[1] https://otx.alienvault.com/pulse/56aa595e4637f23551f3c21d/related

response means being run in a simulated environment, therefore the malware terminated its execution.

- Compare the hostname with the victim's hostname.
- Check if a debugger is attached to the running process. Anti-debug evasion.
- Check if a user is registered in the victim host, analyzing the `passwd` file.
- Check for active user name.
- Verify if a security product is installed in the machine, looking for specific keywords in file paths.
- Access the field of a configuration file in `/etc` location. This is equivalent to check for a registry key in Windows systems.
- Get current date and time. This is common for logic bomb malware.

Flag blocks emulate malware capabilities.

- Download from the network, through FTP protocol, N files with malicious content.
- Steal files: upload M private files to the FTP server.
- Modify the `passwd` file.
- Multiple calls to hashing functions (e.g. SHA-265), which is common in cryptominer malware [RŠL21].
- Read a file, encrypt its content and write it to another file, delete the cleartext file. This is a common pattern that ransomware (e.g. WannaCry, Ryuk) perform a large number of times to encrypt most of the content in the filesystem of a victim host.
- Open a large number of connections with a domain. This is typical of (distributed) Denial of Service (DoS) attacks.

Common blocks perform trivial operations to emulate any sort of behavior that should not be deemed as a malicious capability such as reading files, preforming data structure operations like sorting a list, looking of an element in an array.

4.2 Experiments

This section gives a detailed explanation of the experiments performed, the hyperparameters used and the Deep Neural Network architecture used to implement Deep Reinforcement Learning algorithms. Eventually, the results are presented, comparing algorithms and methods on the task of finding hidden malicious capabilities in evasive samples. The methods are then compared with an ablation study.

Code Block Embedding

As explained in the previous chapter, the code block embedding task is inspired by NLP methods and has the goal of producing a numerical representation for categorical data, the opcodes instructions, while preserving most of its semantics. In the experiments I have used two embedding approaches: TfIdf with dimensionality reduction and Doc2VecC, both trained on the code blocks extracted from the binaries of the synthetic dataset. These methods have been chosen because they have a small number of trainable parameters, which reduces the risk of overfitting on the relatively small dataset I generated, while having a small inference time. The downside of these embedding approaches, compare to more powerful ones like BERT, is their limited capacity of capturing complex semantic relations in text. However, assembly language is less complex and ambiguous with respect to the natural language. Since the embedding method produce vectors of real numbers, the RL algorithms have to deal with a continuous state space, which is addressed by using function approximation. The best dimension for Doc2VecC opcode embeddings has been found to be 31, whereas the vectors produced by the TfIdf vectorizer have been reduced to the first 18 principal components of the PCA method. The number of principal components has been chosen to preserve 85% of the explained variance.

In Figure 4.3, the embedding space produced by Doc2VecC for the opcodes has been projected to a two-dimensional space using the t-SNE method for non-linear dimensionality reduction. It is interesting to notice how the embedding space preserves the semantic information carried by opcodes and function calls. In fact, it is possible to isolate some clusters of well-known operations, including network communication, file and directory operations, evasive and malicious behaviors, exit operations (typical in case of error), and the Conditional Jump Instructions (CJIs). The regularization used by Doc2VecC produces embeddings with lower norm for frequent terms. In fact, the CJIs are embedded in the central cluster, consistently with the expectations.

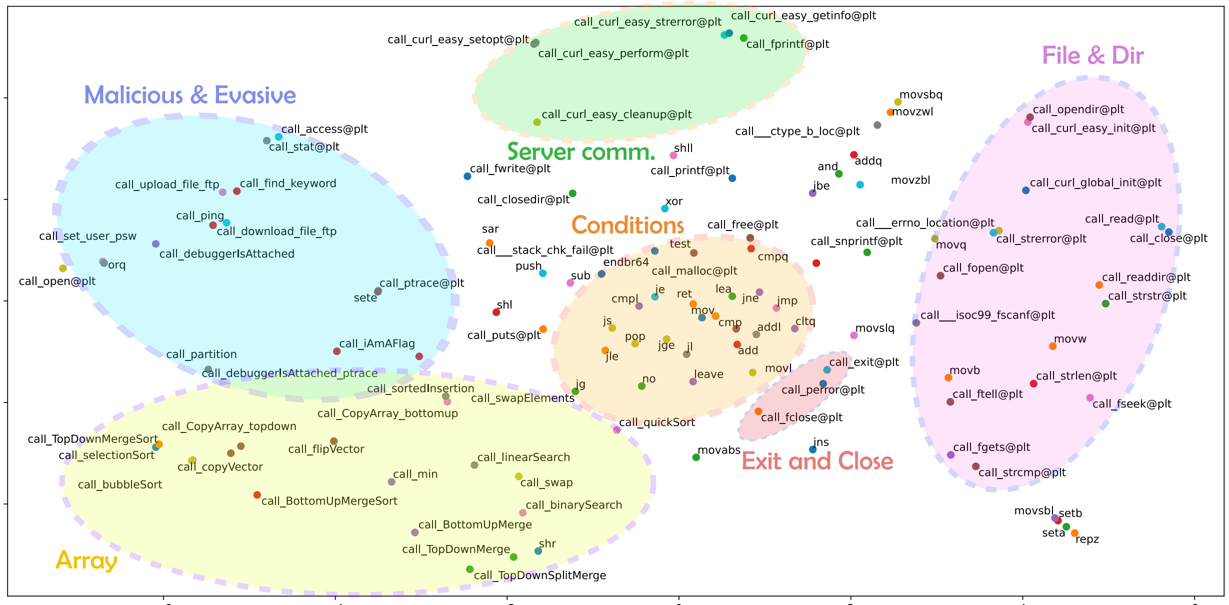


Figure 4.3: Embedding space produced by Doc2VecC method, projected on a 2D space with t-SNE. The embedding method preserves the semantics of opcodes and function calls, making it possible to identify some clusters.

Static vs. Dynamic Training

In this work, I have developed an interpreter to allow the RL agent to interact with the GDB debugger, analyzing the content of a binary sample and hijacking the execution flow in the conditions, at runtime. However, the dynamic interaction with the debugger entails a considerable time overhead, which makes training extremely slow. To get around of this problem, the training has been performed on a static graph representation of the binary samples, which produced a speedup of more than 200 times. The trained models can still be deployed in a dynamic environment, interacting with a debugger, using the developed interpreter. Furthermore, to simplify the adaptation of the static representation of a binary to its dynamic version, the action space has been reduced to the following actions: *jump*, which goes to the code block executed when the condition is true, and *don't jump*, which goes to the alternative block of code. This also simplified the analysis.

Vanilla DQN

The experiments involving the search for the best RL method have started from the popular DQN [Mni+13], which provides a good baseline. It implements Q-learning, using a Deep Neural Network (DNN) as a function approximator. As stated in the Chapter 11 of the book

[SB18], the jointly usage of bootstrapping, off-policy learning and function approximation can generate a considerable source of instability, hindering learning. The authors of DQN propose three methods to stabilize learning: experience replay, clamping the TD error (the loss) between -1 and 1, and the utilization of an auxiliary network called *target network*, used to decorrelate the computation of the training target. The main network is the *policy network* $\hat{q}(\cdot, \cdot; \mathbf{w})$, which is used to select the greedy action when interacting with the environment, from which the name "policy". The parameters of this network, $\mathbf{w}$, are learned online. In contrast, the target network has parameters $\mathbf{w}^-$, which are periodically replaced with the parameters of the policy network. The DQN computes the learning target as:

$$R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a; \mathbf{w}^-)$$

A first improvement over DQN is Double DQN, initially proposed by [vHGS15], which brings the idea of Double Q-learning to the domain of Deep Q-learning. Double Q-learning aims at mitigating the *maximization bias* introduced by the max operator, which arises at the beginning of training. Although the maximization bias often vanishes throughout training, it slows learning since it leads the agent to overestimate the value of some state-action values, due to sampling error. Double DQN ameliorates the formulation of the target, to counter the maximization bias. The target if rewritten as:

$$R_{t+1} + \gamma \hat{q}(S_{t+1}, \arg\max_a \hat{q}(S_{t+1}, a; \mathbf{w}); \mathbf{w}^-)$$

To solve the problem of dealing with states that violate the Markov property, namely

$$\mathbb{P}(S_t \mid S_{t-1}, A_{t-1}) \neq \mathbb{P}(S_t \mid S_0, A_0, S_1, A_1, \ldots S_{t-1}, A_{t-1})$$

the Markov Decision Problem (MDP) has been extended to the Partially Observable MDP, where a state is approximated as a set of the previous k observations. The ideal case requires k to be equal to the current length of a transition, but a remarkable improvement has already been observed when setting $k = 4$. In Alpha Go Zero [Sil+17], the authors propose to use $k = 8$. However, this requires additional trainable parameters, which proved to cause overfitting in this setting, where the dataset size and samples variety are reduced. As a result, a state is the result of stacking the last 4 observations, which are sent to a 1D Convolutional Neural Network (CNN). The observations are stacked and not concatenated for two reasons: first, the dimension o of an observation is variable, depending on the embedding method used, second, convolutional layers have less trainable parameters due to parameter sharing. Stacking allows to apply a CNN. The architecture of the CNN is showed in details in Table 4.1. The output size of the last layer is equal to the size of the discrete actions space a, whereas h is a tunable parameter and controls the size of the linear layer.

Layer	Output size	Details
Input	$4 \times o$	-
1D Conv, ReLU, Batchnorm	$8 \times \cdot$	kernel 4, stride 1
1D AvgPool, ReLU, Batchnorm	$8 \times \cdot$	kernel 2, stride 2
1D Conv, ReLU, Batchnorm	$16 \times \cdot$	kernel 4, stride 1
AvgAdaptPool, ReLU, Batchnorm	h	adaptive kernel
Linear (output)	a	$h \times a$

Table 4.1: CNN architecture used for the experiments.

The CNN architecture presented in Table 4.1, trained with Double Q-learning constitutes the baseline to be compared with the other models and strategies I have explored. For this reason, it is referred as *Vanilla DQN* and the other models explored in this work improve over this baseline. Similarly to DQN, the training is supported by a replay memory from which batches of transitions are sampled randomly, during training. The TD error, which constitutes the objective function to minimize is

$$\begin{aligned}\delta &= \textbf{Target} - \hat{q}(S_t, A_t; \mathbf{w}) \\ &= R_{t+1} + \gamma \hat{q}(S_{t+1}, \arg\max_a \hat{q}(S_{t+1}, a; \mathbf{w}); \mathbf{w}^-) - \hat{q}(S_t, A_t; \mathbf{w})\end{aligned}$$

and as in the original paper, it is clamped in the interval $[-1, 1]$, before computing the gradients. In addition, clamping the gradients in the same interval proved to be an effective strategy to prevent diverging behaviors during learning. A small batch size has the advantage of reducing the chances of getting trapped in local minima, however it also produces noisier estimates of the gradients, which in this case hindered convergence. Setting the batch size to 32 improved stability. The optimizer is Adam with initial learning rate of 1e-3. The loss criterion is the L1 smooth loss, also known as Huber loss, when $delta = beta = 1$, as in this case. The advantage of this loss, over the classic Mean squared Error (MSE) loss, is that the outliers falling outside the $[-1, 1]$ interval, have a smaller impact, with the effect of stabilizing learning. For the Vanilla DQN training procedure, the TD errors are clamped in the $[-1, 1]$ interval, so the Huber loss is the same as the MSE. However, in further experiments with different models, the TD error is clamped in a wider interval. The size of the replay buffer has been set to 100K transition. The RL task is framed with an episodic formulation and the discount factor has been set to $\gamma = 0.999$. The optimal value for the hidden dimension of the CNN, h, has been found to be 48.

DQN with Prioritized Replay Memory

I the work of [Sch+16] is shown how the sampling procedure from the replay memory, typical of experience replay methods, can be improved by increasing the probability of sampling the transitions with the highest TD error, as they are the ones with more learning potential. There are two strategies of attributing uneven sampling probabilities among the transitions in the replay buffer. The first, called Proportional priority, gives to each transition a priority equal to the absolute value of its TD error $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant that ensures that no transition has null priority. However, this is subject to outliers, with the risk of sampling always from the same subset of transitions. Another strategy consists of giving a priority proportional to the rank of a transition, $p_i = \frac{1}{rank(i)}$, where the rank is the order in the list of transitions, sorted by deceasing TD error. This approach is less subject to outliers. The rank based prioritization can be ameliorated by attributing a different rank to B clusters of transitions, grouped by similar TD error. B can be chosen to be equal to the batch size, so that each batch is the result of a stratified sampling, where a transition is sampled uniformly form each cluster. This ensures that each batch contains at least one transition for different intervals of TD errors.

The probability of sampling a transition from the replay buffer, given its priority p_i is computed as:

$$P(i) = \frac{p_i^\alpha}{\sum_j^N p_j^\alpha}$$

where N is the number of transitions in the replay buffer. When $\alpha = 0$, the transitions are sampled uniformly. To counterbalance the bias introduced by uneven probabilities, the authors propose to use the Importance Sampling (IS) weight that has to be multiplied for the TD error, before computing the loss. The IS weights are computed as:

$$w_i = (NP(i))^{-\beta}$$

When $\beta \to 1$, the IS correction becomes more aggressive. Thus, at the beginning of training $\beta \leftarrow \beta_0$ and then it is linearly incremented to attain 1 at the end of training.

In my experiments, the proportionally prioritized experience replay model is called PrDQN, whereas the rank-based with stratified sampling is RnDQN. For the first, I have chosen the values of $\alpha = 0.7$ and $\beta_0 = 0.5$, whereas for the latter I have used $\alpha = 0.6$ and $\beta_0 = 0.4$. Furthermore, the training hyperparameters are the same as for Vanilla DQN. The only difference from the baseline model is that the TD errors have been clamped in the $[-10, 10]$ interval.

DQN(λ)

The work of [DA19] extends DQN to utilize the λ-return as a target, as presented in Equation 3.30. This methods improves over deep Q-learning by reducing the instability introduced by bootstrapping. Furthermore, λ-return better propagates the reward information to previous state-action pairs, improving the credit assignment problem and producing better estimates for the state-action values overall. However, this comes at an higher computational cost, with respect to Q-learning. The authors propose a recursive formulation of the return, that reduces the complexity of its computation

$$G_t^\lambda = R_{t+1} + \gamma\lambda \left[G_{t+1}^\lambda - \max_a \hat{q}(S_{t+1}, a; \mathbf{w}) \right]$$

that depends only one the return of the next transition, which is 0 if the next state is a terminal state. A trajectory is pushed to the replay buffer after it has reached the terminal state and the λ-returns is computed backwards. To make the G_t^λ of past transitions a good training target, its value has to be periodically refreshed throughout training, which entails a sweep over the whole memory. In my experiments, I set the capacity of the replay buffer to be 5K, which proved to be enough to reach good results, while keeping low the burden of the refresh operation. Again, the TD error have been clamped in the $[-10, 10]$ interval, as for prioritized replay buffer methods. After some coarse-grained tuning, I found the best value of λ to be $\lambda = 0.7$.

Continuous Reward

To tackle the problem of sparse rewards, which are given only when a capability is found and at the end of the episode, I have implemented a continuous reward function, with the goal of giving some additional hint to the agent to better understand when it is "close" to a malicious block. In addition, the continuous reward contains a negative term, proportional to the number of actions taken, which increases as the budget of remaining actions lowers. Eventually, an intrinsic reward term has been added, inversely proportional to the number of visits of a state, with the goal of fostering exploration or rarely traversed states. The continuous reward received by the agent, after taking action a_i and transitioning to state s_i is:

$$\psi_t(s_i, a_i) = \max_j \left\{ e^{-\gamma_R \, ||\varepsilon(s_i) - f_j||_2^2} \right\} - \frac{t}{T} + \frac{\sigma}{N(s_i)^2}$$

where the first term is directly proportional to the distance, in the embedding space, of s_i from the blocks f_j containing a flag. The optimal value found for γ_R is 5. The second term increases in magnitude as the number of actions performed during an episode t attains the actions budget T. Eventually the intrinsic reward term is weighted by $\sigma = 0.5$ and $N(s_i)$ is the number of times state s_i has been visited during an episode. This function is the result of extensive experimentation, guided by an informal trial-and-error procedure.

4.2.1 Results

This sections reports the results achieved by the models during both training and evaluation on unseen binaries. In fact, the dataset has been split into a train set (of 10 samples) and a test set (of 5 samples). No validation set has been used and the tuning of the hyperparameters, or the choice of the best model has been carried out taking into account the performances on the train set. The training has been performed by *simultaneously* processing all the samples in the training set, in a *multi-agent* RL fashion, using one agent per training sample. At each time step, each agent takes an action on the binary it has been assigned to, and the resulting transitions are collected in the replay buffer. This allows to learn sample-invariant features.

The chosen evaluation measure is the evolution of the reward throughout training. There are two types of reward: the terminal reward is given at the end of an episode and is only proportional to the number of flags found in a binary sample. Thus, it is a good measure of the effectiveness of a RL method in finding hidden malicious capabilities. On the other hand, the cumulative reward, includes the terminal reward and adds to it all the other rewards received throughout an episode. The cumulative reward curves showed in the plots below, are the result of the average of the cumulative rewards accumulated by each agent throughout an episode. A similar reasoning holds for the terminal reward. Each reward curve reported in this section has been obtained as the average of 5 re-runs of the same method with a different random seed initialization. This allowed to compute 95% confidence intervals, which are omitted for the reward curves, to improve readability.

Training

The results obtained during training can be clustered in two main groups, differentiating the ones using a continuous reward function from the ones using a discrete reward function. Comparing the rewards evolution for discrete versus continuous reward, in Figure 4.4 and Figure 4.5, it is possible to notice how, at the beginning of training, the methods using continuous reward receive a higher cumulative reward since the beginning of training. However, the reward function is hard to tune and can easily introduce some unwanted noise, which makes training more unstable. Overall, the methods employing discrete reward perform better and this may be attributed to the deceiving effect of intrinsic reward and, more in general, to the difficulty of defining a good continuous reward function. The ideal continuous reward improves the credit assignment process, while introducing a low bias.

It is interesting to notice that Tf-Idf embedding performs better than Doc2VecC, on average, for most methods. This can be justified because the vocabulary has a reduced size, easing the task of PCA dimensionality reduction, while not exploiting the potential of Doc2VeC.

The models employing prioritized experience replay achieved good performances, comparable to the more expensive DQN(λ) model, which is the one achieving the overall best performances. This is reasonable, as it employs a training target which is more stable than the use used by TD(0) methods, and is more precise.

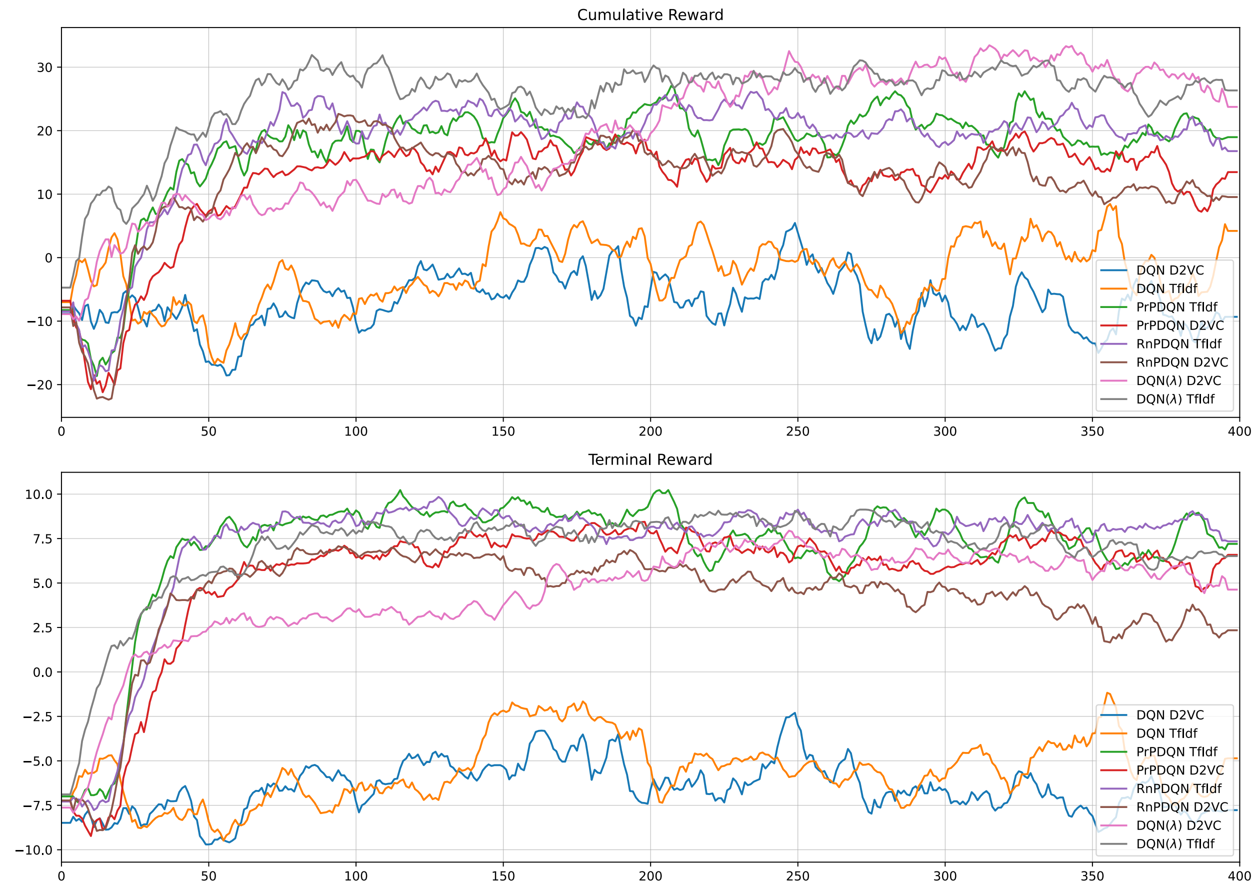


Figure 4.4: Reward evolution over training episodes. Discrete reward.

Also the losses of the methods employing continuous reward, in Figure 4.7, provide additional evidence supporting that this reward function introduces some source of instability. In fact, the loss curves of the methods using a discrete reward function are smoother and well-behaved, as depicted in Figure 4.6.

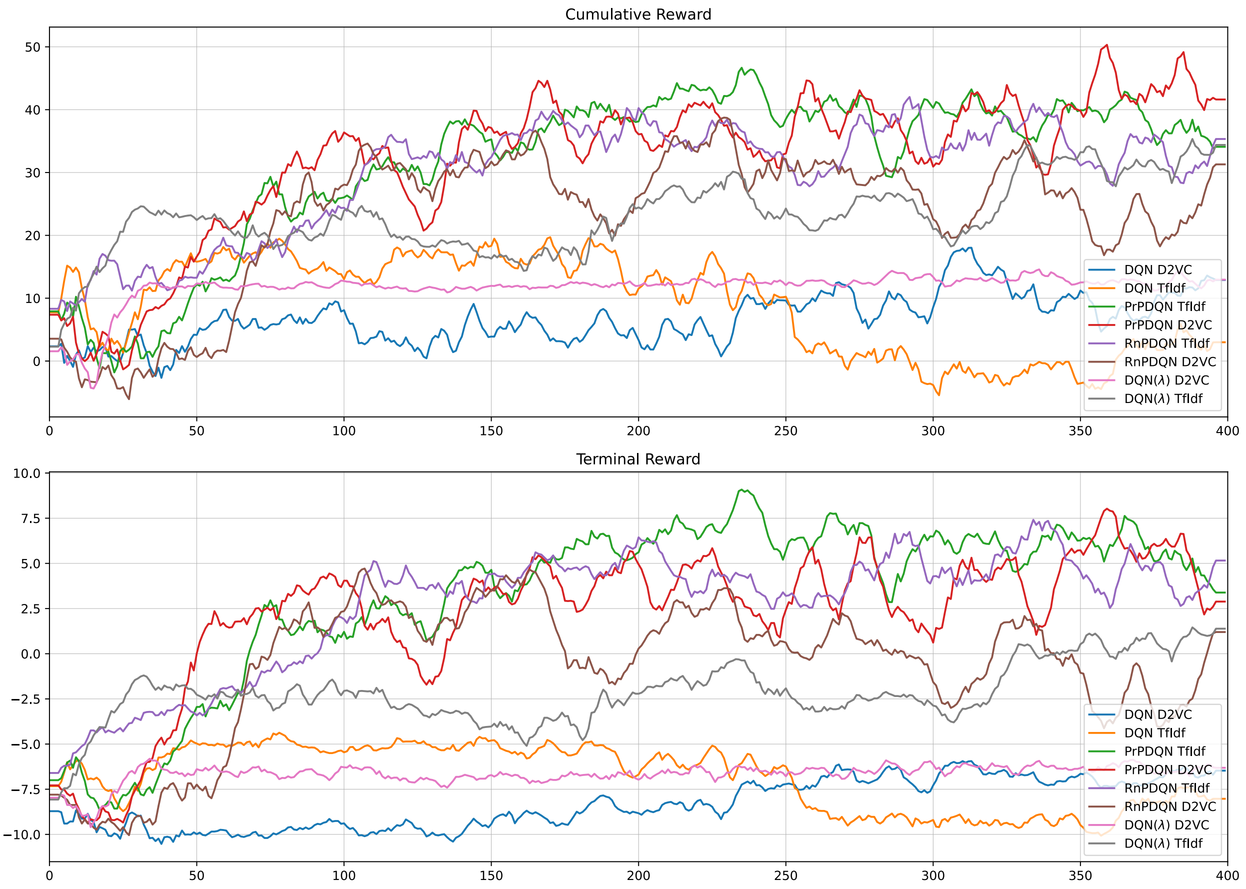


Figure 4.5: Reward evolution over training episodes. Continuous reward.

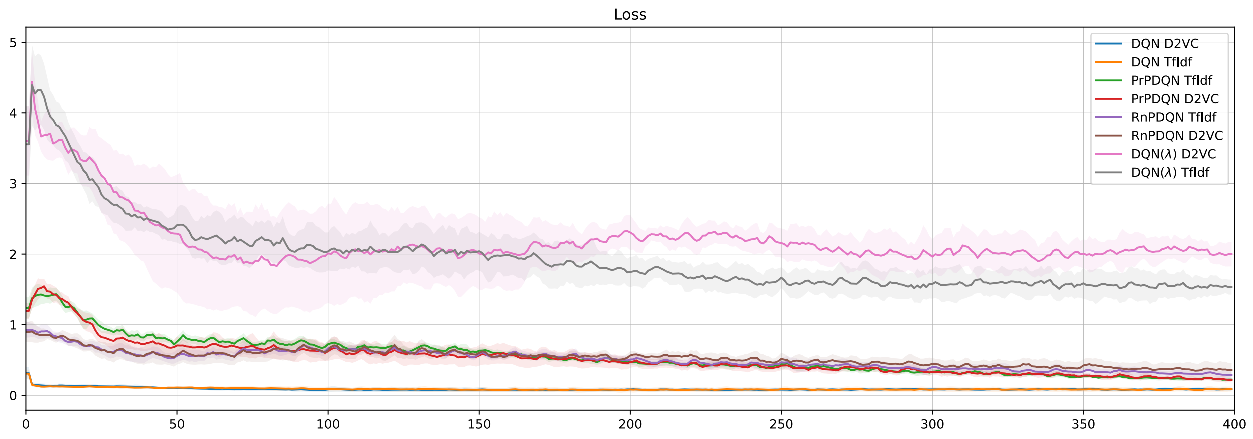


Figure 4.6: Loss evolution over training episodes. Discrete reward. The shaded areas are the 95% confidence intervals.

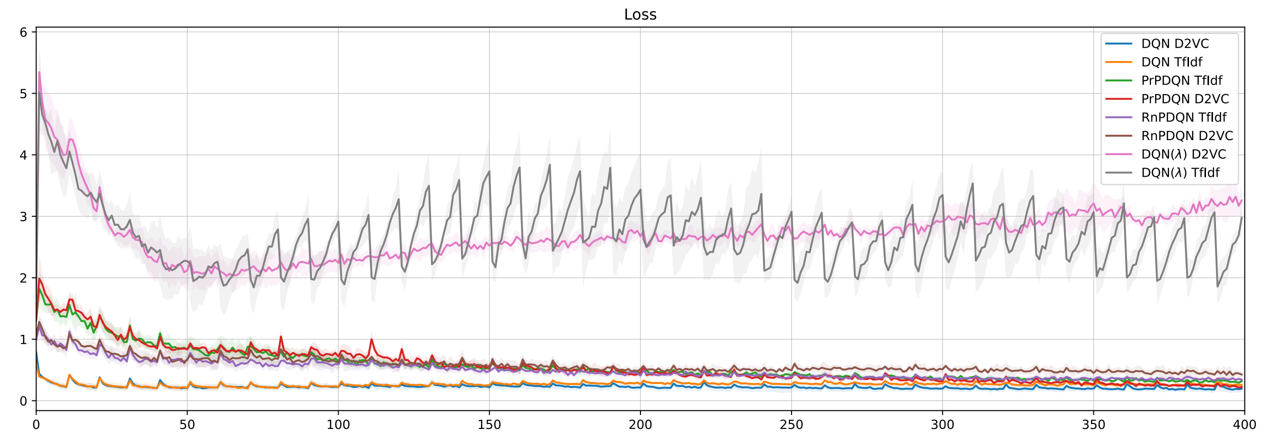


Figure 4.7: Loss evolution over training episodes. Continuous reward. The shaded areas are the 95% confidence intervals.

Evaluation

The evaluation task consists of comparing the performances attained on unseen samples by the best model, with respect to a random policy, which selects a action with equal probability at each step. The best model configuration, DQN(λ) with Tf-Idf embedding, is applied with decreasing ϵ, which is the probability of taking a random action instead of the greedy action. The goal is to show the evolution of the reward curves when the policy becomes more and more greedy.

As Figure 4.8 depicts, when the policy applied to the samples becomes closer and closer to the greedy policy, the reward increase, highlighting how the learned action values generalize well across different binaries. It is important to mention that the binaries belonging to this synthetic dataset are composed of the same building blocks, randomly selected in different order. Therefore, the performances shown on the test set may be too optimistic, compared to real-world scenarios. Still, this results underlines that this research direction is promising, which was the main goal of this work.

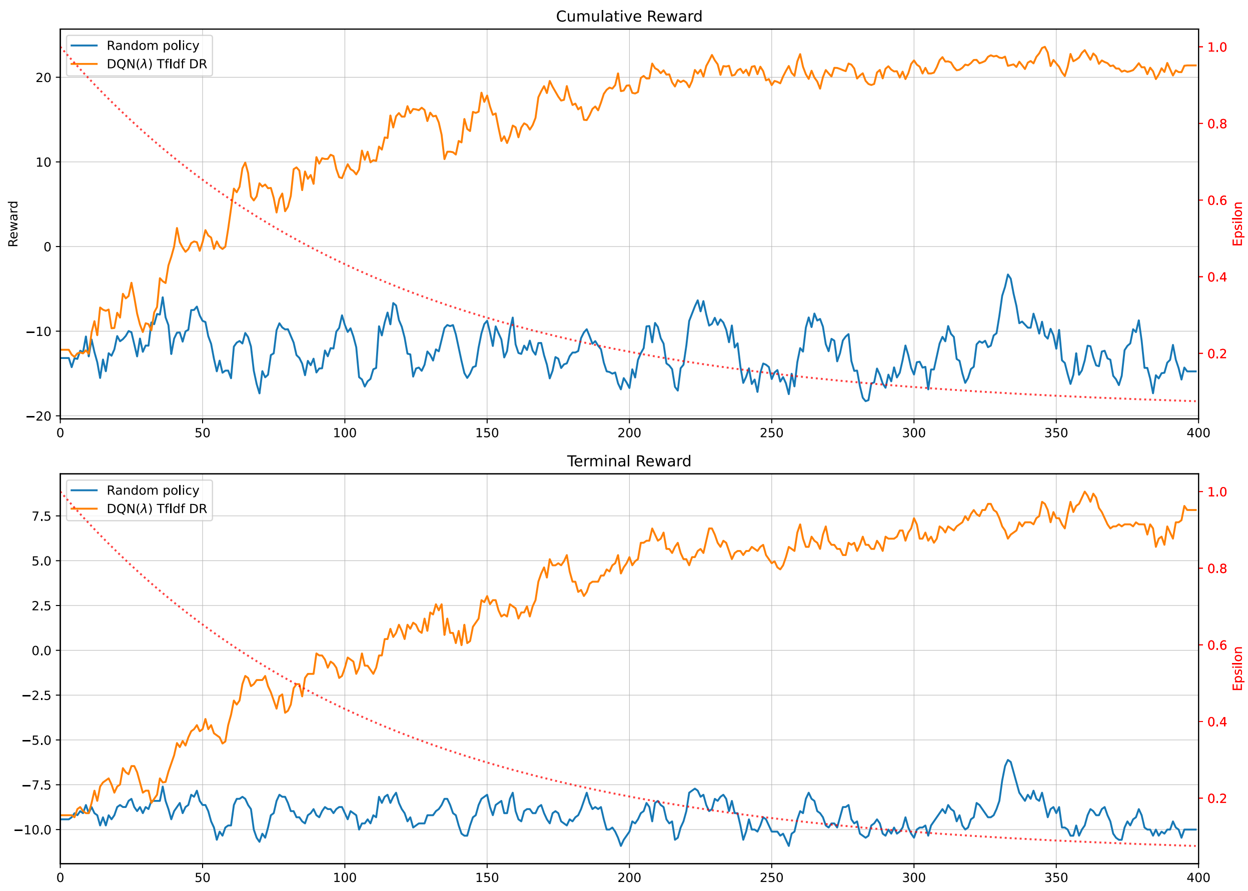


Figure 4.8: Performances on unseen samples. Best model versus greedy policy.

Ablation study

In this section, an ablation study is performed on the two best models, to analyze the contribution of each component, including the embedding method and the reward function. For DQN with prioritized experience replay, in Figure 4.9, employing a continuous reward function (CR) show a better cumulative reward, when compared to discrete reward (DR) models, which may be deceiving. In fact, this trend is flipped when analyzing the terminal reward, which is the most important one, being proportional to the number of hidden malicious capabilities found in a binary. The terminal reward is independent from the reward function used and offers, therefore, is a reliable measure to compare different approaches. Again, it is possible to see how models using Tf-Idf embedding often perform better than the ones utilizing Doc2VecC embedding.

A similar phenomenon is noticeable in Figure 4.10 for the DQN(λ) model. However, in this case the cumulative rewards are less correlated to different reward schemes, whereas the terminal reward is highly influenced by them. This may suggest that the longest horizon used to compute the lambda return is considerably affected by the continuous reward function, while being more prone to propagate the information coming form sparse rewards of the discrete reward function.

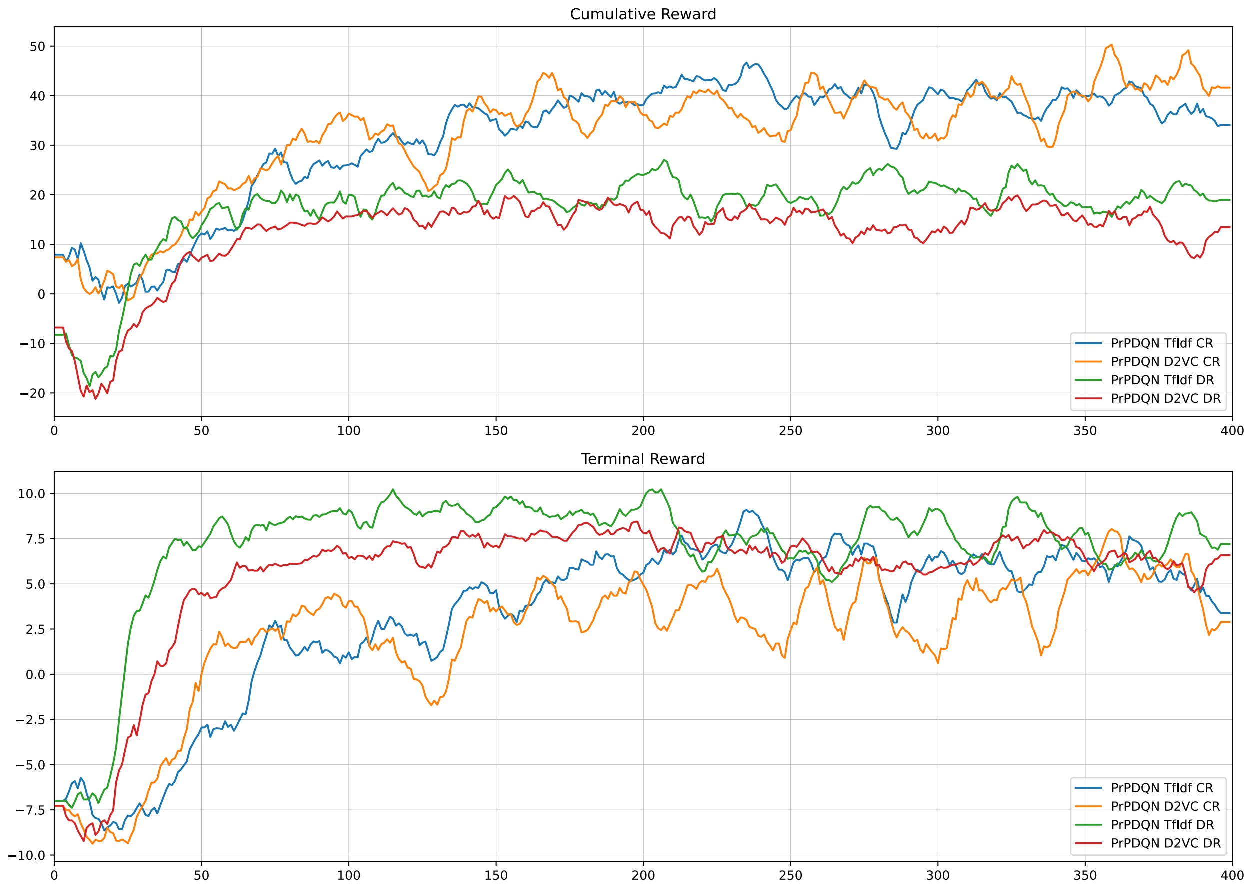


Figure 4.9: Comparison of different configurations of embedding method and reward function for the DQN model with proportionally prioritized experience replay. CR: continuous reward, DR: discrete reward.

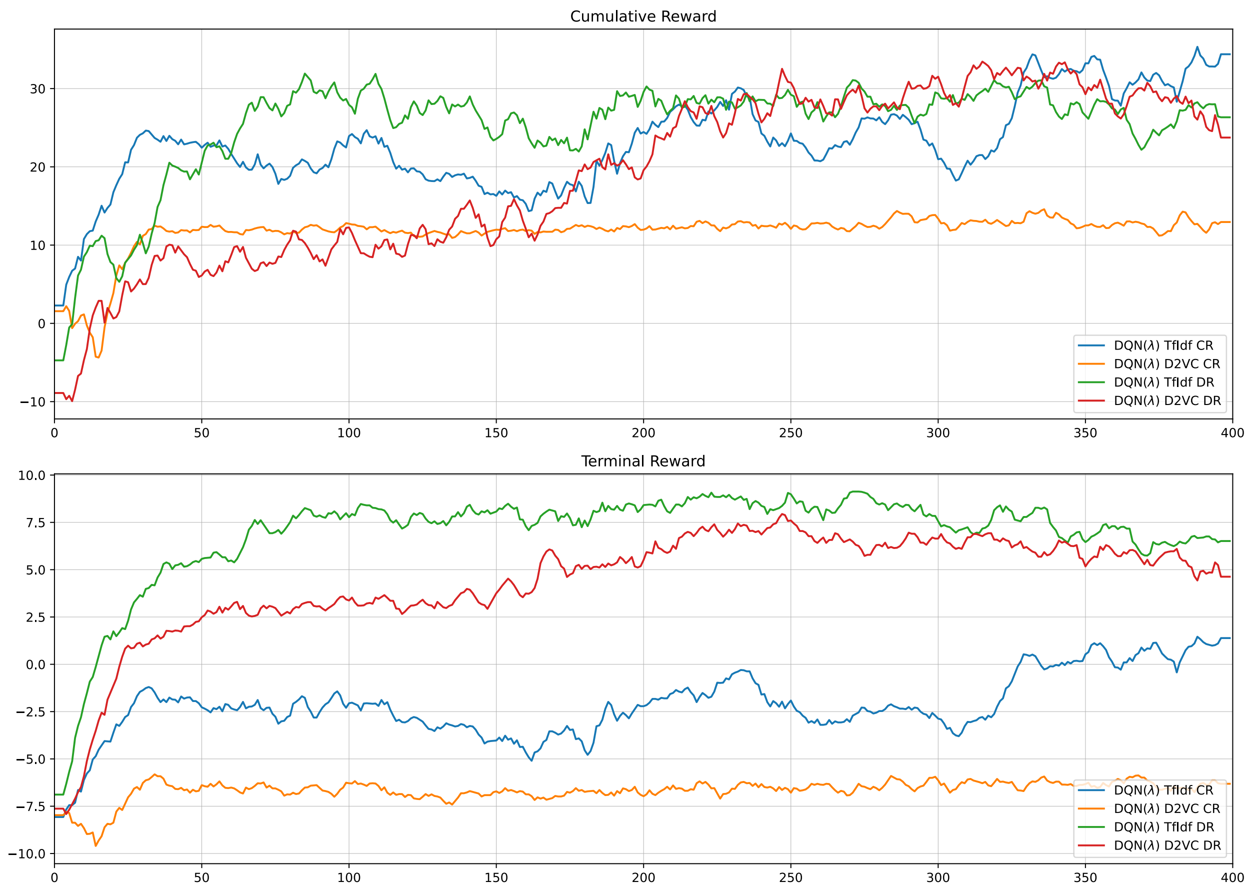


Figure 4.10: Comparison of different configurations of embedding method and reward function for the DQN(λ) model. CR: continuous reward, DR: discrete reward.

4.2.2 Discussion

The results achieved by PrDQN and DQN(λ) algorithms, on both training and evaluation tasks, show that it is indeed possible to train RL models on the task of finding hidden malicious capabilities in evasive binary samples. Although this work is a proof of concept, and the experimental setting cannot be directly compared with the ones employed by state-of-the-art works, it proves that this research direction is promising and deserves further work. Interesting research directions for future work include a deeper investigation of language models to be applied to the task of code block embedding. A larger language model, such as BERT [Dev+19], could be pretrained on a corpus of code blocks extracted from a large dataset of binaries, and fine-tuned on the RL task. The self-supervised trainig of the language model could include techniques from the *domain adaptation* field, learning the same embedding for code blocks with different syntax implementing the same behavior, following the intuition of [Com+10].

On the RL side, further work could explore alternative approaches, taking inspiration form the more advanced Alpha Go Zero [Sil+17], Agent 57 [Bad+20a] and Mu Zero [Sch+20].

Chapter 5

Conclusions

This work has been carried out in the framework of a research internship in the *AI4Sec* research team of Huawei's Munich Research Center. This team focuses its research on the application of machine learning methods to the challenging field of cyber security, with the goal of developing intelligent solutions able to cope with increasingly complex malware. Currently, one of the most promising trends in Machine Learning field is Reinforcement Learning (RL), which proved to be capable of solving relatively complex tasks, outperforming humans in some tasks. Some examples are Alpha Zero and Alpha Go, which beat Chess and Go word champions respectively. Some experts argue that RL is the form of machine learning methods closer to how humans learn, which in principle, makes it one of the most favourable candidates when aiming at Artificial General Intelligence. Malware analysis is a challenging topic which nowadays requires security analysts to perform most of the tedious and time-consuming work, when analyzing suspicious binaries.

The goal of this research internship has been to analyze the malware analysis domain, looking for a suitable application of RL, which may pave the way for future research. At present, the literature addressing RL applications to cyber security is scarce, which made this research hard and rewarding at the same time. In fact, the lack of literature, may reveal an appealing unexplored research niche, while at the same time it may be a symptom of an intrinsic difficulty of applying RL to this domain. This work proves that there is indeed some space for improvement over the state of the art, while acknowledging that the engineering effort required in this domain is considerably high, compared to other machine learning applications.

This thesis performs a comparative analysis of malware analysis techniques, following their temporal evolution, starting from static analysis until state-of-the-art dynamic analysis techniques. Before identifying a promising direction, I have explored the application of RL to static binary code deobfuscation, static API invocation deobfuscation, fuzzing and trigger-based evasive malware. To tackle trigger-based evasive malware I have decided to equip an RL agent with a debugger to dynamically interact with the malicious binary. The agent hijacks the conditions to explore the conditional graph, looking for hidden malicious behavior protected by evasive conditions. As discussed in the previous chapter, the training of RL methods is not free from pitfalls and the experimenter has to deal at the same time with multiple sources of instability. However, the proof of concept developed during my internship has achieved its main goal, namely proving that the chosen research direction is promising and deserves further research. The experiments carried out in this work highlight that RL methods can be successfully applied to the task of finding hidden malicious capability in evasive malware samples. The proposed model, supported by the achieved results, performs an additional step towards a solution to tackle dynamically evasive malware.

Appendix A

Implementation

This appendix describes the implementation of the RL environment. Section A.1 gives a broad overview of the developed modules' architecture. The first step has been the choice of the debugger, as explained in Section A.2. In Section A.2.1 are explained the semantics of a state, how a state is defined at low-level and how the hijacking action is implemented in the debugger. In Section A.2.2 I motivate why a jump back action is needed, I give a formal definition of it and I explain how it is implemented at a lower level. In Section A.2.3 is introduced a main data structure used by the debugger interpreter, the *conditions chain*, which keeps track of the exploration history. Eventually, Section A.3 describes the design of an high-level RL environment class, which will be used in the training notebook.

A.1 Architecture Overview

The architecture is composed of three main modules, as presented in Figure A.1: the debugger module, the debugger interpreter and the training notebook where the RL algorithms are implemented. The debugger runs the target binary and simulates the memory, the stack and the processor registers. It provides the tools to interact at low-level with the binary (e.g., stepping instruction-by-instruction, setting breakpoints and tampering with the memory of the debugged binary).

The debugger process is run as a sub-process of a Python process that operates with low-level debugger commands to extract information and aggregates it as states, after a feature engineering has been performed. The inverse operation is also performed in this stage, namely implementing the high-level actions performed by the agent as fine granularity operations on the debugged binary. Another key task of this module is to perform some backend operations to manage the data structures used to support the translation of high-level actions of the agent

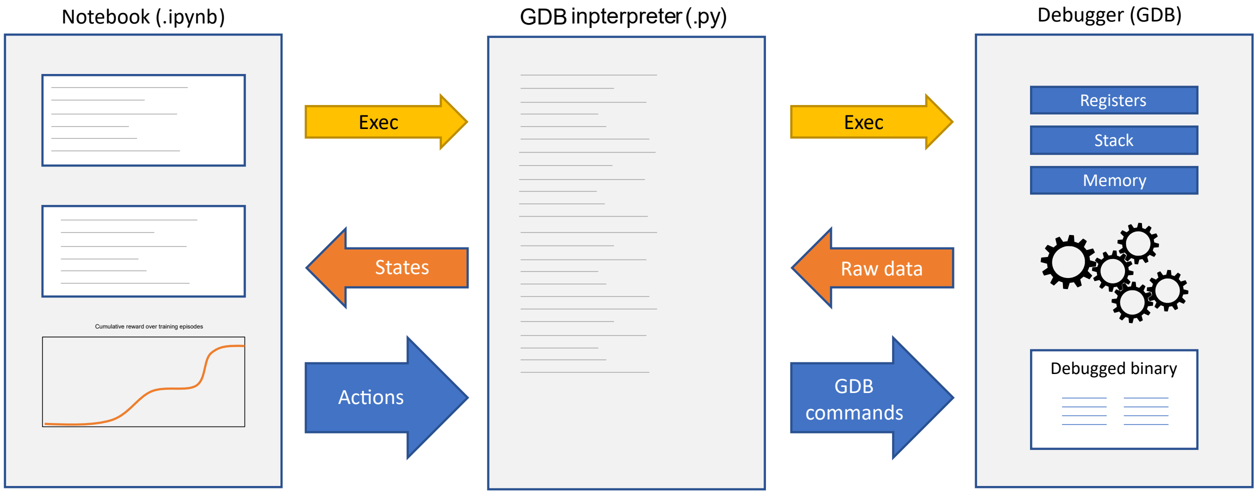


Figure A.1: The RL environment is defined in a notebook that executes the debugger interpreter module as a sub-process and communicates with it through piped `stdin` and `stdout`. In turn, the interpreter launches the debugger as a sub-process and employs a similar communication scheme.

in low-level debugger commands and vice versa. The main idea behind this interpreter is to perform features engineering and hide the low-level complexity so that the agent can perform higher level atomic actions, with just a command. It can be considered as a gateway used by the agent to access the environment. The importance of this interpreter is crucial since it is prohibitive to provide the RL agent with a raw binary as a state. Even splitting the binary in smaller blocks would entail a large dimensionality of the state space, making training unfeasible. The description of the implementation of this module is the main topic of this monthly report.

The RL training and evaluation process is carried out from a notebook where the debugger interpreter is launched as a sub-process. Here the agent receives states and performs actions. These are communicated with the interpreter sub-process through two pipelines on the sub-process' `stdin` and `stdout`.

A.2 Interaction With the Debugger

Currently, to the best of our knowledge, no tools are available for the task of interacting with a GDB via a RL agent. Therefore this required me to develop it, starting from an already existing debugger. I have explored available solutions and assessed their fit to this task depending on the following criteria:

- Developer API support.
- Documentation availability and clarity.
- Community support.
- Cross-platform support (e.g. Windows and Linux binaries).
- Previous personal experience.

I have compared the following debuggers: Immunity Debugger, OllyDbg, Radare's debugger and GDB. In the end, I have chosen GDB since it provides Python APIs, its documentation is complete and clear, the community is active and many examples or support can be found online. Furthermore, I have some experience on how to use it.

A key feature of GDB is its possibility to provide a machine-oriented interface for information-rich low-level interaction, using a format easily parsed by software. This is called GDB/MI interface. This feature made it possible to switch from using the internal Python APIs, which employ an outdated version of Python, to the `pygdbmi` Python library that launches the debugger as a sub-process and parses GDB machine interface string output, returning structured data types (Python dicts) that are JSON serializable. This library performs a bi-directional communication with the debugger sub-process, redirecting its `stdin` and `stdout` to two pipelines.

This Python library provides a reliable way of communicating with the debugger using structured Python dictionaries. Another advantage of relying on GDB/MI comes from the real-time asynchronous events notification. For instance, every time a new dynamic library is imported (loaded in memory or linked to the process memory), this machine interface returns a structured notification. Other examples can be: hit a break-point, termination of the executed binary.

A.2.1 Conditions

The task of the RL agent is mainly devoted to hijacking the control flow, namely tampering with the expected behavior of the conditional jump instructions (CJIs), until a termination condition is met. Therefore, each RL environment state has to be associated with a different conditional jump instruction (CJI). To provide the RL agent with more context information about the CJI, both the features of a conditional jump and the preceding instructions, back to the previous conditional jump instruction, are given to the agent as a state. This enables the agent to gather additional semantic information about the code block that contains the CJI.

Algorithm 4 describes the logic behind the GDB interpreter package. In particular, the algorithm explains how the instructions, disassembled step-by-step (line 4), are analyzed and aggregated to build the next state, how the states and the actions are communicated on the sub-process pipelines (lines 16 and 17) and how the received actions are implemented.

Every time a CJI is encountered, the features collected up to this point are organized as a state and sent to the agent. A state contains also ϕ_{CJI}, the details of the currently encountered CJI, which include a bit which is set to 1 if the CJI is going to jump, 0 otherwise. While the agent is stepping instruction-by-instruction, a function call may be encountered or an asynchronous event may happen. In this stage, any usual asynchronous event (e.g., loading of dynamic library or termination of the program) is properly handled.

When a `call` instruction is encountered, the call target is analyzed: it may be an API or a user-defined function within the examined binary. Depending of the type of call target, the debugger interpreter can perform two actions:

- Step-over the function code. The debugger executes the function in background and waits for it to finish. This command tells the debugger to treat a function as a single instruction. This is the preferred behavior when encountering a known library function, since its content and its semantics are familiar. An example of functions that do not need to be analyzed by the agent, stepping-into them, are APIs and system calls.

- Step-into the function code. The debugger jumps to the first instruction of the function and enables the agent to analyze it instruction-by-instruction. This is favourable when the binary is calling an unknown function that may hide some malicious behavior: since its content is unknown, this kind of function is always explored.

The experimenter decides which are the call targets that the agent has to explore, stepping-into them. Algorithm 4 also introduces the *Jump Back* action: when the agent performs this action it returns to the previously visited state, which is the previous condition. Section §A.2.2 explains in more detail how this action is performed.

Hijacking the CJI

CJIs perform a jump to an address location depending on the outcome of a logic operation performed on the flags register. The action performed by the agent by hijacking the control-flow is equivalent to tampering with the flags register's bits to revert the outcome of this logical expression. As shown in Algorithm 4, the first step consists of identifying whether the CJI performs a jump, then a proper tampering is performed to either invalidate or not the conditional expression checked by the CJI.

Algorithm 4 Generic interaction loop

```
Input: $\pi$ = Entry point                              ▷ Initialize the program counter
 1: while Execution not finished do
 2:     $\phi \leftarrow \{\}$                          ▷ Initialize state's features representation
 3:     while True do
 4:         $I \leftarrow$ disassembleNextInstruction($\pi$)      ▷ Step instruction-by-instruction
 5:         $\phi \leftarrow \phi +$ genericFeatures($I$)   ▷ Generic features engineering of current instruction
 6:         $\pi \leftarrow \pi +$ len($I$)     ▷ Increment the program counter: point to next instruction
 7:         if $I$ is call then                 ▷ If the instruction is a calling a function
 8:             $\phi \leftarrow \phi +$ analyzeCallTarget($I$)    ▷ Add called function information to features
 9:         end if
10:         if $I$ is CJI then
11:             $\phi_{CJI} \leftarrow$ analyzeCji($I$)           ▷ Compute CJI features.
12:             $\phi \leftarrow \phi + \phi_{CJI}$               ▷ Add CJI information to features
13:             break        ▷ Halt the step-by-step disassembling. A new state is ready
14:         end if
15:     end while
16:     sendState($\phi$, stdout)          ▷ Send current state to agent (stdout pipeline)
17:     $\alpha \leftarrow$ receiveAction(stdin)      ▷ Receive action from agent (stdin pipeline)
18:     switch $\alpha$ do
19:         case Hijack
20:             if isJumping($I$) then                 ▷ Check if the CJI is going to jump
21:                 preventJump($\phi_{CJI}$)      ▷ Tamper with flags register to prevent jump
22:             else
23:                 forceJump($\phi_{CJI}$)          ▷ Tamper with flags register to force jump
24:             end if
25:         case Jump back
26:             $\pi \leftarrow$ jumpBack($\pi$)         ▷ Set program counter to previous condition
27:         case Quit
28:             quit()      ▷ Terminate the execution of debugged binary and debugger
29: end while=0
```

Example A.1 includes a conditional jump instruction and identifies the flags to be updated to force or prevent a jump.

Example A.1. *Conditional Jump Instruction:* `jle`.
Jump if: `ZF = 1 or SF != OF`
To prevent the jump (invalidate the expression): `set ZF = SF = OF = 0`
To force the jump: `set ZF = 1`.

Instruction Blocks as States

A state is defined as the instructions block that begins after a CJI and terminates with the *active*[1] CJI. A state is associated with the active CJI since the agent has to take an action to influence that CJI. Without loss of information, to simplify the implementation, this definition has been relaxed by including the previous CJI in the instructions block as well. Thus in this early stage of experiments, a state is defined as the following.

Definition A.1. *State.*
A state is a block of assembly instructions, such that the fist instruction of the block is the previously encountered CJI whereas the last instruction is the last encountered CJI, which is also the active instruction (the one currently referenced by the program counter). The other instructions composing the state are the ones encountered executing the binary instruction-by-instruction, from one CJI to the other. A state can be enriched by adding an additional bit, that represents whether the active CJI instruction is going to jump (1) or not (0).

An example of (GDB's disassembled) code arranged as a block, associated with a condition, to define a state is shown in Listing A.1.

It is important to underline that the binary is executed instruction-by-instruction and each state is put together one instruction at a time, without resorting to static code analysis to gather information from the next instructions or code blocks that have not been executed yet. This is because some static analysis evasions may be present, such as Control-flow Graph obfuscations or hijackings. Some examples are non-returning calls, call stack tampering, exception-based control-flow hijack. The reader is referred to [RM13] for a detailed explanation. I assume that the debugger is immune to static obfuscations, being a dynamic analysis tool.

In the first version of the RL environment, which perform a naïve features engineering, a state is represented as a categorical value defined as the hash of the tuple `(previous CJI address, current CJI address, current CJI jump intention)`.

[1]The active CJI is the CJI currently referenced by the program counter.

```
0x000000000000118d <main+68>:    je     0x11ac <main+99>
0x000000000000118f <main+70>:    movl   $0x0,-0xc(%rbp)
0x0000000000001196 <main+77>:    lea    0xe6d(%rip),%rdi
0x000000000000119d <main+84>:    call   0x1050 <puts@plt>
0x00000000000011a2 <main+89>:    addl   $0x1,-0xc(%rbp)
0x00000000000011a6 <main+93>:    cmpl   $0x2,-0xc(%rbp)
0x00000000000011aa <main+97>:    jle    0x1196 <main+77>
```

Listing A.1: The *current* CJI is at the address `<main+97>` and the instructions that make its context are the ones in range `[<main+70>, <main+93>]`. The instructions block that constitutes the current state is defined by the range `[<main+68>, <main+97>]`.

Example A.2. *State features engineering.*
Following the example in Listing A.1:
S_i = `hash((<main+68>, <main+97>, True))`

A.2.2 Jump Back Action

Throughout the exploration of the binary, the agent may discover that it had undertaken a wrong path that may likely lead to nothing interesting. Therefore, the agent is equipped with the ability of jumping back to the explored conditions' path at any moment. This capability is motivated by both the necessity of reducing the exploration cost and the desire of giving the agent a sufficient flexibility of exploring the binary control-flow, looking for the hidden target flag. This prevents the need of restarting a debugging episode from scratch every time a wrong path is taken. Jumping back may reduce the burden of path exploration when the agent can infer from the states' features that going further is no more interesting and an alternative path should have been taken in previous steps.

Definition A.2. *Valid jump back.*
A jump back operation is valid when it restores the program counter[2] to a previously-executed condition and can be reverted by a *single* action: either hijacking it or not.

The jump back operation should ensure that, when resuming the previous condition, the simulated memory, stack and registers are fully restored, reflecting the same internal state when

[2]The program counter, also called instruction pointer, is the register that stores the address of the currently executed instruction.

that condition was encountered the first time. This is not trivial to ensure because it would require to dump the whole GDB simulated memory and reload it when needed. To address this issue in a more elegant way I have resorted to GDB's built-in checkpointing functionality.

GDB Checkpoints

At a low-level, GDB performs the checkpoint of the debugged process memory by invoking on it the `fork()` system call. This will spawn a new child process that will be kept hanging until the checkpoint is resumed. Performing a `fork()` for each encountered condition becomes infeasible as the number of conditions in the binary grows requiring more memory, making it hard in practice to work with real-world binaries. To overcome this issue I have introduced a tunable hyperparameter that defines the frequency of checkpointing every time a condition is met, leaving some conditions un-checkpointed. When one of these has to be resumed because of a jump back action, the closest checkpointed condition among the previous ones is resumed and the binary is executed in background, without any features collection, until the desired condition's code block is met. Reducing the number of checkpoints lowers the memory needs for the RL environment execution, but it has the drawback of increasing the latency when jumping back to an un-checkpointed condition, due to the overhead of executing some code in background before reaching the desired condition. Every time a checkpoint is resumed a new one is created for future resuming, whereas its parent process is killed (unless it is the debugger process), to free memory space. When a jump back is performed, the last condition and its active checkpoints are freed from memory. Algorithm 5 contains the pseudo-code for the resumption procedure of a checkpoint.

As checkpoints are not available on the Windows' GDB version, I will use the Linux environment to run any future experiment.

Algorithm 5 resumeCheckpoint(): Resuming a checkpointed condition

```
Input: $C_{ID}$                                  ▷ Receive the checkpoint ID to resume
  $\pi \leftarrow$ runCheckpoint($C_{ID}$)                                ▷ Restart checkpoint
  $P_{ID} \leftarrow$ activeCheckpoint()          ▷ Get parent process ID. Can be a checkpoint
  if $P_{ID} \neq$ getDebuggerPID() then    ▷ If the parent process is not the debugger process
      kill($P_{ID}$)                                 ▷ Kill the parent checkpoint process
  end if
  setActiveCheckpoint($C_{ID}$)                    ▷ Broadcast the new parent process ID
  $C_{ID} \leftarrow$ createCheckpoint()                          ▷ Create a new checkpoint
Output: $C_{ID}, \pi$                ▷ Return the new checkpoint ID and the program counter
```

A.2.3 Conditions Chain Exploration

The main data structure that supports the exploration of the dynamic Control-flow Graph, while keeping track of the visited conditions, is the *conditions chain*. This is a list of conditions that records the whole exploration history, enabling the agent to jump back to the previously visited conditions.

The conditions chain maintains general statistics about the set of visited conditions. Every time a new condition is encountered, it is appended to the tail of this chain and every time a jump back is performed, the tail condition is removed and the execution is resumed at the previous condition. The agent can jump back any number of times in a row: if the agent tries to jump back from the head of the conditions chain, no jump back will be performed and it will remain in the same state.

To reduce the memory needs of this RL environment only one condition every k is checkpointed. When the agent jumps back to a condition that was not checkpointed, the checkpoint of the closest checkpointed condition among the previous ones is resumed. Subsequently, the debugger runs the binary until the condition that precedes the target one is encountered. From this point on, the default behavior is restored: the debugger steps instruction-by-instruction and collects features regarding the encountered assembly instructions. The operation of stepping one instruction at a time will halt when the target CJI is encountered. At this point, the information gathered stepping instruction-by-instruction is returned as a state, following the procedure presented in Algorithm 4. Example A.3 shows how a jump back to an un-checkpointed condition is performed in the conditions chain. In this example, not all the conditions have a checkpoint.

Example A.3. *Jump back to un-checkpointed condition.*
Conditions chain: $(\mathbf{C_1}, C_1, C_2, C_3, \mathbf{C_4}, C_5, C_6, C_7)$
Checkpoint frequency: $k = 4$. The checkpointed conditions are in bold.
Active condition: C_7. It is the one currently referenced by the program counter.
The jump back operation will (1) remove C_7 from the conditions chain; then (2) resume the closest checkpointed condition, $\mathbf{C_4}$; subsequently (3) run in background the instructions between $\mathbf{C_4}$ and C_5 and eventually (4) step instruction-by-instruction between C_5 and C_6 arranging the collected information as a code block. The resulting block is returned as the state associated with the condition C_6, according to Definition A.1.

The integrity of jump back, according to Definition A.2, has to be guaranteed even in presence of loops, namely when the same condition is encountered multiple times before jumping to a new unexplored code block. The algorithm appends a condition address to the conditions chain every time a condition is met. However, in a loop the same condition is met a number of times equal to the number of iterations, resulting in appending multiple times the same condition to chain. When a jump back is performed, the execution is resumed from the penultimate condition in the chain, which in this case would bring the agent to the same condition. This would violate Definition A.2. A visual example of this is presented in Example A.4.

Example A.4. *Repeated append of the same condition in the chain.*
Consider the listing below:

```
char curr_time[20];

// First evasive condition: C1
if (! is_debugger_attached()){

    // Loop until the second evasive condition, C2, is met
    do{
        get_curr_time(curr_time);
    }while(strcmp(curr_time, "11:59") != 0);

    // Third evasive condition: C3
    if(strcmp(gethostname(), "CHOOKOO") == 0){
        // Check last evasive condition: C4
        if (are_private_files_present()){
            // Perform the malicious action
            steal_data();
        }
    }
}
else{
    // A debugger is analyzing me! Evade
    exit(0);
}
```

Listing A.2: Example of evasive malware that checks an evasive condition in a loop.

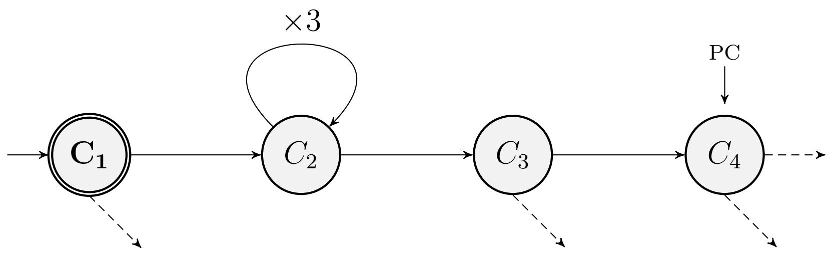
If a condition is appended to the conditions chain every time it is met, after n iterations of the do-while loop in Listing A.2, the conditions chain would be $(C_1, C_2', ..., C_2^n)$. At this point, the program counter references the address of C_2 and preforming a jump back would resume the execution from the second-last condition, which is still C_2, violating Definition A.2.

If we prevent a condition from being repeated multiple times in a row in the conditions chain, the integrity of the jump back is not guaranteed as well. In fact, when resuming a condition C_n after a loop, when the previous checkpointed condition C_{n-j} is before the loop, the repetition of the execution history between C_{n-j} and C_n will perform exactly j steps, getting trapped in the loop from some of them. As a consequence, the jump back will bring the agent to a condition farther than expected, back in the execution history. This jump back will not be reverted by any single action, violating Definition A.2. An example of this is explained in Example A.5.

Example A.5. *Naïve jump back in presence of loops.*
This example is based on the the behavior described in Listing A.2. If a condition is never appended twice in the conditions chain, when the agent reaches the fourth evasive condition, the conditions chain is: (C_1, C_2, C_3, C_4). Assuming C_1 is the only checkpointed condition, jumping back from C_4 to C_3 involves the resumption of the checkpoint of C_1 and the execution in background of the instructions between C_1 and C_3. When executing the instructions in background starting from C_1, the algorithms computes the length of the conditions chain and expects to reach the desired condition after 2 conditions are met: C_2 and C_3. However, C_2, which is checked in the while loop, is encountered n times in a row. As a consequence, the execution in background will get trapped in the loop and the jump back will likely resume the execution from C_2 rather than C_3. This violates Definition A.2.

This requires the introduction of a further sophistication of the jump back mechanism. To tackle the loops problem, each condition is equipped with a *memory* that keeps track of all the times that condition had been met in a row and the actions the agent took accordingly. This prevents the repetition of the same condition in the conditions chain and when the execution history between two conditions is repeated, a condition is replicated a number of times equal to the number of records in its memory. This enables to always correctly resume the execution history between to non-contiguous conditions on the condition chain, even in presence of loops, guaranteeing that the jump back operation is always satisfying Definition A.2. Algorithm 6 summarizes in a structured way the behavior of the jump back action explained above. Example A.6 shows a use case of the implementation of the jump back action on a simple Control-flow Graph containing a loop.

Example A.6. *Jump back in presence of loops (with memory).*
Consider the conditions graph depicted in the figure below, which represents the behavior of the binary when the conditions $\mathbf{C_1}$, C_2, C_3 and C_4 are encountered. This is a way to represent in a more abstract way the code presented in Listing A.2. The condition C_2 is checked in a loop that performs 3 iterations, whereas the dashed arrows represent the conditional paths not visited. The condition $\mathbf{C_1}$ is the only one with a checkpoint. The program counter (PC) is pointing to the tail of the conditions chain, namely the last encountered condition, C_4.



The corresponding conditions chain with *memory* is:

$$\Big(\big(\mathbf{C_1}, (h_{1,1})\big), \big(C_2, (h_{2,1}, h_{2,2}, h_{2,3})\big), \big(C_3, (h_{3,1})\big), \big(C_4, (h_{4,1})\big)\Big)$$

where $h_{i,j}$ stores the j-th interaction of the agent with condition C_i.
When jumping back from C_4 to C_3, the checkpoint in $\mathbf{C_1}$ is resumed and the following interactions are replayed in background:

$$\big((\mathbf{C_1}, h_{1,1}), (C_2, h_{2,1}), (C_2, h_{2,2}), (C_2, h_{2,3})\big)$$

whereas the code between the last occurrence of C_2 and C_3 is executed instruction-by-instruction, as explained in Algorithm 4, generating a state for the condition C_3. Jumping back from C_2 to $\mathbf{C_1}$ in one step is possible since the condition C_2 is not replicated in the conditions chain. When jumping back from C_3 to C_2 two scenarios are possible: resuming C_2 at the first iteration of the loop or at the last iteration. In the first case, the conditions chain would be:

$$\Big(\big(\mathbf{C_1}, (h_{1,1})\big), \big(C_2, (h_{2,1})\big)\Big)$$

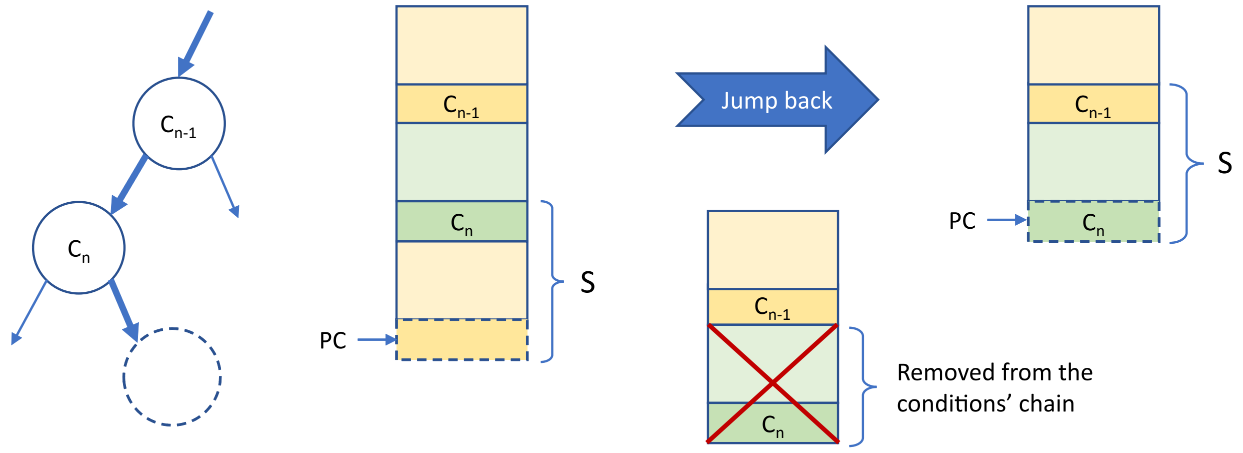
whereas in the second case it would look like:

$$\Big(\big(\mathbf{C_1}, (h_{1,1})\big), \big(C_2, (h_{2,1}, h_{2,2}, h_{2,3})\big)\Big)$$

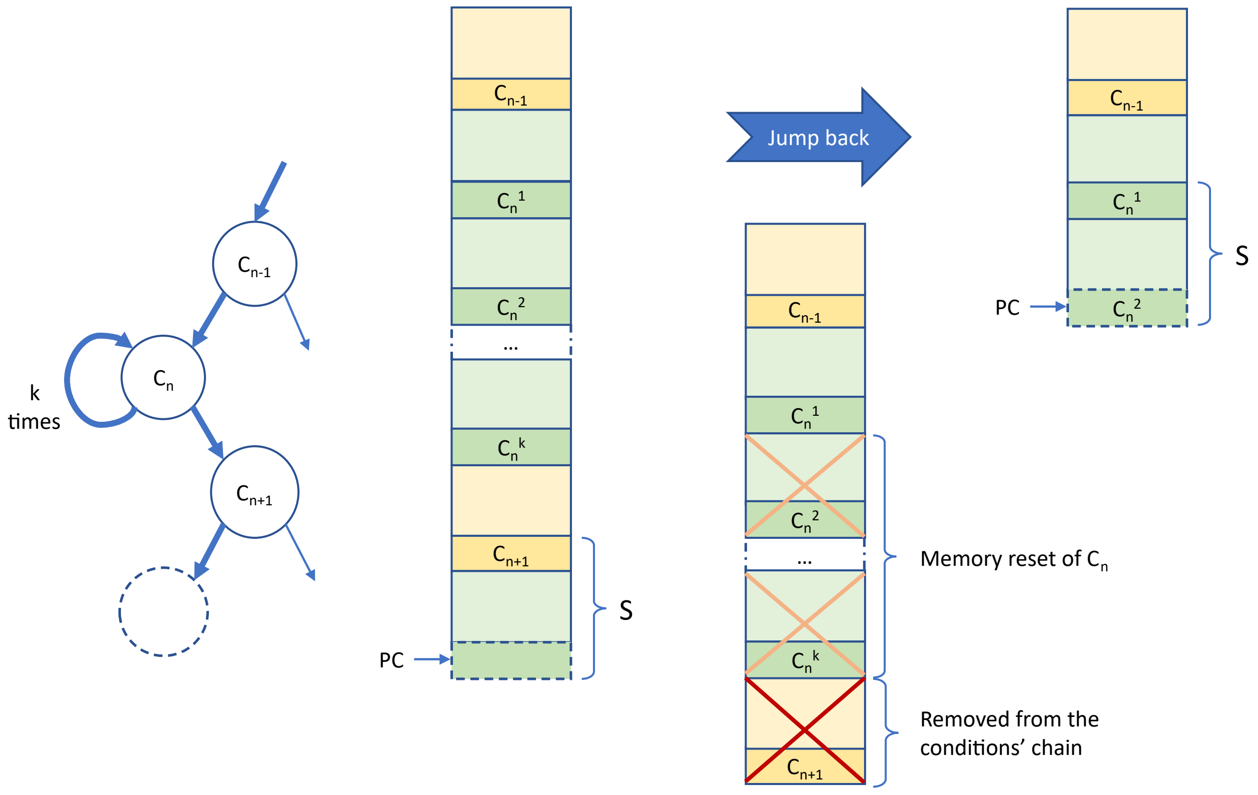
In principle, there is not a preferable option among the two. Since the agent has the ability to break a loop in any moment, we want the agent to take the shortest path to the hidden flag. Hence, we deem preferable the first option. The action of clearing the memory of a condition entry, leaving just the first record, is referred in Algorithm 6 as *memory reset*.

When jumping back to a loop condition that has in its memory n interaction records, the condition's memory is cleared keeping only the first record. This operation, is defined as *memory reset* in Algorithm 6, brings the agent back to the first iteration of the loop. This is a preferable approach since we want the agent to find the shortest path to the hidden flag. Resuming the previous loop condition from the first iteration gives the agent the chance to break the loop soon, if needed.

There are two scenarios where a jump back operation has to be suited to: with a linear conditions sequence and in presence of a loop. A linear sequence of conditions is the result of a sequence of nested if-then-else constructs: their CFG is a tree with no backward edges (loops). As a consequence, the conditions chain is simple and the jump back operation is trivial. In presence of a loop, the target condition's memory has to be reset and a special attention has to be devoted to the replay of the execution history among two non-contiguous conditions, in the conditions chain. It is important to prevent the execution in background from being trapped in loops. Figure A.2 shows the visualizations of these two scenarios.



(a) Jump back in a linear condition sequence.



(b) Jump back in a cyclic condition sequence.

Figure A.2: Jump back scenarios. The box above a condition in the conditions chain represents the disassembled instructions between the previous condition and the current one. S is the state that is sent to the agent after a proper features engineering, according to §A.2.1, whereas PC is the program counter. Different colors for different contiguous conditions.

Algorithm 6 jumpBack()

```
Input: $CC, \pi$                                   ▷ Conditions chain ($CC$) and program counter ($\pi$)
  free($CC[-1]$)                                   ▷ Free last condition in the chain
  $C \leftarrow$ lastCheckpointedCondition($CC$)   ▷ Get last checkpointed condition
  $\pi \leftarrow$ resumeCheckpoint($C$.id())      ▷ Resume last checkpointed condition's checkpoint
  resetMemory($CC[-1]$)                            ▷ Reset memory of the tail condition
  while $C != CC[-1]$ do                           ▷ While current condition is not the tail
      $\pi, I \leftarrow$ runUntil($C$)            ▷ Execute until the condition's address is reached
      while hijackedMemory($C$).length $! = 0$ do  ▷ Until there is memory for condition $C$
          Hijacked $\leftarrow$ hijackedMemory($C$).pop(0)   ▷ Pop next action from memory
          if Hijacked then                         ▷ If the memorized action was an hijack
              $\phi_{CJI} \leftarrow$ analyzeCji($I$)   ▷ Get CJI features
              if isJumping($I$) then               ▷ Check if the CJI is going to jump
                  preventJump($\phi_{CJI}$)        ▷ Tamper with flags register to prevent jump
              else
                  forceJump($\phi_{CJI}$)          ▷ Tamper with flags register to force jump
              end if
          end if
      end while
      $C \leftarrow$ getNext($CC$, $C$)            ▷ Get next condition in the chain after $C$
  end while
Output: $CC, \pi$                                  ▷ Updated chain and program counter
```

A.3 Reinforcement Learning Environment

Following the structure of OpenAI's gym Python library, I have developed an environment class that implements the reward function according to the MDP formalization in Section 3.3.3. Each state is built according to the definition provided in the same section.

The environment class hides the details of the communication with the debugger interpreter. It sends the actions to the debugger interpreter, receives the resulting states and computes the reward according to a reward function. This class enables the agent to interact with the environment by means of two methods: `reset()` and `step(action)`. The first one resets the environment restarting the debugger interpreter, the debugger and the debugged binary sub-processes, which are also depicted in Figure A.1. This method will return the initial state of the binary. The `step(action)` method receives an action and forwards it to the debugger interpreter, waiting for the next state. This method returns the next state features representation, a reward, a flag that indicates whether the state is terminal and a dictionary object for additional information that may be used for secondary tasks.

Bibliography

[Afi+19] Amir Afianian et al. “Malware Dynamic Analysis Evasion Techniques: A Survey”. In: *ACM Computing Surveys* 52.6 (Nov. 14, 2019), 126:1–126:28. ISSN: 0360-0300. DOI: 10.1145/3365001. URL: https://doi.org/10.1145/3365001.

[Ahm+21] Mohsen Ahmadi et al. *MIMOSA: Reducing Malware Analysis Overhead with Coverings*. 2021. arXiv: 2101.07328 [cs.CR].

[ALM17] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. “A Simple but Tough-to-Beat Baseline for Sentence Embeddings”. In: *ICLR*. 2017.

[And+18] Hyrum S. Anderson et al. *Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning*. Jan. 30, 2018. arXiv: 1801.08917 [cs]. URL: http://arxiv.org/abs/1801.08917.

[Bad+20a] Adrià Puigdomènech Badia et al. *Agent57: Outperforming the Atari Human Benchmark*. Mar. 30, 2020. arXiv: 2003.13350 [cs, stat]. URL: http://arxiv.org/abs/2003.13350.

[Bad+20b] Adrià Puigdomènech Badia et al. *Never Give Up: Learning Directed Exploration Strategies*. Feb. 14, 2020. arXiv: 2002.06038 [cs, stat]. URL: http://arxiv.org/abs/2002.06038.

[BGS18] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. *Deep Reinforcement Fuzzing*. Jan. 14, 2018. arXiv: 1801.04589 [cs]. URL: http://arxiv.org/abs/1801.04589.

[Bru+08] David Brumley et al. “Automatically Identifying Trigger-Based Behavior in Malware”. In: *Botnet Detection: Countering the Largest Security Threat*. Ed. by Wenke Lee, Cliff Wang, and David Dagon. Advances in Information Security. Boston, MA: Springer US, 2008, pp. 65–88. ISBN: 978-0-387-68768-1. DOI: 10.1007/978-0-387-68768-1_4. URL: https://doi.org/10.1007/978-0-387-68768-1_4.

[BY17] Alexei Bulazel and B. Yener. "A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web". In: *ROOTS*. 2017. DOI: 10.1145/3150376.3150378.

[Che+18] Binlin Cheng et al. "Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 15, 2018, pp. 395–411. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243771. URL: https://doi.org/10.1145/3243734.3243771.

[Che+21] Binlin Cheng et al. "Obfuscation-Resilient Executable Payload Extraction From Packed Malware". In: 30th {USENIX Security Symposium ({USENIX Security 21). 2021, pp. 3451–3468. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/cheng-binlin.

[Che17] Minmin Chen. "Efficient Vector Representation for Documents through Corruption". In: July 7, 2017. arXiv: 1707.02377 [cs]. URL: http://arxiv.org/abs/1707.02377.

[Com+10] Paolo Milani Comparetti et al. "Identifying Dormant Functionality in Malware Programs". In: *2010 IEEE Symposium on Security and Privacy*. 2010 IEEE Symposium on Security and Privacy. May 2010, pp. 61–76. DOI: 10.1109/SP.2010.12.

[Coz+18] Emanuele Cozzi et al. "Understanding Linux Malware". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018 IEEE Symposium on Security and Privacy (SP). San Francisco, CA: IEEE, May 2018, pp. 161–175. ISBN: 978-1-5386-4353-2. DOI: 10.1109/SP.2018.00054. URL: https://ieeexplore.ieee.org/document/8418602/.

[CS20] Emily J. Chaffey and D. Sgandurra. "Malware vs Anti-Malware Battle - Gotta Evade 'em All!" In: *2020 IEEE Symposium on Visualization for Cyber Security (VizSec)* (2020). DOI: 10.1109/VizSec51108.2020.00012.

[DA19] Brett Daley and Christopher Amato. "Reconciling \lambda -Returns with Experience Replay". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://papers.nips.cc/paper/2019/hash/9f396fe44e7c05c16873b05ec425cbad-Abstract.html.

[DEl+19] Daniele Cono D'Elia et al. "SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed)". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, July 2, 2019, pp. 15–27. ISBN: 978-1-4503-6752-3. DOI: 10.1145/3321705.3329819. URL: https://doi.org/10.1145/3321705.3329819.

[DEl+20] Daniele Cono D'Elia et al. "On the Dissection of Evasive Malware". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 2750–2765. ISSN: 1556-6021. DOI: 10.1109/TIFS.2020.2976559.

[Dev+19] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. May 24, 2019. arXiv: 1810.04805 [cs]. URL: http://arxiv.org/abs/1810.04805.

[Dow+21] Evan Downing et al. "{DeepReflect: Discovering Malicious Functionality through Binary Reconstruction". In: 30th USENIX Security Symposium (USENIX Security 21). 2021, pp. 3469–3486. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/downing.

[Fan+19] Zhiyang Fang et al. "Evading Anti-Malware Engines With Deep Reinforcement Learning". In: *IEEE Access* 7 (2019), pp. 48867–48879. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2908033.

[Gal+22] Nicola Galloro et al. "A Systematical and Longitudinal Study of Evasive Behaviors in Windows Malware". In: *Computers & Security* 113 (Feb. 1, 2022), p. 102550. ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102550. URL: https://www.sciencedirect.com/science/article/pii/S0167404821003746.

[Gar+07] Tal Garfinkel et al. "Compatibility Is Not Transparency: VMM Detection Myths and Realities". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. HOTOS'07. USA: USENIX Association, May 7, 2007, pp. 1–6.

[Kaw+13] Yuhei Kawakoya et al. "API Chaser: Anti-Analysis Resistant Malware Analyzer". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 123–143. ISBN: 978-3-642-41284-4. DOI: 10.1007/978-3-642-41284-4_7.

[KIM18] Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshu. *Taint-Assisted IAT Reconstruction against Position Obfuscation*. 2018. URL: https://www.jstage.jst.go.jp/article/ipsjjip/26/0/26_813/_article.

[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. “Renovo: A Hidden Code Extractor for Packed Executables”. In: *Proceedings of the 2007 ACM Workshop on Recurring Malcode*. WORM ’07. New York, NY, USA: Association for Computing Machinery, Nov. 2, 2007, pp. 46–53. ISBN: 978-1-59593-886-2. DOI: 10.1145/1314389.1314399. URL: https://doi.org/10.1145/1314389.1314399.

[Kuc+21] Alexander Kuchler et al. “Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes”. In: *Network and Distributed System Security (NDSS) Symposium,* NDSS 21. Feb. 2021.

[KVK11] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. “BareBox: Efficient Malware Analysis on Bare-Metal”. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC ’11. New York, NY, USA: Association for Computing Machinery, Dec. 5, 2011, pp. 403–412. ISBN: 978-1-4503-0672-0. DOI: 10.1145/2076732.2076790. URL: https://doi.org/10.1145/2076732.2076790.

[KVK14] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. “BareCloud: Bare-metal Analysis-based Evasive Malware Detection”. In: 23rd USENIX Security Symposium (USENIX Security 14). 2014, pp. 287–301. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat.

[Lin92] Long-Ji Lin. “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching”. In: *Machine Learning* 8.3 (May 1, 1992), pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699.

[LKM11] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. “Detecting Environment-Sensitive Malware”. In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 338–357. ISBN: 978-3-642-23644-0. DOI: 10.1007/978-3-642-23644-0_18.

[LM14] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. May 22, 2014. arXiv: 1405.4053 [cs]. URL: http://arxiv.org/abs/1405.4053.

[Maf+21] Lorenzo Maffia et al. *Longitudinal Study of the Prevalence of Malware Evasive Techniques*. Dec. 21, 2021. arXiv: 2112.11289 [cs]. URL: http://arxiv.org/abs/2112.11289.

[Man+19] Valentin J. M. Manes et al. *The Art, Science, and Engineering of Fuzzing: A Survey*. Apr. 7, 2019. arXiv: 1812.00140 [cs]. URL: http://arxiv.org/abs/1812.00140.

[MCJ07] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware". In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). Dec. 2007, pp. 431–441. DOI: 10.1109/ACSAC.2007.15.

[Men+21] Grégoire Menguy et al. *AI-Based Blackbox Code Deobfuscation: Understand, Improve and Mitigate*. Feb. 9, 2021. arXiv: 2102.04805 [cs]. URL: http://arxiv.org/abs/2102.04805.

[Mik+13] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. Sept. 6, 2013. arXiv: 1301.3781 [cs]. URL: http://arxiv.org/abs/1301.3781.

[Mir+17] Najmeh Miramirkhani et al. "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP). May 2017, pp. 1009–1024. DOI: 10.1109/SP.2017.42.

[Mni+13] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Version 1. Dec. 19, 2013. arXiv: 1312.5602 [cs]. URL: http://arxiv.org/abs/1312.5602.

[NR20] Thanh Thi Nguyen and Vijay Janapa Reddi. *Deep Reinforcement Learning for Cyber Security*. July 21, 2020. arXiv: 1906.05799 [cs, stat]. URL: http://arxiv.org/abs/1906.05799.

[Pal+09] Roberto Paleari et al. "A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators". In: *In Proceedings of the USENIX Workshop on Offensive Technologies (WOOT*. 2009.

[PBM17] Dorottya Papp, Levente Buttyán, and Zhendong Ma. "Towards Semi-automated Detection of Trigger-based Behavior for Software Security Assurance". In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. New York, NY, USA: Association for Computing Machinery, Aug. 29, 2017, pp. 1–6. ISBN: 978-1-4503-5257-4. DOI: 10.1145/3098954.3105821. URL: https://doi.org/10.1145/3098954.3105821.

[Pen+14] Fei Peng et al. "{X-Force: {Force-Executing Binary Programs for Security Applications". In: 23rd USENIX Security Symposium (USENIX Security 14). 2014, pp. 829–844. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[Pet+18] Matthew E. Peters et al. *Deep Contextualized Word Representations*. Mar. 22, 2018. arXiv: 1802.05365 [cs]. URL: http://arxiv.org/abs/1802.05365.

[Pol+17] Mario Polino et al. "Measuring and Defeating Anti-Instrumentation-Equipped Malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, 2017*. 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assess, DIMVA 2017. Springer, Jan. 1, 2017, pp. 73–96. DOI: 10.1007/978-3-319-60876-1_4. URL: https://research.utwente.nl/en/publications/measuring-and-defeating-anti-instrumentation-equipped-malware.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. EMNLP 2014. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

[Raf+17] Edward Raff et al. *Malware Detection by Eating a Whole EXE*. Oct. 25, 2017. arXiv: 1710.09435 [cs, stat]. URL: http://arxiv.org/abs/1710.09435.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. "Detecting System Emulators". In: *Information Security*. Ed. by Juan A. Garay et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 1–18. ISBN: 978-3-540-75496-1. DOI: 10.1007/978-3-540-75496-1_1.

[RM13] Kevin A. Roundy and Barton P. Miller. "Binary-Code Obfuscations in Prevalent Packer Tools". In: *ACM Computing Surveys* 46.1 (July 11, 2013), 4:1–4:32. ISSN: 0360-0300. DOI: 10.1145/2522968.2522972. URL: https://doi.org/10.1145/2522968.2522972.

[Roy+06] P. Royal et al. "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware". In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (2006). DOI: 10.1109/ACSAC.2006.38.

[RŠL21] Michele Russo, Nedim Šrndić, and Pavel Laskov. “Detection of Illicit Cryptomining Using Network Metadata”. In: *EURASIP Journal on Information Security* 2021.1 (Dec. 4, 2021), p. 11. ISSN: 2510-523X. DOI: 10.1186/s13635-021-00126-1. URL: https://doi.org/10.1186/s13635-021-00126-1.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Red. by Francis Bach. 2nd ed. Adaptive Computation and Machine Learning Series. Cambridge, MA, USA: A Bradford Book, Nov. 13, 2018. 552 pp. ISBN: 978-0-262-03924-6.

[Sch+16] Tom Schaul et al. *Prioritized Experience Replay.* Feb. 25, 2016. arXiv: 1511.05952 [cs]. URL: http://arxiv.org/abs/1511.05952.

[Sch+20] Julian Schrittwieser et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *Nature* 588.7839 (Dec. 24, 2020), pp. 604–609. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4. arXiv: 1911.08265. URL: http://arxiv.org/abs/1911.08265.

[Sil+17] David Silver et al. “Mastering the Game of Go without Human Knowledge”. In: *Nature* 550.7676 (7676 Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270.

[TIR+20] Pietro Francesco TIRENNA et al. “Techniques for Malware Analysis Based on Symbolic Execution”. July 2020. URL: https://webthesis.biblio.polito.it/15305/1/tesi.pdf.

[Uga+15] Xabier Ugarte-Pedrero et al. “SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers”. In: *2015 IEEE Symposium on Security and Privacy.* 2015 IEEE Symposium on Security and Privacy. May 2015, pp. 659–673. DOI: 10.1109/SP.2015.46.

[Van+21] Charles-Henry Bertrand Van Ouytsel et al. *Analysis of Machine Learning Approaches to Packing Detection.* May 2, 2021. arXiv: 2105.00473 [cs]. URL: http://arxiv.org/abs/2105.00473.

[Vas+17] Ashish Vaswani et al. *Attention Is All You Need.* Dec. 5, 2017. arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning.* Dec. 8, 2015. arXiv: 1509.06461 [cs]. URL: http://arxiv.org/abs/1509.06461.

[Wan+16] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. Apr. 5, 2016. arXiv: 1511.06581 [cs]. URL: http://arxiv.org/abs/1511.06581.

[Xu10] Xin Xu. “Sequential Anomaly Detection Based on Temporal-Difference Learning: Principles, Models and Case Studies”. In: *Appl. Soft Comput.* (2010). DOI: 10.1016/j.asoc.2009.10.003.

[ZLC20] Lan Zhang, Peng Liu, and Yoon-Ho Choi. *Semantic-Preserving Reinforcement Learning Attack Against Graph Neural Networks for Malware Detection*. Dec. 29, 2020. arXiv: 2009.05602 [cs]. URL: http://arxiv.org/abs/2009.05602.