Università degli Studi di Verona

Dipartimento di Informatica

Ph.D. Thesis

Intelligent Agents for Active Malware Analysis

Author:
Riccardo Sartea

Advisor:
Prof. Alessandro Farinelli

May 7, 2020

Abstract

The increasing use of digital infrastructures makes cyber-security a key issue for society. In particular, malicious software, i.e., malware, represent one of the biggest threats nowadays. For this reason, studying the behavior of malware is extremely important to defend against them. In this context, malware analysis aims at grouping malicious software with respect to common behaviors or to a predefined set of classes. In fact, malware can be grouped in families (or types), that are behavioral categories in which malicious software fall into. The analysis process is critical in order to be able to build countermeasures empowering defense systems such as antivirus, firewalls or Intrusion Detection Systems (IDSs). Identifying which category a new malware belongs to allows to use existing countermeasures to defend against such newly discovered threat. Current approaches for malware analysis heavily rely on Artificial Intelligence (AI) techniques, since it would be impossible for human security experts to manually analyze the millions of new malware discovered every day. Many of such malware are more sophisticated than others, as they require some kind of inputs (or events) to meet the condition for their activation. Human security experts rely on their expertise to choose the best sequence of inputs to extract the most information about a malware. In this perspective, AI approaches based on multi-agent systems can be a crucial asset as they allow to model the analysis as a process involving interacting agents, i.e., the analyzer and the malware in this case.

On this background, Active Malware Analysis (AMA) is a methodology that aims at acquiring knowledge about malware by executing actions on the system. Such actions are selected to trigger a response in the malicious software. Specifically, the interaction involves an analyzer agent whose aim is to acquire information on the malware agent, that instead wants to perform its malicious goals. The main strength of this approach is the capability of extracting behaviors that would otherwise remain invisible to a simple passive observation. A key problem for AMA is to design intelligent strategies that allow to select the most informative actions for the analysis, in contrast to the majority of the existing techniques that perform random or predefined triggering actions. While there are many works on static analysis (where the source or binary code of the malware is inspected without executing it) and many others on dynamic analysis (where the malware runs inside a controlled environment and is observed), only a few employ AMA and even less use intelligent strategies.

The main contribution of this thesis is to give a novel perspective on AMA modeled as a decision making process between intelligent agents. We propose solutions aimed at extracting the behaviors of malware agents with advanced AI techniques. In particular, we devise novel action selection strategies for the analyzer agents that allow to analyze malware by selecting sequences of triggering actions aimed at maximizing the information acquired. The goal is to create informative models representing the behaviors of the malware agents observed while interacting with them during the analysis process. Such models can then be used to effectively compare a malware against others and to correctly identify the malware family.

More in detail, the contributions can be divided in three major areas

1. *Dynamic generation of malware behavioral models:* we propose a formalization for the malware behavioral model based on a composition of multiple Markov chains and devise a new action selection strategy for the analyzer based on Monte Carlo tree search. This allows us to dynamically generate the malware model at runtime while interacting with the malware, without the specific need of any prior knowledge

2. *Long-term behavioral analysis:* we propose to use the long-term transition probability values of moving between states of the Markov chain as features. Additionally, we design a transformation procedure for the behavioral models to enforce the absorbency property of Markov chains, enabling the use of standard techniques to compute the long-term transition probability

3. *Bayesian Active Malware Analysis:* we propose a new formalization for AMA based on Bayesian games, focusing on the decision making strategy for the analyzer. The formalization is built upon the link between malware families and the notion of types in Bayesian games. A key point is the design of the utility function, which reflects the amount of uncertainty on the type of the malware after the execution of an analyzer action. This allows us to identify the malware family with fewer triggering actions by devising an algorithm to play the game with the aim of minimizing the entropy of the analyzer's belief

All the solutions presented in this thesis have been implemented in a comprehensive framework for AMA that has been used to analyze a dataset of real Android malware. Overall, our approaches provide a significant contribution to malware analysis in a large and significant scale, and offer a new perspective to the synergy between cybersecurity and artificial intelligence.

Contents

Chapter 1

Introduction

In recent years, the increasing reliance on computer systems and the increasing use of internet, wireless networks, autonomous systems, as well as the growth of smart and tiny devices as part of the Internet of Things (IoT), resulted in a corresponding increase in the number of cyber-security flaws. Automated techniques for malware analysis are fundamental to help human security experts in studying the huge amount of new threats discovered every day and that would be otherwise impossible to handle, e.g, about 150 million of new malware discovered in 2019 only considering Windows and Macintosh operating systems (AV-Test, 2019). Therefore, AI based tools are a necessity that allow to speed up the process, providing as an output the family of the malware analyzed based on the similarity with known malicious software of the same type. This is a crucial step to take the required countermeasures that avoid (or limit) the damages caused by malware. Such countermeasures are then implemented in real time defense systems such as antivirus, firewalls, IDSs, etc. Therefore, all the information extracted with malware analysis is extremely valuable as it serves as a basis for cyber-security defenses.

In this thesis we propose automated techniques for AMA to aid with the manual analysis of an unknown software that may be malicious, i.e., performing interaction tests with the software while adapting to what the defense system can observe during such tests. Among the preferred targets of malware there is the Android system, which is one of the most diffused operating system on the planet empowering billions of smartphones and IoT devices. Consequently, the majority of new malware released on a daily basis are aimed at attacking the Android platform (Cheung, 2018). For this reason we focus on malware analysis for the Android system, as it would benefit the most from new research at security level. Nonetheless, the majority of techniques presented in this work can be applied to other systems as well since the theoretical approaches proposed are platform independent.

1.1 Malware analysis

Malware analysis is a crucial part of cyber-security as it involves acquiring information about unknown malicious software. Observing and understanding the behaviors of malware allows to build effective countermeasures to avoid damages to systems and people relying on them. In particular, one of the goals of malware analysis is to infer the type of a malicious software with respect to common behaviors or to a predefined set of types. Indeed, malware can be grouped in families (or types), that are behavioral categories in which malicious software fall into (Elisan, 2015). The grouping is fundamental, as if a malware is regarded as being part of a known existing family, the same countermeasures already developed for the samples belonging to such a family could probably be used to defend against the new malware. Moreover, new defense strategies can be developed starting from the existing ones if a malware sample is recognized as similar to those of a certain category. This approach is effective as recently, the focus in malware design has shifted from the creation of new types of malicious payloads, i.e., the code slice of a malware aimed at causing harm, to the engineering of the stealthy system that makes the malware to remain hidden, while the payload is reused from older deployed malware as is or with minor modifications (Upchurch and Zhou, 2016).

There are two fundamental approaches to malware analysis: static and dynamic. Static analysis examines a malware without actually running it. Dynamic analysis instead executes the malware to observe its behavior within a safe and controlled environment (a sandbox). A common problem of static methodologies comes from analyzing malware with encrypted malicious code deployed at runtime or obfuscated. Since encrypted or obfuscated code is not in a readable form (unless the decryption key or the transformation applied to obfuscate are known), static code inspection routines are unable to extract viable information. On the other hand, since the malware does not need to be run, static analysis is usually faster and can inspect pieces of code that may not be always executed at runtime. Dynamic analysis instead is capable of inspecting encrypted or obfuscated code since it focuses on the actions performed on the system by the malware without looking at the code. Nonetheless, dynamic analysis suffers from some limitations. For example it is difficult to observe executions of the program that cover the entire code, i.e., the code coverage problem. However, the code coverage limitation is much less prominent in Android malware analysis since what is observed is usually relevant in the overall behavior, as the software are developed for the specific smartphone usage. Dynamic analysis is also slower than the static one since it requires to execute a program and wait for interesting behaviors to become visible.

Actually, until the recent past, the majority of automated dynamic analysis techniques were passive, meaning that no interaction was performed during the analysis process: the malware was executed and its behavior observed for a fixed amount of time. However, some malware are more sophisticated and often require inputs in order to show their malicious behaviors. In these cases, the payload is released only after a triggering action is performed on the system, and most often different parts of the payloads are sensitive to different triggers, i.e., multiple triggers are required to observe all the malicious behaviors of a malware (Moser et al., 2007). Figure 1.1 shows a possible threat model of a malware that steals messages from an infected smartphone and forwards them to an external server under the control of the attacker. The malicious behavior is observed only when a new message arrives, otherwise the malware remains inert and hidden.

Usually, if a malware requires any user inputs to show its payload, a human security expert would manually study it in order to perform the triggering actions based on what he believes to be the most informative, i.e., those actions that allow to acquire the most information on the malicious behaviors. This process requires intelligence and is extremely time consuming. In this context, recent work propose the use of AMA techniques in which autonomous agents powered by AI try to simulate what a human security expert would do when analyzing malware by actively performing triggering actions. However, most of the techniques for AMA present no rational target-oriented strategy to stimulate the malware under analysis: they either reproduce specific activation conditions to trigger malicious payloads relying on past samples of real user behaviors that have been recorded (Suarez-Tangil, Conti, et al., 2014), or perform a sequence of random triggering actions (Bhandari et al., 2018; A. Martìn et al., 2018) hoping to trigger some interesting reactions.

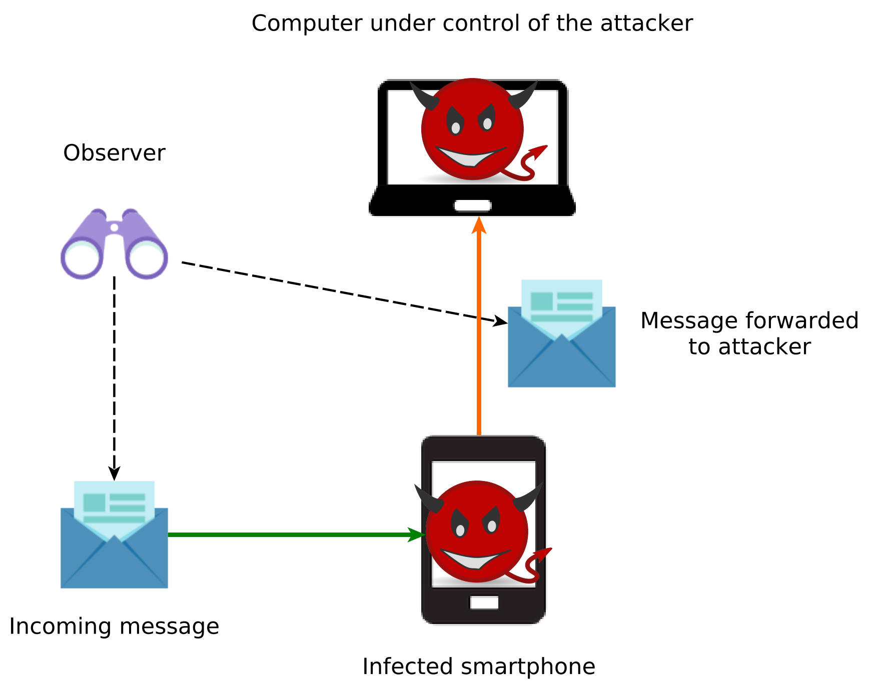

Computer under control of the attacker
Observer
Message forwarded to attacker
Incoming message
Infected smartphone

FIGURE 1.1: Threat model of malware responding to an incoming message as triggering action

1.2 Intelligent active malware analysis

An intelligent strategy for action selection can make a huge difference in the analysis result, in particular considering the number of analyzer actions required to reach a good level of malware identification. Using Figure 1.1 as an example, if the analyzer successfully triggers the malware simulating an incoming message to the smartphone, the malicious behavior can be observed. Now, in some cases the malware reacts to a trigger only with some probability, sometimes slightly changing the execution trace (although not the final goal of forwarding the message). Therefore it makes sense for the analyzer to repeat the same trigger enough times to gain a reasonable confidence about the malware response. When this happens, to continue in performing the same trigger again does not give the analyzer valuable information anymore, since she already knows what happens in response. Conversely, selecting another triggering action could be much more valuable in terms of information gain.

In this perspective, our efforts are aimed at designing novel AMA techniques in which the malware and the analyzer are modeled as intelligent agents that interact during the analysis process. This formalization allows to use a broad variety of AI techniques for multi-agent systems in which intelligent and autonomous entities interact within a complex dynamic environment learning information and adapting their behavior accordingly. As in standard malware analysis, the goal is to group the malware with respect to the family they belong to. However, an important point is that such grouping can be performed by considering the runtime behaviors of the malware encoded as policies, i.e., a mapping between states and actions. A first attempt to realize an intelligent strategy for the analysis is presented in (Williamson et al., 2012), where AMA is formalized as a stochastic game in which the analyzer performs an action on the system and the malware responds with a sequence of actions that change the status of such system. The work of (Williamson et al., 2012) is only a first step in this direction as it suffers of some limitations, the main one being the requirement of a fixed model manually designed by a human security expert. With our solutions instead, we go beyond such preliminary work as the malware behavioral models are dynamically generated at runtime, opening to a wide range of possible methodological contributions, some of which are investigated in this thesis, while others are left for future research.

1.3 Contributions of this thesis

The main contribution of this thesis is to give a novel perspective on AMA aimed at extracting the behaviors of malware agents with advanced AI techniques. In particular, we devise new AMA approaches based on intelligent decision making strategies for the analyzer agents in order to select triggering actions that allow to acquire the highest amount of information possible on the adversary.

1.3.1 Dynamic generation of malware behavioral models

In the first part of the thesis we define a behavioral model for malware that represents the observed execution traces, composed by sequences of Application Programming Interface (API) calls, as Markov chains. Therefore, the model does not depend on the analysis system (except for the operating system that must obviously remain the same to make a comparison meaningful, e.g., Android), but only on the malicious software itself. For example, it is possible to compare models generated on an old version of Android, to models generated on a newer version that has additional functionalities and components. The models can be compared and grouped so as to apply standard classification algorithms to identify the respective malware family.

In order to be able to dynamically generate the malware behavioral models at runtime, we develop a novel Reinforcement Learning (RL) algorithm based on Monte Carlo Tree Search (MCTS) to implement the strategy for the analyzer. This allows to compute the value of an analyzer action, with an entropy-based reward, by simulating the possible responses of the malware in terms of execution traces. Information theory is useful to decide which action the analyzer should try to acquire the most information based on the current state of the malware model that is being generated. This allows to create models that are as informative as possible, hence more distinctive when multiple malware have to be grouped or compared to each other.

1.3.2 Long-term behavioral analysis

We previously explained why grouping malware of the same type together is important for the analyzer. As a consequence, sophisticated malware try to prevent analyzers from gathering information that would allow to perform such grouping either aborting execution (or not releasing the payload) if an emulated environment is detected, or employing more complex stealthy mechanisms. Malware injection and dynamic obfuscation are two specific types of stealthy mechanisms: in the first, malware are inserted in the code of a (usually bigger) benign application, so that the malicious part represents a small portion of the overall behavior; in the latter the malware performs random actions or other non-harmful behaviors while releasing the payload, in order to confuse the analyzer.

Figure 1.2 shows the process of injection performed by a malicious agent. The red circles can be the malicious behavior that an analyzer wants to identify or the noise that must be cleared out in order to focus only on the behavior represented by the

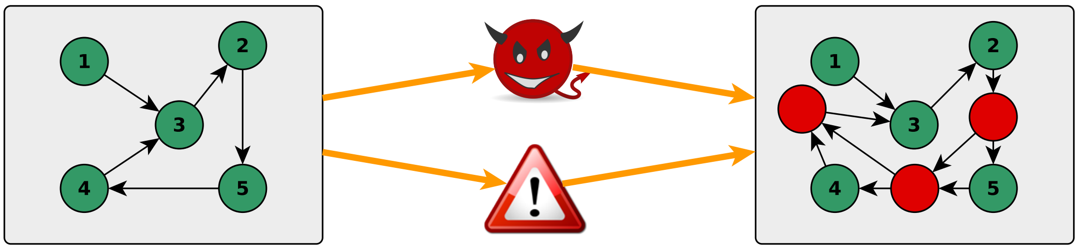


FIGURE 1.2: Example of injection within a behavioral model performed by a malicious agent on purpose. In the case of malware injection, the analyzer is interested only in the malicious behavior represented by the red circles, whereas for dynamic obfuscation the analyzer wants to focus only on the overall behavior, filtering out the red circles

green circles. Notice how the original connections between the green circles on the left-hand side are still present in the right-hand side, although harder to identify because of the injection of red circles in between. In this context, we propose a long-term analysis of the behavioral models that exploits some well known properties of Markov chains in order to lessen the impact of noise in the classification. Focusing on the long-term allows to compute the probability of going from every state s_i to any other state s_j when the process depicted by the Markov chain reaches a stable configuration (a fixpoint), regardless of the states that are in between the couple (s_i, s_j). Our fixpoint of choice is that of complete absorption of the process, guaranteed to be reached for any Markov chain holding the absorbency property. However, the behavioral models we deal with are almost never absorbing since they are generated through observation of the actions of an agent we have no control on. For this reason, we devise a transformation algorithm for the Markov chain that enforces the absorbency property starting from any generic Markov chain and that allows to extract the long-term features of interest. Although the motivation is mainly to analyze malware injection or to bypass dynamic obfuscation mechanisms, our approach is designed in order to be applied to any Markov chain based model, independently of the application domain or from the generation process. Notice that being able to filter out noise is useful also for cases where a genuine agent that is executing a difficult task deviates from her policy by mistake, consequently inserting noise in the behavioral model.

1.3.3 Bayesian Active Malware Analysis

An intelligent analyzer should adapt to what she observes during the analysis process. In our scenario, we would like to adapt the choice of triggering actions to the type of malware the agent is facing. As malware families represent different categories of behavior, it also means that different families usually respond to different set of triggers. The association between family and triggers is information available a priori since datasets usually report this, as well as any antivirus report about a malware sample. In this perspective, we propose a new formalization for AMA as a Bayesian game that, in contrast to existing techniques, exploits the information about the malware family (type) at runtime, in the form of a belief that the analyzer agent maintains on the type of malware she is facing. The aim is to select triggering actions more effectively by updating the belief accordingly to the malware reaction.

Although Bayesian games have been previously used in the context of security games (Tambe, 2011; Jain et al., 2008; H. Xu et al., 2016), this is the first time (to the best of our knowledge) that they are applied to malware analysis, in particular investigating the link between malware family and the notion of types in Bayesian games.

1.3.4 Summary of the contributions

In summary, the contributions that this thesis makes to the state-of-the-art are the following:

1. In the context of the dynamic generation of malware behavioral models
   - We define a behavioral model based on multiple Markov chains where each one represents the observation of a behavior in response to a specific analyzer action. This allows the analyzer agent to compute the value of an action with respect to the malware reaction and consequently compute a decision making strategy to conduct the analysis
   - We develop a RL approach for AMA based on MCTS that can dynamically generate the malware behavioral model at runtime, i.e., while interacting with the malware
   - We implement an analysis framework (SECUR-AMA) and compare with different state-of-the-art malware analysis methodologies both static and dynamic. Results show that our proposed solution favourably compares with existing techniques in the problem of malware analysis of real samples
2. In the context of long-term behavioral analysis:
   - We use the long-term transition probability as a new type of feature to classify behavioral models
   - We define a transformation for Markov chains enforcing the absorbency property. This allows us to use standard techniques to derive the long-term transition probability for generic Markov chains without requiring any specific properties
   - We empirically evaluate our approach on behavioral models of real malicious software agents. Results show that our technique provides informative features suitable to successfully identify known behaviors, allowing to diminish the effect of noise injected into the behavioral models by malicious software agents. This is an improvement over current state-of-the-art malware analysis techniques that often struggle to analyze sophisticated malware employing advanced stealthy mechanisms. We also perform experiments on models of players interacting within classical games in order to evaluate the applicability of our approach to generic use cases, not necessarily related to malware analysis
3. In the context of a new AMA formalization with Bayesian games:
   - We propose Bayesian Active Malware Analysis (BAMA), a novel technique for dynamic active malware analysis, formalized as a Bayesian game and built upon the link between malware family and the notion of types in Bayesian games
   - To exploit our formalization we devise a decision making strategy for the analyzer based on an entropy minimization principle applied to the analyzer's belief about the family of the malware that is currently being faced

- We empirically evaluate BAMA comparing with different state-of-the-art malware analysis techniques, including SECUR-AMA. Results show significant improvements in the learning speed and in the best classification score compared to other methodologies

4. All the solutions presented in the previous points have been implemented in a comprehensive framework for AMA that has been used to analyze a subset of the families included in a dataset of real Android malware (F. Wei et al., 2017)

1.4 Organization of the thesis

The thesis is divided in 5 parts: Part I provides background knowledge and related work necessary to better understand our proposed approaches, while Parts II, III, IV and V present the contributions of the thesis.

1. In Part I, Chapter 2 provides the main concepts of information theory, game theory, MCTS as well as key properties and theoretical results related to Markov chains that we exploit in this work. Chapter 3 discusses the state-of-the-art on malware analysis and agent behavioral modeling based on observation in order to position our work with respect to existing literature

2. Part II presents the formal behavioral model based on Markov chains used to represent agents' behavior (Chapter 4). The analysis framework implemented to analyze real Android malware and to generate the respective models is described in Chapter 5. The concepts and tools presented in Part II are referred throughout all the thesis since our approaches make extensive use of the behavioral model formalization and the analysis framework as well

3. Part III details the SECUR-AMA approach to AMA (Chapter 6) and its empirical evaluation with real Android malware comparing to other malware analysis techniques (Chapter 7)

4. Part IV focuses on the long-term behavioral analysis approach for Markov chain based models (Chapter 8) and its evaluation both with classical games and with real Android malware (Chapter 9)

5. In Part V we present BAMA, a new formalization of AMA using Bayesian games (Chapter 10). The evaluation is conducted in Chapter 11 where we apply BAMA to the analysis of real Android malware and compare with other malware analysis techniques, including SECUR-AMA

Finally, Chapter 12 draws conclusions with final consideration and future research directions.

1.5 Publications

Most of the content presented in this thesis has been published in top-level conferences and journals. In detail, the content of Part II regarding the proposed behavioral model (Chapter 4) and the analysis framework (Chapter 5) and Part III where we describe SECUR-AMA (Chapter 6) and its evaluation (Chapter 7) has been published in (Sartea et al., 2016; Sartea and Farinelli, 2017). The comprehensive approach (Chapters 4, 5, 6, 7) instead has been published in (Sartea, Farinelli, and Murari, 2020). The content of Part IV regarding the long-term behavioral analysis (Chapter 8) and its evaluation (Chapter 9) has been published in (Sartea and Farinelli, 2018; Sartea et al., 2019). The remaining contributions in Part V, where we present BAMA (Chapter 10) and its evaluation (Chapter 11) have been published in (Sartea, Chalkiadakis, et al., 2020). The mentioned publications are listed in the following:

1. Riccardo Sartea, Mila Dalla Preda, Alessandro Farinelli, Roberto Giacobazzi, and Isabella Mastroeni (2016). "Active Android Malware Analysis: An Approach Based on Stochastic Games". In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering.* SSPREW '16. Los Angeles, California, USA: ACM, 5:1–5:10. ISBN: 978-1-4503-4841-6. DOI: `10.1145/3015135.3015140`. URL: `http://doi.acm.org/10.1145/3015135.3015140`

2. Riccardo Sartea and Alessandro Farinelli (2017). "A Monte Carlo Tree Search approach to Active Malware Analysis". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 3831–3837. DOI: `10.24963/ijcai.2017/535`. URL: `https://doi.org/10.24963/ijcai.2017/535`

3. Riccardo Sartea and Alessandro Farinelli (2018). "Detection of Intelligent Agent Behaviors Using Markov Chains". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems.* AAMAS '18. Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, pp. 2064–2066. URL: `http://dl.acm.org/citation.cfm?id=3237383.3238073`

4. Riccardo Sartea, Alessandro Farinelli, and Matteo Murari (2019). "Agent Behavioral Analysis Based on Absorbing Markov Chains". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems.* AAMAS '19. Montreal QC, Canada: International Foundation for Autonomous Agents and Multiagent Systems, pp. 647–655. ISBN: 978-1-4503-6309-9. URL: `http://dl.acm.org/citation.cfm?id=3306127.3331752`

5. Riccardo Sartea, Alessandro Farinelli, and Matteo Murari (2020). "SECUR-AMA: Active Malware Analysis Based on Monte Carlo Tree Search for Android Systems". In: *Engineering Applications of Artificial Intelligence* 87, p. 103303. ISSN: 0952-1976. DOI: `https://doi.org/10.1016/j.engappai.2019.103303`. URL: `http://www.sciencedirect.com/science/article/pii/S0952197619302635`

6. Riccardo Sartea, Georgios Chalkiadakis, Alessandro Farinelli, and Matteo Murari (2020). "Bayesian Active Malware Analysis". In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems.* AAMAS '20. Accepted for publication. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems

Part I

Background and Related work

Chapter 2

Background

In this chapter we formally describe the main concepts used in the methodologies discussed in this thesis. Section 2.1 details some concepts of information theory that are useful to define reward functions based on information gathering; Section 2.2 presents different game theoretical models that can be used to represent the interaction between non-cooperating agents; Section 2.3 introduces Monte Carlo methods to efficiently search within high dimensional domains such as complex games; Section 2.4 explains the model of Markov chain along with some of its important properties and how to extract them. In particular, the concepts of information theory in Section 2.1 are extensively used to model the utility functions for the analyzer agents in the SECUR-AMA approach described in Part III and the BAMA approach described in Part V. The same parts also leverage on the concepts of game theory in Section 2.2 to model the interaction between analyzer and malware agents. Monte Carlo methods and in particular MCTS are employed as decision making strategy for the analyzer in SECUR-AMA (Part III). Markov chains (Section 2.4) serve as a basis for the malware behavioral models (explained in Chapter 4) that are used throughout all the thesis, and as theoretical pillar on which the long-term analysis of Part IV is built on.

2.1 Information Theory

Information theory has useful tools that can be used to deal with decision making under uncertainty, i.e., how to deal with uncertain outcomes or to measure or quantify the information carried by a probabilistic event, that is fundamental when working in complex domains. We start by defining the *Gamma* - Γ and *Digamma* - ψ functions (Yin et al., 2018) that will in turn be used to define different probability distributions.

Definition 2.1 (Gamma function)**.** *For a complex number z with positive real part the gamma function is defined as*

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx$$

For any positive integer n the gamma function is

$$\Gamma(n) = (n-1)!$$

Definition 2.2 (Digamma function)**.** *The digamma function is defined as the logarithmic derivative of the gamma function*

$$\psi(z) = \frac{d}{dz} \ln \Gamma(z)$$

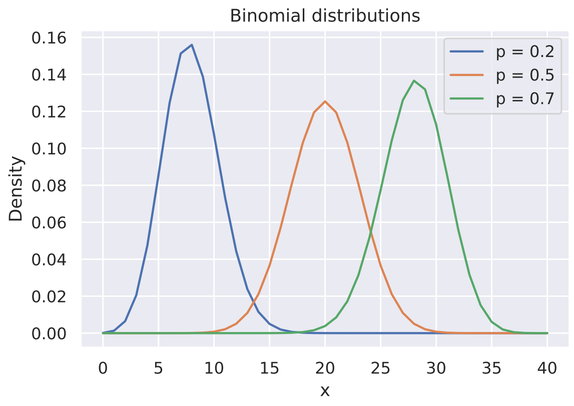


FIGURE 2.1: Binomial distribution densities with $n = 40$

2.1.1 Binomial distribution

The binomial distribution is used to model events with a binary possible outcome, i.e., success/failure of an event (Kachitvichyanukul and Schmeiser, 1988). A simple example is the outcome of a coin toss, either head or tail: a fair coin has a uniform probability distribution between head and tail, i.e., $P(Head) = P(Tail) = \frac{1}{2}$

Definition 2.3 (Binomial distribution)**.** *A binomial distribution with parameters $n \in \mathbb{N}$ and $p \in \mathbb{R}_{[0,1]}$, denoted with $Bin(n, p)$, is the discrete probability distribution of the number of successes in a sequence of n independent experiments (Bernoulli trials). A random variable sampled from a binomial distribution $x \sim Bin(n, p)$ has probability density*

$$Bin(x \mid n, p) = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} p^k (1-p)^{n-k}$$

Notice that $\frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} = \binom{n}{k}$ is the binomial coefficient. The interpretation of the binomial distribution is that an event occurs k times with probability p^k while the other occurs $n-k$ times with probability $(1-p)^{n-k}$. Figure 2.1 shows an example of three different binomial distribution probability densities.

2.1.2 Multinomial distribution

The multinomial distribution is a generalization of the binomial distribution of Section 2.3 to events with more than 2 possible outcomes (Ng et al., 2011). A simple example is that of a die toss: a fair die has a uniform probability distribution among all the possible 6 outcomes, i.e., $P(1) = \cdots = P(6) = \frac{1}{6}$.

Definition 2.4 (Multinomial distribution)**.** *The multinomial distribution $Mult(\boldsymbol{p}, n)$ where $\boldsymbol{p} = (p_1, \ldots, p_k)$ with $p_i \geq 0$ for $i = 1, \ldots, k$ and $\sum_{i=1}^{k} p_i = 1$, and $n \geq 1$, is a discrete distribution over k dimensional non negative integer vectors $\boldsymbol{x} \in \mathbb{N}^k$ where $\sum_{i=1}^{k} x_i = n$. A random variable sampled from a multinomial distribution $\boldsymbol{x} \sim Mult(\boldsymbol{p}, n)$ has probability density*

$$Mult(\boldsymbol{x} \mid \boldsymbol{p}, n) = \frac{\Gamma(n+1)}{\prod_{i=1}^{k} \Gamma(x_i + 1)} \prod_{i=1}^{k} p_i^{x_i}$$

Each entry $x_i \in \boldsymbol{x}$ counts the number of times value i was drawn among the total of n. Figure 2.2 shows an example of multinomial (trinomial) distribution probability density.

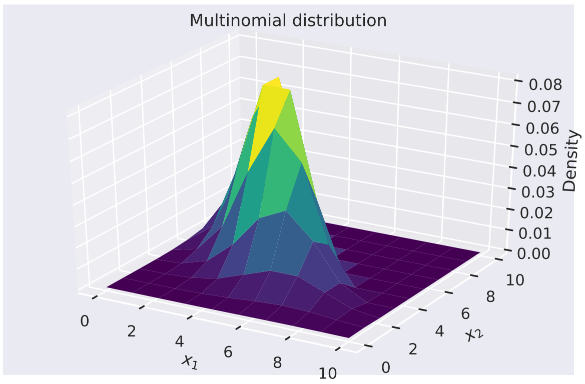


FIGURE 2.2: Multinomial distribution density with $\boldsymbol{p} = (\frac{1}{2}, \frac{1}{3}, \frac{1}{6})$. Dimension x_3 is not shown, but can be inferred by the constraint $x_1 + x_2 + x_3 = 10$

An agent that, for example, has to bet on a coin or die toss would necessarily use the information about the probability distribution associated to such coin or die. Intuitively, if a fair coin is used, the agent receives maximum expected reward betting half of the times on tail and the other half on head. However, if a coin is not fair, the agent needs to handle uncertainty on the probability distribution itself rather than using specific values assigned to each possible outcome to take her decisions. This can be achieved leveraging on distributions that, when sampled, give a probability distribution as result, rather than a single value. Imagine a die factory: every die produced has an associated probability distribution over its 6 outcomes, but the factory itself follows a probability distribution (a *Dirichlet* in this case, see Section 2.1.4) that governs the shape of the probability distribution of every single die. On average, the produced dice follow a uniform multinomial probability distribution, but due to production errors there is some uncertainty involved.

2.1.3 Beta distribution

The beta distribution is a continuous probability distribution that when sampled gives a binomial distribution as result, i.e., the value p for an event to succeed (Ng et al., 2011). The probability of the event to not succeed is simply computed as $(1 - p)$.

Definition 2.5 (Beta-distribution)**.** *A beta distribution with parameters* $\alpha, \beta > 0$ *is denoted as* $Beta(\alpha, \beta)$*. A random variable* $0 \leq x \leq 1$ *sampled from a beta distribution* $x \sim Beta(\alpha, \beta)$ *has probability density*

$$Beta(x \mid \alpha, \beta) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

where

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

Γ *is the Gamma function*

Figure 2.3 shows an example of four different beta distribution probability densities. The expectation over a beta distribution, i.e., the average sampling result, can be computed considering Definition 2.6.

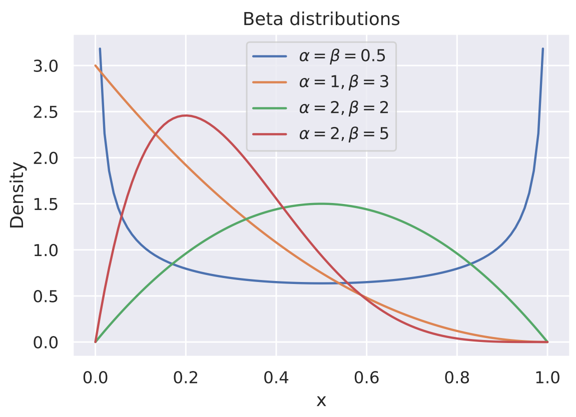


FIGURE 2.3: Beta distribution densities

Definition 2.6 (Expectation of a beta distribution)**.** *The expected binomial distribution parameter $p = x$ obtained by sampling a beta distribution is*

$$\mathbb{E}(Beta(x \mid \alpha, \beta)) = \frac{\alpha}{\alpha + \beta}$$

The uniform binomial distribution is obtained (on average) when $\alpha = \beta$.

2.1.4 Dirichlet distribution

The Dirichlet distribution is a generalization of the beta distribution for events with more than 2 possible outcomes that when sampled results in a multinomial distribution in output.

Definition 2.7 (Dirichlet distribution)**.** *A Dirichlet distribution of order $k \geq 2$ with parameters $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k) \in (\mathbb{R}_{>0})^k$ is denoted as $Dir(\boldsymbol{\alpha})$. A k-dimensional random variable $\boldsymbol{\theta} = (\theta_1, \dots, \theta_k)$ with $\theta_i \geq 0$ for $i = 1, \dots, k$ and $\sum_{i=1}^{k} \theta_i = 1$ sampled from a Dirichlet distribution $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$ has probability density.*

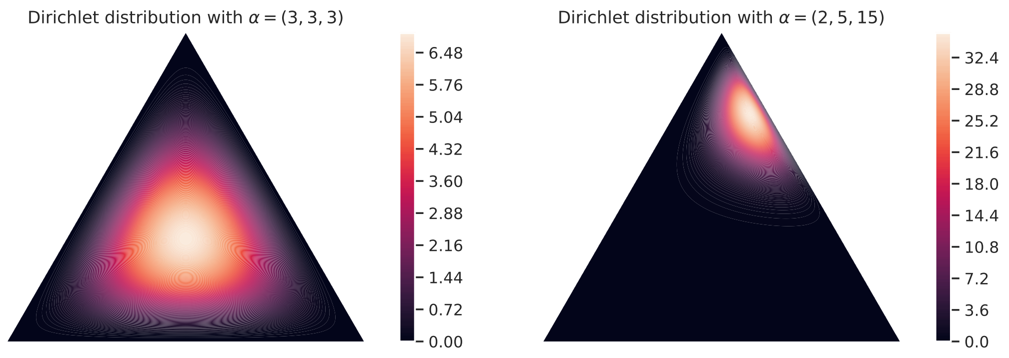
$$Dir(\boldsymbol{\theta} \mid \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1}$$

where

$$B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{k} \Gamma(\alpha_i)}{\Gamma(\alpha_0)} \quad \text{and} \quad \alpha_0 = \sum_{i=1}^{k} \alpha_i$$

Γ *is the Gamma function*

There are some technicalities that arise from Definition 2.7: first of all, the support of a Dirichlet probability density function is actually the $(k-1)$-dimensional simplex, which lives in the k-dimensional space. Hence, the density should be a function of $k-1$ of the k variables, with the k-th equal to 1 minus the sum of the others, so that the overall sum is equal to 1. To avoid over complicating the definition however, we write the density as a function of the entire k-dimensional vector; secondly, the support is actually the open simplex, i.e., $x_i > 0$ for $i = 1, \dots, k$, meaning also that no $x_i = 1$ can exist in the result (Ng et al., 2011). Figure 2.4 shows an example of two different Dirichlet distribution probability densities. The expectation over a Dirichlet distribution, i.e., the average sampling result, can be computed with Definition 2.8.



(A) Dirichlet distribution with $\boldsymbol{\alpha} = (3, 3, 3)$ (B) Dirichlet distribution with $\boldsymbol{\alpha} = (2, 5, 15)$

FIGURE 2.4: Examples of Dirichlet distributions with different $\boldsymbol{\alpha}$. Edges of the triangles represent the values for $x_1, x_2, x_3 \in \mathbb{R}_{[0,1]}$ where $x_1 + x_2 + x_3 = 1$

Definition 2.8 (Expectation of a Dirichlet distribution)**.** *The expected multinomial distribution $\boldsymbol{x}$ obtained by sampling a Dirichlet distribution is*

$$\mathbb{E}(Dir(\boldsymbol{x} \mid \boldsymbol{\alpha})) = (x_1, \ldots, x_k) \quad \textit{where} \quad x_i = \frac{\alpha_i}{\alpha_0}$$

The uniform multinomial distribution is obtained (on average) when $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k)$ with $\alpha_i = \alpha_j$ for all $i, j \in [1, k]$

2.1.5 Conjugate prior

When performing inference an agent updates her *prior* probability distribution with the observation obtained during execution. For example, the agent betting on a coin toss that uses a prior beta distribution of $Beta(2, 5)$, where $\alpha = 2$ is the parameter governing the likelihood of event $Head$ and $\beta = 5$ the one of $Tail$, believes that the outcome of a toss is distributed as $P(Head) = 0.2$ and $P(Tail) = 0.8$ (as an average sampling from $Beta(2, 5)$, see Definition 2.6). However, experience can be used to refine the initial prior belief based on the observation of the outcomes of multiple tosses of the coin, transforming the prior belief in a *posterior* belief (usually more precise). A classical method to perform inference is to apply the Bayes' theorem (Lee, 2012).

Theorem 2.1 (Bayes' theorem)**.**

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

where

- *H is the hypothesis*
- *$P(H)$ is the prior probability on the hypothesis*
- *E is the evidence (new data coming from observations)*
- *$P(H \mid E)$ is the posterior probability of the hypothesis considering the new evidence*
- *$P(E \mid H)$ is the likelihood of observing the new evidence E given the hypothesis*

– *$P(E)$ is the marginal likelihood, the probability of the new evidence given the model without considering the hypothesis*

There are some prior distributions that are interesting as they are *conjugate* priors, meaning that by applying the Bayes' theorem, the resulting probability distribution (posterior) is of the same family of the prior. Conjugate priors are convenient from the algebraic point of view since they give rise to closed-form expressions that allow to avoid numerical integration. The beta and Dirichlet distributions are conjugate priors of the binomial and multinomial distributions respectively and they are widely used to perform Bayesian inference. The application of the Bayes' theorem results in an updating process for the new observations in the beta and Dirichlet priors that is simple and computationally efficient since it only requires to add the counts of the events observed to the parameters of the corresponding distributions. If the new observation vector is $\boldsymbol{o} = (o_1, \ldots, o_k)$, where o_i is the number of times event i has been observed, the Dirichlet updates its parameters $\boldsymbol{\alpha} = (\alpha_1 + o_1, \ldots, \alpha_k + o_k)$. For the beta distribution the update method is the same, where the observations of the two possible events update the α and β parameters respectively. For this reason, parameters of beta and Dirichlet distributions are also called *pseudo-counts.*

2.1.6 Entropy

Entropy is a measure of the information associated to an event. Intuitively, if an event is very likely to be observed, the information acquired by observing it is low (there is no surprise). Conversely, if an event is rare, learning that it will happen in the future (or that happened in the past) is much more valuable. Shannon's definition of entropy is the following

Definition 2.9 (Shannon entropy)**.** *For a continuous probability density $\rho(x)$ the entropy value is defined as*

$$H(\rho) = -\int \rho(x) \ln \rho(x) dx$$

For a discrete variable $\boldsymbol{x} = (x_1, \ldots, x_n)$ the entropy value is defined as

$$H(\boldsymbol{x}) = -\sum_{i=1}^{n} P(x_i) \ln P(x_i)$$

Entropy stands for the measure of *mean uncertainty* for an experiment, or as *mean information* that can be acquired by performing such experiment. The maximum entropy (or maximum uncertainty) is achieved when the underlying probability distribution is uniform, i.e., every event is equally possible, as it means that no information is known (Garbaczewski, 2006). Figure 2.5 shows how the entropy for a binomial distribution changes based on the value of parameter p: maximum entropy is achieved when $p = \frac{1}{2}$, i.e., uninformative distribution. Entropy is often used as a utility function in scenarios in which an agent has to take decisions to lower the uncertainty on the environment, e.g., entropy-based exploration for robotics (Otte et al., 2014), in which a robot moves to portions of the space with higher associated uncertainty (entropy) in order to refine its knowledge.

There are some differences in the entropy between a discrete or a continuous random variable (differential entropy). First of all, differential entropy can assume negative values, and this is the case both for the beta and Dirichlet distributions (Sections 2.1.7 and 2.1.8). Secondly, in the discrete case, entropy quantifies randomness of a

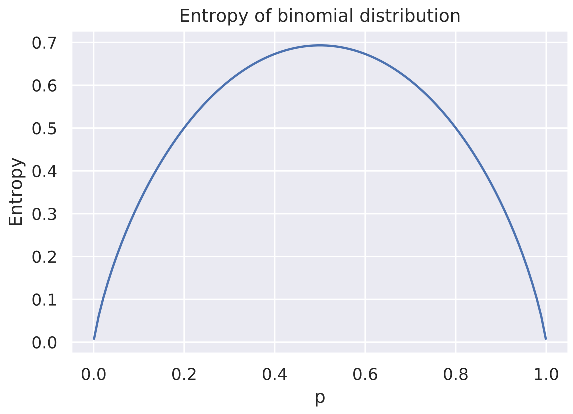


FIGURE 2.5: Entropy of binomial distribution

system in an *absolute* way, whereas in the continuous case this quantification has only a *relative* meaning. Consequently, differential entropy can not represent the absolute amount of information carried by a system, unless carefully interpreted, however, it can be used to know which variable has greater or greatest entropy (Sobczyk, 2001).

2.1.7 Entropy of beta distribution

The beta distribution has a closed-form formulation for the entropy which is the following (Ebrahimi et al., 2011)

Definition 2.10 (Entropy of beta distribution)**.** *The (differential) entropy of the beta distribution Beta(α, β) is*

$$H_B(\alpha, \beta) = \log B(\alpha, \beta) - (\alpha - 1)\psi(\alpha) - (\beta - 1)\psi(\beta) + (\alpha + \beta - 2)\psi(\alpha + \beta)$$

ψ is the Digamma function

The entropy of the beta distribution is reflected in the entropy of the binomial distribution that is sampled: the lower (higher) the entropy of the beta distribution, the lower (higher) the entropy of its expectation (the binomial distribution result of the average sampling).

2.1.8 Entropy of Dirichlet distribution

The entropy of a Dirichlet distribution is a generalization of the entropy of the beta distribution (Ebrahimi et al., 2011).

Definition 2.11 (Entropy of Dirichlet distribution)**.** *The (differential) entropy of the Dirichlet distribution Dir$(\boldsymbol{\alpha})$ of order k is*

$$H_D(\boldsymbol{\alpha}) = \log B(\boldsymbol{\alpha}) + (\alpha_0 - k)\psi(\alpha_0) - \sum_{i=1}^{k}(\alpha_i - 1)\psi(\alpha_i)$$

ψ is the Digamma function

Also in this case the entropy of the Dirichlet distribution is reflected in the entropy of the multinomial distribution that is sampled: the lower (higher) the entropy of the Dirichlet distribution, the lower (higher) the entropy of its expectation (the multinomial distribution result of the average sampling) (Garbaczewski, 2006)

2.1.9 Kullback–Leibler divergence

It may be necessary to measure the "distance" between two probability distributions, e.g., as loss function to train a data model, to study randomness in continuous time-series. The Kullback–Leibler divergence is often used as distance measure in such cases (Kullback and Leibler, 1951).

Definition 2.12 (Kullback-Leibler divergence)**.** *The Kullback-Leibler divergence is a measure of how one probability distribution is different from a second reference probability distribution*

$$D_{KL}(P \parallel Q) = -\sum_{x \in \mathcal{X}} P(x) \log \left(\frac{Q(x)}{P(x)} \right)$$

The Kullback-Leibler divergence is always non negative and asymmetric, i.e., $D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$ and measures how much a distribution is different from another in terms of the quantity of information that is lost when one distribution is approximated with the other.

2.2 Game theory

Game theory involves the study of mathematical tools used to model interaction between agents, regarded as *decision makers.* Such models apply to a wide variety of domains, e.g., games, cyber-security, social security. The basic assumptions on which the theory builds upon are that agents pursue well defined exogenous objectives while being *rational* and *intelligent* (Osborne and Rubinstein, 1994).

- *Rationality:* an agent is rational if she takes decision that maximize her own intents. More in details, each agent has a utility function that maps every state of the world to real numbers in order to express the desirability of a state. A rational agent aims at maximizing such utility function choosing an action resulting in more desirable states
- *Intelligence:* an agent is intelligent if she knows everything about the environment and she can use such knowledge to reason about a situation to achieve her objectives

Rationality and intelligence together are employed by agents in the process of selecting their actions: in the *theory of rational choice*, an agents chooses the best action that is at least as good as any other available action according to the agent's preferences. In particular, preferences are represented with *payoff functions* that associate a value to each action based on the outcome (Myerson, 1997). In the context of game theory, a *game* is a model of interaction between agents, referred to as *players.*

2.2.1 Strategic games

A *strategic game* is a model that captures interaction between the players by allowing each one to be affected by the actions of all the others, not only her own actions, and where every player has preferences about the action profile.

Definition 2.13 (Strategic game)**.** *A strategic game with n players is a tuple $G = (N, \boldsymbol{A}, \boldsymbol{u})$ where*

- *N is the set of n players*

		Suspect 2	
		D	C
Suspect 1	D	$3,3$	$0,4$
	C	$4,0$	$1,1$

Table 2.1: Normal form of the Prisoner's Dilemma

- $\boldsymbol{A} = A_1 \times \cdots \times A_n$ *where* A_i *represents the actions available for player* i
- $\boldsymbol{u} = (u_1, ..., u_n)$ *is a profile of utility functions with* $u_i : \boldsymbol{A} \to \mathbb{R}$ *utility function for player* i

A common interpretation of a strategic game is that it is a model of an event that occurs only once, each player knows the details of the game and the fact that all the players are rational. Each player chooses her action once and for all, and the players choose their actions simultaneously, in the sense that no player is informed, when choosing her action, of the action chosen by any other player (Osborne and Rubinstein, 1994).

A well known strategic game is the Prisoner's Dilemma (Nowak and Sigmund, 1993). Two suspects are held by the police in separate cells and there is enough evidence to convict each of them of a minor offense, but not enough evidence to convict either of them of the major crime unless one of them acts as informer against the other. If they cooperate with each other without saying anything to the police, each will be convicted of the minor offense and spend one year in prison. If only one of them tries to cooperate while the other defects instead, the defecting suspect will be released whereas the other will spend four years in prison. Finally, if both suspects defect, they will both spend three years in prison. The game formalization is reported in Definition 2.14

Definition 2.14 (Prisoner's Dilemma).

- $N = \{n_1, n_2\}$ *the two suspects*
- $\boldsymbol{A} = A_1 \times A_2$ *where* $A_1 = A_2 = \{C, D\}$*. C stands for cooperate and D for defect*
- $\boldsymbol{u} = (u_1, u_2)$ *where* $u_1(D,D) = u_2(D,D) = 3$*,* $u_1(C,C) = u_2(C,C) = 1$*,* $u_1(D,C) = 0$*,* $u_2(D,C) = 4$*,* $u_1(C,D) = 4$*,* $u_2(C,D) = 0$

In this example lower payoff values are better as they represent years of conviction. A compact way to represent a game is through its normal form, such as in Table 2.1, where rows and columns correspond to the possible actions of suspect 1 and 2 respectively, while the values in every cell are the payoffs of suspect 1 and 2 in such order. The Prisoner's Dilemma models a situation in which there are gains from cooperating but also an incentive to defect and "run free". This game is important as many situations have the same structure, e.g., joint project, duopoly, and it is often used as simple example to study concepts such as equilibrium, fairness or others as there is a large literature of works that analyzed such game (Nowak and Sigmund, 1993; Friedman, 1971; Jurišić et al., 2012).

2.2.2 Nash equilibrium

Although we do not deal with the concept of equilibrium in our work, in this section we present such concept as it is useful to understand the future research directions arising from this thesis. As we assume the rationality of the players, i.e., each player chooses the best available action according to her *beliefs* and preferences, we may want to study which actions will be chosen by the players in a strategic game. Indeed, the best action for any given player usually depends on the other player's actions. Therefore, when choosing an action a player must consider what all the other players will choose, i.e., a player forms her own belief. The assumption is that a belief is derived from the past experience playing a game without becoming familiar with the behavior of a specific opponent. In summary, every player chooses her actions according to the theory of rational choice, given the belief on the other player's actions. Importantly, we assume that such belief is correct (Osborne and Rubinstein, 1994).

Definition 2.15 (Nash equilibrium). *A Nash equilibrium is an action profile a^* with the property that no player i can do better by choosing an action different from a_i^*, given that every other player j adheres to a_j^**

If every player has no reason to choose any action different from her component of the action profile a^*, there is no pressure on the action profile to change. In a Nash equilibrium, players have no incentive to deviate from it (Nash, 1950). In the normal form of the game of Prisoner's Dilemma (Table 2.1) it can be seen that the unique Nash equilibrium is the action profile $a^* = (D, D)$ since:

- (D, D): if suspect 1 chooses to defect, suspect 2 is better off choosing to defect as well in order to spend 3 year in prison instead of 4. The same consideration stands if suspect 2 chooses to defect. This action profile is a Nash equilibrium since players have no incentive to deviate from it
- (C, C): if suspect 1 chooses to cooperate with the other one, suspect 2 would run free if defecting instead of cooperating, thus having an incentive to deviate. The same consideration stands if suspect 2 chooses to cooperate
- (C, D): if suspect 2 chooses to defect, suspect 1 is better off choosing to defect as well in order to spend 3 year in prison instead of 4. Suspect 1 than has an incentive to deviate
- (D, C): if suspect 1 chooses to defect, suspect 2 is better of choosing to defect as well in order to spend 3 year in prison instead of 4. Suspect 2 than has an incentive to deviate

The incentive to be released eliminates the choice of the mutually desirable outcome (C, C). Indeed, when reasoning about Nash equilibrium, only payoff values are considered instead of the social aspect, e.g., fairness, or efficiency, e.g., Pareto optimality (Mock, 2011).

In the Prisoner's Dilemma, the Nash equilibrium contains the best action for each player not only if the other one chooses the equilibrium action, but also when choosing the other one. The action dictated by the Nash equilibrium is then optimal regardless of the action expected to be played by the opponent. In most games this does not happen since usually, playing the action from the Nash equilibrium is optimal only if the opponent plays at the Nash equilibrium as well. Such consideration is fundamental: if a player bases her belief on the fact the the opponents behave rationally and have perfect knowledge of the game, such player may incur in losses if the assumptions do not hold in reality. For instance, an inexperienced player of chess will most likely not behave as a master player would in a given situation, potentially countering moves that would have been the best choice against a "perfect" player. Therefore, even though taking actions based on the Nash equilibrium is a reasonable solution in some cases, it is extremely important to consider if reasoning about equilibria is applicable and effective for a particular problem.

There may be multiple Nash equilibria for a game, and an equilibrium profile could be composed of mixed strategies, i.e., probabilistic action choices. The existence of a Nash equilibrium is guaranteed if mixed strategies are allowed and there are a finite number of players and actions (Nash, 1950). Computing a Nash equilibrium however is a hard problem in the general case (Chen et al., 2009), but we do not get into the details of how to compute it since it is out of the scope of this thesis.

2.2.3 Extensive games

A strategic game dos not consider the sequentiality of decision making. In *extensive games* instead, the sequential structure is explicitly considered, allowing players to change their mind as events unfold instead of selecting a plan of action once and for all.

Definition 2.16 (Extensive game with perfect information)**.** *An extensive game with perfect information with n players is a tuple*
$G = (N, H, Z, P, \boldsymbol{u})$ *where*

- N *is the set of* n *players*
- H *is the set of sequences (histories)*
- $Z \subset H$ *is the set of terminal histories, i.e., the set of all sequences of actions that may occur*
- $P : H \backslash Z \to N$ *is the player function that assigns a player to every non-terminal history. The player assigned to any history* h *is the player who takes an action after* h
- $\boldsymbol{u} = (u_1, ..., u_n)$ *is a profile of utility functions with* $u_i : Z \to \mathbb{R}$ *utility function for player* i

An extensive game models a situation in which every player knows all the actions chosen before (has perfect information) and does not move simultaneously with other players (Osborne and Rubinstein, 1994). The entry game of Definition 2.17 is an example of extensive game: an incumbent faces the possibility of entry by a challenger. For example, a firm may be considering to enter into an industrial sector currently occupied by a monopolist. The challenger may enter or not. If she does enter, the incumbent may either acquiesce or fight.

Definition 2.17 (Entry game)**.**
- $N = \{Challenger, Incumbent\}$
- $H = \{\emptyset, In, Out, (In, Acquiesce), (In, Fight)\}$
- $Z = \{Out, (In, Acquiesce), (In, Fight)\}$
- $P(\emptyset) = Challenger$, $P(In) = Incumbent$
- $u_1((In, Acquiesce)) = 2$, $u_2((In, Acquiesce)) = 1$, $u_1((In, Fight)) = 0$, $u_2((In, Fight)) = 1$, $u_1(Out) = 1$, $u_2(Out) = 2$

A useful illustration of an extensive game uses a tree representation such as the one of Figure 2.6

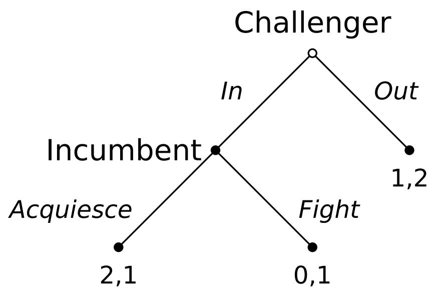


FIGURE 2.6: Extensive form of the entry game

		Incumbent	
		Acquiesce	*Fight*
Challenger	*In*	$2,1$	$0,1$
	Out	$1,2$	$1,2$

TABLE 2.2: Normal form of the entry game in strategic form

The notion of Nash equilibrium can also be applied to extensive games by leveraging on the concept of *strategy*. A player's strategy specifies the action that player p chooses for every history h such that $P(h) = p$, forming a plan of action specifying how to react to the opponent's actions. The Nash equilibrium concept for extensive games with perfect information is similar to that of strategic games: a strategy profile from which no player has an incentive to deviate. In particular, transforming an extensive game in its strategic form (discarding the representation of sequentiality) we obtain the normal form of Table 2.2, from which the set of Nash equilibria can be easily computed and that is guaranteed to coincide with the set of Nash equilibria of its extensive form[1] (Osborne and Rubinstein, 1994). The Nash equilibria for the entry game are two

- $(In, Acquiesce)$: if the challenger enters, the incumbent gets a higher payoff by acquiescing
- $(Out, Fight)$: if the challenger knows that the incumbent will fight, it is better for the challenger to stay out (better payoff)

2.2.4 Bayesian games

The underlying assumption for the existence of a Nash equilibrium is that every player's belief about the other player's actions is correct and reliable. However, in many practical cases such assumption is too strict or even unrealistic. A *Bayesian game* is a generalization of the notion of a strategic game where players have *incomplete* information about some aspects of the environment relevant to decision making, e.g., the exact payoff function of an opponent. In a Bayesian game model, every player may have different types, each one encoding private information on the player, e.g., payoff function. To model a Bayesian game it is useful to add a special player called *nature* that randomly chooses a type for each player according to the prior probability distribution p that is assumed to be known by every player. Such assumption allows to think about a game with incomplete information as a game with *imperfect* information where nature makes the first move, but there is no certainty on the exact

[1]We skip the concept of sub-game perfect equilibrium to avoid getting into many details that are not used in this thesis

type of the other players, only a probability distribution over them (hence imperfect information) (Harsanyi, 1967).

Definition 2.18 (Bayesian game)**.** *A Bayesian game with n players is a tuple $G = (N, \boldsymbol{A}, \boldsymbol{\Theta}, \boldsymbol{u}, p)$ where*

- *N is the finite set of n players*
- *$\boldsymbol{A} = A_1 \times \cdots \times A_n$ where A_i represents the actions available for player i*
- *$\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$ where Θ_i represents the types available for player i*
- *$\boldsymbol{u} = (u_1, ..., u_n)$ is a profile of utility functions with $u_i : \boldsymbol{A} \times \boldsymbol{\Theta} \rightarrow \mathbb{R}$ utility function for player i*
- *p is the common prior over $\boldsymbol{\Theta}$*

Every player player can be of several types, each one corresponding to a possible payoff function for that player. A player's type θ_i is only observed by player i and encodes all relevant information about some important private characteristic of such player. Utility functions take into account not only player's actions but also their types. Bayesian games get their name from the fact that each individual player incorporates the common prior with her own experience using the Bayes' theorem.

The entry game of Definition 2.17 can be modeled as a Bayesian game where there may be two types of incumbents: a tough and a normal one. A challenger entering in a market against a tough incumbent will receive a lower payoff compared to entering a market against a normal incumbent. Definition 2.19 details the game and Table 2.3 shows its normal form.

Definition 2.19 (Bayesian entry game)**.**

- *$N = \{challenger, incumbent\}$*
- *$\boldsymbol{A} = A_1 \times A_2$ where*
  - *$A_1 = \{In, Out\}$*
  - *$A_2 = \{Acquiesce, Fight\}$*
- *$\boldsymbol{\Theta} = \Theta_1 \times \Theta_2$ where*
  - *$\Theta_1 = \{\theta\}$ only one type of challenger*
  - *$\Theta_2 = \{Tough, Normal\}$*
- *$\boldsymbol{u} = (u_1, u_2)$ where: $u_1(Normal, (In, Acquiesce)) = 2$, $u_2(Normal, (In, Acquiesce)) = 1$, $u_1(Normal, (In, Fight)) = 0$, $u_2(Normal, (In, Fight)) = 1$, $u_1(Normal, Out) = 1$, $u_2(Normal, Out) = 2$, $u_1(Tough, (In, Acquiesce)) = 1$, $u_2(Tough, (In, Acquiesce)) = 1$, $u_1(Tough, (In, Fight)) = -1$, $u_2(Tough, (In, Fight)) = 1$, $u_1(Tough, Out) = 1$, $u_2(Tough, Out) = 2$*
- *$p(Tough) = p(Normal) = 0.5$*

The computation of the payoff in Bayesian games uses expectation over the rewards since uncertainty is involved. For example, assuming that the incumbent will fight, the expected payoff for an entering challenger is: $u_1(\Theta_2, (In, Fight)) = p(Tough) \cdot u_1(Tough, (In, Fight)) + p(Normal) \cdot u_1(Normal, (In, Fight)) = 0.5 \cdot$

$p(Tough) = 0.5$		Incumbent *Acquiesce*	Incumbent *Fight*
Challenger	*In*	$1, 1$	$-1, 1$
	Out	$1, 2$	$1, 2$

(A) *Tough* incumbent. $p(Tough) = 0.5$

$p(Normal) = 0.5$		Incumbent *Acquiesce*	Incumbent *Fight*
Challenger	*In*	$2, 1$	$0, 1$
	Out	$1, 2$	$1, 2$

(B) *Normal* incumbent

TABLE 2.3: Bayesian entry game

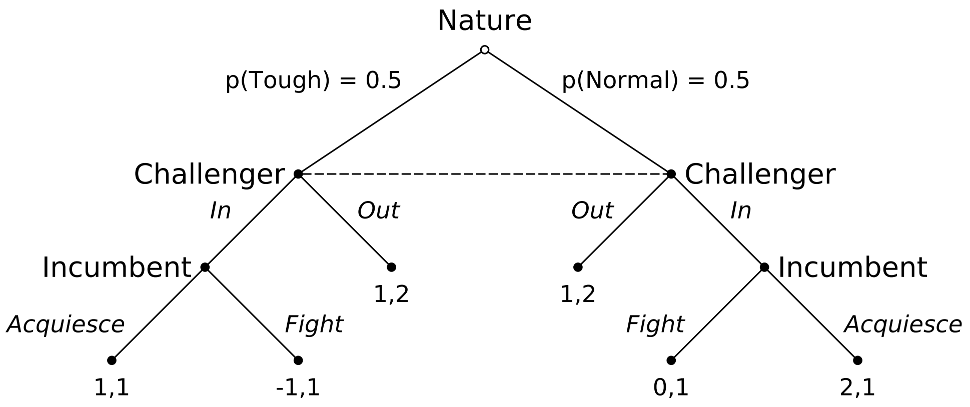


FIGURE 2.7: Extensive form of the Bayesian entry game

$-1 + 0.5 \cdot 0 = -0.5$. The notion of Nash equilibrium can be applied also to Bayesian games: a *Bayesian Nash equilibrium* is strategy profile s^* (one for each player type) such that no type has incentive to deviate given the beliefs about the types and the strategy of the other players. In the Bayesian entry game, the action profile $s^* = (In, (Acquiesce, Acquiesce))$ is a Bayesian Nash equilibrium since the incumbent's action is always optimal regardless of the challenger's choice and, given that the incumbent acquiesces for both of her types, the challenger has no incentive to deviate since she would receive a lower payoff by staying out: $\mathbb{E}(In, (Acquiesce, Acquiesce)) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2} > \mathbb{E}(Out, (Acquiesce, Acquiesce)) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1$. Also $s^* = (Out, (Acquiesce, Fight))$ is a Bayesian Nash equilibrium: $\mathbb{E}(Out, (Acquiesce, Fight)) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1 > \mathbb{E}(In, (Acquiesce, Fight)) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = \frac{1}{2}$

Also Bayesian games can be represented in extensive form (Figure 2.7) by explicitly modeling the order of decisions taken by players. In particular, nature can be seen as a player that moves first and uses a fixed mixed strategy with the same probability distribution of the prior p instead of maximizing a payoff[2] (Osborne and Rubinstein, 1994).

2.2.5 Repeated games

A sequence of plays of the same game can be modeled by a strategic game only under the assumption that there is no strategic link between the plays. If this is not the case, the game can be modeled as a *repeated game*. The strategic game that is repeatedly played is called *stage game*. The basic idea of this theory is that a player may be deterred from exploiting the short-term payoff under the "threat" of retaliation by the other players in the following rounds. Suppose that the Prisoner's Dilemma (Definition 2.14) is played repeatedly and suspect 1 adopts a retaliation (Friedman,

[2]The corresponding notion of sub-game perfect equilibrium in Bayesian games is the sequential equilibrium, a refinement of perfect Bayesian equilibrium that imposes restrictions on strategies and beliefs

1971) strategy in which she chooses C so long as suspect 2 chooses C, and as soon as suspect 2 chooses D for the first time, suspect 1 chooses D forever from that point onward. If suspect 2 always chooses C, resulting in an outcome of (C, C), the trail of payoff is $(1, 1, 1, \dots)$ for both players. If suspect 2 changes to D at some point, her trail becomes $(\dots, 1, 0, 3, 3, \dots)$ that results in a worse trail as long as the horizon is large enough to affect the overall value. If the horizon is infinite, the utility function uses a discount factor $\delta \in \mathbb{R}_{[0,1)}$ in order to compute finite payoff values for infinite sequences as the discounted sum for the sequence of the actions played

$$\sum_{t=1}^{\infty} \delta^{t-1} u_i(\boldsymbol{a}^t) \tag{2.1}$$

The discounted average is a useful representation of payoffs that allows to compare the payoff of a single stage with the average payoff of a trail

$$(1-\delta) \sum_{t=1}^{\infty} \delta^{t-1} u_i(\boldsymbol{a}^t) \tag{2.2}$$

A repeated game is an extensive game with perfect information and simultaneous moves where the history is a sequence of action profiles of the stage game. Definition 2.20 explains the components of a repeated game (Osborne and Rubinstein, 1994).

Definition 2.20 (Repeated game)**.** *An (infinitely) repeated game with n players is a tuple $F = (G, N, H, Z, \boldsymbol{A}, \boldsymbol{u})$ where*

- *G is a strategic game to be repeatedly played*
- *N is the set of n players*
- *H is the set of terminal histories composed by infinite sequences $(\boldsymbol{a}^1, \boldsymbol{a}^2, \dots)$ of action profiles in G*
- *$Z \subset H$ is the set of proper sub-histories of every terminal history*
- *$\boldsymbol{A} = A_1 \times \cdots \times A_n$ where A_i represents the actions available for player i after any history*
- *$\boldsymbol{u} = (u_1, ..., u_n)$ is a profile of utility functions with $u_i : H \to \mathbb{R}$ utility function for player i that evaluates each terminal history according to her discounted (δ) average*

In the repeated Prisoner's Dilemma (Definition 2.21), the strategy profile in which both suspects adopt the retaliation strategy is a Nash equilibrium since the outcome is (C, C), resulting in a discounted average of 1. If one suspect deviates by selecting another strategy producing a different outcome, there is one stage (at least) in which D is chosen. Therefore, in all subsequent stages the other suspect (following the retaliation strategy) chooses D. Consequently, the trail of payoffs will have a higher (then worse) discounted average due the fact that (at least) one of the actions will be D forever (Osborne and Rubinstein, 1994).

Definition 2.21 (Repeated Prisoner's Dilemma)**.**

- *G is the Prisoner's Dilemma strategic game*

- $N = \{n_1, n_2\}$ *the two suspects*
- $H = ((a_1, a_2)^1, (a_1, a_2)^2, \dots)$ *terminal histories*
- $Z \subset H$ *is the set of proper sub-histories of every terminal history*
- $\boldsymbol{A} = A_1 \times A_2$ *where* $A_1 = A_2 = \{C, D\}$
- $\boldsymbol{u} = (u_1, ..., u_n)$ *is a profile of utility functions with* $u_i : H \to \mathbb{R}$ *utility function for player* i *that evaluates each terminal history according to her discounted average according to the utility functions of the Prisoner's Dilemma strategic game*

2.2.6 Stochastic games

In some cases the repeated interaction between players takes place in different *states.* This means that the strategic (stage) game that is repeated may be different in every state, possibly because of some different conditions such as payoff values, or even because the entire stage game could change, e.g., from chess to checkers.

Definition 2.22 (Stochastic game)**.** *A stochastic game G, with n players, is a tuple* $G = (N, S, \boldsymbol{A}, \boldsymbol{u}, T)$ *where:*

- N *is the finite set of* n *players*
- S *is the state space composed by stage games*
- $\boldsymbol{A} = A_1 \times \cdots \times A_n$ *where* A_i *represents the actions available for player* i *in state* $s \in S$
- $\boldsymbol{u} = (u_1, \dots, u_n)$ *where* $u_i : S \times \boldsymbol{A} \times S' \to \mathbb{R}$ *is the utility function for player* i
- $T : S \times \boldsymbol{A} \times S' \to \mathbb{R}_{[0,1]}$ *is a probabilistic transition function between states of the game*

From Definition 2.22, a stochastic game clearly defines players and their available action sets, the state space and the transition function from state to state. The utility function for each player is based on the action performed in a specific state and the outcome (the state reached after the transition). At each stage the game is in state s, the n players choose actions $\boldsymbol{A} = (a_1, ..., a_n)$, the game transitions to state s' according to the transition function T and players receive payoffs $(r_1, ..., r_n)$ according to the utility functions $\boldsymbol{u} = (u_1, ..., u_n)$. In other words, the joint actions of the players move the game from one state to another with a probability expressed by the transition function. Furthermore, each player receives a payoff that is based on the transition that took place (Mertens and Neyman, 1981).

An example of stochastic game is reported in Definition 2.23

Definition 2.23 (Example of stochastic game)**.** *This example of stochastic game is taken from the Game Theory course of Cardiff University*

- $N = \{n_1, n_2\}$
- $S = \{x, y\}$ *visible in Table 2.4*
- $\boldsymbol{A} = A_1 \times A_2$ *where*
  - $A_1(x) = \{a, b\}$, $A_1(y) = \{e\}$

		n_2 c	n_2 d
n_1	a	$8,4$ / $0.5, 0.5$	$5,3$ / $1,0$
n_1	b	$1,5$ / $1,0$	$2,6$ / $1,0$

(A) State x

		n_2 f
n_1	e	$0,0$ / $0,1$

(B) State y

TABLE 2.4: Example of stochastic game

		n_2 c	n_2 d
n_1	a	$8 + 0.33v, 4 + 0.33u$	$5 + 0.66v, 3 + 0.66u$
n_1	b	$1 + 0.66v, 5 + 0.66u$	$2 + 0.66v, 6 + 0.66u$

TABLE 2.5: Stage game of Table 2.4a with $\delta = 0.66$

  - $A_1(x) = \{c, d\}$, $A_1(y) = \{f\}$

- $\boldsymbol{u} = (u_1, u_2)$ *visible in Table 2.4, first couple of numbers (p, q) in each cell where p is the payoff for n_1 while q the payoff for n_2*
- T *visible in Table 2.4 second couple of numbers (l, m) in each cell where l is the probability of transitioning to state x while m the probability of transitioning to state y*

The concept of Nash equilibrium for stochastic games is a strategy profile σ^* from which no player has incentive to deviate given that all other players are following the strategy profile of the Nash equilibrium. In particular, a strategy σ_i^* for player i is a probability distribution over the set A_i given a state $s \in S$ (Markov strategy). Since a stochastic game is a repeated game, we have to use discounting (as per Section 2.2.5) in order to compute the payoffs considering the previous stages. A Nash equilibrium satisfies U_i^* of Equation 2.3 that is the expected utility of player i when all players follow the strategy profile of the Nash equilibrium (a_{-i}^* is the optimal strategy profile for players different from i).

$$U_i^*(s) = \max_{a_i \in A_i(s)} \left(\sum_{s' \in S} u_i(s, (a_i, a_{-i}^*), s') T(s, (a_i, a_{-i}^*), s') + \delta \sum_{s' \in S} T(s, (a_i, a_{-i}^*), s') U_i^*(s') \right) \tag{2.3}$$

For a discount factor $\delta = 0.66$, the Nash equilibrium of the game is the strategy profile $\sigma^* = (a, d)$ where the only state considered is x since in y players receive no rewards and such state is never left (the outgoing probability value is 0). Equation 2.3 gives the following conditions for Nash equilibrium associated to each strategy profile (imposing that $U_i^*(x)$ is greater than or equal to the utility of the other strategy profiles).

1. (a, c): $v \leq 21$ and $u \leq 3$
2. (a, d): $u \geq 3$
3. (b, c): $v \geq 21$ and $5 \geq 6$ (unsatisfiable)
4. (b, d): $5 \leq 2$ (unsatisfiable)

If the future gains of player n_1 in state x is v and those of player n_2 is u we have the situation of Table 2.5 and checking the implications of each profile of being a Nash equilibirium given the previous inequalities we obtain

1. (a,c): $8 + 0.33v = v \Rightarrow v = 12$ and $4 + 0.33u = u \Rightarrow u = 6$. Contradicts inequality 1
2. (a,d): $3 + 0.66u = u \Rightarrow u = 9$
3. (b,c): unsatisfiable
4. (b,d): unsatisfiable

Therefore strategy profile $\sigma^* = (a,d)$ is the unique Nash equilibrium[3] for Markov strategies (notice that such strategy profile is not the Nash equilibrium for the stage game x alone).

Stochastic games generalize some concepts explained in previous sections:

- When the length of play is 1, stochastic games generalize normal form games
- When the state space has dimension 1, i.e., there is only one stage game to be played, stochastic games generalize repeated games
- Actions, payoffs and transitions depend only on the current state. Markov Decision Process (MDP) are stochastic games with only 1 player
- We can have Bayesian stochastic games by modeling the player types

2.3 Monte Carlo methods

Monte Carlo methods are a powerful set of tools that can be used to reason within high dimensional domains where sampling is a more efficient option than precise inference, e.g., games of Go, poker, other domains with incomplete or imperfect information. Monte Carlo methods are useful to estimate the Q-value of an action in terms of its expected reward with a sampling such as the one proposed in (Gelly and Silver, 2011)

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i \tag{2.4}$$

where

- $N(s,a)$ is the number of times action a has been selected from state s
- $N(s)$ is the number of times state s has been visited
- z_i is the result of the i-th simulation from state s
- $\mathbb{I}_i(s,a)$ is an indicator function equal to 1 if action a was selected from state s on the i-th simulation, 0 otherwise

When the sampling of actions from a given state is uniform the approach is described as *flat Monte Carlo*, which is very effective in games such as Bridge or Scrabble (Ginsberg, 2001; Sheppard, 2002). However, there are cases in which flat Monte Carlo

[3]The corresponding notion of sub-game perfect equilibrium in stochastic games is the Markov perfect equilibrium

fails (Browne, 2011), requiring to improve the reliability of the estimates by biasing the action selection process based on past experience. This is often the case when the underlying reward distributions are unknown, hence requiring a balance between exploration and exploitation such as in the famous *multi-armed bandit* problem.

2.3.1 Multi-armed bandit

The multi-armed bandit problem is a class of decision problems in which one needs to select among K actions (the K arms of a multi-armed bandit slot machine) in order to maximize the cumulative reward by consistently taking the optimal action. The action selection is difficult because the underlying reward distributions are unknown, requiring to estimate potential rewards and to balance exploration of arms with still uncertain rewards with the exploitation of arms that are known to give high rewards. This is known as the *exploration-exploitation dilemma* (Auer et al., 2002). A common way of approaching the multi-armed bandit problem is to use a policy determining which arm to play based on past rewards with the aim of minimizing the player's *regret* defined as

$$R_N = \mu^* n - \mu_i \sum_{i=1}^{K} \mathbb{E}[T_i(n)] \tag{2.5}$$

where

- μ^* is the best possible expected reward
- μ_i is the expected reward of arm i
- K is the number of available arms (actions)
- $\mathbb{E}[T_i(n)]$ is the expected number of plays for arm i in the first n trials

The regret is the expected loss due to not playing the best action. It has been shown that no policy can achieve a regret growing slower than $O(\ln n)$ for many classes of reward distributions (Lai and Robbins, 1985).

2.3.2 Upper confidence bound

In (Auer et al., 2002), authors define the Upper Confidence Bound (UCB) that any given arm will be optimal, called *UCB1* which has a logarithmic growth of regret without any prior knowledge on the reward distributions. The optimal policy is obtained by playing the arm i maximizing UCB1.

Definition 2.24 (UCB1)**.**

$$UCB1 = \overline{X}_i + \sqrt{\frac{2 \ln n}{n_i}}$$

where

- n is the number of plays so far
- n_i is the number of times arm i was played
- $\overline{X}_i$ is the average reward from arm i

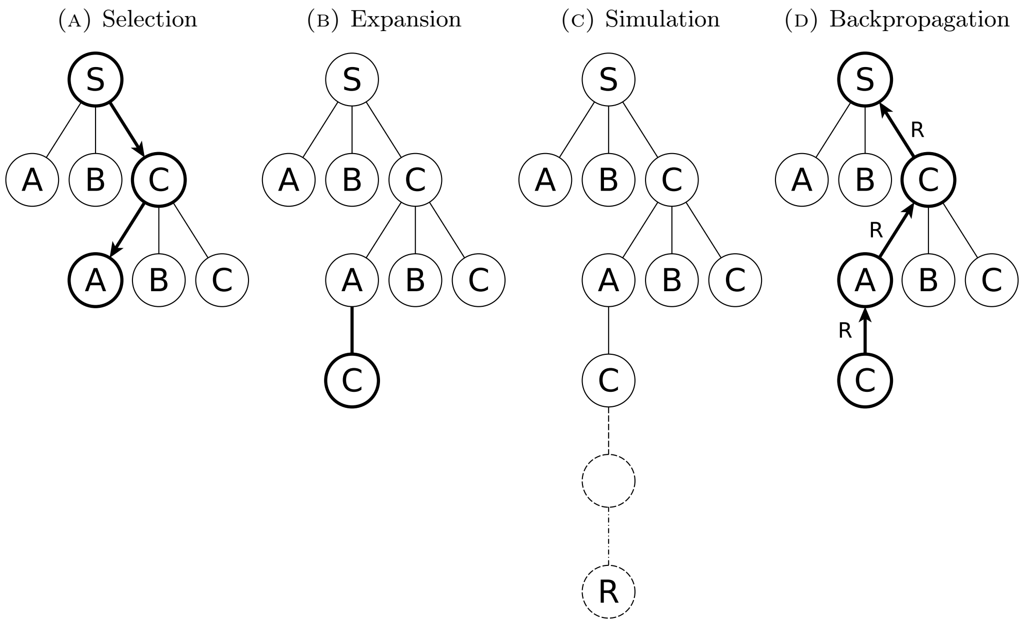


FIGURE 2.8: Schema of Monte Carlo Tree Search with action set {A, B, C}

The left-hand term encourages the exploitation of choices associated to higher rewards, while the right-hand term encourages the exploration of the less visited choices, therefore with more uncertainty associated to their rewards. UCB1 has a simple formalization, it is efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret. Due to such properties, UCB1 is often used within a generic search framework known as *Monte Carlo Tree Search* (Section 2.3.3).

2.3.3 Monte Carlo tree search

MCTS combines the precision of the tree search with the generality of random sampling by taking random samples in the search space and building a search tree according to the results (Coulom, 2007). The strength of MCTS is that the search space is not explored exhaustively, but the focus is on the most promising sub-spaces detected by using the rewards computed at the end of the simulations conducted. In particular, state and action spaces are searched by incrementally exploring the environment using simulation of executions and estimating the rewards of the actions taken into account. MCTS relies on two fundamental concepts:

- The true value of an action may be approximated using random simulation
- The true values may be used to efficiently adjust the policy in a best-first strategy

MCTS had a great impact on AI, especially in domains that can be represented as trees of sequential decisions, but where an exhaustive search would be unfeasible because of the domain size, e.g., games, planning problems. The major breakthrough that brought MCTS to the attention for its capabilities was the application to the game Go (Kocsis and Szepesvári, 2006), which is really hard for computers to play because of high branching factor, deep tree, and lack of known efficient heuristic value functions for non-terminal board positions.

The core structure of the algorithm is a tree which is initially empty and contains only the root node. The nodes of the tree represent the possible actions that lead to a particular state of the domain, while directed links form paths of subsequent states obtained by a sequence of actions. MCTS proceeds by repeating 4 basic steps until a predefined computational limit is reached, e.g., time, memory or iteration constraints (Guillaume Chaslot et al., 2008):

1. *Selection:* a *tree policy* is recursively applied from the root node descending through the tree until the most promising expandable node is reached (Figure 2.8a). The tree policy estimates the utility value of nodes already contained within the search tree. This is required in order to select the node with the highest estimated value to expand
2. *Expansion:* at least one child node is added to the previously selected node according to the set of available analyzer actions (Figure 2.8b)
3. *Simulation:* a simulation is conducted from every expanded node by following a *default policy*. The default policy is meant to be the routine that manages the simulation until a stop condition is met, then, estimates a reward accordingly (Figure 2.8c)
4. *Backpropagation:* the simulation reward is propagated from the expanded node up to the root while updating the statistics of the parent nodes traversed during the previous descent (Figure 2.8d)

Algorithm 2.1 Monte Carlo Tree Search

```
Require:
    $s_0$ - start state
    $b$ - computational budget
Ensure: Best action

1: $root \leftarrow$ MakeNode($s_0$)
2: while $b$ not depleted do
3:     $n_s \leftarrow$ Select($root$)
4:     $n_e \leftarrow$ Expand($n_s$)
5:     $r \leftarrow$ Simulate($n_e$)
6:     Backpropagate($n_e$, $r$)
7: return BestChild($root$)
```

Algorithm 2.1 summarizes the MCTS schema. For each iteration of the procedure, the tree is descended and expanded with new nodes (lines 3-4). Then, the reward given by the virtual performance (simulated) of the action-nodes added during the expansion step is estimated (line 5). Finally, the algorithm updates the statistics of the nodes traversed during the descent using the reward value (line 6). The algorithm runs until a predefined computational limit is reached and, at that point, the search is halted and the best-performing action estimated so far is returned[4] (line 7).

2.3.4 Upper confidence bound for trees

The success of MCTS depends on how the tree and default policies are implemented: the latter is almost always designed around the specific application since a simulation

[4]Usually the most visited child of the root is selected as best node. Other heuristics can be use however, such as the node with highest average reward (Coulom, 2007; G.M.J.B. Chaslot et al., 2007).

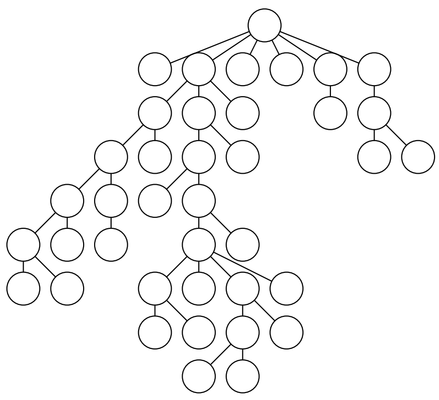


FIGURE 2.9: MCTS asymmetric growth

has to follow the rules and constraints of the environment, whereas the former can be agnostic with respect to the domain in many cases. In (Kocsis and Szepesvári, 2006), authors propose the use of UCB1, as in modeling the selection step as a multi-armed bandit problem, the value of a node is the expected reward estimated by the simulation step, therefore corresponding to random variables with unknown distributions. The selection step descends the levels of the tree choosing the nodes i maximizing the Upper Confidence Bound for Trees (UCT)

$$UCT = \overline{X}_i + 2C_p\sqrt{\frac{2\ln n}{n_i}} \tag{2.6}$$

where n is the number of times the current node has been visited, $\overline{X}_j$ is the average reward of the child node j, n_j is the number of times the child node j has been visited, and $C_p > 0$ is a constant. Not yet visited nodes are assigned the largest possible value in order to force the procedure to consider them at least once before any node at the same level is expanded further. This heuristic is widely used in RL literature and is known as *optimism in the face of uncertainty* (Szita and Lőrincz, 2008). As each node is visited, the denominator of the exploration term (right-hand side) increases, consequently decreasing its contribution. Conversely, if another node at the same level is visited, the numerator increases, increasing the contribution of less visited siblings as well. The work of (Kocsis and Szepesvári, 2006) also showed two fundamental properties for UCT:

- the bound on the regret of UCT holds also in the case of non-stationary reward distributions
- the probability of selecting a sub-optimal action at the root level converges to zero at a polynomial rate as the number of games simulated grows to infinity. Therefore, given enough time and memory, UCT allows the process to converge to *minimax*, making MCTS optimal.

One of the reasons for the efficiency of MCTS with UCT in searching within high dimensional domains can be visualized in Figure 2.9: the tree grows asymmetrically focusing on the most promising branches instead of performing an exhaustive search. The tree that is being generated is unbalanced, as the sequences of actions acquiring higher rewards are those descending through the left side according to the simulations conducted.

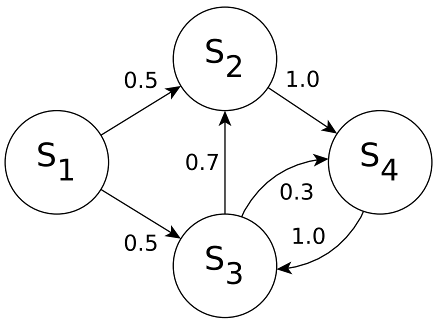


FIGURE 2.10: Markov chain

2.4 Markov chain

The Markov chain is a "simple" yet powerful formal model to represent fully observable states of a system with a random variable that changes over time according to some probability distribution. The definitions, theorems and lemmas of this section, along with their proofs, can be found in (Kemeny and Snell, 1983).

Definition 2.25 (Markov chain). *Let $\boldsymbol{P}$ be a $k \times k$ matrix with elements $\{P_{ij} : i,j = 1,...,k\}$. A random process $(X_0, X_1, ...)$ with finite space $S = \{s_1, ..., s_k\}$ is a Markov chain with transition matrix $\boldsymbol{P}$ if for all n, all $i,j \in \{1,...,k\}$ and all $i_0,...,i_{n-1} \in \{1,...,k\}$ we have*

$$\begin{aligned}&\mathbb{P}(X_{n+1} = j | X_0 = i_0, ..., X_{n-1} = i_{n-1}, X_n = i) = \\ &\mathbb{P}(X_{n+1} = j | X_n = i) = P_{ij}\end{aligned} \tag{2.7}$$

P_{ij} represents the probability of going from state s_i to state s_j at the next step. Equation 2.7 expresses the Markov property, i.e., the conditional probability distribution of the next state depends only on the current one. Such assumption, even though not realistic in some cases, is an acceptable approximation in many application domains. It is useful to represent a Markov chain as graphical model, visualizing states and transitions in the form of a directed graph. Vertices of the graph are the states of the Markov chain whereas edges are the transitions with probability values as labels. An example of Markov chain transition matrix is visible in Equation 2.8 and Figure 2.10 shows the corresponding graphical model.

$$\boldsymbol{P} = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.8}$$

The evolution of the process represented by a Markov chain can be computed using Theorem 2.2. In particular, from a starting probability distribution over the states, we can compute where the process transitions after the next n steps.

Theorem 2.2. *Let $\boldsymbol{P}$ be a transition matrix of a Markov chain and let $\boldsymbol{\mu}$ be the vector representing its initial distribution. Then the probability that the chain is in state s_i after n steps is the i-th entry of the vector:*

$$\boldsymbol{\mu}^n = \boldsymbol{\mu}\boldsymbol{P}^n \tag{2.9}$$

For example, using the transition matrix of Equation 2.8 and making the process start from state s_1, Theorem 2.2 tells us that the probability vector for being in a state after $n = 3$ steps is

$$\boldsymbol{\mu}^n = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} \begin{bmatrix}0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1.0 \\ 0 & 0.7 & 0 & 0.3 \\ 0 & 0 & 1.0 & 0\end{bmatrix}^4 = \begin{bmatrix}0 & 0 & 0.65 & 0.35\end{bmatrix}$$

Starting from s_1, after 3 steps there is a 0.65 chance of being in state s_3 and 0.35 of being in state s_4.

There are different types of Markov chains that depend on their characteristics. In this work we focus in particular on *irreducible* (Section 2.4.1) and *absorbing* Markov chains (Section 2.4.2) since their properties are useful for our methodologies.

2.4.1 Irreducible Markov chain

Irreducible Markov chains are easily understood thinking in terms of their graphical model representation, and in particular leveraging the concept of reachability.

Definition 2.26 (Irreducible Markov chain). *A set of states is irreducible if it is possible to go from each state to any other in an arbitrary (finite) number of steps. A Markov chain is irreducible if it consists of a single irreducible set.*

This means that if every state $s_i \in S$ is connected to any other $s_j \in S$ by a path in the graph representation, S is an irreducible set. A Markov chain can contain only some parts that are irreducible without being irreducible itself[5]. In Figure 2.10, states s_2, s_3, s_4 form an irreducible Markov sub-chain.

The long-term behavior of a process represented by a Markov chain can be computed for steps $n \to \infty$ using Theorem 2.2 until the result converges to a fixpoint (under some conditions detailed below). Such fixpoint is the *stationary distribution.*

Theorem 2.3 (Stationary distribution). *Given a Markov chain $\boldsymbol{P}$, the vector $\boldsymbol{\pi}$ such that $\boldsymbol{\pi}\boldsymbol{P} = \boldsymbol{\pi}$ is the stationary distribution of $\boldsymbol{P}$.*

The stationary distribution $\boldsymbol{\pi}$ represents the fraction of times a Markov chain will spend in each state when the number of steps n becomes large, i.e., as $n \to \infty$. The number of steps required for a Markov chain process to reach the stationary distribution is called *mixing time.*

Lemma 2.1. *For any finite, irreducible Markov chain, $\boldsymbol{\pi}$ is unique.*

As per Lemma 2.1, a Markov chain may have different stationary distributions unless finite and irreducible. In fact, if such properties do not hold, using Theorem 2.2 to compute the stationary distribution starting from different initial probabilities may lead to different results. We can retrieve the stationary distribution from the Markov chain with transition matrix $\boldsymbol{P}$ of Equation 2.8 as it is finite (4 states) and after the first step, the process remains in an irreducible set (s_2, s_3, s_4). In order to apply Theorem 2.3 we can compute the left eigenvector of $\boldsymbol{P}$ with eigenvalue 1.

[5]Some works define ergodic Markov chains the way we defined irreducible ones. We instead consider a Markov chain ergodic if it is aperiodic and positive recurrent, although they are not used in this work.

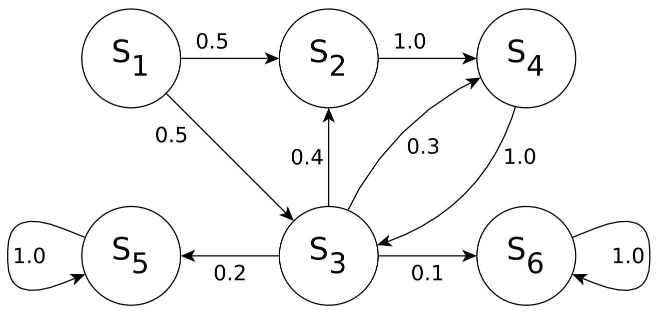


FIGURE 2.11: Absorbing Markov chain

$$\boldsymbol{\pi}\begin{bmatrix}0 & 0.5 & 0.5 & 0\\ 0 & 0 & 0 & 1.0\\ 0 & 0.7 & 0 & 0.3\\ 0 & 0 & 1.0 & 0\end{bmatrix} = \boldsymbol{\pi} = \begin{bmatrix}0 & 0.26 & 0.37 & 0.37\end{bmatrix}$$

In the long-term, the process is more likely to be in states s_3 and s_4 than s_2. Notice that it is impossible to end up in state s_1 after the beginning, since there is no incoming edge for such state. Indeed, the stationary distribution has components with positive values only for the irreducible parts of the Markov chain, i.e., only states s_2, s_3, s_4 in this example as state s_1 is unreachable.

2.4.2 Absorbing Markov chain

Another particular type of Markov chains is represented by absorbing Markov chains, characterized by two kind of states: transient and absorbing.

Definition 2.27 (Absorbing Markov chain)**.** *Given a Markov chain $\boldsymbol{P}$, a state s_i is absorbing if $P_{ii} = 1$, otherwise it is transient. A Markov chain is absorbing if at least one of its states is absorbing and if from every transient state an absorbing one will be eventually reached.*

Figure 2.11 shows an example of absorbing Markov chain where the absorbing states s_5 and s_6 can be reached by every other state (as by Definition 2.27).

If we deal with an absorbing Markov chain, it is usually preferable to reorder the states in a canonical transition matrix in order to clearly identify whether they are transient or absorbing. Such decomposition also comes in handy to easily compute different properties of an absorbing Markov chain (Theorems 2.4, 2.5, 2.6, and 2.7).

Definition 2.28 (Canonical form of an absorbing Markov chain)**.** *If an absorbing Markov chain $\boldsymbol{P}$ has n transient states and r absorbing states, its transition matrix can be rewritten as*

$$\boldsymbol{P} = \left[\begin{array}{c|c} \boldsymbol{Q} & \boldsymbol{R} \\ \hline \emptyset & \boldsymbol{I} \end{array}\right]$$

where:

- *$\boldsymbol{Q}$ is an $n \times n$ matrix of the transition probability between the transient states*
- *$\boldsymbol{R}$ is a $n \times r$ non-null matrix of the transition probability from the transient to the absorbing states*

– $\boldsymbol{\emptyset}$ *is a* $r \times n$ *null matrix*

– $\boldsymbol{I}$ *is a* $r \times r$ *identity matrix*

The canonical form of the Markov chain depicted in Figure 2.11 is reported in Equation 2.10

$$\boldsymbol{P} = \left[\begin{array}{cccc|cc} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0.4 & 0 & 0.3 & 0.2 & 0.1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{2.10}$$

where the decomposition is that of Equation 2.11

$$\boldsymbol{Q} = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0.4 & 0 & 0.3 \\ 0 & 0 & 1 & 0 \end{bmatrix} \boldsymbol{R} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.2 & 0.1 \\ 0 & 0 \end{bmatrix} \boldsymbol{\emptyset} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.11}$$

It is interesting to notice that the evolution of block matrix $\boldsymbol{Q}$ reflects the nature of transient states: since an absorbing state is reached every time a transient state is left (in any number of steps), the probability gradually "vanishes" from $\boldsymbol{Q}$ while being accumulated in the absorbing states. Therefore, after a given number of steps, the process is absorbed, meaning that the probability of being in any of the absorbing states is 1 while it is 0 everywhere else. Lemma 2.2 formalizes this behavior.

Lemma 2.2. *For any absorbing Markov chain in canonical form we have that* $\boldsymbol{Q}^k \to 0$ *as* $k \to \infty$.

Lemma 2.2 is useful to derive Theorems 2.4 and 2.6.

Theorem 2.4 (Fundamental Matrix of an absorbing Markov chain)**.** *The fundamental matrix* $\boldsymbol{N}$ *of an absorbing Markov chain* $\boldsymbol{P}$ *in canonical form is defined as*

$$\boldsymbol{N} = \boldsymbol{I} + \boldsymbol{Q}^1 + ... + \boldsymbol{Q}^k = \sum_{k=0}^{\infty} \boldsymbol{Q}^k = (\boldsymbol{I} - \boldsymbol{Q})^{-1}$$

where $\boldsymbol{I}$ is the identity matrix of the same size of $\boldsymbol{Q}$. Each entry N_{ij} represents the expected number of times that the chain is in a given transient state s_j if starting from the transient state s_i[6].

Lemma 2.3. *The inverse of* $(\boldsymbol{I} - \boldsymbol{Q})$ *is guaranteed to exist for every absorbing Markov chain*

Lemma 2.3 gives a theoretical guarantee that allows to always compute the fundamental matrix that is the common basis from which to derive Theorems 2.5, 2.6, and 2.7. With the example of Equation 2.11, the fundamental matrix is

$$\boldsymbol{N} = \left(\boldsymbol{I}_4 - \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0.4 & 0 & 0.3 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right)^{-1} = \begin{bmatrix} 1 & 1.83 & 3.33 & 2.83 \\ 0 & 2.33 & 3.33 & 3.33 \\ 0 & 1.33 & 3.33 & 2.33 \\ 0 & 1.33 & 3.33 & 3.33 \end{bmatrix}$$

[6]There exists a fundamental matrix also for Regular Markov chains.

In this case, state s_3 is expected to be visited 3.33 times on average before the process is absorbed, regardless of the starting state. State s_1 instead is visited only once and only when the process starts from s_1 itself since it is not possible to return to it.

From any starting state, it may be useful to compute how many steps the process takes before being absorbed.

Theorem 2.5 (Time to absorption)**.**

$$\boldsymbol{t} = \boldsymbol{N}\mathbf{1}$$

Each entry $\boldsymbol{t}_i$ represents the expected number of steps before being absorbed when starting from transient state s_i. $\mathbf{1}$ is a vector where all components have value 1. The time to absorption for the Markov chain of Equation 2.11 is

$$\boldsymbol{t} = \begin{bmatrix} 1 & 1.83 & 3.33 & 2.83 \\ 0 & 2.33 & 3.33 & 3.33 \\ 0 & 1.33 & 3.33 & 2.33 \\ 0 & 1.33 & 3.33 & 3.33 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 8.99 \\ 8.99 \\ 6.99 \\ 7.99 \end{bmatrix}$$

The process is absorbed faster when started from state s_3, whether the longest it takes is about 9 steps when started from s_1 or s_2.

Another useful quantity on which we build upon to extract the long-term probability from Markov chain models is the *transient states probability*.

Theorem 2.6 (Transient states probability)**.**

$$\boldsymbol{H} = (\boldsymbol{N} - \boldsymbol{I})\boldsymbol{N}_{dg}^{-1}$$

Each entry H_{ij} represents the probability of reaching transient state s_j starting from transient state s_i before the process is completely absorbed. $\boldsymbol{N}_{dg}^{-1}$ is the inverse of a diagonal matrix with the diagonal of $\boldsymbol{N}$. Continuing from the example of the Markov chain in Equation 2.11, the transient states probability is

$$\boldsymbol{H} = \left(\begin{bmatrix} 1 & 1.83 & 3.33 & 2.83 \\ 0 & 2.33 & 3.33 & 3.33 \\ 0 & 1.33 & 3.33 & 2.33 \\ 0 & 1.33 & 3.33 & 3.33 \end{bmatrix} - \boldsymbol{I}_4 \right) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2.33 & 0 & 0 \\ 0 & 0 & 3.33 & 0 \\ 0 & 0 & 0 & 3.33 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & 0.79 & 1 & 0.85 \\ 0 & 0.57 & 1 & 1 \\ 0 & 0.57 & 0.7 & 0.7 \\ 0 & 0.57 & 1 & 0.7 \end{bmatrix}$$

State s_2 is guaranteed to be reached before the process is absorbed when starting from any state but s_3 (where the probability value drops to 0.7).

The last quantity for absorbing Markov chains we are interested in is the *absorption probability*, that can be computed with Theorem 2.7.

Theorem 2.7 (Absorption probabilities)**.**

$$\boldsymbol{B} = \boldsymbol{N}\boldsymbol{R}$$

Each entry B_{ij} represents the probability of being absorbed from j-th absorbing state starting from transient state s_i. To conclude with our running example

$$\boldsymbol{B} = \begin{bmatrix} 1 & 1.83 & 3.33 & 2.83 \\ 0 & 2.33 & 3.33 & 3.33 \\ 0 & 1.33 & 3.33 & 2.33 \\ 0 & 1.33 & 3.33 & 3.33 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.2 & 0.1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.66 & 0.33 \\ 0.66 & 0.33 \\ 0.66 & 0.33 \\ 0.66 & 0.33 \end{bmatrix}$$

The process is more likely to be absorbed by state s_5 rather than s_6 with probability values of 0.66 and 0.33 respectively[7]. The starting transient state has no impact on the result in this example as all the rows of $\boldsymbol{B}$ are equal.

[7]The absorbing states of the Markov chain are s_5 and s_6, but their indices in $\boldsymbol{B}$ are 1 and 2 since $\boldsymbol{B}$ only contains columns for the absorbing states.

Chapter 3

Related Work

This chapter reports a variety of state-of-the-art approaches for malware analysis and behavioral modeling of recent years. Section 3.1 describes different techniques for malware analysis grouped in static, dynamic and hybrid; Section 3.2 presents AMA explaining the fundamental differences from previous techniques and why it is an important research line; Section 3.3 instead focuses on approaches for behavioral modeling of agents.

3.1 Malware analysis

Malware, or malicious software, play a part in most computer intrusion and security incidents. Any software that intentionally causes harm to a user, computer, or network can be considered a malware, which is defined by its malicious intent of acting against the requirements of the computer user. Malware do not include software that causes unintentional harm due to some deficiency.

The first line of defense in cyber-security is threat detection, and common antivirus or firewalls perform this activity by using some kind of known information, e.g., hash signatures of binaries, request of critical user permissions. Such prior information is typically obtained by threat analysis, aiming at achieving a better understanding of something recognized as a threat. Nowadays, malware analysis is mostly done by human security experts aided by specific tools, often built ad hoc for each new malware sample detected into the wild. These tools help to identify how malware penetrate a system, how they propagate to other systems and what is the payload. This is a time consuming process, often requiring to manually analyze binary code by executing it and examining logs. Malware *analysis* is the art of dissecting a malware to understand how it works, and consequently, how to identify it and what can be done to defend against it. This process is critical in order to be able to give a fast response to security incidents, since the number of malware is already in the order of millions and they grow at a very fast rate. When analyzing suspected malware, the goal will typically be to determine exactly what it can do, how to detect it, and how to measure and contain its damage. In particular, one of the main goals of malware analysis is to group malicious software with respect to common behaviors or to a predefined set of classes. Indeed, malware can be grouped in families (or types), that are behavioral categories in which malicious software fall into (Elisan, 2015). This is different from malware *detection*, where the aim is to distinguish harmless from malicious software: usually, malware analysis is performed after the result of the detection process flags a software as malicious, so as to identify the specific malware family and possibly use already known countermeasures to defend against the threat.

Most often, the only element available when performing malware analysis is the malware executable, which is not in a human readable form. There are two fundamental approaches to study such executable: *static* analysis and *dynamic* analysis. Static analysis examines a malware without actually running it. Dynamic analysis instead executes the malware to observe its behavior within a safe and controlled environment (a sandbox) (Sikorski and Honig, 2012).

3.1.1 Static analysis

Static techniques include several approaches such as frameworks for analyzing malware code producing a possibly approximated representation of the Control Flow Graph (CFG) for code similarity analysis. The approximation allows to reduce the size of the feature space introducing different layers of abstraction to handle the computational complexity of analyzing the CFG with abstract interpretation (Sharif et al., 2008; A. Lakhotia et al., 2013; Gao et al., 2008). There are also works in which a detection system mines malicious program logic from known malware by extracting compact representations of single programming functions to compare across different applications (Yang et al., 2014). A similar approach instead generates static signatures of every programming function that can be compared to others (Meng et al., 2016). Another branch of research focuses on the Call Graph (CG) of software: Markov chains of the connections between APIs can be extracted from the CG and the transition probabilities are then used to binary distinguish a malware from a harmless software, i.e., without considering the possible families (Mariconti et al., 2017). Alternatively, the CG is instead labeled with a weighted distance between neighbor APIs and such feature is used to classify between malware and benign software (Gascon et al., 2013; M. Zhang et al., 2014). The same goal is pursued in (L. Zhang et al., 2019), where authors aim at using non standard[1] static features and apply the widely used n-gram model (Wressnegger et al., 2013) to eXtensible Markup Language (XML) strings of the application executable resources.

Among the interesting works on static analysis there is DENDROID (Suarez-Tangil, Tapiador, et al., 2014), in which authors propose a static method to determine the distribution of particular structures embedded in Android applications and exploit such features to train a classifier to recognize the similarity degree between malware samples and family representatives. More specifically, DENDROID can be summarized with the following steps: it first decomposes the application under analysis into its code chunks, i.e., basic elements obtained through the retrieval of the CFG, each representing a single method of the program; once every sample of the dataset has been processed in order to gain a corresponding set of code chunks, the analysis proceeds by mining the code chunks inspiring to known text information retrieval methods. This phase requires the construction of a feature vector per sample where each entry represents a measure that recalls the Term Frequency-Inverse Document Frequency (TFIDF) index, each one accounting for a specific code chunk. Once this sub-task is accomplished for the entire dataset, a K-Nearest Neighbor (K-NN) classifier (with $k = 1$) is employed to estimate the malware family a sample belongs to.

The common thread of static analysis techniques is the engineering of feature selection from the binary code without actually executing the program. However, a problem of static methodologies comes from analyzing malware with encrypted malicious code deployed at runtime or obfuscated. Indeed, encrypted or obfuscated code is not in a readable form (unless the decryption key or the transformation applied to obfuscate are known), and for this reason static code inspection routines are unable to extract viable information. Often the code to run is downloaded at runtime, therefore

[1]Features that are not commonly used in malware static analysis

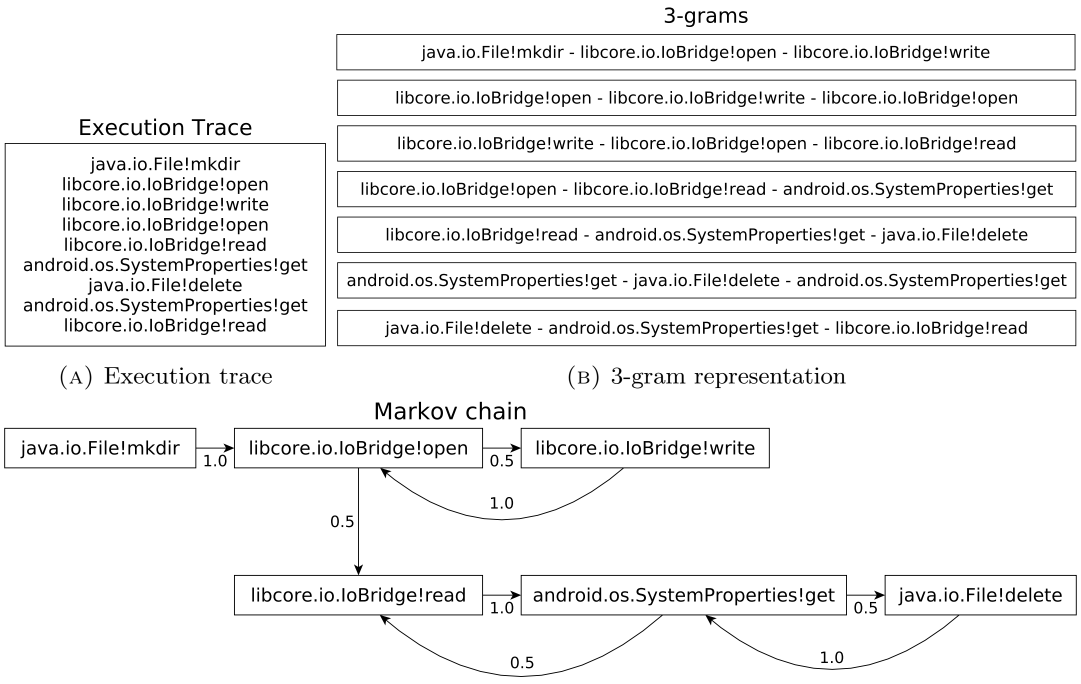
missing from the application binary and it is impossible to analyze at all (without running the application). Another limitation is related to the "actually executed blocks" problem: these approaches analyze a given program by building the associated CFG and then studying its properties, but the CFG (or other information extracted from the binary code) structure does not indicate the real execution flow of that program when it runs[2]. In other words, there is a gap between the code segments appearing in the compiled program and the statements actually performed at runtime, as these are a subset of the former. This characteristic represents a limit since, in a classification task, the feature extraction applied to such models will cause the presence of overflowing useless information in feature vectors, hence creating noise for the training phase and hence yielding potentially misleading responses. In the face of such limitations, in this work instead we focus on dynamic analysis techniques (and also because it naturally adapts to multi-agent approaches we are interested in).

3.1.2 Dynamic analysis

Dynamic techniques are fundamentally different from the static ones as they execute a program to observe its behavior in order to extract the features. An interesting work is (Rieck et al., 2011), in which authors leverage on the concept of n-grams as feature for detection extracted while observing a malware that is running within a sandbox. In particular, from the list of function calls performed during execution, n-grams are retrieved and used for classification with the aim of detecting the malware family of each malicious software analyzed. Even though good results can be achieved with such technique, a significant issue is the exponential space requirements when n increases. Moreover, since n-grams are an approximation of atomic behaviors embedded in malware, it is difficult to decide the proper granularity degree of the information represented through such feature type, i.e., how to select a proper value for n. There are also dynamic approaches that tweak the clock of a virtual machine by speeding up or down the execution of a software in order to be able to study malware with long delays between infection and payload execution (Lin et al., 2018).

Dynamic analysis typically suffers of the opposite problem with respect to static analysis: it is difficult to observe executions of the program that cover the entire code, i.e., the code coverage problem. However, the code coverage limitation is much less prominent in Android malware analysis since what is observed is usually relevant in the overall behavior, as the software are developed for the specific smartphone usage. Additionally, anti-emulation mechanisms are often employed by malware to abort execution if a sandboxed environment is detected instead of a real one. In fact, there is no reason for a malicious software to release its payload within an emulated environment, since almost no one employs them for everyday use. Conversely, malware analyzers, human or AI powered, almost always execute malware inside a controlled sandbox. This solution is adopted not only to precisely record the observed behavior, but also to avoid the disruption of real systems that are much more difficult to repair with respect to an emulator that can easily be reset from a snapshot. Furthermore, all the mentioned techniques suffer from an important limitation: they are *passive*, meaning that no interaction happens between the analyzer and the target program (in contrast to what a human security expert would usually do). Nonetheless, it has been assessed that in many cases interaction is fundamental to extract behaviors that are only exhibited when triggered since malware often try to mask their real intentions (Moser et al., 2007). Recently, a new type of dynamic analysis giving promising

[2]Rather it embeds every possible transition from a code section to another


Execution Trace
java.io.File!mkdir
libcore.io.IoBridge!open
libcore.io.IoBridge!write
libcore.io.IoBridge!open
libcore.io.IoBridge!read
android.os.SystemProperties!get
java.io.File!delete
android.os.SystemProperties!get
libcore.io.IoBridge!read
3-grams
java.io.File!mkdir - libcore.io.IoBridge!open - libcore.io.IoBridge!write
libcore.io.IoBridge!open - libcore.io.IoBridge!write - libcore.io.IoBridge!open
libcore.io.IoBridge!write - libcore.io.IoBridge!open - libcore.io.IoBridge!read
libcore.io.IoBridge!open - libcore.io.IoBridge!read - android.os.SystemProperties!get
libcore.io.IoBridge!read - android.os.SystemProperties!get - java.io.File!delete
android.os.SystemProperties!get - java.io.File!delete - android.os.SystemProperties!get
java.io.File!delete - android.os.SystemProperties!get - libcore.io.IoBridge!read
Markov chain
java.io.File!mkdir
1.0
libcore.io.IoBridge!open
0.5
libcore.io.IoBridge!write
1.0
0.5
libcore.io.IoBridge!read
1.0
android.os.SystemProperties!get
0.5
java.io.File!delete
0.5
1.0

(A) Execution trace

(B) 3-gram representation

(C) Markov chain representation

FIGURE 3.1: Example of different types of features extracted from an execution trace

results has been proposed: *active* malware analysis. For this reason we continue the discussion on dynamic analysis in Section 3.2, explicitly considering active techniques.

3.1.3 Hybrid analysis and feature selection

A combination of static and dynamic analysis features may help mitigating the limitations of each other, and the recent work of (A. Martìn et al., 2019) aims at doing this providing a dataset combining static and dynamic features. Nonetheless, fusing the information acquired by static and dynamic methods in meaningful features, or using static analysis to augment dynamic analysis, requires additional research. Static and dynamic information can also be fused by statically extracting the CG and observing the APIs performed during the execution, creating hybrid execution traces in order to exploit the correlation between different syntax but same semantics (Han et al., 2019). Feature selection is indeed important as attackers may exploit the knowledge about how a detection tool uses such features to mislead the process. A case study of this problem is provided in (Calleja et al., 2018), in which authors use evolutionary algorithms to perform a search that identifies a minimal number of changes to the features in order to induce family misclassification. They also analyze multiple defense models proposing how to strengthen them in order to be more resilient to feature thwarting attacks. In particular, different anti-virus software focus on different features, resulting in malware signatures that can be used to identify different families (classes) of malicious software, e.g., adware, spyware, ransomware, as in the work of (I. Martìn et al., 2019). Finally, notice that the type of features used by static and dynamic analysis often coincide, e.g., n-gram on APIs observed during execution and XML string embedded in the executable file, although the process used to extract them is completely different. Figure 3.1 shows an example of how an execution trace can be represented with different features, namely 3-grams and a Markov chain.

3.2 Active malware analysis

This section presents different techniques for AMA dividing between non-intelligent analyzers that do not reason about the adversary during the analysis process, and intelligent analyzers that employ rational strategies to interact with the malware agents.

3.2.1 Non-intelligent active malware analysis

The goal of AMA is to perform actions in order to trigger a malware into showing behaviors that would otherwise remain hidden. If a malware reacts to a triggering action, it means that the activation condition has been met and a payload is usually deployed. For example, in (Suarez-Tangil, Conti, et al., 2014), authors build an analyzer that aims at reproducing specific activation conditions to trigger malicious payloads relying on stochastic models extracted by past samples of real user behaviors that have been recorded. In particular they make extensive use of geo-localization based on the Global Positioning System (GPS). Another interesting approach is (Bhandari et al., 2018), where random triggers are used in a runtime semantic-aware malware detector resilient to code injection and capable of deciding whether a software is malicious or not. In (A. Martìn et al., 2018), authors describe a dynamic process to conduct the analysis of malware, called CANDYMAN, followed by the classifier training phase. A core aspect of this work is the wide range of supervised algorithms considered and evaluated, involving almost all classical learning methods and also deep learning techniques. In summary, CANDYMAN involves the execution of the malware under analysis in a controlled environment where it can be safely run while observing its behavior. All the data collected are then processed in order to construct a Markov chain model that expresses the malicious dynamics observed during the previous analysis step. Once this modeling phase is terminated, the features embedded in the resulting Markov chains are extracted and used for a classification task. CANDYMAN also applies some approximations on the Markov chains to lower the number of states composing the Markov chains and, in the feature extraction part, to reduce the feature space used to subsequently train a classifier. The active procedure proposed in (A. Martìn et al., 2018) is very effective in assessing malware families, however the triggering policy targets the Graphical User Interface (GUI) with random actions, hence without employing an intelligent strategy.

3.2.2 Intelligent active malware analysis

All the active techniques mentioned so far present no rational target-oriented strategy to stimulate the malware under analysis. Therefore future research is required to devise an approach that can select the triggering actions so to maximize the acquired information. A first step in such a direction comes from the work of (Williamson et al., 2012), that proposes to design an autonomous agent with an active role in the disclosure of malicious intents. The method sees an analyzer being a player aiming at the acquisition of non-trivial information regarding the opposite player, i.e., the malware. The process is formalized as a stochastic game (Definition 2.22) called AMA in which the analyzer performs an action on the system and the malware responds with a sequence of actions that change the status of such system.

Definition 3.1 (Active malware analysis game). *An AMA game is a stochastic game where:*

- *$G(V, E)$ is a weighted graph representing the system on which the analysis is going to be performed, where V is the set of vertices and E is the set of directed* *edges connecting the vertices. Each vertex is a state of core components of the system, e.g., important flags in the registry, and the edges represent the transitions in the state of the system that a malware can induce. The weight on each edge represents the transition probability from the edge source vertex to the destination one*

- $v^m \in V$ *is the current malware* n_2 *vertex position in the graph and the sequence of those positions through the graph, represent affected system components*
- $a_2 = v^m \cup \textsc{neighbours}(v^m)$ *is the strategy space of the malware* n_2 *and represents the changes it can take from its current location*
- $V^H \subset V$ *is the set of actions that can be played, described by the subset of leaf nodes of the graph. The analysis agent,* n_1*, may then execute actions on the system*
- $a_1 = V^H$ *is the strategy space of the analysis agent* n_1*. The agent can create new simulated user activities and remove previous ones. Locations on the graph are used to represent some fake user activity or data in place, with the preceding vertices representing state changes before this information is introduced*
- $S = V \times V^H$ *is the global state space, given by the possible locations of the malware,* v^m*, and of the actions,* v^h
- $\pi : S \rightarrow a_2$ *is the fixed, possibly stochastic, but unknown policy of agent* n_2 *giving the next action(s) from a specific state*
- $U : \pi \rightarrow \mathbb{R}$ *is the reward function for agent* n_1 *associated with learning the policy and preferences of agent* n_2
- $v^0 = s^1$ *will always be the malware starting location in the clean system*
- P^H *is the set of paths from vertex* v^0 *to each of the possible actions* V^H *and each path is defined as a sequence of edges* $e_0, ..., e_k$

The analysis game requires a fixed model manually designed by a security expert that specifies which components of the systems to monitor for changes by the malware and which components the analyzer can interact with to trigger a response in the malware. Such model is represented with a Directed Acyclic Graph (DAG) where leaf vertices are the possible actions of the analyzer, internal vertices are possible actions of the malware, and paths in the graph are sequences of actions that the malware can take to reach a component (leaf vertex) touched by the analyzer. During the analysis game, the analyzer selects a component of the system to interact with and observes the malware reaction updating the transition probabilities on the model graph. At the end of the process, the transition matrix obtained represents the policy of the malware with respect to the pre-specified model, that is fixed and remains the same during the analysis of every malware. Authors implement the policy of the analyzer agent with a procedure called MYOPIC (Algorithm 3.1) where the next analyzer action to play is chosen with a 1-step look ahead heuristic based on the entropy of the malware execution patterns. Using the statistics (historical frequency) on how the malware transitioned between vertices of model, the analyzer action with highest entropy is selected to be played next. The intuition is that actions with higher entropy usually retrieve more information. Consequently, the malware policies extracted by the analyzer are strictly dependent on the quality and level of detail of

Algorithm 3.1 MYOPIC

```
Require:
    $m$ - malware model
    $limit$ - time limit
Ensure:
    $\pi$ - policy in the form of transition function $T$

1: for all actions $a \in m$ do
2:     Set probability distribution $T_a$ as uninformative
3:     $R_a \leftarrow \infty$
4: for $t \leftarrow 1$ to $limit$ do                ▷ Play until time limit is reached
5:     $a \leftarrow$ ChooseBestAction($m$)          ▷ Get best action using model
6:     $trace \leftarrow$ Execute($a$)               ▷ Execute action and observe malware
7:     $R_a \leftarrow m$.Entropy($a$)               ▷ Compute reward
8:     $T_a \leftarrow$ Adapt($T_a, trace$)          ▷ Adapt transition function based on observation
9:     $m$.Update($T_a, R_a, a$)                     ▷ Update model
```

the model employed: since a model represents the behavioral patterns on which to play the analysis game, the accuracy of its specification impacts whether the analysis computes meaningful policies. Indeed, if the model does not contain an important component of the system to observe as a malware possible action, that will be ignored when performed. Human expertise and manual effort is then fundamental to build a good model for a meaningful analysis process. The necessity of the model as fixed input is therefore an important limitation of the work.

In the following we describe a brief example of the reducing entropy heuristic employed in MYOPIC. The malware model used is visible in Figure 3.2, in which the possible analyzer actions are 4 and 5 and possible paths are $\{[1,2,4,5],[1,2,4,6],[1,3,4,5],[1,3,4,6]\}$. The edge conditional probability is computed based on the observation of the malware behavior based on the analyzer action (the historical frequency), assuming an initial uninformative distribution. Specifically, if a malware has visited vertex v for l times and an edge $e \in E_v$ has been taken m times, when the action executed by the analyzer was a, the conditioned probability becomes

$$Pr(e \mid a) = \frac{1+m}{|E_v|+l} \tag{3.1}$$

A path is defined as a sequence of edges $e_0, ..., e_k$ and the probability for a malware of taking a path $p \in P$, when the action executed by the analyzer is $a \in V^H$, can be computed as

$$Pr(p \mid a) = \prod_{e \in p} Pr(e \mid a) \tag{3.2}$$

The entropy of a path $p \in P$, when the action executed by the analyzer is $a \in V^H$, is computed as

$$H(p) = -Pr(p \mid a) \ln Pr(p \mid a) \tag{3.3}$$

Finally, the entropy of analyzer's action $a \in V^H$ is

$$H(a) = \sum_{p \in P} H(p) \tag{3.4}$$

The steps in the example process are

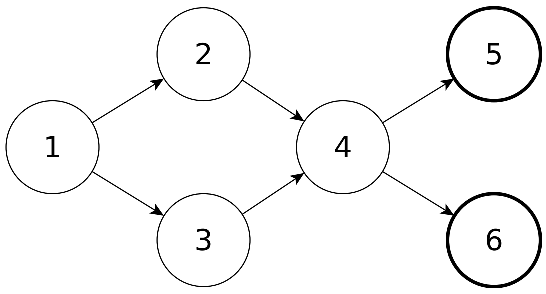


FIGURE 3.2: MYOPIC model example

Action	Edge Probabilities						Path	Path	Total
	1-2	1-3	2-4	3-4	4-5	4-6	Probability	Entropy	Entropy
5	0.5		1		0.2		0.1	0.23	1.2
		0.5		1		0.8	0.4	0.37	
	0.5		1		0.2		0.1	0.23	
		0.5		1		0.8	0.4	0.37	
6	0.4		1		0.3		0.12	0.25	**1.28**
		0.6		1		0.7	0.28	0.36	
	0.4		1		0.3		0.18	0.31	
		0.6		1		0.7	0.42	0.36	

TABLE 3.1: MYOPIC example, step 1

1. after having played the game for a few timesteps, the situation is the one of Table 3.1. The historical frequency says that for analyzer action 5, the malware is more likely to follow either $[1, 2, 4, 6]$ or $[1, 3, 4, 6]$. For action 6 instead, the path followed with highest probability is $[1, 3, 4, 6]$

2. MYOPIC chooses as next action to play, the one giving highest entropy, which is 6 in this case

3. malware reaction is to follow the path $[1, 3, 4, 6]$ changing the transition function values to those visible Table 3.2

4. at this point, the entropy associated to action 5 has not changed while that of action 6 dropped, but without becoming lower than that of action 5. For this reason, MYOPIC chooses action 5 again

5. malware follows again the path $[1, 3, 4, 6]$, resulting in transition function values of Table 3.3

6. the situation has changed: now action 5 is the one with lowest entropy, therefore MYOPIC would continue the game by choosing action 5

Our proposed approaches for AMA differ from (Williamson et al., 2012) as we avoid the weak point represented by the fixed and pre-specified model by dynamically generating the malware model at runtime with a RL algorithm based on MCTS. Specifically, we define a different behavioral model for malware that represents the observed execution traces, composed by sequences of API calls, as Markov chains. Therefore, the model does not depend on the analysis system such as in (Williamson et al., 2012), but only on the malicious software itself. During the generation process, our analyzer agent selects the best action to play taking into account the possible responses

Action	Edge Probabilities						Path	Path	Total
	1-2	1-3	2-4	3-4	4-5	4-6	Probability	Entropy	Entropy
5	0.5		1		0.2		0.1	0.23	1.2
		0.5		1		0.8	0.4	0.37	
	0.5		1		0.2		0.1	0.23	
		0.5		1		0.8	0.4	0.37	
6	0.36		1		0.27		0.1	0.23	**1.23**
		0.64		1		0.73	0.26	0.35	
	0.36		1		0.27		0.17	0.3	
		0.64		1		0.73	0.47	0.35	

Table 3.2: MYOPIC example, step 2

Action	Edge Probabilities						Path	Path	Total
	1-2	1-3	2-4	3-4	4-5	4-6	Probability	Entropy	Entropy
5	0.5		1		0.2		0.1	0.23	**1.2**
		0.5		1		0.8	0.4	0.37	
	0.5		1		0.2		0.1	0.23	
		0.5		1		0.8	0.4	0.37	
6	0.33		1		0.25		0.08	0.2	1.18
		0.66		1		0.75	0.25	0.34	
	0.33		1		0.25		0.16	0.3	
		0.66		1		0.75	0.5	0.34	

Table 3.3: MYOPIC example, step 3

of the malware with multiple steps of simulations with a MCTS. The capability of dynamically generating the model at runtime makes the analysis more flexible since it is not tied to a pre-specified input, with a great impact on final classification results for malware identification.

Broadly, the concept of executing specific actions to perform a better analysis can be linked to the general framework of active learning, and recently there has been a specific interest in applying active learning techniques to malware analysis (Nissim et al., 2014). In that work, authors propose the use of an Support Vector Machine (SVM) classifier to select which samples (already analyzed) should be fed to the classifier, so to refine the classification bounds. In this thesis instead, we aim to generate malware models that can be studied by a human security expert or processed by automated techniques (with clustering or classification) for comparison. Hence, we focus on the decision making side of the analysis by devising an intelligent strategy for the analyzer action selection, differing from active learning approaches where the methodology is usually tied to the specific choice of classifier in order to improve the classification bounds, e.g., K-NN and naïve Bayes (K. Wei et al., 2015), logistic regression (Y. Guo and Schuurmans, 2007), linear regression (Yu et al., 2006), SVM (S. Tong and Koller, 2002).

3.3 Agent Behavioral Modeling

The main focus of this thesis is to analyze and generate behavioral models extracted by observing the effect of the actions performed by a target agent within a fully observable environment. In this context, Markov chains are particularly suited for the task and have been widely used in literature to model the behavior of agents. The work of (Whittaker and Thomason, 1994) views reliability analysis of software programs as a sequence generation and analysis problem, where Markov chains are used to represent execution patterns. A state of the model contains the values of the program's most important variables whereas the transitions consider the effect of each possible input on a state. To verify a program, the Markov chain structure of normal execution is used by a testing process that assigns transition probabilities based on the historical frequency of the tests conducted. If a program behaves as required, the Markov chain of testing will converge to the same transition function of the usage Markov chain. In (Sarukkai, 2000), Markov chains are employed to infer probabilistic link prediction and path analysis in HyperText Transfer Protocol (HTTP) web navigation. From navigation logs of users on a web site, a Markov chain is created for each session where states represent web pages (or resources in general) and transitions[3] instead represent the links that have been followed in the log to move between pages. With the application of Theorem 2.2 it is possible to predict the possible future $n+1, \ldots, n+k$ pages that a user will visit after the current page n. The quality of the prediction is strictly tied to the training process, for which a big amount of training data (user navigation logs) is required; however, the amount of traffic nowadays is very sustained, making the process of data collection for web navigation quite easy. Nevertheless, the number of states representing the resources of a web network may be huge, consequently decreasing the prediction reliability. In order to circumvent this problem, the work of (Zhu et al., 2002) proposes to compress the transition matrix by merging states with similar incoming and outgoing transitions. This approach allows an efficient computation for the prediction of the next probable states at the cost of precision (to a varying degree depending on the approximation). The agent behavioral models obtained are often required to be compared (or analyzed with respect) to each other. In the context of Markov chain models, the works of (Dyer et al., 2006; Busic et al., 2009) provide methods to compare Markov chains relying on the mixing time or directly computing the stationary distribution (Theorem 2.3). However, the existence of a stationary distribution is guaranteed only for Markov chains with specific properties (see Section 2.4). This limits the applicability of (Dyer et al., 2006; Busic et al., 2009) to domains in which the stationary distribution can be computed.

Modeling the behavior of an agent can be a challenging task, especially if such agent has goals that are in contrast to those of the analyzer or observer. An interesting work in this context is presented in (Hernandez-Leal and Kaisers, 2017), where authors propose a framework for stochastic games (Definition 2.22) aimed at learning the policy of multiple unknown adversaries drawn from different populations. In particular, an agent faces an unknown opponent that changes after every few interactions as selected by a random process from a reference population. The agent does not know when a new opponent is drawn, and the population is partitioned into different groups (hence there are different types of opponents). In order to identify the opponents, authors propose an algorithm that first generates policies, game transitions and performances (in an offline phase) based on the known information. During an online phase instead, the game is played choosing one of the policies at disposal with a belief based approach, and then observing the opponent behavior (in terms of transitions and rewards). After the online phase, the algorithm updates the policies with the new acquired information and then plays again. With respect to our reference application, i.e., AMA, an important element that is not addressed in (Hernandez-Leal

[3]Probability values are estimated using the historical frequency

and Kaisers, 2017) is that an opponent may intentionally perform some random or completely unrelated actions in order to mask her real policy, injecting noise in the behavioral model. Consequently, the observer agent can be deceived if she does not consider such potential deviation during the analysis process.

Part II

Behavioral Model and Analysis Framework

Chapter 4

Behavioral Model

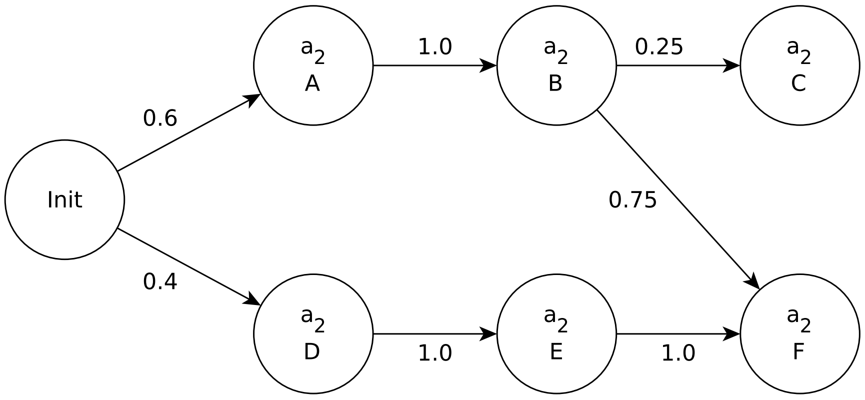
In this chapter we present the formalization of the behavioral model that will be used throughout all the thesis. Although the main focus is malware analysis, and consequently malware models, our formalization is generic enough to be applied to any kind of model that is extracted by the observation of a behavior in response to some interaction (more on this in Chapter 9). Section 4.1 introduces the Markov chain as representation of the observed behavior in response to some actions; Section 4.2 presents the specific case of behavioral models for malware where malicious behaviors are triggered by analyzer actions; Section 4.3 explains how to extract the representation of the transition function as a policy when comparing different behavioral models.

4.1 Behavioral model of the interaction

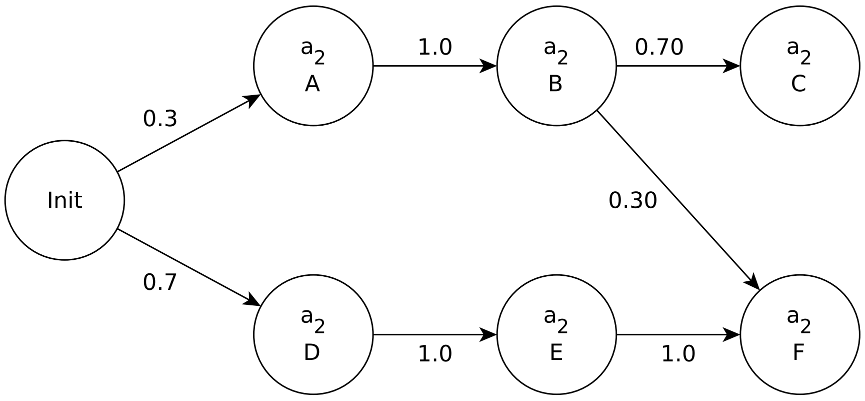
In this thesis we deal with an agent that observes the behaviors of another agent after an interaction between the two takes place. In particular, the action of the first agent a_1 triggers a behavior of the second agent a_2, where a behavior is composed by the sequence of actions of agent a_2 that agent a_1 has observed. In order to model this, we employ the widely adopted model of the Markov chain. As presented in Section 2.4, Markov chains are a formal model to represent the evolution of a process over time. In our case, the process is represented by the behavior of agent a_2 that we want to learn: the states are her possible actions and the transitions represent the connections between such actions. A path in the resulting Markov chain then will be a behavior, i.e., sequence of actions, where we always add an *Init* state from which every behavior starts for consistency. Given a Markov chain then we can study the behavior represented by it, however we are interested in the interaction that made such behavior emerge. In order to do this, we associate every action performed by agent a_1 to the Markov chain representing the behavior of agent a_2 and that was stimulated by such action. More in detail, we condition the probability values of the transitions in the Markov chain model with the action of agent a_1 that triggered that transition. Figure 4.1 shows an example of the model we just described: notice that considering only the probability values with label X we obtain the Markov chain of Figure 4.1a, whereas considering the label Y we obtain the one of Figure 4.1b. Those Markov chains represent two different behaviors of agent a_2 that take place when different actions are performed by agent a_1, i.e., X and Y respectively.

4.2 Markov chain based malware model

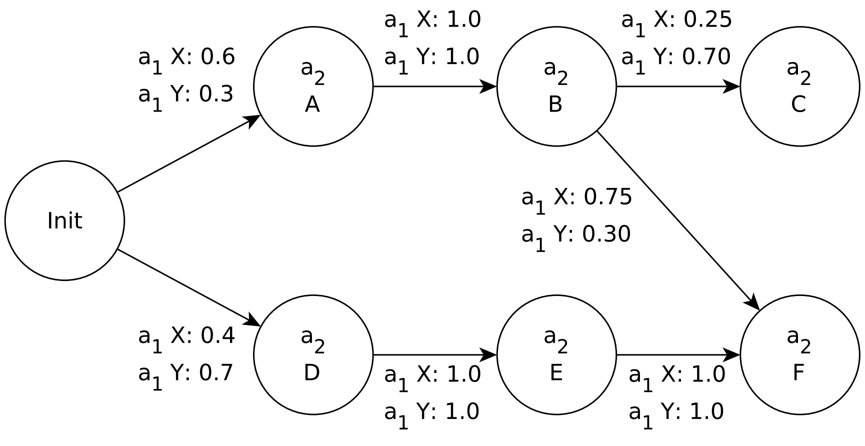
In the context of malware analysis, the process that we described in Section 4.1 involves an analyzer agent and a malware as interacting intelligent agents. The states of the Markov chain then become malware actions whereas the labeling on the edges



(A) Markov chain representing the behavior of agent a_2 in response to the triggering action X of a_1



(B) Markov chain representing the behavior of agent a_2 in response to the triggering action Y of a_1



(C) Behavioral model resulting from the composition of the behaviors depicted in Figures 4.1a and 4.1b

FIGURE 4.1: Example of behavioral model as a result of the observation of multiple behaviors in response to triggering actions

that conditions the transition probability values represent the analyzer actions that triggered the behavior observed in the malware. Figure 4.2 shows an example of the structure of a malware model, in which the partial model on the top is generated from scratch after the observation of the malware reaction to a triggering action X. Next, the model is updated by adding the observation of the reaction to the triggering action Y. In fact, using the historical frequency of the malware actions observed, we can update a model with new interactions and subsequent observations. Therefore, models can be updated anytime in case of necessity, for example if more actions for the analyzer become available after the initial analysis. The reason why we represent states of the Markov chain with malware actions is that sequences of such actions are what constitute different behaviors of the malware: a path in the Markov chain is a possible execution trace, hence a possible behavior we are interested in modeling. Indeed, what we observe of the malware agent are its actions on the system, and different malware perform different sequences of actions when triggered by the analyzer. Obviously, since we use a Markov chain, we approximate what could be the next action of a malware given the last one executed. However, this approximation is acceptable in our application domain since we work with the probability associated with a behavior (therefore a sequence in the model, or part of it), rather than single specific actions alone. Nonetheless, when information associated to the internal state of the agent to analyze can be observed, the states of the Markov chain can be represented with such information instead of the actions performed. A behavior then becomes a sequence of changes in the internal state of the agent. This is the case for the experiments conducted in Section 9.1.3, where states of the Markov chain are represented by the value of the internal accumulated reward for the agent playing a lottery.

Our aim is to make the behavioral model independent from the system on which AMA is performed and for this reason the state space of our approach is different from that of Definition 3.1. As explained in Section 3.2, the state and transition spaces in the work of (Williamson et al., 2012) correspond to the parts of the system (on which the analysis is going to be performed) to monitor for changes during the analysis process. There are two main limitations to this solution: first, if the system for the analysis changes, so does the model, therefore all the policies extracted have to be recomputed in order to reflect the changes; second, the identification of the system resources to monitor and their connections require human effort and expertise. Consequently, we define AMA directly on the Android APIs that compose an execution trace of the malware, so as to not be bound to the specific underlying system for the analysis. For example, one could employ this behavioral model also for Windows malware, where the action set of the malware, i.e., the set possible APIs, will obviously be different from Android, but the formalization remains exactly the same. The result of the analysis then will be a different model for every specific malware analyzed, incorporating the dynamics of the interaction that happened between analyzer and malware. In particular, a sequence of malware actions in the model becomes an execution trace of the malicious program, i.e., a behavior.

4.2.1 Platform independent model

It is worth discussing what we mean by platform independent model. Malware targeting different operating systems are engineered in a different way. Moreover, the specific programming language can cause differences in how malware are written, even with respect to the same platform. There is however a common feature that is always present in every implementation of malware code: API calls are performed during the

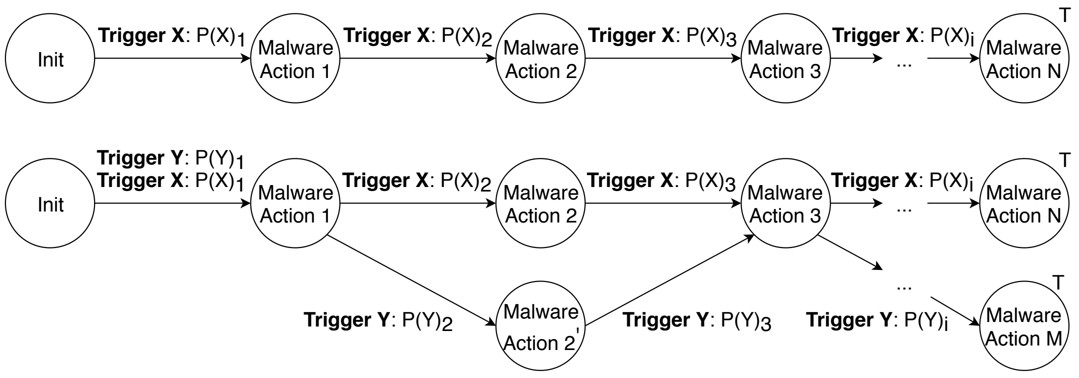
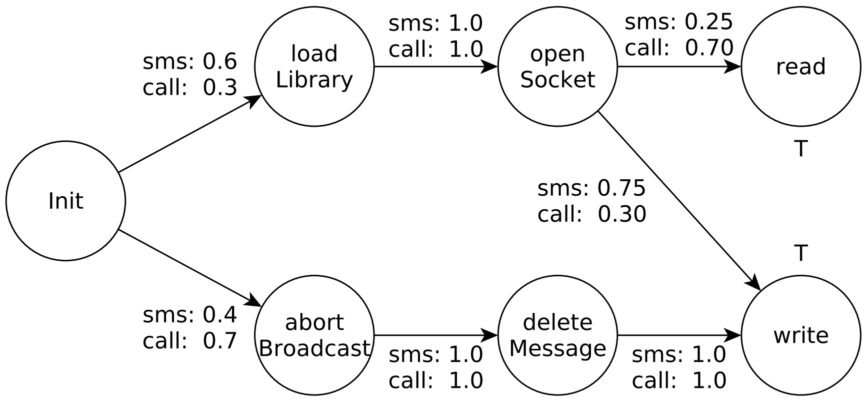


FIGURE 4.2: Instance of a malware behavioral model after the observation of the reaction to the triggering action X (on top) followed by a Y (below)

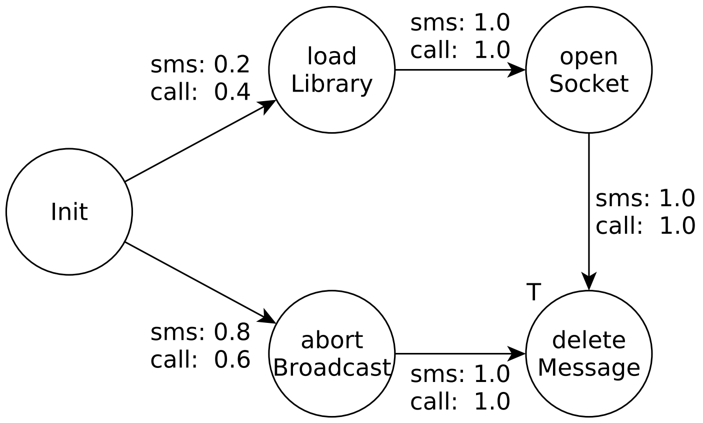
execution of any software (malicious or not). Now, these APIs are different depending on the programming language and/or operating system, nevertheless they are always present and they form sequences that can be represented by Markov chains. In our case, the programming language does not pose any particular problems as for the Android system all the applications are written in Java, therefore the APIs are consistent across every application. For other operating systems, e.g., Windows, it may be easier to observe the malware execution at a different level of abstraction in order to avoid the differences in API layers between software. Indeed, there are different levels of abstraction from which the Markov chain models can be created: instead of representing states with APIs, the system calls performed at kernel level could be employed, as the actual implementation of the APIs rely on system calls, e.g., opening a file, input and output operations etc. This solution makes the observations consistent with respect to the underlying operating system, even when using different programming languages, and so APIs. Another example of abstraction is that of (Williamson et al., 2012), presented in Section 3.2, where components of the underlying systems represent states, e.g., registry flags, files. The methodologies developed in this thesis are built upon the Markov chain representation of the models, independently on the actual meaning of a state, as long as a path in the model can reliably represent the execution flow of the malware, with respect to the analyzer triggering action, observed while generating such path. Naturally, the practical usage of our approaches to different operating systems requires a non indifferent engineering effort to implement the sandboxing environment, to decide the appropriate abstraction level that is informative enough for the type of malware that will be analyzed, to implement the analyzer as an agent that can perform actions on the system of choice, and to tune the various parameters of the algorithms that we developed. What remains unchanged is the theoretical work that serves as a basis to our techniques that rely on theoretical frameworks such as Markov chains and Bayesian or stochastic games and that are not influenced by the level of abstraction (as long as it is meaningful enough, as previously explained).

4.2.2 Malware model design

We design the malware model in order to store the information collected by the analyzer during the analysis process. As previously explained, such information is represented by the execution traces observed in response to the analyzer action that triggered them. Figure 4.3 shows two examples of real Android malware models. Vertices are labeled with malware API calls, while edges connect two consecutive



(A) Malware model example A



(B) Malware model example B

FIGURE 4.3: Example of two malware models based on execution traces as sequences of APIs conditioned by analyzer actions *sms* and *call*

API calls of an execution trace and are labeled with transition probabilities (using the historical frequency) conditioned by the actions executed by the analyzer. The labeling is crucial to represent behaviors that are triggered by specific stimuli. Considering a single analyzer action by keeping only its label on the edges, e.g., *sms*, we obtain a Markov chain (Definition 2.25) representing the behavior observed in response to such action. Hence, models are compositions of multiple Markov chains, one for each analyzer action executed on the system during the analysis. Additionally, if a vertex is labeled with an API call that terminates one or more malware execution traces, such vertex is marked as terminal (T). Any path from the initial vertex to a terminal depicts a possible API execution sequence of the malware, i.e., a behavior. Looking at Figure 4.3b for example, we can see that if the analyzer simulates an incoming *sms*, the malware is more likely to perform the execution trace composed by $loadLibrary, openSocket, deleteMessage$ rather than $abortBroadcast, deleteMessage$.

4.3 Model comparison

Given a malware model it is possible to extract the policy of such malware represented as the transition function between API calls. For any analyzer action a, let $v_{r,a}$ be the number of times vertex v is reached under a, and let $e_{t,a}$ be the number of times the outgoing edge e of v (the set of outgoing edges of v is indicated with E_v) has been traversed under a. Then, given a model $G = (V, E)$, we apply Equation 4.1 to retrieve the conditioned probability value for each tuple (a, v, e) of G:

$$P(e \mid a) = \frac{1 + e_{t,a}}{|E_v| + v_{r,a}} \tag{4.1}$$

Whenever $v_{r,a} = 0$ or $e_{t,a} = 0$, the resulting probability value will be uninformative, reflecting the fact that we have no information on how the malware behaves in such case.

As previously explained, the end goal of AMA is to group malware models based on shared characteristics and behaviors. The type of countermeasures to employ depends on the type of malicious software, and in particular, detecting whether a malicious behavior shares common characteristics with known malware families is extremely important to take effective countermeasures. Our model formalization enables to compare and group (possibly) similar malware by extracting their compatible vectorial representation. First, the model graphs of the malware we want to compare are fused together by merging vertices based on API call labels. Then, the transition function of each model is projected on the merged graph using Equation 4.1. This results in a feature vector for every sample where the features are the probability values associated to the transitions between model states. Such features can then be used to perform tasks such as classification and clustering using any standard machine learning tools and algorithms. To see how the comparison operation works, consider malware models A and B of Figure 4.3: the comparison procedure would fuse the models obtaining a merged graph of the same shape of Figure 4.3a, since it includes the one of Figure 4.3b. Notice that in the general case, this will not happen if there are some states that are not shared between the two models to merge. However, the same process applies: such states would still appear in the merged model and the computation of probability values on outgoing edges follows Equation 4.1. Then, the projection of the transition functions of A and B with Equation 4.1 would result the in the following

$$\begin{array}{lcccccccccccccccccc}
 & & & & \mathit{sms} & & & & & & & & & \mathit{call} & & & & \\
A = & [0.6 & 0.4 & 1 & 0.28 & 0.58 & 0.14 & 1 & 1 & \mid & 0.3 & 0.7 & 1 & 0.62 & 0.30 & 0.08 & 1 & 1] \\
B = & [0.2 & 0.8 & 1 & 0.20 & 0.20 & 0.60 & 1 & 1 & \mid & 0.4 & 0.6 & 1 & 0.17 & 0.17 & 0.66 & 1 & 1]
\end{array}$$

The behavioral model presented in this chapter is used throughout all the thesis: in Part III the models are dynamically generated at runtime and are used to guide the analyzer action selection strategy; in Part IV we propose a long-term analysis of the behavior represented by the models; in Part V, the model is again dynamically generated by the analyzer at runtime, although the analysis process is completely different with respect to Part III, i.e., a new formalization.

Chapter 5

Analysis Framework

In order to conduct experiments on malware analysis we need a safe environment under our control for the execution of the malicious software. One of the standard techniques used in these cases is to employ a sandboxed emulator in which to run malware and observe their behavior in the controlled environment. Section 5.1 gives a brief overview of the Android operating system and the structure of Android applications; Section 5.2 introduces the concept of Android operating system emulation; Section 5.3 presents the analysis framework architecture that we built on the top of the Android operating system; Section 5.4 details the software tools employed to build the sandbox; Section 5.5 lists all the possible triggering actions available for the analyzer to perform during the analysis process.

5.1 Android Operating System

Android is an operating system that runs on the Linux kernel. It has been developed by Google since 2005 and it is released under the open source Apache license. On May of 2019, Google announced that there are currently 2.5 billion active Android devices, that are popular with technology companies that require a ready made, low cost and customizable operating system for high tech devices. Its open nature has encouraged a large community of developers and enthusiasts to use the open source code as a foundation for community driven projects, which add new features for advanced users or bring Android to devices originally shipped with other operating systems. At the same time, as Android has no centralised update system, most Android devices fail to receive security updates: research in 2015 concluded that almost 90% of Android phones in use had known but unpatched security vulnerabilities due to lack of updates and support. Consequently, Android has become one of the preferred targets of malware designers (Cheung, 2018).

5.1.1 Android architecture

The main hardware platform for Android is the Advanced RISC Machine (ARM) architecture and its kernel is based on one of the Linux kernel. On the top of this, there are the middleware, libraries and APIs written in C, and application software running on an application framework which includes Java compatible libraries. Applications are written in Java, but to achieve better runtime performance, the entire bytecode is compiled into machine code upon application installation.

Android's standard C library, Bionic, was developed by Google specifically for Android, as a derivation of the standard C library code. Bionic has been designed with several major features specific to the Linux kernel and optimized for low frequency Central Processor Units (CPUs) (Yaghmour, 2013).

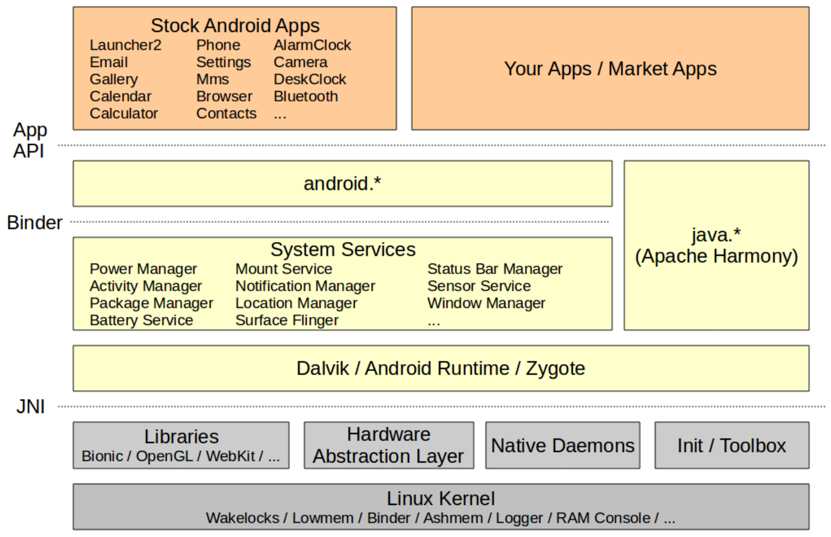


FIGURE 5.1: Android operating system architecture

5.1.2 Android security

Android applications run in a sandbox, an isolated area of the system that does not have access to the rest of the system's resources, unless access permissions are explicitly granted by the user when the application is installed. The sandboxing and permissions system lessens the impact of vulnerabilities and bugs in applications, but developer confusion and limited documentation has resulted in applications routinely requesting unnecessary permissions, reducing its effectiveness. Malware designer often exploit the permission system tricking the standard user in allowing an application to have unrestricted access to the system, making easier for malicious applications to release their payload.

Research from security company Trend Micro lists premium service abuse as the most common type of Android malware, where text messages are sent from infected phones to premium rate telephone numbers without the consent or even knowledge of the user. Other malware display unwanted and intrusive adverts on the device, or send personal information to unauthorised third parties. One of the last and most dangerous discoveries in Android malware, are those intercepting bank, such as the ones containing One Time Password (OTP), to steal money (Yaghmour, 2013). Notice that the malware types just described require user interaction to show the malicious behaviors: for this reason an analysis based on triggering actions is fundamental to be able to extract comprehensive information about malware.

5.1.3 Android application

An Android application (app) is distributed as a package with *.apk* extension and has the following structure:

- Java bytecode, compiled from Java source code, in a *classes.dex* file;
- resource files like images, sounds and others;
- *.xml* files, included the *AndroidManifest.xml*.

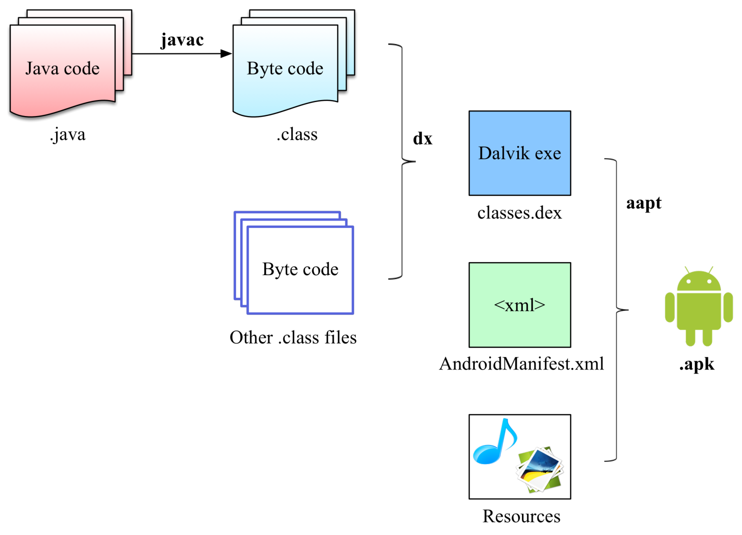


FIGURE 5.2: Android application structure

Android apps are made of loosely tied components. Components of one application can invoke or use components of others. There is no single entry point to an Android app, instead, there are predefined events called *intents* that developers can tie their components to, thereby enabling their components to be activated on the occurrence of the corresponding events.

Components

There are four types of components:

- **Activities**: the main building block of all visual interaction in an Android app;
- **Services**: Android services are akin to background processes or daemons in the Unix world;
- **Broadcast Receivers**: Broadcast receivers are akin to interrupt handlers. When a key event occurs, a broadcast receiver is triggered to handle that event on the app's behalf;
- **Content Providers**: Content providers are essentially databases. Usually, an application will include a content provider if it needs to make its data accessible to other apps.

Intents

Intents are the late binding mechanisms that allow components to interact. An application may request to open a web page even if it has not the capability to do so. Components can be declared as capable of dealing with given intent types using filters in the *manifest* file. The system will thereafter match intents to that filter and trigger the corresponding component at runtime. The intents, together with the components are what make an application interact with the system and accept input from the user. Therefore they represent the entry point for most triggered malicious behavior that an analyzer wants to intercept and observe.

Manifest

It informs the system of the application's components, the capabilities required to run the app, the minimum level of the API required, any hardware requirements, etc. The manifest is formatted as an XML file and resides at the top most directory of the application's sources as *AndroidManifest.xml.* It is used by many mobile antivirus to statically obtain information about possible application behaviours or to generate a signature.

5.2 Android emulation

Executing a malware, even if purposely, requires a safe environment in which analysis can be performed without permanent damages. This allows to avoid the disruption of the physical machine in use or others linked to that since malware can act in many unexpected ways. A possible solution could be building a completely isolated computer network as a test platform without internet access, called air gapped network. An approach like this, lowers the risk of spreading an infection, but is highly limiting: the majority of malware needs an internet connection to show their complete malicious potential. Hence, performing an analysis in such an environment gives imprecise and incomplete results. Moreover, if one or more of the computers get damaged in an irreparable way (as it is likely to happen), they need to be completely reinstalled and this is a time consuming task. Drastic consequences are not the only reason a computer operating system must be reverted to a situation back in time, before something happened. It is often the case that, for studying a malware, it is necessary to run it multiple times from exactly the same state of components. This is hardy achievable using normal computers.

A safe environment in which executing malicious software, resolving the problems presented before, is the sandbox, or rather a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third parties, suppliers, untrusted users and untrusted websites. A sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as scratch space on disk and memory. Network access, the ability to inspect the host system or read from input devices are usually disallowed or heavily restricted. A sandbox is implemented by executing the software in a restricted operating system environment, thus controlling the resources that a process may use. To build our analysis framework we make use of an Android emulator, which is a piece of software that simulates the Android hardware. An emulator does not execute code directly on the underlying hardware (since it may be different), instead, instructions are intercepted, translated to a corresponding set of instructions for the target platform, and finally executed on the hardware. Thus, whenever a sensitive instruction is executed (even when it is unprivileged), the system emulator is invoked and can take an appropriate action.

5.3 Framework architecture

The framework we built as a controlled sandbox is composed of three main modules: the environment manager (the guest machine emulator), the behavioral observation module and the decision making routine for the analyzer, i.e., triggering action selection. Figure 5.3 depicts the framework architecture reporting the three mentioned modules. In particular, given an application to analyze, first the emulator is started and after the boot is completed the application is installed and executed. To avoid

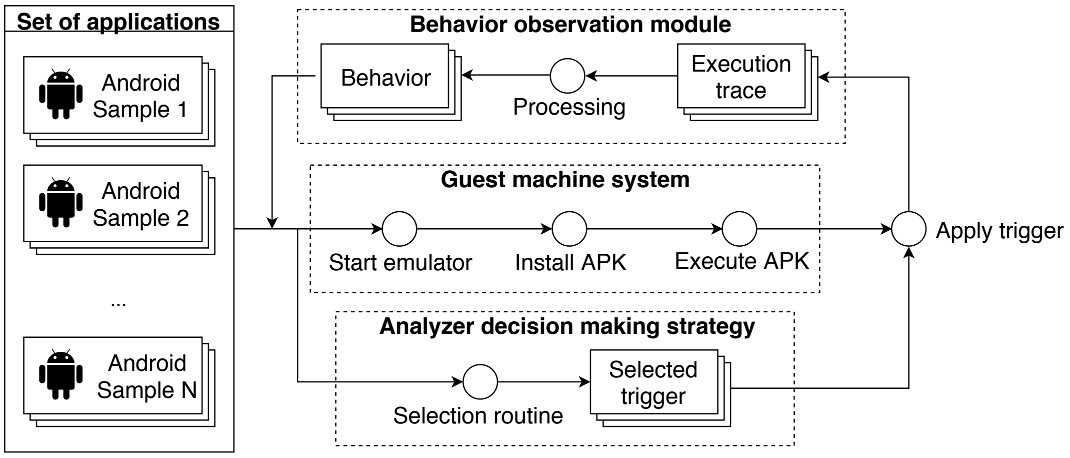


FIGURE 5.3: Framework architecture

downtime, the analyzer trigger selection operation is executed in parallel with the setup of the environment operation. At this point, the environment is ready and a trigger has been selected, hence it can be applied by the analyzer in order to extract the response as an execution trace. Every module can be easily substituted with another of the same type: for example, the trigger selection depicted in the figure is a MCTS for the approach described in Chapter 6, whereas it plays a stage of Bayesian game in the approach drawn in Chapter 10. One could to the same for the guest machine, which is Android in our case, creating a module to work with Windows or other operating systems.

5.4 Experimental platform

The experimental platform we built to implement the framework, is composed by a set of tools based on an emulator acting as a sandbox. The virtual environment is a Genymotion[1] emulator mounting an Android 6.0 image. The main tools employed include Xposed[2], a modular framework that can monitor and alter the behavior of the system and applications hosted within the guest side of the emulator, and Oracle, an ad-hoc Xposed module we developed to hook API calls performed by the malicious software under analysis. This last component provides also some measures to neutralize common anti-detection techniques employed by malware to avoid dynamic analysis if an emulated environment is detected. Oracle is designed to inject itself in the applications to analyze and intercepts all the parts of such application that make it interact with the system: activities, services, receivers, providers, intents (as described in Section 5.1.3).

5.5 Triggering actions

The action set for the analyzer agent is composed of 17 different actions that mimic a standard user's behavior:

[1]`https://www.genymotion.com`
[2]`https://repo.xposed.info`

1. send SMS
2. receive SMS
3. make call
4. receive call
5. add contact
6. enable wifi
7. disable wifi
8. activate screen
9. deactivate screen
10. install application
11. uninstall application
12. change GPS position
13. enable GPS
14. disable GPS
15. internet navigation from default browser
16. charge battery
17. discharge battery

The choice of these triggering actions is important as they are purposely generic, i.e., not application specific, and can be applied to a wide variety of applications of different nature, e.g., games, banking applications, web navigation etc. Moreover, those actions are commonly performed by the standard user during normal usage of the smartphone.

The framework presented in this chapter is used in all the empirical evaluations of our proposed approaches. Specifically, the different techniques that we designed have been implemented in modules for this framework that has subsequently been used to analyze a dataset of real Android malware composed of about 1400 samples partitioned into 24 different families that have been selected from a bigger dataset (F. Wei et al., 2017). The rationale for the selection process of the malware families to analyze is explained in Section 7.1.

Part III

Dynamic Generation of Malware Behavioral Models

Chapter 6

A Monte Carlo Tree Search Approach to Active Malware Analysis

In this chapter we propose a novel AMA technique based on MCTS that dynamically generates the malware model at runtime. The partial model generated up to a certain point of the analysis is also used to guide the reminder of the analysis since it serves as a basis for the action selection strategy of the analyzer. Section 6.1 presents our SECUR-AMA framework, an improved AMA approach for Android systems; Section 6.2 details the analysis pipeline with respect to the framework described in Chapter 5; Section 6.3 provides a running example of the SECUR-AMA analysis process with particular attention to the MCTS execution.

We differ from previous approaches described in 3.2 as they rely on human expertise to decide which actions should be executed by the analyzer (also in the form of a manually designed model), or implement automated random action selection strategies. In this chapter we show that using an intelligent strategy for the analyzer decision making strategy (such as the one we propose) has a significant impact on the results in terms of malware identification.

6.1 SECUR-AMA

The SECUR-AMA analysis framework has been published in (Sartea, Farinelli, and Murari, 2020) as a refinement of the prototype version we previously proposed in (Sartea and Farinelli, 2017). Following (Williamson et al., 2012), introduced in Section 3.2, we formalize our framework as a stochastic game between an analyzer agent and a malware, where the former tries to acquire information about the latter by stimulating it and observing the reaction. This analysis design is justified by works where practical problems of interest are represented as multi-agent systems (in our context we deal with the specific case of two-agent systems) in which intelligent and autonomous entities interact within a complex dynamic environment learning information and adapting their behavior accordingly. However, as explained in Section 4.2, we consider the execution trace of API calls as response from the malware instead of monitoring system resources.

Definition 6.1 (SECUR-AMA game). *The analysis game of SECUR-AMA is a stochastic game with*

- $N = \{n_1, n_2\}$ *where* n_1 *is the analyzer and* n_2 *is the malware*
- $\boldsymbol{A} = A_1 \times A_2$ *where*

  - A_1 *consists of all the possible triggering actions for the analyzer (make a call, connect to WiFi, etc.)*
  - A_2 *consists of all the possible execution traces of the malware*

- S *is the set of states, each one defined by an instance of malware model*
- $\boldsymbol{u} = (u_1, u_2)$ *where* $u_1 : S \times \boldsymbol{A} \times S' \to \mathbb{R}^+$ *is the utility function for the analyzer*
- $T : S \times \boldsymbol{A} \times S' \to \mathbb{R}_{[0,1]}$ *is a probabilistic transition function*

SECUR-AMA is clearly an instance of a stochastic game of Definition 2.22 between two players: the analyzer n_1 and the malware n_2. The action set A_1 available to the analyzer comprises all the possible triggering actions that can be performed on the system and that could possibly cause a reaction in the adversary (make/receive call, enable/disable GPS, etc.). The malware action set A_2 instead includes all the possible execution traces (sequences of API calls) that a malware can exhibit. A state of the game is defined by an instance of malware model that the analyzer generates during the analysis process (Section 4.2). The utility function is defined only for the analyzer (Section 6.1.2) and is based on the entropy gain between the model of current state s and next one s': such transition between states is governed by function T using the joint actions of the players. While playing SECUR-AMA we do not take into account rewards for the malware since our focus is to analyze it by extracting the highest amount of information on the adversarial behaviors, rather than to counter a malware by minimizing its disruption potential. At the same time, malware are not aware of the fact that they are playing a game, therefore they do not "compute a strategy" based on what the analyzer may do in each situation. As such, computing an equilibrium for the game would not be beneficial for the analyzer. Hence, the formalization reflects the point of view of the analyzer on the game and the reward function is designed to guide the information gathering on the malware.

The aim is to learn the malware behavioral model by using the minimum amount of analyzer actions. Nevertheless, in contrast to (Williamson et al., 2012), where a fixed and pre-specified model is provided in order to play the analysis game, we do not have access to such information (the "playground"). Consequently, the main problem to solve is how to handle the huge action space available to the malicious agent while planning the most promising analyzer action, as there are many possible sequences of API calls that can be triggered in response. To tackle the mentioned problems we decided to leverage on MCTS (Section 2.3.3) as it is a suitable approach to visit big search spaces and also very flexible depending on how the tree and default policies are implemented (Browne et al., 2012). We present the analysis technique with a top-down approach starting with the high-level view of the process and then explaining the technical details.

6.1.1 Monte Carlo analysis

The Monte Carlo Analysis (MCA) procedure is presented in Algorithm 6.1, where the malware model is generated incrementally by collecting the information extracted from the previous responses to the analyzer actions. Initially, the algorithm starts with an empty, hence uninformative, model (line 1). Next, the analyzer chooses the best action[1] to play on the system (in order to stimulate the malware) with a MCTS that runs multiple simulations using the information contained in the current model (line 4). Once the resulting choice is returned, it is concretely executed on the system

[1]The one supposed to obtain the most information possible from the malware.

and the malware response is recorded as a sequence of API calls in a log file (line 5). Parsing the log, the current model is updated both on structure and statistics as transition probabilities between API calls (line 6). At this point the game moves to the next state associated to the new malware model. The algorithm ends when a computational stopping condition is met, usually corresponding to a game length, thus a fixed number n of analyzer triggering actions. The output is given by the last model obtained.

Algorithm 6.1 Monte Carlo Analysis

```
Require:
    $n$ - game length
Ensure: Malware model

1: $model \leftarrow \emptyset$                          ▷ Start with empty model
2: for $n$ times do
3:     $tmpmodel \leftarrow model$.COPY()
4:     $a \leftarrow$ MCTS($tmpmodel$)                     ▷ Choose next action
5:     $trace \leftarrow$ EXECUTE($a$)                     ▷ Observe malware reaction
6:     $model$.UPDATE($trace, a$)
7: return $model$
```

6.1.2 Monte Carlo tree search implementation

As previously mentioned, MCTS is a very flexible algorithm that can be adapted to many application domains depending on the implementation of the tree and default policies presented in their generic tasks in Section 2.3.3. In the following we detail our policy design choices for SECUR-AMA.

Tree policy

The role of the tree policy is to descend the search tree from the root to a leaf node by following the most promising branch. This allows to generate a tree without performing an exhaustive search, saving precious computational resources. The decision of which node to select at each level of the descent is modeled as a multi-armed bandit problem addressing the exploration-exploitation dilemma. In fact, a trade-off between selecting actions that appear to be promising and actions belonging to sub-spaces not well sampled yet and that may turn out to be optimal in the future, represents a suitable solution to our problem. Taking inspiration from the proposal of (Kocsis and Szepesvári, 2006), we employed UCT, often applied to multi-armed bandit contexts, to compute an estimate of the optimality value associated to every possible choice. For details refer to the background Section 2.3.4

Default policy

The default policy is responsible for conducting a simulation from the state of the game corresponding to an expanded node of the tree to a leaf corresponding to the termination of the game plus one more step (motivated in Section 6.1.2). The process simulates multiple interactions between the analyzer agent and the malware, at the end of which a reward for the former is estimated based on the outcome related to the conclusion of the game. As the reward is used by the tree policy to focus only on promising branches of the tree (Equation 2.6), a bad default policy could make the entire MCTS imprecise or even useless. In our implementation the simulation is composed as a sequence of iterations where the analyzer selects a triggering action and the malware responds to it with an execution trace of API calls. This informally described procedure is reported in Algorithm 6.2 and is part of the MCTS routine (line 4 of Algorithm 6.1).

Algorithm 6.2 Default Policy

```
Require:
    $n$ - expanded node
    $tmpmodel$ - temporary model
Ensure: Resulting model of the simulation

1: $a \leftarrow n.action$
2: repeat
3:     $trace \leftarrow \textsc{Simulate}(a)$
4:     $tmpmodel.\textsc{Simulate}(trace, a)$
5:     $a \leftarrow \textsc{ChooseAction}(tmpmodel)$
6: until end of the game
7: return $tmpmodel$
```

There are two crucial operations to perform in the default policy, the first of which being SIMULATE (line 3) to predict malware responses. Given the number of all possible execution traces, we devised a simple yet effective simulation strategy: if the analyzer triggering action a has never been concretely executed by the analyzer on the system previously in the game, we produce random API sequence for the malware response. Otherwise, a past observed API sequence is produced with the same historical probability associated to the observation of such trace in response to a. Consequently, at the beginning of the game the simulation is purely random, but it becomes more and more accurate as the game evolves. The second important operation is CHOOSEACTION (line 5) which implements the analyzer strategy while playing the game. Recall that every state of the underlying game is associated to a malware model, therefore every node of the search tree is also associated to a temporary malware model m corresponding to the state of the game represented by such node (composed by a mix of simulated malware responses generating during the MCTS and actually observed ones). We adopted an information-centric reward based on entropy aimed at selecting the action a that has maximum entropy value $H_m(a)$ in the current temporary model m.

$$H_m(a) = -\sum_i D_m(a)_i \ln D_m(a)_i \tag{6.1}$$

$D_m(a)$ is the probability distribution for reaching a vertex i, labeled as terminal (T) in the model m, starting from the initial vertex while considering the analyzer action a. The motivation for this choice of value function is that usually higher entropy is an indicator of a more informative action. Since the aim is to generate a model for the malware that is as informative as possible (without having access to the code, but only to the observation of behaviors), minimizing the entropy of the model under construction is a reasonable solution for the analyzer to confidently learn how a malware agent reacts to the stimuli. For the model depicted in Figure 4.3a, the computation of the entropy for action *sms* uses Theorem 2.2 as follows for $n = 3$ steps

$$\begin{aligned}
D_m(sms) &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0.6 & 0 & 0 & 0.4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0 & 0.75 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^3 \\
&= \begin{bmatrix} 0 & 0 & 0 & 0 & 0.15 & 0.85 \end{bmatrix} \underset{read,\, deleteMessage}{\Longrightarrow} \begin{bmatrix} 0.15 & 0.85 \end{bmatrix} \\
H_m(sms) &= -0.15 \cdot ln(0.15) + 0.85 \cdot ln(0.85) = 0.423
\end{aligned} \tag{6.2}$$

Reward backpropagation

The reward for the analyzer is meant to represent the amount of information extracted by performing a triggering action in the environment. In particular, during the back-propagation step, the reward accumulated in each node of the path between the root and the expanded node is the entropy gain $H_m(a) - H_{m'}(a)$ computed with Equation 6.1, where a and m are the action and the temporary model associated to a node, while m' is the temporary model obtained at the end of the simulation step. Notice that for the nodes at maximum depth (length of the game) $H_m(a) - H_m(a) = 0$, hence we make the simulation of MCTS go one step further (to depth $n+1$) and compute the entropy gain from that model. Such value gives an estimate of how much information the analyzer might acquire by playing a certain sequence of actions from the current state of the game to the end of it.

6.2 Analysis pipeline

Figure 6.1 depicts the analysis pipeline by reporting three main operations: setup of the environment, generation of the behavioral model and analyzer trigger selection. In particular, given an application to analyze, first the emulator is started and after the boot is completed the application is installed and executed. To avoid downtime, the analyzer trigger selection operation is executed in parallel with the setup of the environment operation. At this point, the environment is ready and a trigger has been selected, hence it can be applied by the analyzer in order to extract the response as an execution trace that will update the current application model. The process is iterated for a predefined number of times $n = 30$, corresponding to the length of the analysis game. The MCTS then is set to reach a maximum depth of 30 and the reward to backpropagate is computed from the model obtained at that point. Based on our architecture and pipeline, the stopping condition for the MCTS instead is met when the emulator has finished booting and the application is installed and running, ready to be stimulated. In case of unsatisfying results and depending on the computational power available, it is possible to let the MCTS run longer, up to a predefined number of iterations or by increasing the waiting time before selecting the outcome.

6.3 Running Example

The process of stimulating a malware is iterated for a predefined number of times $n = 30$, corresponding to the length of the analysis game. The MCTS then is set to

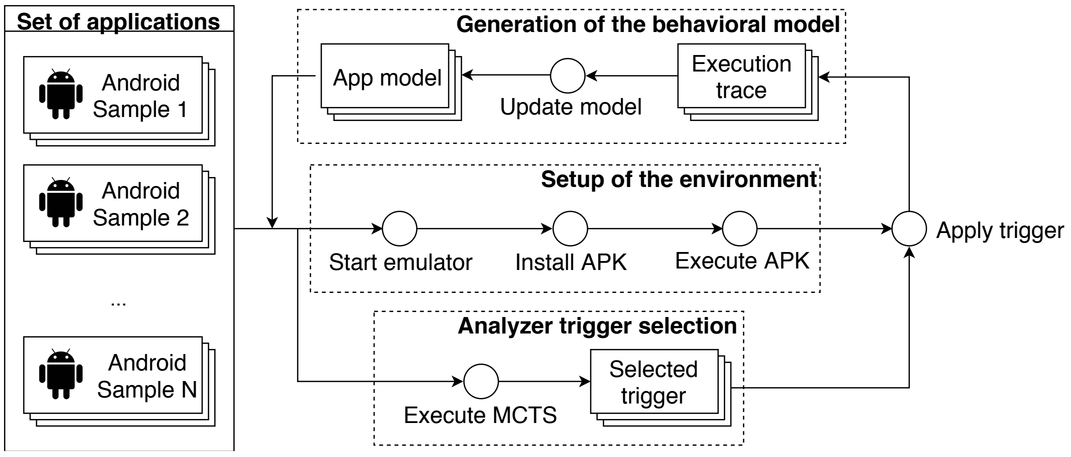


FIGURE 6.1: SECUR-AMA analysis pipeline

reach a maximum depth of 30 and the reward to backpropagate is computed from the model obtained at that point. Based on our architecture and pipeline, the stopping condition for the MCTS instead is met when the emulator has finished booting and the application is installed and running, ready to be stimulated. In case of unsatisfying results (in terms of correctness of the models and with respect to the final classification score) and depending on the computational power available, it is possible to let the MCTS run longer, up to a predefined number of iterations or by increasing the waiting time before selecting the outcome. In fact, increasing the computational power of the analyzer allows to compute the next action to perform more accurately since MCTS is a CPU bound process and the more iterations can be performed in the time that has been allocated, the better the tree will estimate future interactions between analyzer and malware. In the next paragraph we provide a running example of the analysis process.

Suppose that the stochastic game of SECUR-AMA is currently in state s after 5 stages (5 analyzer-malware interactions) out of 30 (total game length). The state s is defined by the malware model s_m generated so far and the analyzer has a history of past malware execution traces s_h observed in response to specific triggering actions. The analyzer has now to select the 6^{th} triggering action to play next by running a MCTS with input model s_m and maximum depth $30 - 5 = 25$. Figure 6.2 provides a running example of a MCTS action selection operation. Every node of the tree contains the name of its triggering action and a couple (n, X) where n is the number of visits and X the accumulated reward for that node. The example starts at the beginning of the 16^{th} iteration (notice that $n = 15$ for the root node). The selection step visible in Figure 6.2a descends the tree by computing the UCT (Equation 2.6) for every node at the same level to select the one with highest value. While descending, the input model s_m is updated with a malware execution trace in response to the analyzer triggering action a corresponding to the node traversed (using the history s_m or simulating a new trace as per Section 6.1.2). The green arrows highlight the selected descent path. The expansion of the selected node is represented in Figure 6.2b where also the n values of the nodes in path have been updated (green) according to the selection step. Figure 6.2c shows that from the expanded node, the remaining number of interactions to the end of the game (23 to reach $30 + 1$) are simulated based on the history of the current analysis s_h (see the default policy of Section 6.1.2), resulting in a model m'. In particular, at the first step of the simulation the analyzer selects the triggering action with associated highest entropy in the malware

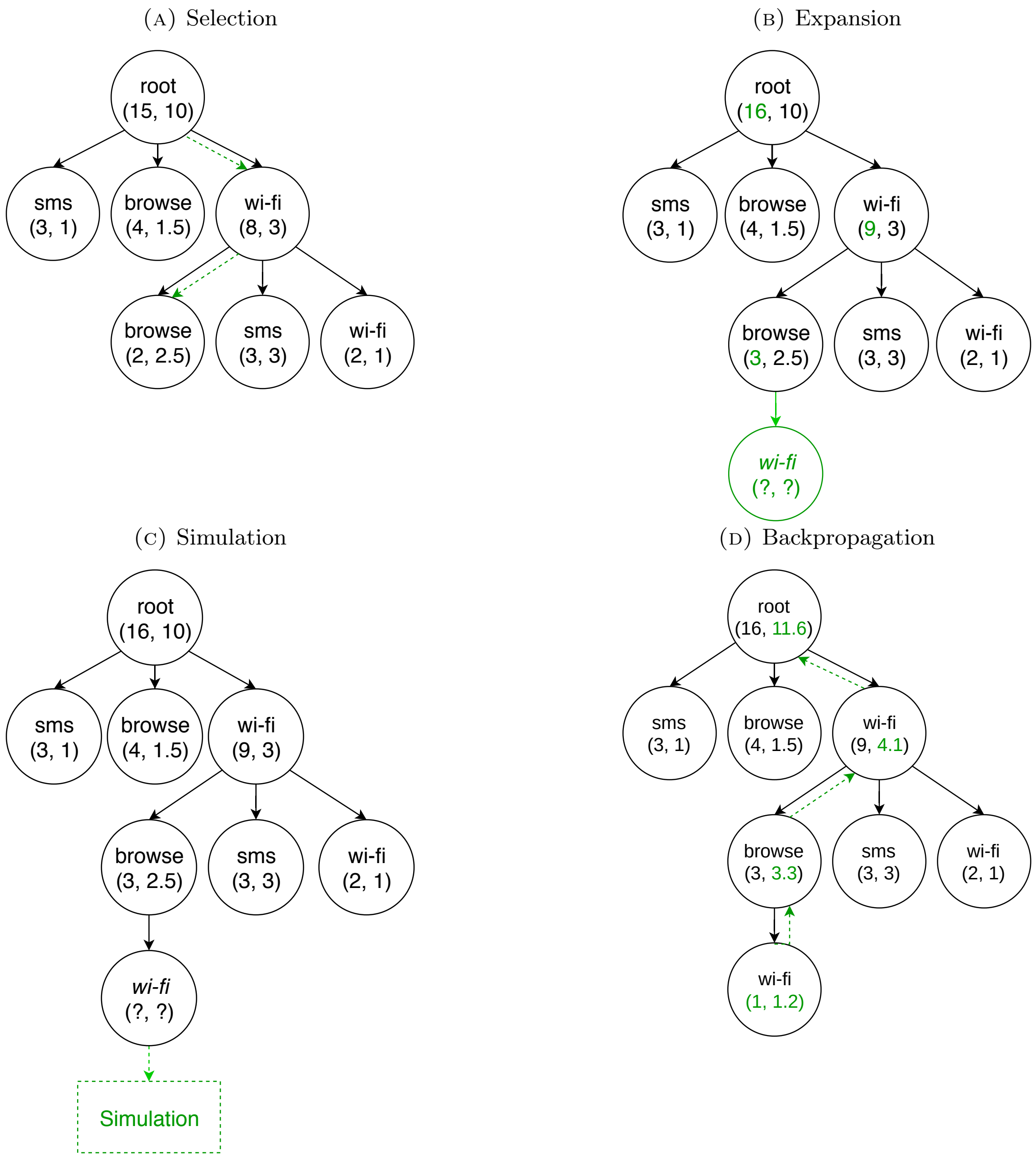


FIGURE 6.2: MCTS running example

temporary model corresponding to the expanded node, simulates a malware response to such action, updates the temporary model and moves the game to the next state to continue the simulation. In the backpropagation step (Figure 6.2d), for each node the information gain is computed with the entropy difference $H_m(a) - H_{m'}(a)$, where a and m are the triggering action and the temporary model tied to a node respectively, from which rewards are updated accordingly (green). For an example of entropy calculation see Equation 6.2 in Section 6.1.2. The 16th iteration is completed and if the MCTS process was to be stopped now, the selected action would be the most visited child node of the root, i.e., "wi-fi".

Chapter 7

Empirical Evaluation of SECUR-AMA

This chapter reports a detailed empirical evaluation of SECUR-AMA compared with other state-of-the-art techniques, namely DENDROID (Suarez-Tangil, Tapiador, et al., 2014) and CANDYMAN (A. Martìn et al., 2018). In malware analysis, the first step to analyze an unknown software is to decide whether it could be malicious (Aafer et al., 2013). Having decided that a program contains malicious code, a main task is represented by the identification of the specific malicious family a software belongs to in order to use proper (possibly already known) countermeasures to handle the potential threat. Our method aims to accomplish this identification goal employing standard machine learning techniques to train various classifiers, namely K-NN, Random Forest and Linear SVM. In particular, we want to obtain models that are as informative as possible, i.e., a reliable representation of malware behaviors in response to specific analyzer actions, leveraging on the information entropy. Lower entropy means probability values on the transitions between APIs that are further away from uninformative. Therefore, entropy is an internal measure that SECUR-AMA optimizes (minimizing it) during the analysis process, whereas the F_1-score is used as external measure to evaluate the results of the classification. Section 7.1 reports the details of the dataset of real Android malware used in the experiments; Section 7.2 explains the experimental methodology adopted to conduct the empirical evaluation; Section 7.3 reports the obtained results; Section 7.4 presents the runtime performance of the analysis process; Section 7.5 concludes the chapter with final considerations and possible future directions.

7.1 Dataset

The dataset we used to test our approach has been built by selecting approximately 1400 malicious programs partitioned into 24 different families from the samples collected in (F. Wei et al., 2017). The number of families is comparable to those of other dynamic analysis techniques (CANDYMAN for example reports experiments for 24 families as well). One of the reasons that guided our selection process is that, in contrast to the majority of the works on Android dynamic analysis, we built our framework to use the 6^{th} (more recent) version of the Android operating system, instead of the 4^{th}. This is a specific choice we made in order to conduct an empirical evaluation that is realistic with respect to the malware that is spreading nowadays (or in recent times). This is very different from all the other dynamic analysis techniques (at least those that are publicly accessible), that are only able to analyze older malware that are no more around or new malware executing on older versions of the operating system (when possible at all). Many malware that were written for Android 4 or below are not able to perform malicious behaviors anymore on newer versions since the security layer of Android has radically changed and improved. In this regard, many of the malware contained in older datasets, e.g., “Genome”, simply do not run on newer versions of Android. Notice that this problem is specific for the dynamic analysis paradigm, whether static analysis techniques can analyze older malware without any particular issues since they study the binary file without executing it. Considering all of the above, static analysis techniques can report results about many thousands of malware analyzed, in contrast to us.

Since our analysis is active, the main principle that guided the selection process was the presence of triggering mechanisms inside the malware payload. For instance, the family *Finspy* concerns the logging and exfiltration of personal information of the user on an Android device, thus it is sensitive to calls, SMS activities, browser navigation history updates, etc. For our classification problem, some of the families included in this experiment can be seen as challenging to correctly classify since they employ specific mechanisms aimed to deceive the analysis. In particular, *Gorpo* and *Kemoge* exploit a combination of anti-analysis techniques such as the dynamic loading of the malicious code at runtime and the execution of noisy API calls that are not useful to implement the malware payload but serve as a method to mislead an analyzer that focuses on the sequence of actions performed by the malicious sample. Hence, resulting models will contain malicious behaviors interweaved by APIs that can induce an analyzer to ignore malicious characteristics. Moreover, *AndroRat* and *GoldDream* families distinguish themselves on the type of infection vector. Such classes are composed by small malware injected into complex harmless applications[1] such as games. This peculiar feature causes the models generated by SECUR-AMA to have only few branches depicting malicious behaviors (potentially negligible), thus making their identification hard. The rest of the families involved in the dataset can be considered less sophisticated because they do not employ advanced anti-detection mechanisms and do not hide themselves through injection. Figure 7.1 shows the composition of the dataset in terms of relative frequency for each family. It can be noticed the unbalanced nature of the composition with families with more than 200 samples and others with as few as 5. Such dataset characteristic reflects how malicious software appear in the real world in relation to their families: recently, the focus in malware design has shifted from the creation of new types of malicious payloads, i.e., the code slice of a malware aimed at causing harm, to the engineering of the stealthy system[2], while the payload is reused from older deployed malware as is or with minor modifications (Upchurch and Zhou, 2016). As a consequence of this trend, most of the malware in the wild fall in few families, inducing a disproportion among the cardinality of specimens contained in the different families (Walenstein and Arun Lakhotia, 2006; Walenstein et al., 2007).

7.2 Experimental methodology

In order to empirically assess the contribution of our proposal we perform a comparison of SECUR-AMA with relevant techniques proposed in literature (Suarez-Tangil, Tapiador, et al., 2014; A. Martìn et al., 2018). The main goal of such methods is the same: given a set of known malware K, where each element is labeled with its actual malicious family identifier, and a set of unknown (unlabeled) malware U, for each sample $u \in U$ infer for the malware family which u should belong to by using the knowledge provided by K. We selected state-of-the-art approaches that employ

[1]We refer to small and complex programs in qualitative terms, using as discriminant the cardinality of instructions composing such objects.

[2]The code of a malware that makes it to remain undetected during its execution.

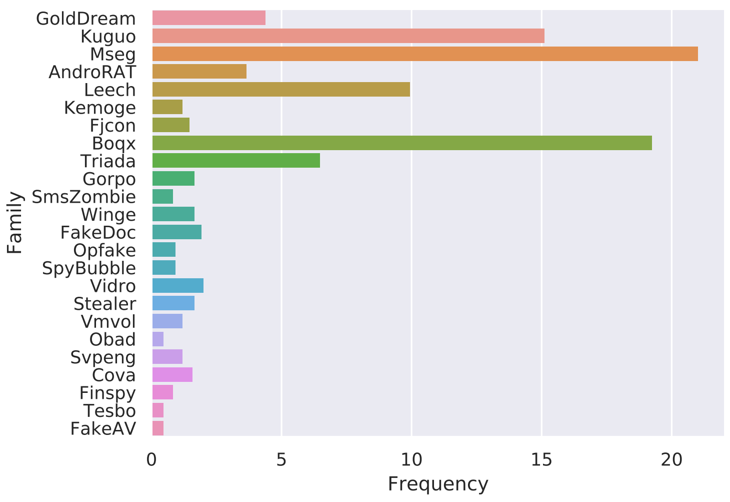


FIGURE 7.1: Dataset composition

different paradigms in the analysis process, differentiating from the AMA approach of SECUR-AMA. In (Suarez-Tangil, Tapiador, et al., 2014) authors propose a static method, called DENDROID (Section 3.1.1), to determine the distribution of particular structures embedded in the applications of the dataset and exploit such features to train a classifier to recognize the similarity degree between malware samples and family representatives. The work presented in (A. Martìn et al., 2018) instead belongs to the dynamic paradigm, in which authors describe a dynamic process to conduct the analysis of malware, called CANDYMAN, followed by the classifier training phase. A core aspect of this work is the wide range of supervised algorithms considered and evaluated, involving almost all classical learning methods and also deep learning techniques (Section 3.2). The analysis steps of CANDYMAN might closely recall the main concepts composing the methodology of SECUR-AMA. Indeed, the authors choose to model a Markov chain using the state space corresponding to the malicious APIs captured during the observations. Nonetheless, there are some important points that distinguish our approach. First of all, CANDYMAN randomly selects the triggering actions that aim at simulating users' behaviors. In contrast, in SECUR-AMA the analyzer agent implements a specific strategy, based on information gain, through multiple MCTS stages. This leads to different resulting models: for CANDYMAN, a single Markov chain is obtained for each sample and there is no labeling of the transition probabilities between Markov chain's states; SECUR-AMA instead retrieves a set of Markov chains (each one specifying the behaviors observed in response to an analyzer action, hence associated to a triggering action label). As a consequence, we can consider CANDYMAN as naively active as it does not use an informed strategy to interact with the malware agent. Moreover, CANDYMAN applies some approximations on the Markov chains to lower the number of states composing the Markov chains and, in the feature extraction part, to reduce the feature space used to subsequently train a classifier[3]. During the replication of the work of (A. Martìn et al., 2018) we focused on preserving the same analysis process described by the authors, but some details in the implementation have been changed due to programmatic reasons related to some parts of our framework. In particular, the state space of the Markov chains we

[3]To summarize, CANDYMAN employs the API labels retrieved through Droidbox as state space for the Markov chains, which represent an abstraction of real Android APIs. Moreover, once the feature vector is built, it erases from the feature model all transitions that occur less than ϵ times.

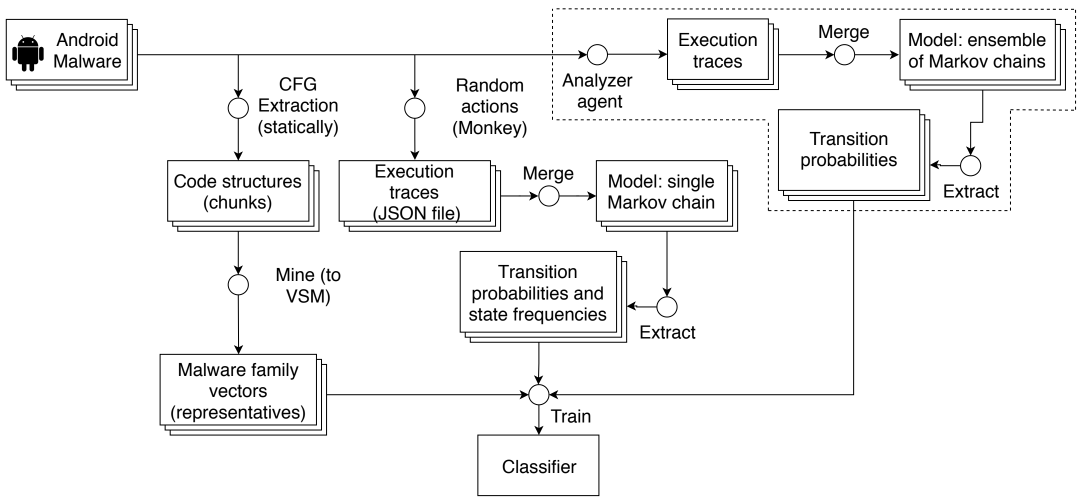

Android Malware
CFG Extraction (statically)
Random actions (Monkey)
Analyzer agent
Execution traces
Merge
Model: ensemble of Markov chains
Transition probabilities
Extract
Code structures (chunks)
Execution traces (JSON file)
Merge
Model: single Markov chain
Mine (to VSM)
Transition probabilities and state frequencies
Extract
Malware family vectors (representatives)
Train
Classifier

FIGURE 7.2: Overview of the experimental methodology for behavioral analysis. SECUR-AMA is contained within in the dashed area.

generate does not use event labels provided by the DroidBox tool[4], but rather adopt the actual Android API calls captured during the malware execution. Furthermore, some experimental setup choices made in (A. Martìn et al., 2018) have been changed due to the characteristics of our setting, e.g., families with less than 20 malware are not discarded as we want to be able to handle also unbalanced datasets. However, the mentioned changes do not affect the overall results obtained from the classifiers we trained, which are comparable to the ones obtained in (A. Martìn et al., 2018).

In Figure 7.2 we report a schematic overview of the experimental methodology. First, every malicious sample is subject to a mining phase to extract information about the malware, which is different for each method implemented, i.e., code chunks for DENDROID, log files of execution traces for both CANDYMAN and SECUR-AMA. Next, outcomes of CANDYMAN and SECUR-AMA are parsed in order to obtain a behavioral model of the sample (DENDROID skips this operation since the behavioral model is considered to be the set of chunks). Finally, features are extracted from each model and used to train a classifier to identify the malicious family an unknown sample belongs to. Both DENDROID (Suarez-Tangil, Tapiador, et al., 2014) and CANDYMAN (A. Martìn et al., 2018) are focused on the Android platform and they provide a classification approach to produce malicious family detectors for each examined sample. In our empirical evaluation we also employ supervised learning algorithms[5]. In particular, we decided to conduct experiments using the best performing classifier pointed out in each paper, i.e., K-NN with $k = 1$ and Random Forest for DENDROID and CANDYMAN respectively. We adopted Stratified K-Fold Cross Validation with $K = 5$ to provide training and testing sets with 5 different random splits. Quality of the results is assessed with unweighted[6] standard measures, i.e., precision, recall, and F_1-score. Implementations of classifiers, quality measures and cross-validation make use of Scikit-Learn (Pedregosa et al., 2011). For each technique tested, we report the performance values achieved for every malware family in the dataset and the overall average score.

[4]`https://github.com/pjlantz/droidbox`

[5]DENDROID efficiently supports only the 1-NN algorithm, hence we report only results for such classification technique in the corresponding table.

[6]The cardinality of each class does not affect the computation of the overall scores, hence every class has the same weight in the aggregated measures.

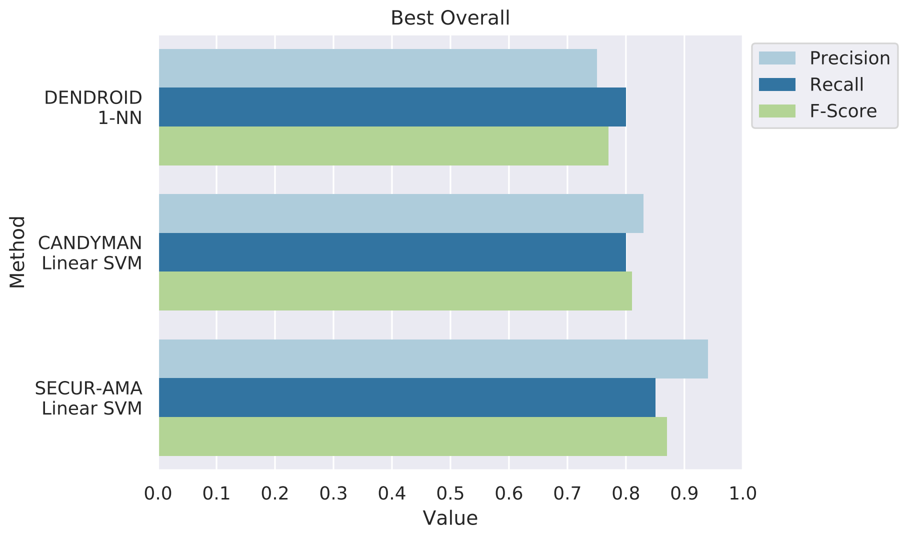


FIGURE 7.3: Malware classification comparison

7.3 Results and discussion

All three techniques are compared with 1-NN (chosen since DENDROID is built to work only with it). CANDYMAN and SECUR-AMA are also compared with Random Forest and Linear SVM where the first is shown to perform better with CANDYMAN in (A. Martìn et al., 2018), whereas the second performs better with SECUR-AMA. The overall best results for every approach are reported in Figure 7.3 where it is visible that SECUR-AMA with Linear SVM performs better than the others. However, every method has its strengths and weaknesses that are worth discussing more in detail.

Table 7.1 reports the results of the comparison between all three techniques where 1-NN is used as classifier, Table 7.2 reports the results of the comparison between CANDYMAN and SECUR-AMA with Random forest classifier, and finally, Table 7.3 reports again the results of the comparison between CANDYMAN and SECUR-AMA, but with a Linear SVM classifier. The two active techniques, i.e., CANDYMAN and SECUR-AMA perform better than the static one, i.e., DENDROID. As explained in Section 3.1.1, common problem of static methodologies comes from analyzing malware with encrypted malicious code deployed at runtime or obfuscated. This lowers the classification score of DENDROID for many of the families in the dataset. Dynamic analysis instead typically suffers of the opposite problem: it is difficult to observe executions of the program that cover the entire code, i.e., the code coverage problem. Such problem is noticeable in the *Opfake*, *Winge* and *Triada* families as they are designed to receive commands from an external server controlled by an attacker in order to be triggered and show their malicious behavior. DENDROID is able to analyze such behaviors by looking at the code without executing it, resulting in a better classification score for such families regardless of the classifier used for CANDYMAN and SECUR-AMA. However, the code coverage limitation is much less prominent in Android malware analysis since what is observed is usually relevant in the overall behavior, as the software are developed for the specific smartphone usage. There are obviously some exceptions, nonetheless, experiments confirm the viability of dynamic analysis in our case to focus on what is important to learn for the classification task. Other families for which the static analysis of DENDROID performs better than the other two techniques are *AndroRAT* and *GoldDream*, characterized by small

	DENDROID			CANDYMAN			SECUR-AMA		
Family	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
AndroRAT	**0.96**	**0.96**	**0.96**	0.79	0.75	0.77	0.78	0.78	0.78
Boqx	**0.92**	**0.95**	**0.93**	0.77	0.88	0.82	0.85	0.85	0.85
Cova	**0.78**	1.00	**0.88**	0.53	1.00	0.69	0.71	1.00	0.83
FakeAV	1.00	**0.89**	**0.94**	1.00	0.80	0.89	1.00	0.80	0.89
FakeDoc	0.90	1.00	0.94	1.00	0.95	0.98	1.00	1.00	**1.00**
Finspy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Fjcon	**0.83**	**1.00**	**0.91**	0.71	0.75	0.73	0.59	0.81	0.68
GoldDream	**1.00**	**1.00**	**1.00**	0.54	0.67	0.60	0.63	0.79	0.70
Gorpo	0.46	0.67	0.55	**0.92**	0.67	0.77	0.87	**0.72**	**0.79**
Kemoge	0.56	**0.67**	0.61	1.00	0.62	**0.76**	1.00	0.43	0.60
Kuguo	0.00	0.00	0.00	**0.81**	0.73	0.77	0.78	**0.77**	**0.78**
Leech	0.00	0.00	0.00	**0.97**	0.97	0.97	0.95	**1.00**	0.97
Mseg	0.87	**1.00**	**0.93**	0.86	0.80	0.83	0.87	0.87	0.87
Obad	0.00	0.00	0.00	1.00	**1.00**	**1.00**	1.00	0.80	0.89
Opfake	1.00	**1.00**	**1.00**	0.67	0.40	0.50	1.00	0.30	0.46
SmsZombie	0.75	0.86	0.80	1.00	1.00	1.00	1.00	1.00	1.00
SpyBubble	**0.94**	0.70	0.80	0.29	0.50	0.37	0.67	**1.00**	0.80
Stealer	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Svpeng	1.00	0.80	0.89	1.00	1.00	1.00	1.00	1.00	1.00
Tesbo	**0.31**	**0.57**	**0.40**	0.00	0.00	0.00	0.00	0.00	0.00
Triada	**0.95**	**0.95**	**0.95**	0.75	0.73	0.74	0.79	0.76	0.78
Vidro	1.00	1.00	1.00	0.96	1.00	0.98	1.00	1.00	1.00
Vmvol	0.90	**1.00**	0.94	**1.00**	0.85	0.92	1.00	1.00	**1.00**
Winge	1.00	**1.00**	**1.00**	1.00	0.61	0.76	0.93	0.78	0.85
	0.75	0.80	0.77	0.82	0.78	0.79	**0.85**	**0.81**	**0.82**

TABLE 7.1: Malware classification comparison - 1-Nearest Neighbor

malicious code injected into bigger benign applications. Such configuration prevents dynamic techniques to obtain perfect scores, because of the mentioned problem of "noisy" execution traces: the part of the model of interest (containing the malicious behavior) is too small compared to its entirety, making the classifier fail.

The strength of classical dynamic analysis in handling encrypted or obfuscated code with respect to static approaches contributes to better scores when considering the whole dataset, as visible in Tables 7.2 and 7.3. The main difference between CANDYMAN and SECUR-AMA is that the first performs sequences of thousands of random events focused on the user interface of the specific application under analysis, whereas SECUR-AMA strategically selects generic user-like actions (without considering the user interface) in the order of tens so as to trigger the malware. SECUR-AMA results in more satisfactory performances, justifying a triggering mechanism driven by rational policy to improve in efficiency over a random action selection. The capability of targeting the graphical parts of an application, although randomly as CANDYMAN does, have the clear advantage of being able to analyze malware that are explicitly triggered by actions on the user interface, such as for many games injected with malicious code. For SECUR-AMA it is not straightforward to implement the same kind of interaction since such an action is heavily context-dependent, hence there is the risk to gather no information if it is performed without checking whether the system

	CANDYMAN			SECUR-AMA		
Family	Precision	Recall	F_1-score	Precision	Recall	F_1-score
AndroRAT	1.00	**0.78**	**0.87**	1.00	0.76	0.86
Boqx	0.85	0.95	0.90	**0.93**	0.95	**0.94**
Cova	**1.00**	**1.00**	**1.00**	0.89	0.94	0.91
FakeAV	1.00	0.80	0.89	1.00	0.80	0.89
FakeDoc	1.00	0.95	0.98	1.00	**1.00**	**1.00**
Finspy	1.00	1.00	1.00	1.00	1.00	1.00
Fjcon	**1.00**	0.75	**0.86**	0.87	**0.81**	0.84
GoldDream	0.71	0.81	0.76	**0.82**	**0.98**	**0.90**
Gorpo	1.00	0.50	0.67	1.00	**0.56**	**0.71**
Kemoge	1.00	**0.23**	**0.38**	1.00	0.21	0.35
Kuguo	**0.82**	0.93	0.87	0.79	**0.96**	0.87
Leech	1.00	0.97	0.99	1.00	**1.00**	**1.00**
Mseg	0.91	0.93	0.92	**0.96**	**0.94**	**0.95**
Obad	1.00	**1.00**	**1.00**	1.00	0.80	0.89
Opfake	1.00	0.20	0.33	1.00	**0.30**	**0.46**
SmsZombie	1.00	1.00	1.00	1.00	1.00	1.00
SpyBubble	0.67	0.80	0.73	**0.83**	**1.00**	**0.91**
Stealer	1.00	1.00	1.00	1.00	1.00	1.00
Svpeng	1.00	1.00	1.00	1.00	1.00	1.00
Tesbo	0.00	0.00	0.00	0.00	0.00	0.00
Triada	0.79	0.75	0.77	**0.88**	**0.82**	**0.85**
Vidro	1.00	1.00	1.00	1.00	1.00	1.00
Vmvol	1.00	1.00	1.00	1.00	1.00	1.00
Winge	**1.00**	0.67	0.80	0.94	**0.83**	**0.88**
	0.91	0.79	0.82	0.91	**0.82**	**0.84**

TABLE 7.2: Malware classification comparison - Random Forest

is in the right context. A solution to address this problem could be the definition of a generic "use GUI" analyzer action, accountable for any interactions involving a screen event. However, such trigger would be too broad as there are many distinct possible interactions that are substantially different from application to application, e.g., pressing a button within a program may have totally uncorrelated meanings with respect to another program. For instance, the consequences of pushing the fee button in a ransomware form are completely different and unrelated to the effects of pressing a button in the main menu of a game application.

Figure 7.4 shows the learning rate for SECUR-AMA with Linear SVM classifier, i.e., how the performance changes with the number of actions performed, in terms of classification measures. The best is reached at around 20 actions performed by the analyzer and results stabilize from that point onward. Figure 7.5 depicts the normalized confusion matrix of SECUR-AMA with Linear SVM classifier for a more detailed visualization of results presented in Table 7.3.

We may conclude the discussion by saying that our methodology accomplishes the objective to propose a valid analysis to deal with triggered malware. Nevertheless, the performances achieved by the comparison techniques confirm their effectiveness as well: in spite of their overall lower learning metrics, they require a lower amount

	CANDYMAN			SECUR-AMA		
Family	Precision	Recall	F_1-score	Precision	Recall	F_1-score
AndroRAT	**0.94**	0.76	**0.85**	0.89	**0.80**	0.84
Boqx	0.88	0.91	0.90	**0.96**	**0.96**	**0.96**
Cova	0.83	1.00	0.92	**0.89**	1.00	**0.94**
FakeAV	1.00	0.80	0.89	1.00	0.80	0.89
FakeDoc	1.00	0.95	0.98	1.00	**1.00**	**1.00**
Finspy	1.00	1.00	1.00	1.00	1.00	1.00
Fjcon	0.43	0.72	0.55	**0.93**	**0.81**	**0.87**
GoldDream	0.64	0.73	0.68	**0.90**	**0.94**	**0.92**
Gorpo	0.79	0.83	0.81	0.79	0.83	0.81
Kemoge	**1.00**	**0.62**	**0.76**	0.80	0.31	0.44
Kuguo	0.83	0.84	0.84	**0.90**	**0.95**	**0.93**
Leech	0.98	0.97	0.98	**0.99**	**1.00**	**1.00**
Mseg	0.96	0.89	0.93	**0.99**	**0.97**	**0.98**
Obad	1.00	1.00	1.00	1.00	1.00	1.00
Opfake	**1.00**	0.40	0.57	0.83	**0.50**	**0.63**
SmsZombie	1.00	1.00	1.00	1.00	1.00	1.00
SpyBubble	0.22	**1.00**	0.36	**0.83**	0.50	**0.63**
Stealer	1.00	1.00	1.00	1.00	1.00	1.00
Svpeng	1.00	0.91	0.96	1.00	**1.00**	**1.00**
Tesbo	0.00	0.00	0.00	**1.00**	**0.20**	**0.33**
Triada	**0.88**	0.63	0.74	0.73	**0.92**	**0.81**
Vidro	1.00	1.00	1.00	1.00	1.00	1.00
Vmvol	1.00	0.85	0.92	1.00	**1.00**	**1.00**
Winge	0.80	0.70	0.76	**0.94**	**0.83**	**0.88**
	0.83	0.80	0.81	**0.94**	**0.85**	**0.87**

TABLE 7.3: Malware classification comparison - Linear SVM

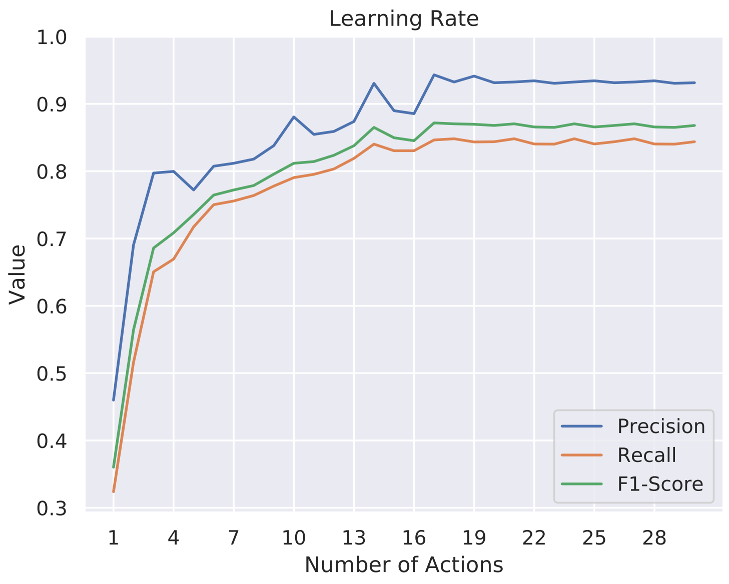


FIGURE 7.4: Learning rate: precision, recall and F_1-score evolution against actions performed

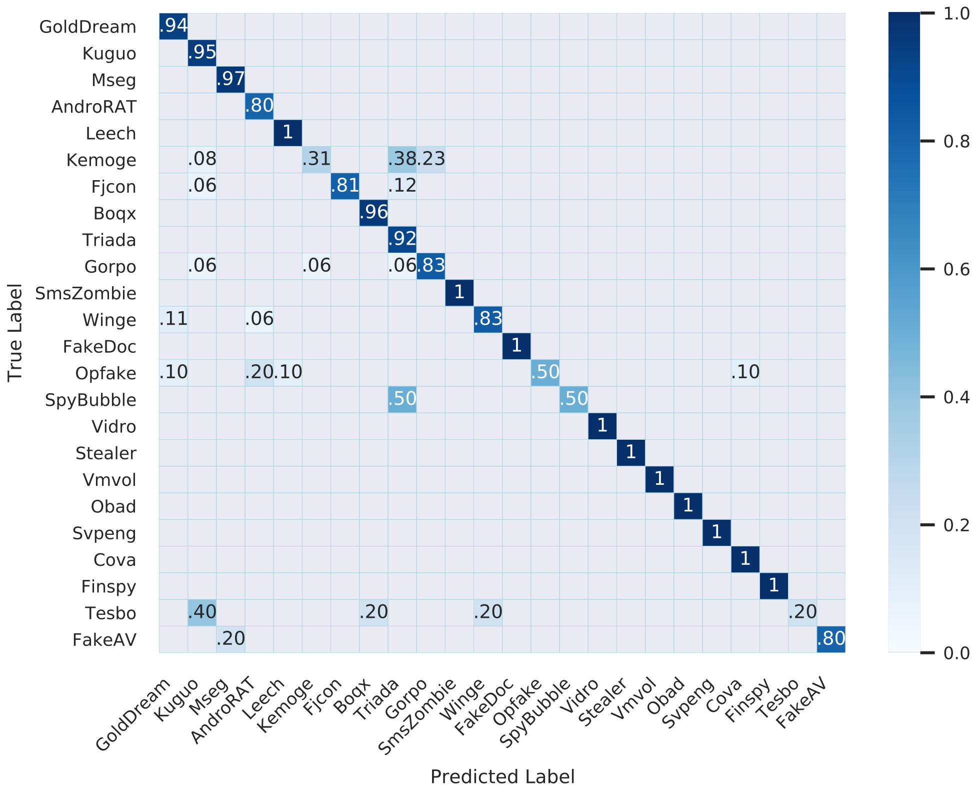


FIGURE 7.5: Normalized confusion matrix for SECUR-AMA with Linear SVM classifier. Values smaller than 0.05 are masked

of data to train a classifier with respect to SECUR-AMA[7] and, in case of malware that do not present any event listeners to show the payload, perform similarly or even better than our active approach. We believe that SECUR-AMA is a valid addition to complement existing techniques by providing satisfying results on a real malware dataset.

7.4 Runtime Performance

SECUR-AMA is built in order to make the analyzer perform its selected action and then observe the malware response for 10 seconds. Indeed, in the context of Android malicious applications it is often the case that if a target malware is reactive, it will probably respond as soon as triggered by the analyzer and will perform its goal in a reasonable amount of time. After each interaction the emulator is reset (using a snapshot) to a clean state in about 10 seconds. The optimal length of the analysis game has been found empirically to be around 20 actions for the analyzer in our experimental setting, as visible in Figure 7.4, resulting in about 6.5 minutes for the analysis of a single malware on a standard machine with 4 cores and 16GB of RAM. Hence, the time resources required by SECUR-AMA are comparable to those specified for the experimental settings of CANDYMAN in (A. Martìn et al., 2018).

[7]The process depicted in (A. Martìn et al., 2018) requires a classifier to be trained with a single transition matrix per malware family, whereas our analysis involves multiple Markov chain transition matrices.

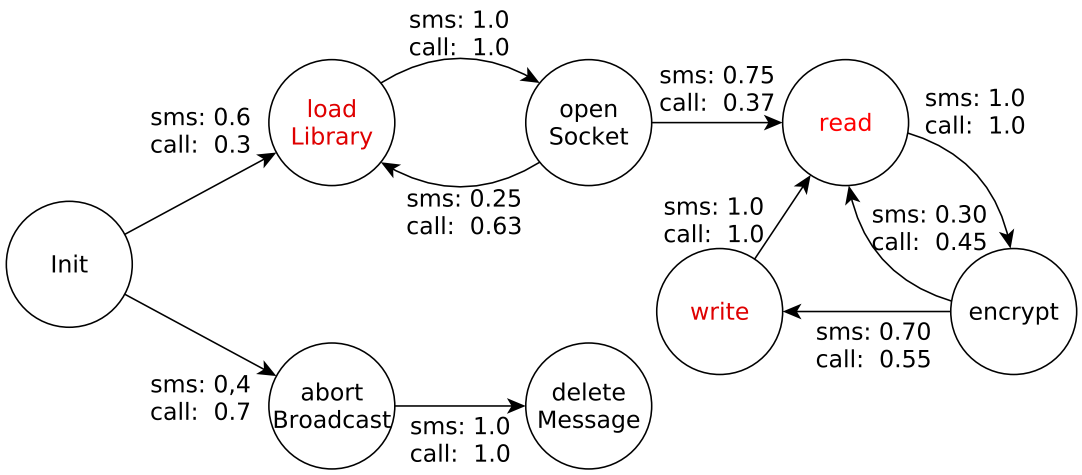


Figure 7.6: Example of malware injection

7.5 Conclusions

We propose SECUR-AMA, an alternative AMA technique based on MCTS to collect information about malicious software. SECUR-AMA models a malware as a set of Markov chains representing the behavior of the malware in response to triggering actions performed by the analyzer. Malware models are generated at runtime using the information collected during the analysis and such information is used to guide the remainder of the analysis in an iterative process. This formalism allows to represent the features obtained from the generated models in order to proceed to their classification with standard machine learning techniques, i.e., K-NN, Random Forest and Linear SVM. We conducted a throughout evaluation by comparing SECUR-AMA with other state-of-the-art techniques of different nature on a real malware dataset. Results show that our approach is a viable addition to existing techniques and is better performing in many cases.

Furthermore, the models generated by SECUR-AMA are agnostic with respect to the malware families (and so is the decision making process for the analyzer). This means that instead of using classification to study the models after the analysis process, non-supervised machine learning algorithms can be employed, e.g., clustering approaches. This allows to group the models with respect to similarity between each other regardless of the malware families. To clarify, malware models would be grouped together based on how similar they are to each other, and not to how similar they are to the examples of the families that have been shown during the training phase. This can result in groups of malware that do not represent any of the known families because they are different, effectively discovering new malware families. We went for a supervised (classification) approach because it allows us to reliably validate the results obtained against a ground truth.

Nevertheless, SECUR-AMA also suffers from some limitations mentioned in Section 7.3: malware often employ anti-detection techniques that inject noise in their execution trace, or the malicious code itself is injected into a bigger harmless application. Such factors make difficult for defense systems to detect threats due to the presence of "noise" inside the behavioral models generated. Consider for example the malware model of Figure 7.6: if the malicious code injected is composed only of the states in red color, i.e., *loadLibrary, read, write*, all the other states and transitions involving them should be discarded (or ignored) in order to correctly identify the malicious part. The same stands if all the states in black color (and the transitions involving them) are randomly produced at runtime by the malicious code as an anti-detection mechanism that introduces noise trying to hide the malicious behavior. In Chapter 8 we address this problem, introducing a long-term analysis of the behavioral models generated by SECUR-AMA in order to focus on the interesting parts of the models, ignoring the possible noise.

Part IV

Long-Term Behavioral Analysis

Chapter 8

Long-Term Analysis of Behavioral Models

In this chapter we aim at improving the identification of behavioral models generated by SECUR-AMA, for the particular cases of malicious code injection and Dynamic Obfuscation. The main hypothesis of our approach is that in the long-term, the intended behavior of the target agent[1] will emerge, allowing the analyzer to filter out possible misleading observations. In this perspective, we compute long-term transition probability values as features for classification instead of using the standard transition matrix as proposed for SECUR-AMA (Chapter 6). Specifically, we estimate the values of going from each state to every other, giving the process represented by the Markov chain enough time to reach a fixpoint, i.e., when the result would not change anymore from that point onward. Although our main application domain is to the malware behavioral models of SECUR-AMA, we design our methodology in order to make it applicable independently of the technique used to obtain the behavioral models of the target agents (as long as they can be interpreted as Markov chains). Hence, the focus of this chapter is not to improve the process of obtaining the behavioral models (although obviously some techniques can give better final results for particular domains), but rather the approach used to analyze them after they have been generated. Consequently, we can not make any assumptions on the various properties that the Markov chains may hold. Section 8.1 defines the problem and summarizes our solution approach; Section 8.2 details the methodology employed to compute and extract the long-term transition probability values. The approach presented in this chapter has been published in (Sartea and Farinelli, 2018; Sartea et al., 2019).

8.1 Problem definition

In scenarios in which intelligent agents interact within complex uncertain environments gathering information and adapting their behaviors accordingly, an important issue is to identify whether a known behavior appears within a behavioral model that has been learned through observations (Hiraishi and Kobayashi, 2014; Arifoglu and Bouchachia, 2017). As previously mentioned, a key challenge in this context is to deal with the presence of noise, e.g., while performing a difficult task an agent might make mistakes trying to follow her policy, consequently injecting noise into the behavioral model learned by observing the execution of that task. Moreover, noise injection could be intentional, e.g., a malicious agent might try to mask her real intentions and to deceive potential observers (Marpaung et al., 2012). The last example represents a

[1]We generically refer to the agent that is being analyzed by the analyzer as the “target agent”. In the case of malware analysis this is the malware

limitation of the SECUR-AMA framework, as explained in Section 7.5 with specific reference to Figure 7.6.

In this context, the methods for behavioral modeling mentioned in Section 3.3 either impose constraints on the models to be analyzed, e.g., (Whittaker and Thomason, 1994) requires the Markov chain of the behavior we want to capture to be a sub-graph of the a bigger Markov chain containing multiple behaviors, or perform transformations that are application specific, e.g., (Zhu et al., 2002; Mayil, 2012) require cycles to be removed and arbitrary states to be grouped together, hence neglecting significant information for what concerns behavior identification. Other works (Dyer et al., 2006; Busic et al., 2009) provide methods to compare Markov chains relying on the mixing time and by directly computing or estimating the stationary distribution. However, the behavioral models we deal with are generated by generic techniques, hence we have no guarantee for the existence of a meaningful stationary distribution, as it requires specific properties to hold. We also differ from (Hernandez-Leal and Kaisers, 2017) as we explicitly take into account adversarial agents that may intentionally perform some random or completely unrelated actions in order to mask their real policy, injecting noise in the behavioral model. Consequently, an observer agent can be deceived if she does not consider such potential deviation during the analysis process.

To address the problems presented above, we aim to compute the long-term transition probability features defined as follows

Definition 8.1 (Long-term transition probability). *The long-term transition probability for a Markov chain is the probability of going from any state s_i to any other state s_j given an infinite number of steps*

A crucial technical difficulty related to this idea is the computation of the long-term transition probability values as features for generic Markov chains. In fact, while there are methods to compute different long-term characteristics, e.g., the stationary distribution (Theorem 2.3) or the time to absorption (Theorem 2.5), their application is subject to the presence of specific properties, irreducibility and absorbency respectively (Definitions 2.26 and 2.27), that generic Markov chains may not have. For example, in our experiments we have large models that are generated through the observation of an agent's policy. For such models, the absorbency property, which is a fundamental requirement to compute long-term transition probability, never holds. To overcome this problem we propose a transformation of the Markov chain that enforces the absorbency property and allows to derive the long-term transition probability for the original Markov chain[2], i.e., preserving the long-term semantics of the embedded behavior.

8.2 Long-term behavior extraction

Similarly to SECUR-AMA, we are given a set of known behavioral models K partitioned into a set of classes C, and a set of unknown (unlabeled) behavioral models B of target agents. All models are represented with Markov chains and have been generated by observing the changes in the environment as a consequence of the interaction between an analyzer agent and a target agent. Our goal is then to assign the unknown elements of B to the known classes of C. The solution we propose is to employ supervised learning techniques to train a classifier for B given K. In contrast to SECUR-AMA, the probability values of transitioning between every state that are

[2]We focus on the long-term transition probability and not on the stationary distribution for example, because the former is more informative for our aim (example in Section 8.2).

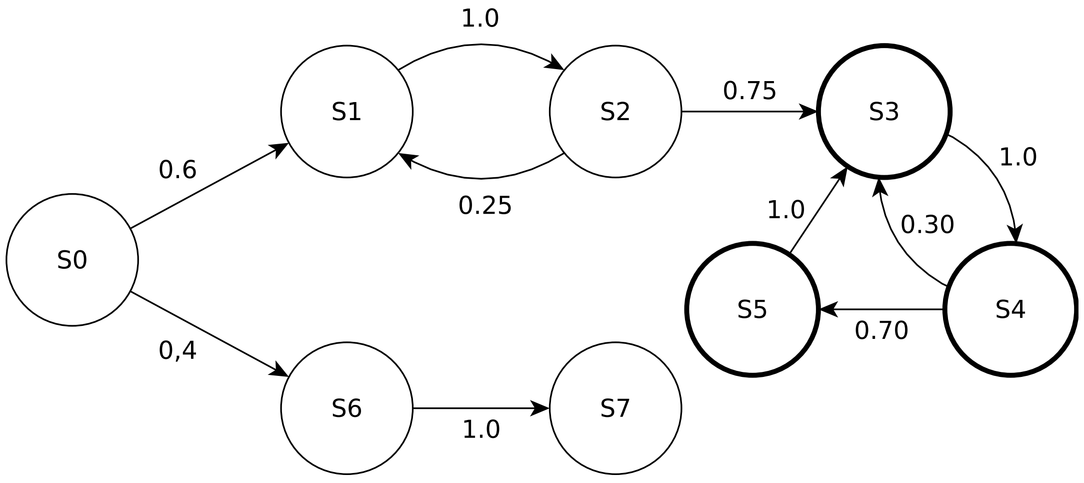


FIGURE 8.1: Markov chain with states in bold (S3, S4, S5) forming a terminal SCC

specified by the transition matrix of the Markov chains are not used as features to train the classifier since (as detailed in Chapter 7) such features may be unreliable as they only represent short-term transition probability, hence neglecting important information about the long-term behavior of the agent. This new approach instead, aims at extracting such long-term behavior from every model by using the long-term transition probability. For example, in a learning by demonstration setting for a complex task, a teacher agent might make mistakes while trying to follow its default policy, injecting noise in its execution trace. A deliberately harmful scenario instead is when a malware designer intentionally inserts fake API calls to deceive malware detection tools. We use the long-term transition probability instead of the stationary distribution, as the latter is not guaranteed to be meaningful in the behavioral models we are given to analyze. Looking at the Markov chain in Figure 8.1, the stationary distribution (computed using Theorem 2.3) is $\pi = [0, 0, 0, 0.37, 0.37, 0.26, 0, 0]$, where π_i corresponds to the probability of being state Si. Notice that π is non-zero only for the set of states forming an irreducible (Definition 2.26) Markov chain ($S3, S4, S5$). Hence, we lose the information that all the states where $\pi_i = 0$ can be reached, even though never visited again afterward. The long-term transition probability instead would tell us that, considering state S_0 for example, the probability values of reaching the other states are $L_{0i} = [0, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.4]$. Such information for every couple of states is fundamental to our approach.

We exploit some well known properties to compute the long-term behavior of the agents by using the transient states probability (Theorem 2.6). However, this approach can be used only for absorbing Markov chains, but since our behavioral models come from generic extraction techniques, there are no guarantees that the corresponding Markov chains are absorbing (in our case studies we are never given Markov chains already absorbing). The methodology is independent from how the Markov chain was generated as no assumption is required about the meaning of the states, or the structure of the Markov chain. In fact, we use representations with different meanings for the three experimental settings (see Chapter 9). To overcome this problem, we define a procedure to transform any Markov chain into an absorbing one. The goal is to design a transformation procedure that takes as input any Markov chain, i.e., without imposing any constraint on it, and derives the long-term transition probability.

8.2.1 Absorbing transformation

Algorithm 8.1 (AbsEnforcer) details our transformation procedure. Given a Markov chain M of n states, a corresponding absorbing Markov chain M' with $g \leq n$ transient states and an absorbing state s_a is created. Only the block matrix Q of the canonical form (Definition 2.28) is returned since it is the only part used in the subsequent computations. R $(g \times 1)$, I (1×1) and $\emptyset$ $(1 \times g)$ can be easily derived knowing that exactly one absorbing state exists in M' and that every row of M' must sum to 1 (if a state had an outgoing probability value of 0 it would not reach an absorbing state, hence M' would not be absorbing). The first step is to compute the Strongly Connected Components (SCCs) of the Markov chain (Tarjan, 1971). A SCC is a set of states where every state can be reached by any other in a finite number of steps and we distinguish between terminal and non-terminal SCC. In Figure 8.1, states $(S1, S2)$ form a non-terminal SCC, whereas states $(S3, S4, S5)$ form a terminal SCC.

Algorithm 8.1 AbsEnforcer

Require:
M - transition matrix of a Markov chain
Ensure:
Q - block matrix of new absorbing Markov chain M'

```
1: $sccs \leftarrow$ Tarjan($M$)                                ▷ Find SCCs
2: for all $T \in sccs$, with $T$ terminal do
3:     $s_m \leftarrow s \in_R T$                               ▷ Randomly select a merge state
4:     for all $s_i \in M$, with $s_i \notin T$ do
5:         $p \leftarrow 0$
6:         for all $s_j \in T$ do                               ▷ Remove edges and record weights
7:             $p \leftarrow p + M_{ij}$
8:             $M_{ij} \leftarrow 0$
9:         $M_{im} \leftarrow p$                                ▷ Redirect edges to $s_m \in T$
10:    Merge $T$ into the single state $s_m$
11:    $M_{mm} \leftarrow 0$                                    ▷ Connect $s_m$ to $s_a$ with $P = 1$
12: $Q \leftarrow M$
13: return $Q$
```

Definition 8.2. *We say a SCC A is terminal if there does not exist a path from a state $s_i \in A$ to a state $s_j \notin A$, otherwise A is defined as non-terminal.*

Given a Markov chain M, for each terminal SCC T, AbsEnforcer merges all the states $s_i \in T$ into a single state $s_m \in T$, connecting it to an absorbing state s_a with a new edge. Since we work with the canonical form, and we impose $M_{mm} = 0$ (line 11), the state s_m is consequently connected to s_a with probability value 1, i.e., in the R block matrix not explicitly represented. The update of M (lines 4-9) redirects edges entering any state $s_i \in T$ to the designated merged state $s_m \in T$ before making it the only state of T (line 10). Figure 8.2 shows an application example of AbsEnforcer to the Markov chain in Figure 8.1. The resulting transition matrix M' in canonical form is visible in Equation 8.1. Indices for states $S6$ and $S7$ are 4 and 5 respectively in M' as a consequence of merging the terminal SCC. Block matrices Q, R, $\emptyset$ and I are the top-left, top-right, bottom-left, and bottom-right blocks of M' respectively, as

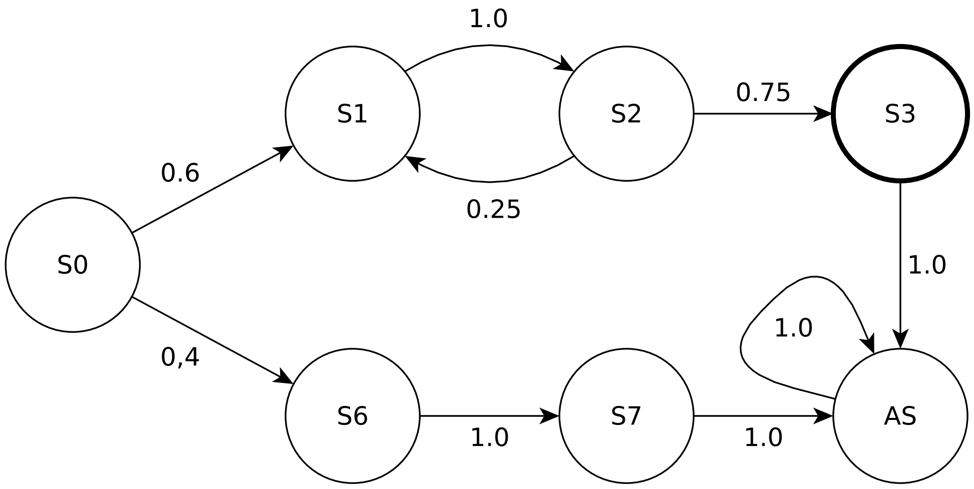


FIGURE 8.2: Absorbing transformation applied to the Markov chain of Figure 8.1. State S3 has been selected as s_m for the terminal SCC (S3, S4, S5)

of Definition 2.28.

$$M' = \left[\begin{array}{cccccc|c} 0 & 0.6 & 0 & 0 & 0.4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0.75 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{8.1}$$

In the following we prove that the output of AbsEnforcer is an absorbing Markov chain M' w.r.t. a generic Markov chain M. This is fundamental as it allows to apply Theorems 2.4 and 2.6 and then to derive the long-term transition probability for M.

Theorem 8.1. *The application of* AbsEnforcer *to a Markov chain M always results in an absorbing Markov chain M'*

Proof. Every Markov chain M contains at least one terminal SCC. Notice that a single state is a SCC since there is always a 0-length path from it to itself. AbsEnforcer creates a new Markov chain M' by merging each terminal SCC $T \in M$ into a chosen state $s_m \in T$, and redirecting all incoming edges of the removed states toward s_m. The definition of probability is maintained accumulating the weights of all the edges redirected for each source state and assigning the same sum of weights to the new edge toward s_m (lines 4-9). Additionally, s_m is also connected to the absorbing state s_a with $P = 1$ as a consequence of removing any outgoing edge from it (line 11). Every state $s_i \in M'$ is either contained in a terminal SCC T for M, or in a non-terminal SCC U for M. In the first case s_i is a merged state $s_m \in M'$ for T. Consequently there exists a direct edge in M' such that $s_i = s_m \to s_a$. In the second case there exists a path in M' such that $s_i \rightsquigarrow s_m$, where s_m is the result of merging a terminal SCC of M. This is true because since the number of states is limited, following an outgoing path from U, a terminal SCC will eventually be reached. Then $s_i \rightsquigarrow s_m \to s_a$ in M'. Therefore, every state of M' is either the absorbing state s_a or a transient state that will eventually reach the absorbing state s_a. Hence, from Definition 2.27, M' is an absorbing Markov chain. □

The transformation described above, even though removing states forming terminal SCCs, allows to derive the long-term transition probability for the original Markov chain, including all the removed states. To show this we make use of Lemma 8.1. Also notice that every state of M is a transient state in M' (possibly merged in a s_m).

Lemma 8.1. *Within a terminal SCC T, the long-term transition probability values of going from any state $s_i \in T$ to any state $s_j \in T$ converge to 1 as the number of steps $n \to \infty$.*

Proof. By Definition 2.26, a terminal SCC T is an irreducible Markov chain, meaning that it is possible to go from each state to every other with non-zero probability. Moreover, being terminal, there exists no outgoing path from T. Consequently, starting from any state $s_i \in T$, the probability value of reaching any other state $s_j \in T$ increases, approaching 1, as the number of steps n increases. □

As from Theorem 8.1, AbsEnforcer produces an absorbing Markov chain (specifically, its Q block matrix). Therefore, Theorems 2.4 and 2.6 can be applied to its output in order to compute the long-term transition probability as of Definition 8.1.

Corollary 8.1 (Long-term transition probability). *Given a Markov chain M, the long-term transition probability value L_{ij} of going from state $s_i \in M$ to state $s_j \in M$ can be computed from the transient states probability H (Theorems 2.4 and 2.6) with $Q =$ AbsEnforcer(M) as follows*

$$L_{ij}(H) = \begin{cases} 1 & \textit{if } s_i \textit{ and } s_j \textit{ are in the same terminal SCC in } M \\ 0 & \textit{if } s_i \textit{ is in a terminal SCC } T \textit{ in } M \textit{ and } s_j \notin T \\ H_{im} & \textit{if } s_i \textit{ is not in a terminal SCC in } M \textit{ and } s_j \textit{ was} \\ & \textit{merged into a state } s_m \textit{ in } M' \\ H_{ij} & \textit{otherwise} \end{cases}$$

The first case is a direct application of Lemma 8.1, whereas the third one is a consequence: in the long-term, the probability value of reaching a state of a terminal SCC T is the same of reaching any other state of T, as in the long-term they can reach each other with probability value 1. The second case is trivial: states within a terminal SCC can only reach other states of the same SCC. The fourth case is where no adjustment has to be made and the standard transient states probability can be used.

The application of Definition 8.1 to a model transformed with AbsEnforcer allows to recover the long-term behavior associated to the original Markov chain. Therefore the original long-term semantics is preserved, or rather it can be fully reconstructed.

8.2.2 Feature extraction

The feature extraction process is tailored on our supervised learning approach. As we aim at recognizing known behaviors, we require a "blueprint" D as input, along with the actual model x from which to extract the feature vector. The blueprint is used to retrain from x, only the long-term transition probability values between the states we are interested in. Hence, the only information that D needs to contain are states and corresponding edges between states. The probability values on the edges (for D) are not required since they are not used in the feature extraction process. Essentially, the blueprint D is a "shape" on which to project the long-term transition probability extracted from a model x to analyze. Figure 8.3 shows an example of blueprint.

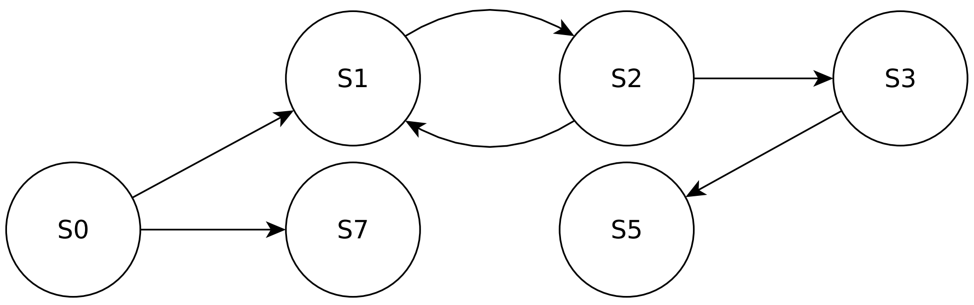


FIGURE 8.3: Example of blueprint D

Algorithm 8.2 (EXTRACTOR) details the feature extraction procedure for an unknown model x, given a blueprint D. The first step is to apply ABSENFORCER to transform M into an absorbing Markov chain M', obtaining its block matrix Q (line 1). Now we can apply Theorems 2.4 and 2.6 to Q as second step, retrieving the transient states probability values H of going from each state to every other for M' (lines 2-3). The last step is to extract the long-term transition probability (Definition 8.1) from H, for the states of x that also appear in the blueprint D (lines 5-7). In our experiments we label the states of the models we generate in a consistent manner, therefore, to check if a state of D exists also in x we perform a simple label comparison (line 6).

Algorithm 8.2 EXTRACTOR

```
Require:
    $D$ - $G(V, E)$ blueprint model with $k = |V|$
    $M$ - Markov chain of model $x$
Ensure:
    $F$ - feature vector

1: $Q \leftarrow$ ABSENFORCER($M$)                          ▷ Algorithm 8.1
2: $N \leftarrow (I - Q)^{-1}$                              ▷ Theorem 2.4
3: $H \leftarrow (N - I)N_{dg}^{-1}$                        ▷ Theorem 2.6
4: $D' \leftarrow k \times k$ empty matrix
5: for all edges $(s_i, s_j) \in E$ do
6:     if states $s_i, s_j$ exist also in $M$ as $s_v, s_u$ then
7:         $D'_{ij} \leftarrow L_{vu}(H)$                   ▷ Definition 8.1
8: return FLATTEN($D'$)
```

The application of the complete feature extraction procedure to the model x of Figure 8.1 with the blueprint D of Figure 8.3 is reported below. The first step is to enforce the absorbency property, obtaining a result visible in Figure 8.2 and corresponding to Equation 8.1. Then, by applying Theorems 2.4 and 2.6 to the block matrix Q (top-left block of Equation 8.1) we obtain the transient states probability matrix H of Equation 8.2.

$$H = \begin{bmatrix} 0 & 0.6 & 0.6 & 0.6 & 0.4 & 0.4 \\ 0 & 0.25 & 1 & 1 & 0 & 0 \\ 0 & 0.25 & 0.25 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.2}$$

Finally, the projection of H over the blueprint D making use of Definition 8.1 results in matrix L of Equation 8.3. Recall that state $S5$ was merged into $S3$ as effect of the absorbing transformation, hence $L_{i5} = L_{i3}$ for every state s_i, whether states $S4$ and $S6$ are not contained in D (indices 4 and 5 then correspond to states $S5$ and $S7$ respectively in L). Matrix L will then be flattened into the final feature vector.

$$L = \begin{bmatrix} 0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.4 \\ 0 & 0.25 & 1 & 1 & 1 & 0 \\ 0 & 0.25 & 0.25 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.3}$$

The exploitation of standard techniques for absorbing Markov chains enable to efficiently extract the long-term transition probability instead of, for example, performing multiple Depth-First Searchs (DFSs) till a convergence point. We can now solve our classification problem by training a classifier using the features extracted by Extractor. We first create blueprints for every class in C. These can be manually crafted by a domain expert or, as we did in the experiments, can simply be created from the known models K by choosing representatives for the classes and retaining their graphs (states and edges, no probability values). It is also possible to select more than one representative per class and to perform a merge to obtain a single blueprint for such class. Then all the blueprints for the classes in C are merged together to obtain a single blueprint D. Successively, we train a classifier extracting the training features from each $d \in K$ by calling Extractor(D, M_d) and using the knowledge of which class $c \in C$, d belongs to. We then classify the unknown behavioral models $x \in B$ by extracting their features, i.e., by calling Extractor(D, M_x), and then querying the trained classifier. M_d and M_x are the transition matrices of d and x respectively.

Chapter 9

Empirical Evaluation of the Long-Term Behavioral Analysis

This chapter presents the empirical evaluation of the long-term behavioral analysis described in Chapter 8. Section 9.1 reports the results obtained with application of our approach to classical games; Section 9.3 reports instead the results for real Android malware; Section 9.2 provides a pathological example to show the limitations of this approach; Section 9.4 concludes the chapter with final consideration and future directions.

We divide the empirical analysis in two types of experiments: in the first one we focus on agents interacting within classical games, i.e., the iterated Prisoner's Dilemma (Nowak and Sigmund, 1993), Rock Paper Scissors (RPS) and a repeated lottery game, while in the second one we analyze real Android malware trying to identify malicious behaviors. The first experimental setting is interesting as it shows that our approach can be used in a generic domain where multiple agents interact and observe each other. Moreover, this gives us the opportunity to study our proposed technique in a well known domain. The second experimental setting instead is crucial in real world IT defense systems and aims at solving the problem regarding the presence of noise inside the models generated by SECUR-AMA (Chapter 6). Figure 9.1 shows an overview of the empirical evaluation we conducted, where an analyzer agent performs the MCA action selection strategy of SECUR-AMA described in Section 6.1.1 to generate the behavioral models of different agents, i.e., players of classical games, malicious software agents. Again, the specific technique employed to generate the behavioral models is independent from our methodology used to analyze them. From such models we apply our approach to extract the long-term transition probability values as informative features, and compare the learning quality with respect to other different features proposed in literature, i.e., 1-step transition probabilities (the classical transition matrix) used also in SECUR-AMA and n-grams (Nowak and Sigmund, 1993; Friedman, 1971).

Our analyzer agent selects the actions to perform in order to gain as much information as possible on the adversary, trying to minimize the entropy of the behavioral model being generated. A behavioral model is represented with a set of Markov chains extracted from the observation of an agent's actions (see Section 4.2). For all the experiments we trained a Linear SVM performing a K-fold cross validation with $K = 5$. Classification quality is evaluated using precision, recall, and F_1-score. Since each behavioral model encodes multiple Markov chains, EXTRACTOR is applied to each of them individually in order to extract a feature vector for each Markov chain. Then, we perform a concatenation of the feature of each Markov chain in a single vector for classification. A complete example of the extraction process is reported in Section 8.2.

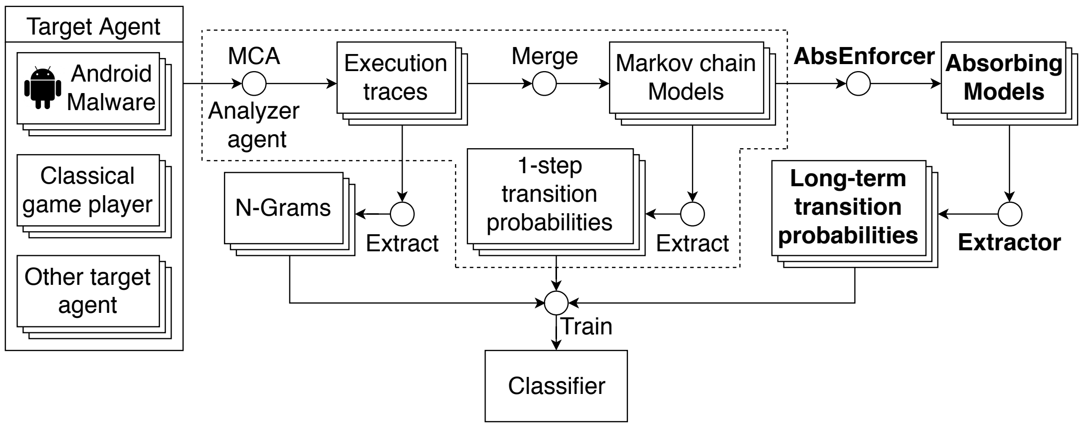


FIGURE 9.1: Overview of our methodology for behavioral analysis. The dashed area contains the method of SECUR-AMA, whereas bold text represent the long-term analysis

9.1 Classical games

While there are various approaches in the context of games to find strategies that optimize agents' payoffs studying equilibria (Nash, 1950), here we focus on the identification of known strategies only by observing the interaction between players, therefore we are not interested in the computation of an equilibrium.

9.1.1 Iterated Prisoner's Dilemma

In the iterated (or repeated) Prisoner's Dilemma (Section 2.2.5), two players are given the choice to cooperate (C) or to defect (D) in a repeated interaction of the same stage game. Hence, we design 6 strategies previously used in literature (Nowak and Sigmund, 1993; Friedman, 1971) for player B (target agent), while player A (analyzer agent) chooses its actions following the MCA action selection strategy of SECUR-AMA. The six strategies are:

1. tit-for-tat: play the action played by the adversary in the last game
2. retaliation: cooperate until the adversary defects to the police for the first time and then defect forever
3. random: cooperate and defect are chosen with 0.5 and 0.5 probability values respectively
4. always cooperate
5. always defect
6. mixed: cooperate and defect are chosen with $\frac{1}{5}$ and $\frac{4}{5}$ probability values respectively

States are labeled with the joint actions that made the game reach such state, whereas edges are labeled with the action of player A that triggered such transition. Figure 9.2 shows an example of an observed random behavioral model for player B. State CD for example is the result of actions *(cooperate, defect)* by player A and B respectively. Every strategy has been played 20 times in an iterated Prisoner's Dilemma of length 100, obtaining 120 behavioral models for player B. The aim then is, given such models, to classify them over the 6 known strategies (classes). The blueprint (Section 8.2.2)

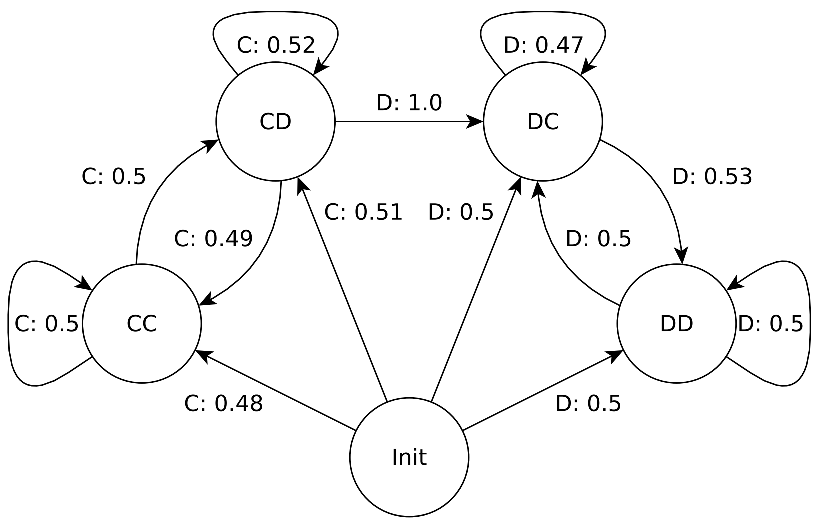


FIGURE 9.2: Example of an observed random behavioral model for player B in the iterated Prisoner's Dilemma

	Strategy	Precision	Recall	F_1-score
1	Tit-for-tat	1.00	1.00	1.00
2	Retaliation	1.00	1.00	1.00
3	Random	1.00	0.65	0.79
4	Always C	1.00	1.00	1.00
5	Always D	1.00	1.00	1.00
6	Mixed	0.74	1.00	0.85
	Total	0.94	0.94	0.94

TABLE 9.1: Player's strategy identification for the iterated Prisoner's Dilemma

has been created selecting random representatives from each strategy and merging all their graphs together. Table 9.1 reports the evaluation of the process. Results show that strategies 1, 2, 4 and 5 are perfectly identified by our method. However, since strategies 3 and 6 differ only in the probability value they assign to actions cooperate and defect, if the Markov chain states are highly connected, e.g., if they form a single SCC, the long-term behavior tends to flatten the differences between transition probability values, making harder to distinguish behaviors in the long-term compared to the short. This is a limitation of our approach that arises in the pathological case of models composed only by few terminal SCCs. However, this is unlikely to happen with more complex models and real world scenarios, as our next experiments will show.

9.1.2 Rock Paper Scissors

The second evaluation setting involves the famous game of RPS in its iterated version (Norris, 2017). Two players are given the choice to play rock (R), paper (P) or scissors (S), with a payoff matrix visible in Table 9.2. Also in this case we simulated an interaction between two players: player A (analyzer agent) chooses its actions following the MCA action selection strategy of SECUR-AMA, whereas player B follows one of four possible known strategies, specifically:

1. tit-for-tat: play the action played by the adversary in the last game

		Player 2		
		R	P	S
	R	$0, 0$	$-1, 1$	$1, -1$
Player 1	P	$1, -1$	$0, 0$	$-1, 1$
	S	$-1, 1$	$1, -1$	$0, 0$

TABLE 9.2: Normal form of the RPS game

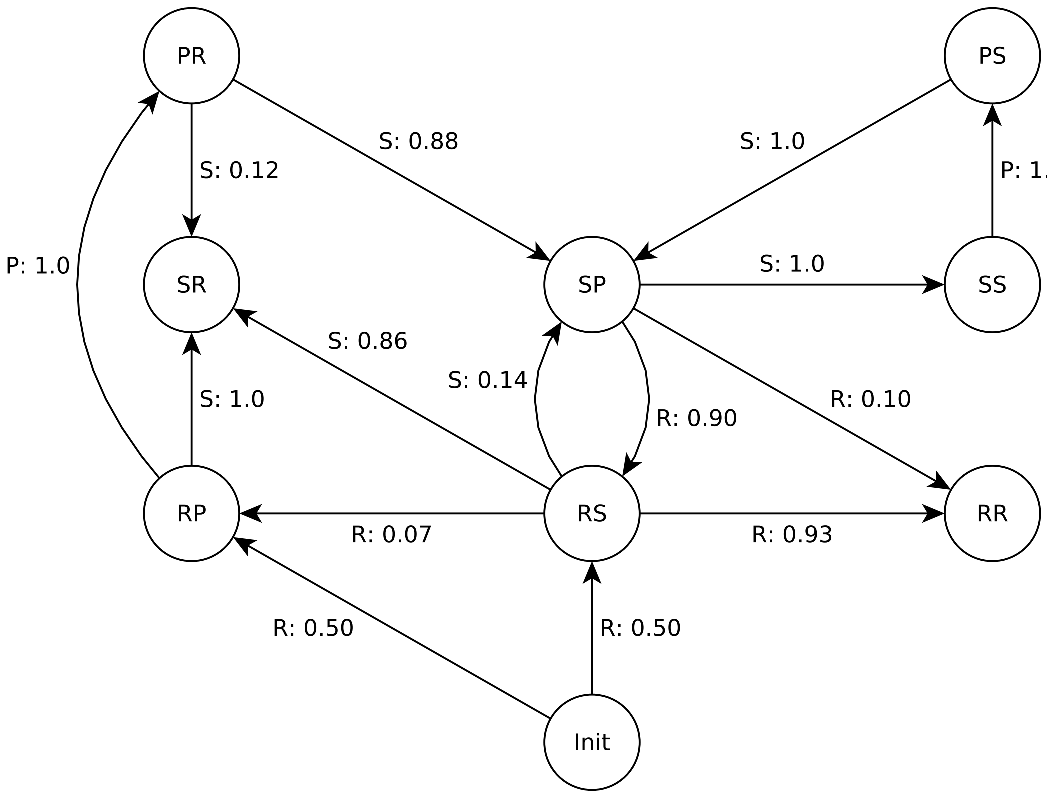


FIGURE 9.3: Example of an observed tit-for-tat behavioral model for player B in the iterated RPS

2. counter: play the action that neither players used in the last game
3. mirror: play the same action of the last game if it was a winning choice
4. random: R, P, S are played $\frac{1}{3}$, $\frac{1}{3}$ and $\frac{1}{3}$ probability values respectively

The Markov chain representation of the game is similar to the Prisoner's Dilemma, but the chains are 3 (because of the 3 possible actions of player A) and with more states. Again, every strategy has been played 20 times in an iterated RPS of length 100, obtaining 80 behavioral models for player B.

Table 9.3 reports the evaluation of the process. The aim then is again to classify the models over the 4 known strategies. The blueprint D has been created selecting random representatives from each strategy and merging all their graphs together. Also in this case results are satisfying, although the problem of model collapse is still present as visible from the lower precision in the strategies. In the next experiments (Sections 9.1.3 and 9.3) the models will be more complex and reflecting more realistic scenarios, therefore less impacted by the problem of model collapse.

	Strategy	Precision	Recall	F_1-score
1	Tit-for-tat	1.00	1.00	1.00
2	Counter	0.78	0.98	0.86
3	Mirror	0.79	0.99	0.87
4	Random	0.79	0.99	0.88
	Total	0.87	0.99	0.90

Table 9.3: Player's strategy identification for the iterated RPS

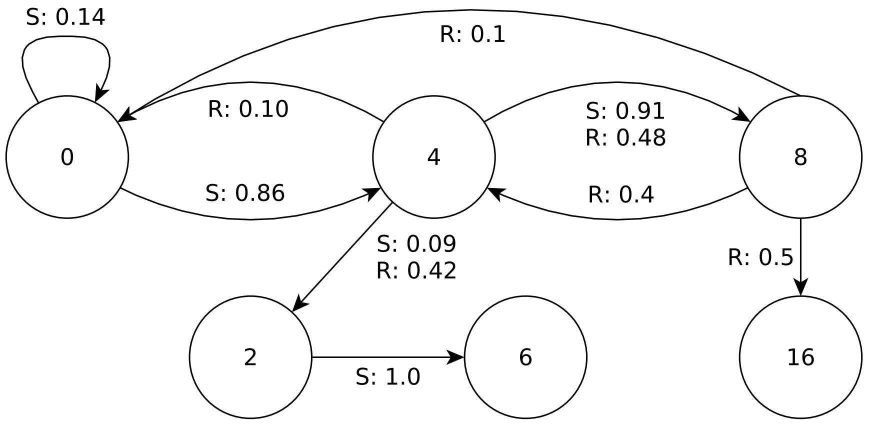


Figure 9.4: Example of an observed behavioral model in the repeated lottery

9.1.3 Repeated lottery game

As a third evaluation setting, we define a custom repeated lottery game (Mas-Colell et al., 1995) as follows: at every iteration, a player chooses between a safe (S) and a risky (R) lottery, accumulating the reward at each stage. S gives reward 4 with $P = 0.9$ and halves the current accumulated reward with $P = 0.1$. R gives a reward of 8 with $P = 0.5$, halves the current accumulated reward with $P = 0.4$, and sets it to 0 with $P = 0.1$. Figure 9.4 shows an example of an observed behavioral model for the repeated lottery game. States are labeled with the current accumulated reward, whether edges are labeled with the lottery chosen by the player. This experiment is useful also to show that the meaning of a state is not important with respect to the method that we use to extract the long-term features. Indeed, here we do not represent states with actions, but with the internal state of the agent that we have access to. Although, what is important is that a path in the model is characteristic of the strategy employed, therefore the model can be used to identify specific strategies. In this case, behavioral models are generated by the analyzer only by observing the player agent, i.e., no interaction is involved between the two. We design some similar strategies with respect to the previously used games:

1. always S
2. always R
3. R until loss (always R until the first loss happens, S from that stage onward)
4. S until loss (always S until the first loss happens, R from that stage onward)
5. random: S and R are chosen with 0.5 and 0.5 probability values respectively
6. mixed: S and R are chosen with $\frac{4}{5}$ and $\frac{1}{5}$ probability values respectively

	Strategy	Precision	Recall	F_1-score
1	Always S	0.86	0.90	0.88
2	Always R	0.95	1.00	0.98
3	R until Loss	0.89	0.85	0.87
4	S until Loss	1.00	0.95	0.97
5	Random	1.00	0.95	0.97
6	Mixed	0.95	1.00	0.97
7	R between $[10, 20]$	1.00	1.00	1.00
8	R between $[10, 50]$	1.00	1.00	1.00
9	R between $[20, 40]$	1.00	1.00	1.00
	Total	0.96	0.96	0.96

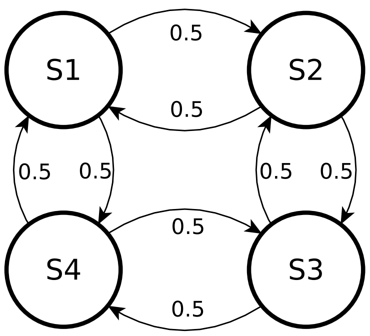
Table 9.4: Player's strategy identification for the repeated lottery game

7. R is played when the current accumulated reward is within $[10, 20]$, S otherwise
8. R is played when the current accumulated reward is within $[10, 50]$, S otherwise
9. R is played when the current accumulated reward is within $[20, 40]$, S otherwise

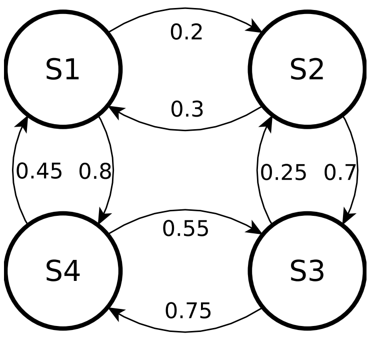
Every strategy has been played 20 times in a repeated lottery game of length 500, obtaining 180 behavioral models for the player agent. The aim, again, is to classify such models over the 9 known strategies (classes). This experiment is different from the previous one as the generated behavioral models are much bigger (up to 1913 states) and contain a various number of terminal and non-terminal SCCs, resulting in a comprehensive evaluation setting for our approach. Also in this case, the blueprint D has been created selecting random representatives from each strategy and merging all their graphs together. Table 9.4 reports the results of the process, where it is visible that the strategies are overall well classified. We notice that strategies 5 and 6 are identified more clearly compared to Table 9.1. As we mentioned before, in bigger and realistic models, the flattening problem of the long-term transition probability is much less prominent. Strategies 1 and 3, and strategies 2 and 4 instead, can be confused with each other depending on when the first loss happens during the game. Strategy 4 for example becomes exactly strategy 2 after the first loss. If this change happens at the very beginning of a game, the two strategies become indistinguishable. In contrast to the previous experiments, given the more complex behavioral models generated in this evaluation setting, the problem of model collapse does not appear.

9.2 Investigating a pathological case: model collapse

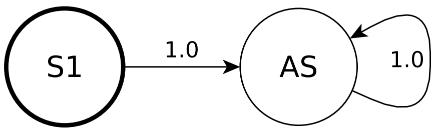
In the previous sections we mentioned the problem of model collapse, giving a brief intuition of what this is about. In this section we provide a detailed explanation of such problem to understand better on the implications. Figure 9.5 presents a visualization of the model collapse that happens when models are composed only by a few, or in this specific example by only one, terminal SCCs. In particular, Figure 9.5a shows a behavioral model where the transition function is uniformly random: from every state there are always two possible successors, each one equally likely. Figure 9.5b instead shows a behavioral model with the same state space, but the transition function is different in terms of probability distribution with respect to Figure 9.5a: from every state there are always two possible successors, not equally likely as before. Now, comparing the two transition functions as they are, the difference is clearly visible. However, if



(A) Behavioral model M_1 representing a random uniform strategy composed by a single terminal SCC



(B) Behavioral model M_2 representing a random mixed strategy composed by a single terminal SCC



(C) Result of the transformation algorithm applied to the behavioral models M_1 and M_2 of Figures 9.5a and 9.5b respectively

FIGURE 9.5: Example of model collapse in the case of a single SCC and different strategies

we apply ABSENFORCER (Algorithm 8.1) to both models the result will be that of Figure 9.5c in both cases: the terminal SCC containing all the states is merged into a single node $S1$, and the model collapses. Therefore, EXTRACTOR (Algorithm 8.2), that relies on the long-term transition probability of Definition 8.1, would produce[1] feature vectors F_1 and F_2 of Equation 9.1, that are obviously indistinguishable.

$$\begin{aligned} F_1 &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\ F_2 &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned} \tag{9.1}$$

The problem of model collapse is strictly tied to the shape of the Markov chain: fewer terminal SCCs will more likely result in equal feature vectors extracted with the long-term behavioral analysis. In our main realistic application setting though (Section 9.3), models are much more complex than those of the iterated Prisoner's Dilemma and the iterated RPS, where the state space is of size 4 and 9 respectively. Additionally, in such games, transitions are possible between all of the states, making the creation of terminal SCCs even more likely.

9.3 Malware analysis

As mentioned in Section 7.5, some advanced malware inject noise, e.g., sequences of random or non-dangerous actions, in their behavior as an anti-detection mechanism, hence creating an extremely challenging scenario for the analyzers. Another complication for the analyzer comes from small malware injected into bigger and benign applications, e.g., a password stealer inserted into the code of a game. In this case, the major portion of the behavioral model corresponds to actions belonging to the benign gaming application, whereas the few related to the password stealing process appear

[1]Assuming a blueprint of the same shape of Figures 9.5a and 9.5b, hence projecting all the edges of the models

Family	3-grams	4-grams	SECUR-AMA	Long-term
AndroRAT	0.83	0.86	0.84	**0.94**
GoldDream	0.88	0.92	0.92	**0.95**
Gorpo	0.77	0.66	0.81	**0.93**
Kemoge	0.58	0.57	0.44	**0.92**
Cova	0.91	0.92	0.94	**0.97**
FakeAV	0.89	0.89	0.89	0.89
Kuguo	0.87	0.85	**0.93**	0.90
SpyBubble	0.94	0.94	0.63	**0.95**
Winge	0.78	0.72	**0.88**	0.86

Table 9.5: Malware classification comparison F_1-score

within them. In this experimental scenario we aim at improving the identification of malware behavioral models extracted by SECUR-AMA for families that are difficult to correctly classify for the reasons explained. The dataset used (F. Wei et al., 2017) is the same of Section 7.1 but here the focus is on 4 particular families: *AndroRAT*, *GoldDream*, *Gorpo* and *Kemoge*. *AndroRAT* and *GoldDream* are an example of small malware injected into bigger applications (games and others). They steal personal information such as contact numbers, sms and call contents. Therefore, they react to the analyzer actions of sending an sms, making a call etc. *Gorpo* and *Kemoge* families instead employ anti-detection techniques such as dynamically loading the malicious code at runtime and performing unrelated actions to intentionally inject noise. We compare the long-term behavioral analysis approach with SECUR-AMA, where the same process is used to generate the malware models (the difference lies in the feature extraction as summarized in Figure 9.1). For a visual example of malware behavioral model refer to Figure 4.3. We also compare with the n-gram features extensively used in literature (Rieck et al., 2011; Wressnegger et al., 2013). As suggested in such works we experimented with SVM and K-NN classifiers using gram lengths in the range $[1, 4]$. To perform a fair comparison, we extract the n-grams directly from the same execution traces used by the other two methods. Regarding our approach, the blueprint D has been generated merging the graphs of the representatives for each family. Specifically, if the standalone (not injected) or clean (without anti-detection mechanisms) version of a malware for a family is known, its behavioral model is used as representative of such family, otherwise random behavioral models are chosen from the same family.

Table 9.5 reports our empirical best results obtained using SVMs and gram lengths 3 and 4. The two methods using features extracted from the Markov chains, i.e., SECUR-AMA and the long-term analysis, have overall better results when compared to n-grams. Using Markov chain transition probability values as features allows to better capture distinctive characteristics of the malware dynamics, improving the classifier performance. Moreover, when comparing SECUR-AMA and this new approach we can notice that the performance of the classifier are always comparable and most of the time are significantly better in favor the long-term. In more detail, the technique based on the long-term behavioral analysis performs significantly better for malware that are injected into benign applications (*AndroRAT* and *GoldDream*) or for malware employing anti-detection techniques preforming a lot of noisy actions (*Gorpo* and *Kemoge*). This confirms that using the long-term transition probability values as features allows to effectively remove noise and significantly improve the classification performances for such families, i.e., we achieve a gain up to 34%. With classical

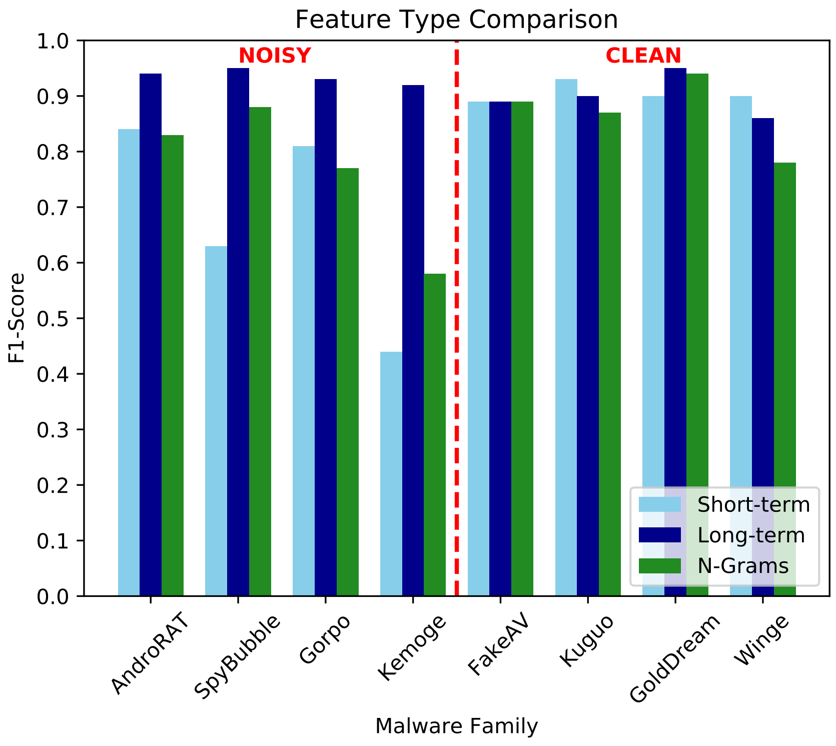


FIGURE 9.6: Visualization of the malware classification comparison F_1-score

malware (lower half of Table 9.5) instead, the long-term behavioral analysis does not consistently provide significant gains and is comparable to the others. This suggests that our proposed methodology should complement existing techniques to provide benefits in specific and important situations, e.g., malware injection and countering anti-detection mechanisms. Figure 9.6 displays a bar chart of the classification performance (F_1-score) comparing the different features tested.

9.4 Conclusions

We propose the use of Markov chains to identify known behaviors of intelligent agents acting within uncertain environments. More in detail, we employ classification to solve the problem and we use the long-term transition probability values as features. We design a transformation to enforce the absorbency property for Markov chains, enabling the computation of such features for generic Markov chains. This makes our approach independent of the specific technique used to extract the behavioral models. We evaluate our methodology in three domains: two player games, a single player repeated lottery game, and malware analysis. The empirical evaluation shows that our approach provides informative features to successfully identify known behaviors. In particular, for the malware analysis scenario this method allows to significantly outperform SECUR-AMA and other standard feature, i.e., n-grams, when considering real-world injected malware samples and advanced anti-detection mechanisms.

However there are some limitations of using the long-term transition probability as feature. For example, as explained in Section 9.1, if a model is composed by only few SCCs it may collapse and flatten all the probabilities extracted from it. An attacker can exploit such weakness by designing malware that perform execution traces forming complete graphs when represented with Markov chains, making the long-term behavioral analysis inconclusive. Moreover, a deeper empirical and theoretical study would be useful in order to assess the degradation of a behavioral model before applying the transformation algorithm. This would also improve the knowledge on whether it is useful to employ the long-term transition probability instead of other features (as seen in Section 9.3, not all the malware families benefit from that). Nonetheless, we believe that the approach detailed in this chapter represents a valid addition to the tools available for analyzing behaviors of intelligent agents based on Markov chains, as confirmed also by the empirical evaluation conducted.

Part V

Active Malware Analysis as a Bayesian Game

Chapter 10

Bayesian Active Malware Analysis

We have seen in previous chapters that malware can be grouped in families (or types), that are behavioral categories in which malicious software fall into. Nevertheless, this grouping applies also to actions that trigger malicious behaviors: every family responds to a certain set of triggers, many of which are shared across different families. For example, both a spyware and a ransomware may react to an incoming sms: the first forwarding it to a third party, whereas the second encrypting its content. However, this information on the grouping of triggering actions has never been used by previous works on malware analysis in order to improve the decision making strategy at runtime. In SECUR-AMA for example, the model is used to select the best action to perform next for the analyzer, without taking in consideration the type of the adversary. Consequently, for all the AMA techniques, if the number of possible analyzer actions is high, many analysis steps may be required to achieve a high classification score.

In this chapter, we present a new approach, BAMA, that uses the available information on the families and on the characteristics of behaviors they contain, i.e., triggers, to guide the analyzer in selecting triggering actions that reflect the current belief regarding which family the malware being analyzed belongs to. The main benefit of this is to be able to identify the correct malware family of the adversary with less interactions by explicitly lowering the uncertainty on its type rather than on its behaviors, i.e., on the model being generated. In particular, our aim is to exploit the intrinsic characteristics of the malware families, where each one is known to respond to specific stimuli depending on the malicious payload. With such information at hand, we model the analysis in order to reason about the malware family (type) at runtime, and to adapt the analyzer action selection strategy accordingly. Section 10.1 defines the problem and summarizes our solution approach; Section 10.2 investigates the sources of uncertainty for our problem and how to deal with them; Section 10.3 details the formalization of BAMA as a Bayesian game; Section 10.4 explains the new analyzer strategy we devised to play BAMA when analyzing malware; Section 11.2 presents the empirical results obtained with application of BAMA to real Android malware comparing with SECUR-AMA and other state-of-the-art approaches.

10.1 Problem definition

There exists an extensive literature that takes into account the type of the other agents to perform inference, with one key framework being that of Bayesian games (Section 2.2.4). These are used to model many domains, such as security games (Tambe, 2011; Jain et al., 2008; H. Xu et al., 2016), coalitional games (Chalkiadakis and Boutilier, 2007; Chalkiadakis et al., 2007), or network security (Jin et al., 2013; Xinxin Liu et al., 2013). In our proposed BAMA approach we build the formalization upon the link between malware family and the notion of types in Bayesian games. In particular, we formalize the analysis as a Bayesian game between an analyzer agent and a malware agent, focusing on the decision making strategy for the analyzer. Such strategy is guided by a utility function specifically designed to reflect the amount of uncertainty on the type of the adversary, according to the analyzer's belief. The aim is to be able to select triggering actions that allow one to infer the type of malware with increasing accuracy at every stage of the game while it progresses.

Our proposed methodology, BAMA, is to the best of our knowledge the first to consider the link between malware families and the notion of types in Bayesian games. The goal is to be able to match an unknown malware sample to the family (class) to which it belongs, by performing as few analyzer triggering actions as possible. In order to achieve this goal, we make use of a priori information about the malware families that BAMA employs. Such information is represented as a list of all the malware families that respond to a specific analyzer triggering action. Indeed, nowadays almost every repository of malware reports detailed information for every family, including the triggering mechanisms, such as the one we use in the experiments (F. Wei et al., 2017). Furthermore, in our application domain (Android systems), the triggers we use are actions that an average smartphone user would perform, e.g., sending an sms, making a call, opening the browser etc., for which an application has to declare listeners in the manifest to intercept (Section 5.1.3), thus easily obtainable by an analyzer. In contrast to previous works (Sartea and Farinelli, 2017; A. Martìn et al., 2018), in order to devise the strategy for the analyzer to identify the malware type, we do not make use of detailed information, i.e., exact list of APIs, of an execution trace, but rather only distinguish between a passive execution trace that could have been observed also without interacting with the malware, and a reactive execution trace that is triggered by the specific action performed by the analyzer. This is an advantage since we require less information and the abstraction between passive and reactive allows to deal with the intrinsic uncertainty of the problem. Before giving the formalization of BAMA (Section 10.3), we first explain the problems arising from the uncertainty on the observations of malware behaviors as execution traces, and how to solve them (Section 10.2).

10.2 Dealing with uncertainty

Our approach makes use of an Android sandbox emulator to run malicious applications, to execute analyzer triggering actions, and to observe and extract the corresponding execution traces (see Section 5.3 for details). However, uncertainty affects the readings of the execution traces from the sandbox due to several reasons. The first is given by the intrinsic slow nature of AMA: the emulator has to be reset to a clean state and rebooted after every interaction, hence it needs to take time. After executing a triggering action, the analyzer waits for a fixed amount of time before registering the malware response.[1] This is motivated by the fact that in Android malicious applications it is often the case that if a malware is reactive, it will probably respond in a reasonable time after having been triggered. Nevertheless, for this same reason execution trace can be "cut" in different places when read multiple times as a response to a trigger. The second reason of uncertainty is that malware also often employ deception techniques that intentionally inject noise (random or non-correlated APIs) within execution traces, to deceive an analyzer by trying to hide their malicious behavior (as extensively studied in Chapter 8). In addition, a malware could

[1]The amount of time is a parameter of the analysis depending on the host system. Usually varies between 10 and 30 seconds.

have been designed to activate the payload in response to a specific trigger with some probability, instead of being deterministic. Finally, malware samples belonging to the same family may differ, possibly not responding to a trigger that is instead common to other samples of the same family, due to code repackaging and other changes inserted by criminals that often modify existing malware rather than creating them from scratch (Upchurch and Zhou, 2016). Hence, based on system workload, timing constraints, deception techniques, and other factors, different execution traces from the same malware could be retrieved in response to the same triggering action. This makes the analysis task harder since the underlining decision making process is based on the observation of malware behaviors that is affected by uncertainty.

As mentioned, we are only interested in distinguishing between passive and reactive responses without analyzing in detail the content of the execution traces. A passive response is extracted before starting the game by simply executing a malware and observing its behavior without any interaction. Thus, we employ the Kullback-Leibler divergence D_{KL} of Definition 2.12 between the distribution of APIs P of the passive execution trace, and the distribution of APIs Q of another execution trace. In particular, we compute a threshold value ϵ on a training set of Android applications (both benign and malicious) by randomly executing triggering actions and measuring the mean value of D_{KL} between the passive and the reactive responses. Given P and Q, if $D_{KL}(P \parallel Q) \geq \epsilon$ the trace from which Q has been extracted is considered reactive, otherwise it is considered passive.

10.3 BAMA formalization

The goal of our proposed technique is to analyze a malware by repeatedly interacting with it in order to infer its type. Thus, BAMA is a game that we model from the point of view of the analyzer, meaning that it reflects how the analyzer sees the whole process. In particular, the utility function is designed from an information-centric perspective aimed at guiding the analyzer in acquiring information on the type of the adversary faced during the game.

Definition 10.1 (BAMA game)**.** *The game of BAMA is a Bayesian game with*

- $N = \{n_1, n_2\}$ *where* n_1 *is the analyzer and* n_2 *is the malware*
- $\boldsymbol{A} = A_1 \times A_2$ *where*
  - $A_1 = \{t_1, ..., t_m\}$ *are all the possible triggering actions for the analyzer (call, wifi, etc.)*
  - $A_2 = \{passive\,(p), reactive\,(r)\}$ *consists of a passive execution trace and a reactive response to a trigger for the malware*
- $\boldsymbol{\Theta} = \Theta_1 \times \Theta_2$ *where*
  - $\Theta_1 = \{\theta_1\}$ *the fixed type of the analyzer*
  - $\Theta_2 = \{f_1, ..., f_k\}$ *where* f_j *with* $j = 1, \ldots, k$ *is a malware family*
- $\boldsymbol{u} = (u_1, u_2)$ *is a profile of utility functions with*
  $u_1 : \boldsymbol{A} \times \boldsymbol{\Theta} \to \mathbb{R}_{\leq 0}$ *utility function for the analyzer*
- $p = Dir(\boldsymbol{\alpha})$ *is the Dirichlet prior over* $\boldsymbol{\Theta}$

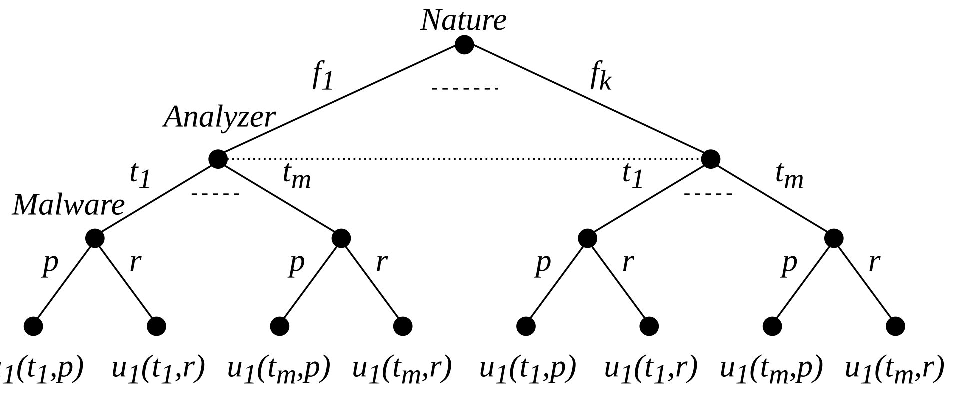

Nature
f_1
f_k
Analyzer
t_1
t_m
t_1
t_m
Malware
p r p r p r p r
$u_1(t_1,p)$ $u_1(t_1,r)$ $u_1(t_m,p)$ $u_1(t_m,r)$ $u_1(t_1,p)$ $u_1(t_1,r)$ $u_1(t_m,p)$ $u_1(t_m,r)$

FIGURE 10.1: A stage of the BAMA game

BAMA is clearly an instance of a Bayesian game of Definition 2.18 between two players: the analyzer n_1 and the malware n_2. The action set A_1 available to the analyzer comprises all the possible triggering actions that can be performed on the system (send/receive an sms, make/receive a call, enable/disable wifi, etc.) and that could possibly cause a reaction in the adversary. The malware action set A_2 instead is an abstraction over all the concrete actions that can be observed, i.e., the execution traces, that are grouped in either *passive* or *reactive*. The type of the analyzer θ_1 is fixed, whereas for the type set Θ_2 available to the malware we build a one-to-one correspondence with the possible malware families. The player's type θ_i encodes all the relevant private information for player i that in our context maps to how a malware responds to the possible analyzer triggering actions. Since $|\Theta_2| \geq 2$, giving a multinomial probability distribution as uncertainty measure over the types, our choice for prior p is the Dirichlet distribution, which is the conjugate of the multinomial distribution. The initialization of p can either be a uniform distribution or else reflect the distribution of the families in the dataset (or in the wild). The analyzer uses the prior p to reason about the next action to play during the game, updating it accordingly to the observation of the outcomes. The utility function is explained in Section 10.3.2 since it is based on the prior update process, therefore we first present that in Section 10.3.1 for a better understanding.

BAMA is intended to be played as a repeated game in multiple stages. In detail, every time a malware sample has to be analyzed, the analyzer starts a BAMA game of length n, i.e., of n stages in total. At each stage l, the analyzer selects a triggering action, observes the malware response, obtains the reward, updates the prior p into p' accordingly and moves to stage $l+1$ against the same malware sample but with the new prior (posterior) p'. At the end of stage n, the prior p is reset to its initial distribution and the analysis process starts again with a new malware sample from stage 1. Figure 10.1 depicts a BAMA stage game. Formally, since the type of the malware sample is initially unknown, we assume its type is drawn by nature at stage 1 and remains fixed for the next n stages (or rather that nature always draws the same type), until the game resets to stage 1 again.

10.3.1 Prior update

The Dirichlet prior is updated into a posterior by adding to parameters $\boldsymbol{\alpha}$ (a vector of pseudo-counts) the count of the new observations per class (Section 2.1.5). However, in contrast to many classical instances of Bayesian games, after receiving a reward we are still uncertain about the type of the adversary since more families can share the same triggers, requiring observations to be treated accordingly. We first define a function $g()$ that maps each analyzer triggering action *to the set of families that are known to respond to it* based on the a priori available information:

$$g : A_1 \to \mathcal{P}(\Theta_2) \tag{10.1}$$

Given a prior p with parameters $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k)$ at stage l, the posterior p' with parameters $\boldsymbol{\alpha'} = (\alpha'_1, \dots, \alpha'_k)$ at stage $l+1$ depends on the action a_2 of the malware after the analyzer played action a_1 at l. The α_j's are one per malware family f_j. The prior update is performed via function $w()$:

$$w(\boldsymbol{\alpha}, a_1, a_2) = \boldsymbol{\alpha'}$$
$$\text{where}$$
$$\alpha'_j = \begin{cases} \alpha_j + \frac{1}{|g(a_1)|} & f_j \in g(a_1) \wedge a_2 = \textit{reactive} \\ \alpha_j + \frac{1}{|g(a_1)|} & f_j \notin g(a_1) \wedge a_2 = \textit{passive} \\ \alpha_j & \text{otherwise} \end{cases} \tag{10.2}$$

with $1 \leq j \leq k$ and $f_j \in \Theta_2$. That is, if a malware actively responds to a triggering action a_1, i.e., $a_2 = \textit{reactive}$, we split the observation across all the families that are known to respond to a_1, which are given by $g(a_1)$. Conversely, if a malware does not respond to a_1, i.e., $a_2 = \textit{passive}$, we split the observation across all the families that are known to not respond to a_1. Notice that, as explained before, given the uncertainty in the readings from the emulator, we have to stay conservative as an execution trace may be detected as *passive* when it was *reactive* instead or vice versa. As such, we never force α'_j to be close to 0 as this can result in wrong inference in the remainder of the game.

10.3.2 Utility function

The key point of our formalization is the utility function u_1 for the analyzer. This is designed with the aim of guiding the selection of triggering actions so as to maximize the information acquired on the type of the adversary. The current belief on the adversary type is encoded by the prior $p = Dir(\boldsymbol{\alpha})$, and the multinomial distribution $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$ has an uncertainty degree tied to the entropy $H_D(\boldsymbol{\alpha})$ of Definition 2.11. The reward received by the analyzer then is the entropy of the posterior obtained by updating p (with Equation 10.2) after $a_1 \in A_1$ and $a_2 \in A_2$ are performed:

$$u_1(a_1, a_2) = H_D(w(\boldsymbol{\alpha}, a_1, a_2)) \tag{10.3}$$

where $w()$ is computed with Equation 10.2. Essentially, the utility is a function that corresponds to the amount of uncertainty on the type of the adversary resulting after the joint actions (a_1, a_2) have been played, hence on how the prior (and consequently the belief on the type of malware) changes due to an analyzer action. Our formalization allows to avoid the need to explicitly capture every single factor that contributes to the uncertainty (time, deception, etc., as mentioned in the first part of Section 10.2) by abstracting between passive and reactive responses and maintaining a prior probability distribution used in the utility function of the analyzer.

10.4 Analyzer strategy

BAMA is a game where the analyzer's end goal is to infer the type of the adversary. Indeed, knowing with enough confidence what is the type of the current adversary, allows the analyzer to pick the correct actions in order to trigger the malware and observe its malicious behavior in response. Nonetheless, in the context of malware analysis it is crucial to characterize the payload of a malware and this is often done inferring its behavioral category with respect to other known malicious samples. However, our aim is not to reach full code coverage, i.e., to enumerate all the reactive traces, as we rather want to gain enough information that allows to correctly classify a malware into its family, and consequently infer its behavior by similarity to others of the same category. If a malware family has 5 known triggers, and after executing 2 of them the prior points precisely, i.e., low uncertainty in the resulting multinomial distribution, to such family, it is useless to perform also the other 3 triggers and observe the corresponding new traces since that would mean spending time in performing interactions that with high probability will confirm the current prior shape (this is obviously an extreme example).

As such, we devise the analyzer strategy with the aim of reducing the entropy in the prior p at every stage. The key point is that after performing a trigger a_1, the analyzer acquires information on the response that changes the prior accordingly (Equation 10.2). Since the aim is to reduce the uncertainty on adversary type when sampling the prior, we base the analyzer strategy on the entropy minimization principle for p, employing a 1-step lookahead that considers the two known possible updates from p at stage l to p' at stage $l+1$ (passive or reactive response). Thus, the analyzer employs its current belief on the type of adversary, and chooses the action that reduces the entropy of p' the most. Formally, action selection starts with a sampling of the current prior p giving $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$, and then picks an entropy minimizing action, as follows

$$\begin{gathered}
\underset{a_1 \in A_1}{\operatorname{argmin}} \; [q \cdot u_1(a_1, \mathit{reactive}) + (1-q) \cdot u_1(a_1, \mathit{passive})] \\
\text{that is} \\
\begin{aligned}
\underset{a_1 \in A_1}{\operatorname{argmin}} \; & [q \cdot H_D(w(\boldsymbol{\alpha}, a_1, \mathit{reactive})) \\
& + (1-q) \cdot H_D(w(\boldsymbol{\alpha}, a_1, \mathit{passive}))]
\end{aligned} \\
\text{with} \\
q = \sum_{f_j \in g(a_1)} \theta_j
\end{gathered} \tag{10.4}$$

where $w()$ is defined in Equation 10.2 and H_D is the entropy of the Dirichlet from Definition 2.11. The value of q sums up to the probability for the adversary to respond to the analyzer triggering action a_1, based on the current prior p and the information on the families and their triggers (function $g()$ of Equation 10.1). Equation 10.4 then corresponds to selecting the action that minimizes the expectation over H_D at the next stage.

After a BAMA game ends, the state of the prior p should reveal the type of the adversary the analyzer has been confronting. However, the same set of triggers may be shared among multiple malware families, making them to be seen as the same type in our formalization. For this reason, we make use of Markov chain based models (Figure 4.3) as a tie breaker. Specifically, such a model is generated using the execution traces observed while playing BAMA, but it is not considered at all during the game: it is only used at the end to clear the uncertainty in the prior, if so required. Moreover, building such models allows us to compare with state-of-the-art techniques in terms of model classification score, as detailed in Section 11.2.

Algorithm 10.1 BAMA Analysis

```
Require:
    $p = Dir(\boldsymbol{\alpha})$ - prior over $\Theta_2$
    $n$ - game length
    $\epsilon$ - threshold value for $D_{KL}$

1: Retrieve distribution $P$ of passive trace
2: for $n$ times do
3:     Sample $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$
4:     Select action $a_1$ with Equation 10.4 using $\boldsymbol{\theta}$
5:     Execute $a_1$ and retrieve distribution $Q$ of the trace
6:     if $D_{KL}(P \parallel Q) \geq \epsilon$ then
7:         $a_2 \leftarrow reactive$
8:     else
9:         $a_2 \leftarrow passive$
10:    $\boldsymbol{\alpha'} \leftarrow w(\boldsymbol{\alpha}, a_1, a_2)$          ▷ Update with Equation 10.2
11:    Update $p$ with $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha'}$
```

Algorithm 10.1 details the BAMA analysis. The first step (line 1) retrieves the distribution of the passive trace for the computation of D_{KL} later. At this point the game begins by sampling the prior p for action selection (lines 3-4). The selected action a_1 is then executed on the emulator and the subsequent execution trace of the malware is retrieved along with its distribution Q (line 5). Based on the value of $D_{KL}(P \parallel Q)$ with respect to the threshold ϵ, action a_2 of the malware is assigned as *passive* or *reactive* (lines 6-9). Finally, the prior parameters $\boldsymbol{\alpha}$ are updated according to the outcome, and the game progresses to the next stage (lines 10-11). At every stage l the analyzer selects the action that minimizes the entropy H_D of the prior p at stage $l + 1$. Reducing the entropy at every stage achieves the result of reducing the uncertainty on the type of adversary when sampling the prior, as confirmed by experiments and visible in Figure 11.1.

The overall BAMA analysis pipeline with respect to the framework described in Chapter 5 is shown is Figure 10.2, where the analyzer's decision making module is represented by a BAMA stage game with the current value of the Dirichlet prior. The behavior observation module differs from SECUR-AMA as it assign the resulting execution trace, after the execution of a triggering action, to either passive or reactive, consequently updating the prior before restarting the emulator in order to play the next stage.

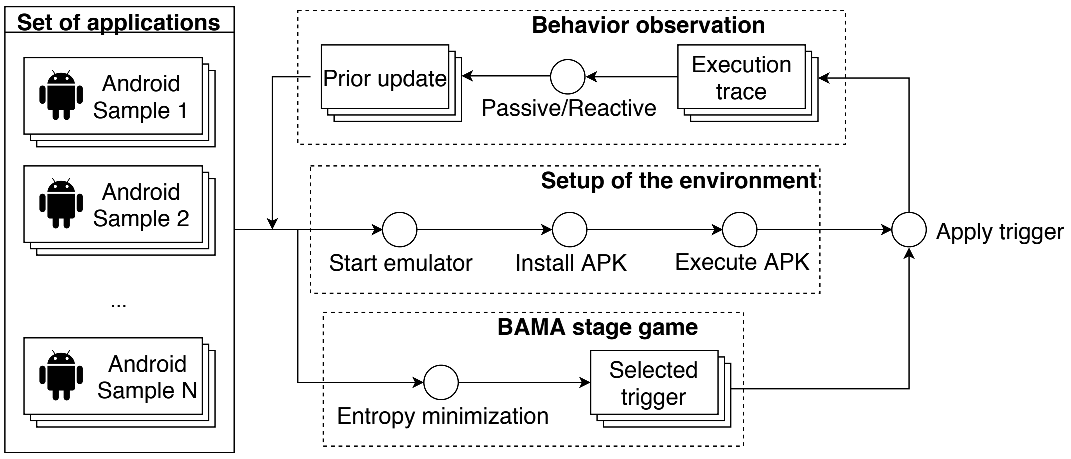


FIGURE 10.2: BAMA analysis pipeline

Chapter 11

Empirical Evaluation of BAMA

In the experimental setting we compare BAMA with other three AMA techniques: MYOPIC (Williamson et al., 2012), SECUR-AMA, and CANDYMAN (A. Martìn et al., 2018). All the techniques tested in the experiments share the end goal of identifying the family of an unknown malicious software by employing different strategies for the analyzers (as previously detailed in Section 3.2). Section 11.1 briefly explains the characteristics of the dataset employed in the experiments; Section 11.2 reports the results obtained with all the tested techniques; Section 11.3 concludes the chapter with final considerations and future directions.

11.1 Dataset

We use the same dataset of the previous chapters (Section 7.1) composed of about 1400 real Android malware partitioned into 24 families. For instance, the family *Finspy* concerns the logging and exfiltration of personal information of the user on an Android device, thus it is sensitive to calls, SMS activities, browser navigation history updates, etc. Furthermore, some of the families included in this experiment can be seen as challenging to correctly classify, since they employ specific mechanisms aimed at deceiving the analysis. In particular, *Gorpo* and *Kemoge* employ a combination of anti-analysis techniques such as the dynamic loading of the malicious code at runtime and the execution of noisy unrelated API calls that are not useful to implement the malware payload but serve as a method to mislead an analyzer that focuses on the sequence of actions performed by the malicious sample. Hence, behaviors related to the damaging payload interwaved by noisy APIs can induce an analyzer to overlook malicious characteristics. Moreover, *AndroRat* and *GoldDream* families distinguish themselves on the type of infection vector, as they are composed by small malware injected into complex harmless applications such as games. This peculiar feature causes only a small portion of the observed execution traces to depict malicious behaviors, while the rest being related to the harmless application that has been injected, thus making the malware identification hard. Malware samples belonging to *Opfake* are designed to receive commands from an external server controlled by an attacker in order to be triggered and show their malicious behavior. This happens also for samples of *Tesbo* that additionally also try to hide themselves by not having a GUI for the user to see. The rest of the families involved in the dataset can be considered less sophisticated because they do not employ advanced anti-detection mechanisms and do not hide themselves through injection into other applications. The detailed report available for each family has been used to build function $g()$ of Equation 10.1 with the a priori information about which families respond to which triggers; $g()$ is used then in turn to construct the initial Dirichlet prior for our experiments (reflecting the dataset composition visible in Figure 7.1).

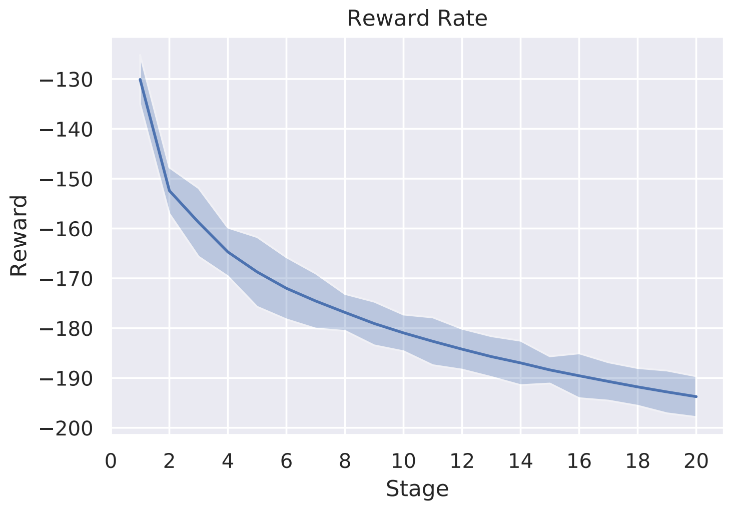


FIGURE 11.1: Rewards obtained over time in a BAMA game

11.2 Empirical evaluation

For the experiments we employed a Stratified K-Fold Cross Validation with $K = 5$ to provide training and testing sets with 5 different random splits. Quality of the results is assessed with unweighted standard measures, i.e., precision, recall, and F_1-score. Implementations of the classifier, i.e., a linear SVM,[1] quality measures and cross-validation make use of Scikit-Learn (Pedregosa et al., 2011).

Results in Figure 11.1 show that rewards, i.e., the entropy of the posterior Dirichlet distribution, clearly decrease at each stage of BAMA, a confirmation that the analyzer progressively reduces the uncertainty on the sampling of the type of adversary. Next, we perform a comparison by classifying the transition matrices of the Markov chain models (Figure 4.3) obtained after every step of the analysis for each of the techniques employed in the experiments. In particular, SECUR-AMA outputs such model by default, we apply the same model generation method to the MYOPIC algorithm, and we build the model also while playing BAMA as explained before (without using it in any way during the analysis). CANDYMAN instead outputs models of the same kind but without the conditioning of the probabilities based on the analyzer action, i.e., the model is composed by a single Markov chain. This allows us to fairly compare the different AMA techniques as they all share the same type of output model. The set of triggering actions for the analyzer is the same for BAMA, SECUR-AMA and MYOPIC, and is composed of 17 different actions that mimic a standard user's behavior: *send/receive sms, make/receive call, switch on/off Wifi/GPS/screen, charge/discharge battery, add/remove contact, install/remove app, set clock* (Section 5.5). CANDYMAN instead uses different triggering actions that are related to the GUI of every specific application, randomly selecting up to 5000 of such actions in 5 minutes. Therefore, we are able to show the classification score rate of BAMA, SECUR-AMA and MYOPIC, as the progression in terms of analyzer actions performed is comparable; while for CANDYMAN we can only show the score after the analysis is complete, since the number and type of triggering actions are of different nature.

[1]We also tested other classifiers but the linear SVM gives the best results.

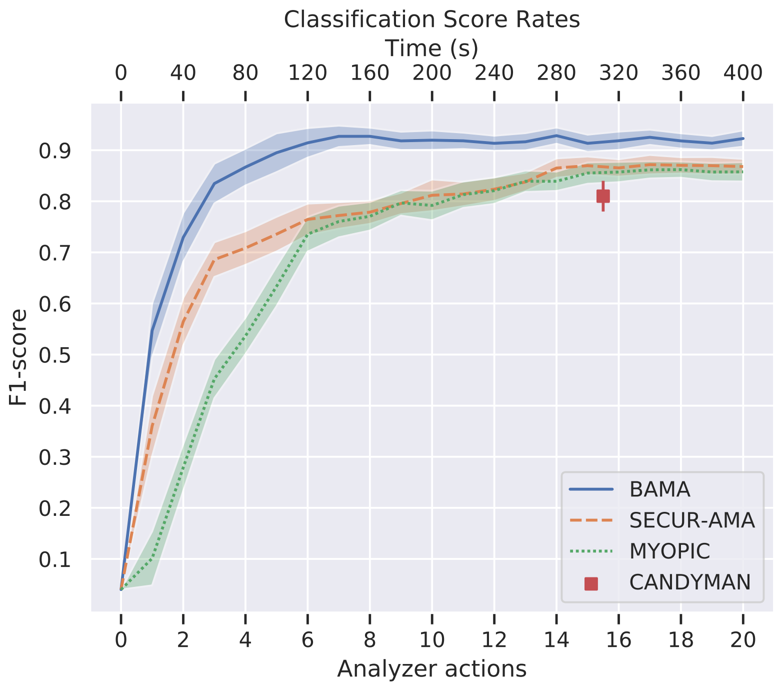


FIGURE 11.2: Comparison of classification score rates

Figure 11.2 shows the classification score rates in term of F_1-score obtained, where the shaded area represent the confidence interval. It is clearly visible that BAMA learns faster compared to the other techniques: it only requires 4 analyzer actions (about 80 seconds) to reach the same best overall classification score (0.87) of SECUR-AMA. Indeed, the speed of the analysis is extremely important: there are many web services that analyze malware at users' request: the faster the response the better. However, the greatest impact is for security firms that have to analyze a huge amount of new malware discovered every day (in the order of thousands), thus, reducing the number of analyzer actions required to identify a malware has a big impact on the overall analysis time.

Furthermore, BAMA also reaches the highest global classification score (0.92), due to the fact that by using the information on the adversary type, final malware models contain less noise, i.e., fewer Markov chains associated to reactions to triggers that are not meaningful for that malware, augmenting the classification process. The difference between BAMA and the other techniques is statistically significant according to a Student's paired two-tailed t-test with $p < 0.05$. It is clear that BAMA improves the strategy of the analyzer: it allows to pick a sequence of actions that is shorter and more precise compared to the other techniques in order to generate a good malware model based on the observation of the traces. The SVM classifiers used on the models extracted by SECUR-AMA, MYOPIC and CANDYMAN reach a worse classification score than when BAMA is used to create the models. Therefore, the BAMA decision making strategy has clear value in terms of results. Table 11.1 shows the best overall F_1-score classification results for the best instance of each technique: 8 actions in 160 seconds for BAMA, 14 actions in 280 seconds for SECUR-AMA, 18 actions in 360 seconds for MYOPIC, and 310 seconds for CANDYMAN.

Although BAMA performs better overall when compared to the other techniques, it does not perform better for every malware family in the dataset. Table 11.2 details the per-family F_1-score for the best instance of each technique tested, i.e., the BAMA,

	BAMA	SECUR-AMA	MYOPIC	CANDYMAN
Analyzer actions	**8**	14	18	-
Seconds	**160**	280	360	310
F_1-score	**0.92**	0.87	0.86	0.81

Table 11.1: Best overall classification results

SECUR-AMA, MYOPIC, and CANDYMAN instances reported in Table 11.1. In the case of *AndroRAT* and *GoldDream* for example, the BAMA entropy minimization of the prior strategy is able to overcome the injection problem (mentioned at the beginning of this section) by executing a limited subset of triggers that are specific to the injected malicious part of the application, allowing it at the same time to identify the correct family. For the other techniques that instead rely on the model for their decision making strategy (whereas BAMA does not use it at all), the presence of a small malicious behavior within a bigger harmless application is harder to detect. The anti-analysis routines employed by the *Kemoge* family make the samples load the code at runtime when triggered by specific actions on the GUI: since CANDYMAN triggering mechanism is specifically designed around the GUI, it is more effective in stimulating the samples belonging to such family. On the other hand, since *Tesbo* does not have a GUI, CANDYMAN is not able to trigger any behaviors from such samples. Nevertheless, all the techniques perform badly on *Tesbo* because of the code coverage problem of dynamic analysis: some malicious behaviors are triggered by commands coming from an external server, therefore they are never exhibited without designing application specific triggers (this is the case also for the *Opfake* family).

The fact that BAMA does not rely on the sequences of API calls but abstracts them in either passive or reactive has the good side effect of being able to better counter the dynamic obfuscation and noise injection mechanisms of some malware. The reason is that an execution trace presenting unrelated noisy APIs most likely has an entropy value different from the passive execution trace, therefore such trace will be tagged as reactive. Consequently, a malware trying to hide its intentions by injecting noise, will reveal itself as responding to the triggering action, which is all what BAMA considers. A main reason for BAMA to perform worse with some families (although still comparably to the other techniques) lies in the stochasticity in the retrieval of malware responses since the threshold value ϵ for the Kullback-Leibler divergence may result in a wrong identification of the trace type and hence impact on trigger selection. Another reason for errors in classification comes from the fact, explained in Section 10.2, that a specific malware sample may respond not only to the triggers for which its family is known to respond to, but also to triggers of other families as well. Conversely, a malware sample could also not respond to some triggers listed for its family. This happens in the experiments and it is a major source of uncertainty, as well as one of the key reasons why the problem of malware analysis based on triggering is difficult and the classification will not be perfect. Figure 11.3 shows how the expectation of the Dirichlet prior (Definition 2.8) is updated after every stage of the BAMA game when analyzing a malware sample of the *Opfake* family. It can be noticed that after an initial adjustment during the first 7 stages, from that point onward the analyzer is quite confident about the malware family type. This result is tied to that of Figure 11.1, as converging to a single family in the belief also reduces the uncertainty in the Dirichlet prior. Finally, Figure 11.4 shows the confusion matrix related to Table 11.2 for a better visualization of the classification results.

Family	BAMA	SECUR-AMA	MYOPIC	CANDYMAN
AndroRAT	**0.93**	0.84	0.84	0.85
Boqx	0.95	**0.96**	0.93	0.90
Cova	**0.97**	0.94	0.89	0.92
FakeAV	0.89	0.89	0.89	0.89
FakeDoc	1.00	1.00	1.00	0.98
Finspy	1.00	1.00	1.00	1.00
Fjcon	**0.94**	0.87	0.79	0.55
GoldDream	**0.94**	0.92	0.87	0.68
Gorpo	**0.87**	0.81	0.82	0.81
Kemoge	0.70	0.44	0.35	**0.76**
Kuguo	**0.94**	0.93	0.91	0.84
Leech	0.99	**1.00**	0.98	0.98
Mseg	**0.99**	0.98	0.97	0.93
Obad	1.00	1.00	1.00	1.00
Opfake	**0.75**	0.63	0.67	0.57
SmsZombie	1.00	1.00	1.00	1.00
SpyBubble	**0.95**	0.63	0.46	0.36
Stealer	1.00	1.00	1.00	1.00
Svpeng	1.00	1.00	1.00	0.96
Tesbo	**0.57**	0.33	0.57	0.00
Triada	**0.91**	0.81	0.84	0.74
Vidro	0.96	1.00	1.00	1.00
Vmvol	1.00	1.00	1.00	0.92
Winge	**1.00**	0.88	0.91	0.76

TABLE 11.2: Per-family F_1-score classification for the best instance of each technique as reported in Table 11.1

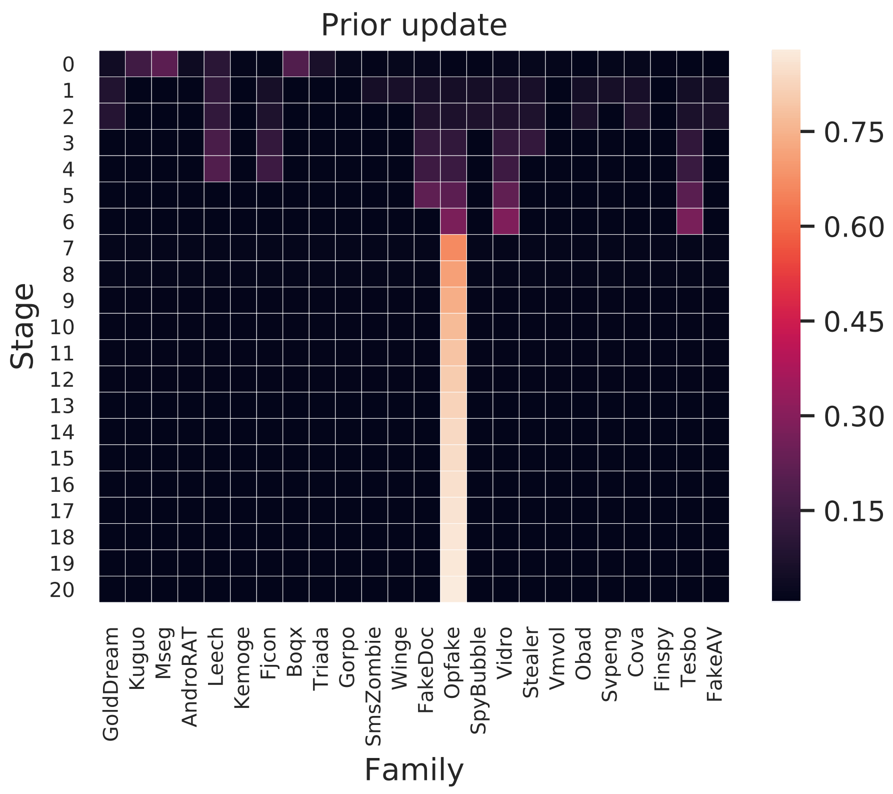
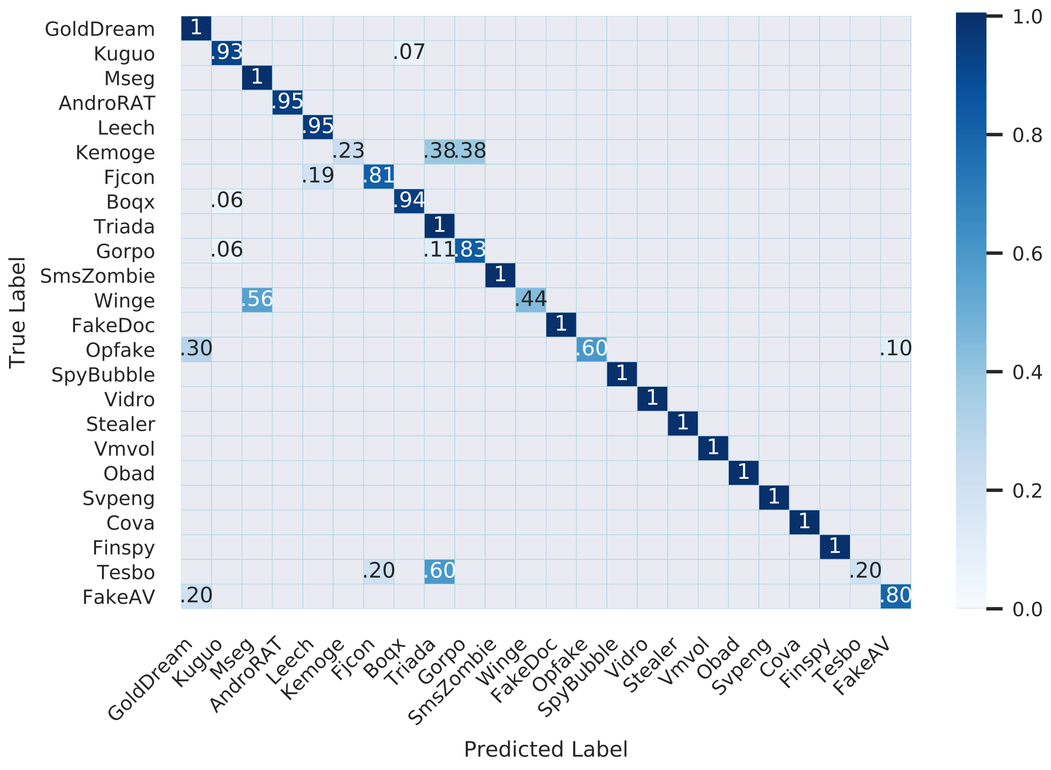


FIGURE 11.3: Heatmap visualizing how the expectation of the prior is updated after every stage of the BAMA game when analyzing a malware sample of the *Opfake* family



True Label	GoldDream	Kuguo	Mseg	AndroRAT	Leech	Kemoge	Fjcon	Boqx	Triada	Gorpo	SmsZombie	Winge	FakeDoc	Opfake	SpyBubble	Vidro	Stealer	Vmvol	Obad	Svpeng	Cova	Finspy	Tesbo	FakeAV
GoldDream	1																							
Kuguo		.93						.07																
Mseg			1																					
AndroRAT				.95																				
Leech					.95																			
Kemoge						.23			.38	.38														
Fjcon					.19		.81																	
Boqx		.06						.94																
Triada									1															
Gorpo		.06							.11	.83														
SmsZombie											1													
Winge			.56									.44												
FakeDoc													1											
Opfake	.30													.60										.10
SpyBubble															1									
Vidro																1								
Stealer																	1							
Vmvol																		1						
Obad																			1					
Svpeng																				1				
Cova																					1			
Finspy																						1		
Tesbo							.20		.60														.20	
FakeAV	.20																							.80

FIGURE 11.4: Normalized confusion matrix for BAMA with Linear SVM classifier. Values smaller than 0.05 are masked

11.3 Conclusions

BAMA is a novel technique for dynamic malware analysis formalized as a Bayesian game between an analyzer and a malware agent. Specifically, the formalization is built upon the link between malware family and the notion of types in Bayesian games. To guide the analyzer we design a utility function that expresses the amount of uncertainty on the type of the adversary after the execution of an action. Such uncertainty is represented by a Dirichlet prior over the possible types of the game, i.e., the possible malware families, that is employed as analyzer's belief. The algorithm devised to play BAMA aims at minimizing the entropy of the analyzer's belief at every stage of the game in a myopic fashion selecting the action the allows to acquire the most information possible about the malware family of the current adversary. Experiments on a dataset of real Android malware show that, when compared to other state-of-the-art techniques, BAMA requires fewer actions (and consequently time) to reach a satisfying classification score for malware identification. As such, our approach paves the way for using Bayesian malware analysis in a large and significant scale.

Nevertheless, one of the strengths of BAMA, i.e., the exploitation of easy to obtain prior information, is also a potential weakness: if such information is incorrect, the analysis could be inconclusive. Indeed, if the function $g()$ that maps triggers to families is imprecise, the analyzer is not able to pick actions that lower the entropy of the Dirichlet prior. However, the amount of wrong mappings inside $g()$ has to be consistent for BAMA to not work at all, since (as explained in Section 10.2) some degree of uncertainty is already taken into account by the formalization, e.g., malware that do not respond to some triggers of their families.

Notice that with BAMA we generate the same malware models of SECUR-AMA, however the decision making process of the analyzer is not agnostic with respect to the families, on the contrary it is heavily reliant on the mapping between triggers and families. For this reason, we believe that SECUR-AMA is more indicated than BAMA for the identification of new unknown families (see Section 7.5).

Chapter 12

Conclusions and Future Work

In this final chapter we draw the conclusions of the work conducted in the thesis (Section 12.1), and we also present an overview of some possible future directions to improve the presented approaches and further develop research in the field of AI and cyber-security (Section 12.2).

12.1 Conclusions

This thesis proposes a novel perspective for AMA, where the malware and the analyzer are modeled as intelligent agents that interact during the analysis process. A main goal of our approach is to analyze the behavior of the malware, i.e., what the malware does during its execution. To fully realize this we propose the use of a probabilistic malware behavioral model, i.e., a Markov chain, and we devise novel methodologies that allow an analyzer agent to select triggering actions that can build such a model while interacting with the malware. The behavioral models generated with our proposed techniques can be grouped and compared to each other in order to successfully identify the family of malware, giving better results with respect to existing state-of-the-art techniques.

The key problem when designing an AMA analyzer is to devise her decision making strategy in order to select triggering actions that allow to acquire the highest amount of information possible on the adversary. Our research is inserted in the mentioned scenario and provides contributions to AMA and behavioral analysis of adversarial agents based on interaction. We propose solutions in three areas: *(1)* dynamic generation of malware behavioral models; *(2)* extraction of a new type of feature for classification of behavioral models derived from the analysis of the long-term behavior; *(3)* new formalization of AMA as a Bayesian game. More in detail, this thesis advances the state-of-the-art with these contributions:

1. Dynamic generation of malware behavioral models
    - We define a behavioral model based on multiple Markov chains where each one represents the observation of a behavior in response to a specific analyzer action. This allows us to efficiently compute the probability distribution over the possible responses to the analyzer actions
    - We develop a RL approach for AMA based on MCTS that can dynamically generate the malware behavioral model at runtime, i.e., while interacting with the malware
2. Extraction of a new type of feature for classification of behavioral models derived from the analysis of the long-term behavior

- We use of the long-term transition probability as a new type of feature to classify behavioral models. Focusing on the long-term allows to enhance the connections between states that are important for the malicious behavior and that arise when reaching a fixpoint of the process depicted by the Markov chain of the model. This lessens the impact of noise within the models that is usually more apparent when considering the classical transition matrix (1-step connections)
- We design a transformation for Markov chains enforcing the absorbency property. This allows us to use standard techniques to derive the long-term transition probability, making our approach independent from the specific process that generated the Markov chain based models

3. New formalization of AMA as a Bayesian game

   - We propose BAMA, a novel technique for AMA, formalized as a Bayesian game between an analyzer agent and a malware agent, focusing on the decision making strategy for the analyzer. The formalization is built upon the link between malware family and the notion of types in Bayesian games. A key point is the design of the utility function, which reflects the amount of uncertainty on the type of the adversary after the execution of an analyzer action
   - We devise an algorithm to exploit the formalization building on an entropy minimization principle applied to the analyzer's belief. This allows us to successfully reduce her uncertainty on the type of the adversary at every stage of the game

We also build a comprehensive analysis framework for Android malware that includes as modules for the analyzer all the solutions presented: SECUR-AMA, the long-term behavioral analysis and BAMA. The empirical evaluations reported in this thesis have been conducted with such framework analyzing a dataset of real malware composed of about 1400 samples divided into 24 families (F. Wei et al., 2017).

This thesis provides novel contributions to AMA showing that the use of intelligent analyzer agents gives significant improvements to the analysis of malware.

12.2 Future Work

The work conducted opens to several future research lines within malware analysis scenarios and also in other domains.

- In future work we will consider the use of static analysis techniques to augment AMA, for example, to extract triggers from malware code in addition to the current triggers based on the simulation of user activity. Indeed, combining static and dynamic analysis would help in mitigating the limitations of the separate techniques, although being not trivial. Nonetheless, fusing the information acquired by static and dynamic methods in meaningful features, or using static analysis combined with dynamic analysis to create a comprehensive analysis approach could really improve the overall methodology

- In this thesis we always considered malware as belonging to one specific family. Sometimes however, many malware of different families are injected within a benign application that serves as infection vector. This poses a challenge for the analyzer since the observed behaviors are partitioned into multiple families instead of only one. Hence, directing research effort into extending the presented approaches in order to deal with such scenario would be a significant contribution

- Regarding the procedure for the computation of the long-term transition probability, there is an interesting theoretical link with the field of abstract interpretation, which extracts and studies semantic properties of computer programs with formal reasoning methodologies that heavily rely on the concept of fixpoint. The transformation algorithm that identifies and merges the terminal SCCs allowing to reason about the connections between states at a fixpoint is indeed a form of abstract interpretation. Therefore, applying techniques of such field to the computations performed on the Markov chain models could help improving the approach to analyze the long-term behavior. We think that this connection is worth further investigation

- Another interesting direction is that of improving SECUR-AMA by augmenting the simulation capabilities of the MCTS with generative models for the execution traces. The idea is to address the problem of generating an execution trace with Natural Language Processing (NLP) techniques, since an execution trace could be interpreted as a sentence where words (tokens) correspond to API calls. This would allow us to take advantage of the vast literature of generative models for NLP (Gatt and Krahmer, 2018) and apply it to our cyber-security domain

- The recent success of Generative Adversarial Networks (GAN) could provide an interesting direction too for the generation of realistic execution traces (J. Xu et al., 2018; Fedus et al., 2018; J. Guo et al., 2018), however the application to sequences is still limited, especially when dealing with discrete tokens (such as APIs). The game theoretic implications of GAN also apply well to AMA and it would be interesting to substitute the analyzer and the malware agents with the two competing network where the first generates sequences of inputs to trigger the second. Indeed, a network as attacker instead of a malware is not an unrealistic case since, nowadays, the extensive use of AI to empower cyber-security systems has given rise to attacks targeting machine learning algorithms rather than the system architectures themselves (Samangouei et al., 2018; Xuanqing Liu and Hsieh, 2019)

- In the context of attacks to machine learning algorithms we believe that the game theoretical formalization could be more resilient to such attacks with respect to other classifiers based on feature selection or extraction. There are works that show how the features computed by a neural network can be studied in order to generate samples that escape or change the classification boundaries as desired by a malicious attacker (Eykholt et al., 2018; L. Tong et al., 2019). While playing a game instead, the analyzer can compute a strategy that already takes into account a strategic malware (attacker) adapting the decision making strategy in order to play at the equilibrium. By definition of equilibrium, an attacker can not do any better by deviating from the best response that the analyzer has computed. This approach would not make the analyzer perfect, neither always effective, nevertheless it would allow to plan a defense strategy that is geared toward the worst case scenario. In this perspective, in the near future we plan to extend BAMA to consider also malware that actively try to counter a strategic analyzer. By this we do not mean malware that use simple anti-emulation mechanisms, but rather malware that, on top of the usual goal and ability to release the payload, also strategize to counter an analyzer's attempts to reveal their family. Of course, malware of such level of sophistication are not common yet in the real world, but the existing few are arguably among the most dangerous ones. Our formalization allows the easy modeling of such malware, requiring only the careful design of the utility function so that it accurately represents the adversary goals and abilities. Indeed, for strategic malware, responding to an analyzer triggering action becomes a choice that has to balance the trade off between releasing the payload and not revealing the belonging family. The utility function of the malware then will have to consider that. Designing the decision making strategy of BAMA to consider the malware as strategic will allow the analyzer to reason about (or play at) the equilibrium, therefore gaining the maximum out of the analysis process in the worst case

- We plan to investigate an interesting research domain for the techniques proposed in this thesis that goes beyond malware analysis: network security. We believe that AMA techniques could find application as an intelligent firewall interacting with agents over a network, extending the framework to the case of multiple (possibly adversarial) unknown agents, e.g., IoT devices. Our analysis framework would be a starting point, although the extension requires a consistent amount of research

- We also plan to apply AMA to cyber-physical systems and in particular to robot security. We believe that, in the robotic domain, being able to perform actions in the physical world in order to gain information to better refine what caused a possible anomaly would be a great improvement to current anomaly and fault detection techniques. For example, an autonomous drone that is undergoing a GPS spoofing attack could perform the physical action of a fixed trajectory to understand if such attack is actually happening. The cost of acting in the real world, e.g., battery consumption and execution time, should be carefully evaluated by the system in order to balance the cost of the action with the amount of information that can potentially be acquired. This requires a sophisticated decision making mechanism that can act in face of uncertainty. The long-term behavioral analysis could be an important additional tool for attack or anomaly detection in robotic platforms since the complexity of the tasks performed are subject to errors, consequently inserting noise in the observed behavioral models

- Attackers will likely develop new anti-emulation methods, and the analysis framework would benefit from the integration of a proper counter anti-emulation layer to hide the sandbox, making it appear as a physical smartphone to the malware. The current solution is just a collection of masking functions that attackers can easily bypass with a slight change to the malware code. Research in adaptive counter anti-emulation techniques, i.e., stealthy mechanisms for the sandbox that adapt to what the malware does at runtime instead of being fixed, is another interesting direction

Overall, the field of cyber-security (be it related to computer, network or cyber-physical systems) is playing nowadays a fundamental role as application scenario for research in AI, and will do even more so in the future. We believe that the approaches presented in this thesis are a significant contribution to the synergy between cyber-security and AI from which both fields can benefit, providing interesting solutions at the intersection between theory and practice.

Glossary

A

Air Gapped Newtork A network security measure employed on one or more computers to ensure that a secure computer network is physically isolated from unsecured networks, such as the public Internet or an unsecured local area network 62

C

Call Graph The call graph represents, using graph notation, the call dependencies between functions of a program. The call graph does not represent the possible "flows" of a program (as the control flow graph does), i.e., no if-then-else branching or loops are considered, but if a function can possibly call another one, a dependency is created (Allen, 1970) 40

Control Flow Graph Representation, using graph notation, of all paths that might be traversed through a program during its execution. The CFG is essential to many compiler optimizations and static analysis tools. In a control flow graph each node in the graph represents a basic block: jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves (Allen, 1970) 40

D

Duopoly In the duopoly game, two firms produce the same good and can decide to charge either a low (L) price or a high (H) price. The goal of each firm is to earn the highest profit as possible. If both firms choose H, each earns \$1000; if one chooses H and the other L, the one that chose H incurs in a loss of \$200, whereas the other earns \$1200; if both firms choose L, each earns \$600. The game matrix visible in Table 1 models the same situation of the Prisoner's Dilemma (Osborne and Rubinstein, 1994)

		Firm 2	
		H	L
Firm 1	H	$1000, 1000$	$-2000, 1200$
	L	$1200, -200$	$600, 600$

TABLE 1: Duopoly game matrix

19

Dynamic Obfuscation In the most general sense, to obfuscate a program means to transform it into a form that is more difficult for an adversary to understand or change than the original code. In the case of dynamic obfuscation, the software make the execution trace observed at runtime change every time inserting random actions or interweaving different behaviors (Collberg and Nagra, 2009) 89

J

Joint project Two students are working on a joint project. Each of them can either work hard (W) or goof off (G), preferring to goof off if the other works hard (although the outcome of the project would be better if both work hard, but the increment in value is not worth the extra effort). The outcome of both working hard is preferable to that of both goofing off (in which case nothing gets accomplished), whereas the worst outcome for a working hard student is for the other one to goof off (as the first gets exploited). The game matrix visible in Table 2 models the same situation of the Prisoner's Dilemma (Osborne and Rubinstein, 1994)

		Student 2	
		W	G
Student 1	W	$2, 2$	$0, 3$
	G	$3, 0$	$1, 1$

TABLE 2: Joint project game matrix

19

K

k-nearest neighbor The k-nearest neighbor classifier is a supervised approach commonly based on the Euclidean distance (but any other choice suitable for the specific application domain can be used) between test samples a training samples. With k-nearest neighbor where $k = 1$, the predicted class of test sample is set equal to the true class of its nearest neighbor training sample. For $k > 1$ the predicted class of test sample is set equal to the most frequent true class among k nearest training samples (Cover and Hart, 2006)

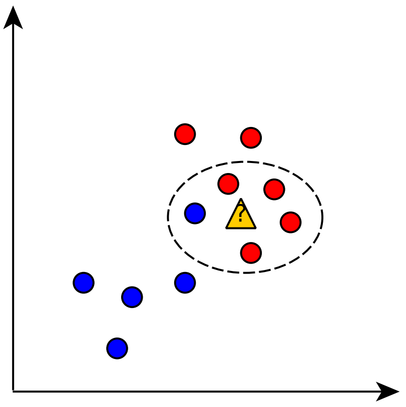


FIGURE 1: k-nearest neighbor decision with $k = 5$. The new sample "?" will be assigned to the red class

40, 79

M

Aperiodic Markov chain The period d_i of a state s_i of a Markov chain $\boldsymbol{P}$ is defined to be the Greatest Common Divisor (GCD) $d_i = gcd\{n \mid P_{ii}^n > 0\}$. A state s_i is aperiodic if $d_i = 1$. A Markov chain is aperiodic if the period of all its states is 1. If an irreducible Markov chain has an aperiodic state, then all of its states are aperiodic and so is the entire Markov chain (Kemeny and Snell, 1983) 34, 131

Ergodic Markov chain A Markov chain is ergodic if it is aperiodic and positive recurrent (Kemeny and Snell, 1983) 34

Positive Recurrent Markov chain A state s_i of a Markov chain is recurrent if the process will return to state s_i with probability 1, otherwise s_i is transient. A Markov chain is recurrent if all of its states are recurrent. Let $T_i = \{n \geq 1 \mid X_n = i\}$ be the time of first return to state s_i. Then s_i is positive recurrent if $\mathbb{E}(T_i \mid X_0 = i) < \infty$, otherwise s_i is null recurrent. A Markov chain is positive recurrent if all its states are positive recurrent (Kemeny and Snell, 1983) 34, 131

Regular Markov chain A Markov chain with transition matrix $\boldsymbol{P}$ is called regular if $\boldsymbol{P}^n$ has only positive components for any integer n (Kemeny and Snell, 1983) 36

Markov perfect equilibrium The Markov perfect equilibrium corresponds to subgame perfect equilibrium for stochastic games with the additional properties: the strategies of players can depend only on the current state (they have to be Markovian); the state can encode only payoff-relevant information, meaning that strategies must depend only on moves of the players without considering cooperation, signaling, etc. (Fudenberg and Tirole, 1991) 28

N

n-gram An n-gram is a contiguous sequence of n items from a given sequence, e.g., words in a text, APIs in an execution trace, opcodes in a program, etc. n-gram models, in the form of Markov models, are widely used to predict the next item in a sequence in the fields of probability, communication theory, natural language processing, computational biology, and data compression. The main limitation of n-grams, despite their relative simplicity, is that by increasing n the spatial requirements grow exponentially (Sidorov et al., 2013) 40–42

P

Pareto optimality An action profile is Pareto optimal (or efficient) if there is no other action profile that increases the payoff of at least one player without decreasing anyone else's (Mock, 2011). For the Prisoner's Dilemma (Table 2.1), the Pareto optimal action profile is (C, C) since it is the best outcome considering both suspect payoffs 20

Perfect Bayesian equilibrium A strategy profile for an extensive game is a perfect Bayesian equilibrium if: each player's strategy specifies optimal actions given her beliefs and the strategies of other players (*sequential rationality*); each player's belief is consistent with her strategy profile (*consistency of beliefs*). The second part requires the Bayes rule to be followed to maintain consistency of the beliefs where appropriate (Osborne and Rubinstein, 1994) 24

R

Random forest classifier Random forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. The fact that a forest of trees instead of a single decision tree is generated helps reducing the overfitting problem (Ho, 1995)

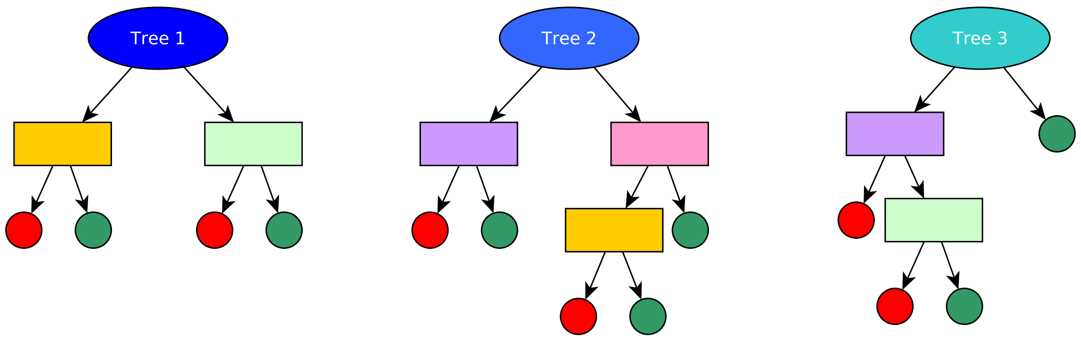


FIGURE 2: Random forest composed by 3 decision trees

79

S

Sequential equilibrium Sequential equilibrium strengthens the perfect Bayesian equilibrium by requiring that beliefs be justifiable as coming from some set of totally mixed strategies that are close to the equilibrium strategies, i.e., a small perturbation of the equilibrium strategies. (Osborne and Rubinstein, 1994) 24

Sub-game perfect equilibrium Nash equilibrium ignores the sequential structure of an extensive game, leading to situations in which player's strategies are optimal at the beginning of the game, but are sub-optimal for some histories h (sub-games). For such cases, the concept of sub-game perfect equilibrium comes in handy: in no sub-game can any player i do better by choosing a strategy different from s_i^*, given that every other player j adheres to s_j^*, where s_* is the strategy profile of the sub-game perfect equilibrium (Osborne and Rubinstein, 1994) 22

Support vector machine Support vector machines are supervised models for classification that construct a set of hyperplanes in order to separate the points of the training set by the largest margin (max margin) with respect to their classes. Support vector machine are originally applied to linearly separable classes, but they have been extended to non-linearly separable ones with the use of kernels to implicitly mapping their inputs into high-dimensional feature spaces., e.g., radial basis function kernel (Cortes and Vapnik, 1995)

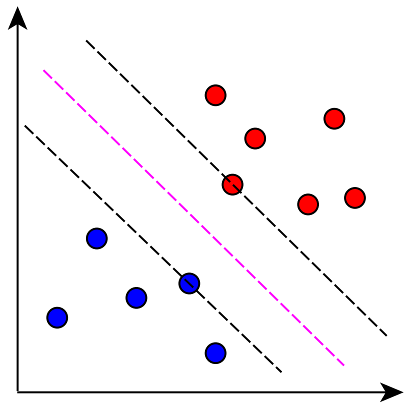


FIGURE 3: Separating hyperplane (purple line) in a 2-dimensional space

47, 79, 103, 118

T

Term Frequency-Inverse Document Frequency Term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. The TFIDF value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word. Denoting D as the corpus of documents and $f_{t,d}$ as the number of times term t occurs in document $d \in D$: (Rajaraman et al., 2014)

$$tfidf(t,d,D) = \left(\frac{1}{2} + \frac{1}{2}\frac{f_{t,d}}{max\{f_{t',d} \mid t' \in d\}}\right) \log \frac{|D|}{|\{d \in D \mid t \in d\}|}$$

40

Acronyms

IoT Internet of Things 1, 128

K

***K*-NN** *K*-Nearest Neighbor 40, 47, 75, 78, 84, 104, *Glossary:* *k*-nearest neighbor

M

MCA Monte Carlo Analysis 68, 97–99

MCTS Monte Carlo Tree Search 4, 6, 7, 11, 30–32, 46, 47, 63, 67–73, 77, 84, 125, 127

MDP Markov Decision Process 28

N

NLP Natural Language Processing 127

O

OTP One Time Password 60, *Glossary:* One Time Password

R

RL Reinforcement Learning 4, 6, 32, 46, 125

RPS Rock Paper Scissors 97, 99–101, 103

S

SCC Strongly Connected Component 92–94, 99, 102, 103, 105, 127, *Glossary:* Strongly connected component

SVM Support Vector Machine 47, 75, 79, 81, 84, 97, 104, 118, 119, *Glossary:* Support vector machine

T

TFIDF Term Frequency-Inverse Document Frequency 40, 133, *Glossary:* Term Frequency-Inverse Document Frequency

U

UCB Upper Confidence Bound 29, 30, 32

UCT Upper Confidence Bound for Trees 32, 69

X

XML eXtensible Markup Language 40, 42, 62

Bibliography

Aafer, Yousra, Wenliang Du, and Heng Yin (2013). “DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android”. In: *Security and Privacy in Communication Networks.* Ed. by Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao. Cham: Springer International Publishing, pp. 86–103.

Allen, Frances E. (July 1970). “Control Flow Analysis”. In: *SIGPLAN Not.* 5.7, pp. 1–19. ISSN: 0362-1340. DOI: `10.1145/390013.808479`. URL: `http://doi.acm.org/10.1145/390013.808479`.

Arifoglu, Damla and Abdelhamid Bouchachia (2017). “Activity Recognition and Abnormal Behaviour Detection with Recurrent Neural Networks”. In: *Procedia Computer Science* 110. 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, pp. 86–93. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2017.06.121`. URL: `http://www.sciencedirect.com/science/article/pii/S1877050917313005`.

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer (May 2002). “Finite-time Analysis of the Multiarmed Bandit Problem”. In: *Machine Learning* 47.2, pp. 235–256. ISSN: 1573-0565. DOI: `10.1023/A:1013689704352`. URL: `https://doi.org/10.1023/A:1013689704352`.

Bhandari, Shweta, Rekha Panihar, Smita Naval, Vijay Laxmi, Akka Zemmari, and Manoj Singh Gaur (2018). “SWORD: Semantic aWare andrOid malwaRe Detector”. In: *Journal of Information Security and Applications* 42.IEEE Trans Inf Forensics Secur 12 8 2017, pp. 46–56. DOI: `10.1016/j.jisa.2018.07.003`. URL: `https://app.dimensions.ai/details/publication/pub.1106321275`.

Browne, Cameron (2011). “The Dangers of Random Playouts”. In: *ICGA Journal* 34.1, pp. 25–26. DOI: `10.3233/ICG-2011-34105`. URL: `https://doi.org/10.3233/ICG-2011-34105`.

Browne, Cameron, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton (2012). “A Survey of Monte Carlo Tree Search Methods.” In: *IEEE Trans. Comput. Intellig. and AI in Games* 4.1, pp. 1–43.

Busic, A., I.M.H. Vliegen, and A. Scheller-Wolf (2009). *Comparing Markov chains: combining aggregation and precedence relations applied to sets of states.* English. Report Eurandom. Eurandom.

Calleja, Alejandro, Alejandro Martìn, Héctor D. Menéndez, Juan E. Tapiador, and David Clark (2018). “Picking on the family: Disrupting android malware triage by forcing misclassification”. In: *Expert Syst. Appl.* 95, pp. 113–126. DOI: `10.1016/j.eswa.2017.11.032`. URL: `https://doi.org/10.1016/j.eswa.2017.11.032`.

Chalkiadakis, Georgios and Craig Boutilier (2007). “Coalitional Bargaining with Agent Type Uncertainty”. In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence.* IJCAI’07. Hyderabad, India: Morgan Kaufmann Publishers Inc., pp. 1227–1232. URL: `http://dl.acm.org/citation.cfm?id=1625275.1625474`.

Chalkiadakis, Georgios, Evangelos Markakis, and Craig Boutilier (2007). “Coalition Formation Under Uncertainty: Bargaining Equilibria and the Bayesian Core Stability Concept”. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’07. Honolulu, Hawaii: ACM, 64:1–64:8. ISBN: 978-81-904262-7-5. DOI: 10.1145/1329125.1329203. URL: http://doi.acm.org/10.1145/1329125.1329203.

Chaslot, G.M.J.B., M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy (2007). “Progressive strategies for Monte-Carlo tree search”. Dutch. In: *Information Sciences 2007*. Ed. by P.P. Wang. Vol. 10. Pagination: 7. World Scientific Publishing Company, pp. 655–661. ISBN: 9812709665.

Chaslot, Guillaume, Sander Bakkes, Istvan Szita, and Pieter Spronck (2008). “Montecarlo Tree Search: A New Framework for Game AI”. In: *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AIIDE’08. Stanford, California: AAAI Press, pp. 216–217. URL: http://dl.acm.org/citation.cfm?id=3022539.3022579.

Chen, Xi, Xiaotie Deng, and Shang-Hua Teng (May 2009). “Settling the Complexity of Computing Two-player Nash Equilibria”. In: *J. ACM* 56.3, 14:1–14:57. ISSN: 0004-5411. DOI: 10.1145/1516512.1516516. URL: http://doi.acm.org/10.1145/1516512.1516516.

Cheung, Matthew (June 2018). *Market Share: Operating Systems, Worldwide, 2017*. https://www.gartner.com/doc/3879167/market-share-operating-systems-worldwide. Gartner, Inc.

Collberg, Christian and Jasvir Nagra (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1st. Addison Wesley Professional.

Cortes, Corinna and Vladimir Vapnik (Sept. 1995). “Support-vector networks”. In: *Machine Learning* 20.3, pp. 273–297. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018.

Coulom, Rémi (2007). “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 72–83. ISBN: 978-3-540-75538-8.

Cover, T. and P. Hart (Sept. 2006). “Nearest Neighbor Pattern Classification”. In: *IEEE Trans. Inf. Theor.* 13.1, pp. 21–27. ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1053964. URL: https://doi.org/10.1109/TIT.1967.1053964.

Dyer, Martin, Leslie Ann Goldberg, Mark Jerrum, and Russell Martin (2006). “Markov chain comparison”. In: *Probab. Surveys* 3, pp. 89–111. DOI: 10.1214/154957806000000041. URL: https://doi.org/10.1214/154957806000000041.

Ebrahimi, Nader, Ehsan S. Soofi, and Shaoqiong (Annie) Zhao (Mar. 2011). “Information Measures of Dirichlet Distribution with Applications”. In: *Appl. Stoch. Model. Bus. Ind.* 27.2, pp. 131–150. ISSN: 1524-1904. DOI: 10.1002/asmb.870. URL: http://dx.doi.org/10.1002/asmb.870.

Elisan, C.C. (2015). *Advanced Malware Analysis*. McGraw-Hill Education. URL: https://books.google.it/books?id=17SUAwAAQBAJ.

Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song (2018). “Robust Physical-World Attacks on Deep Learning Visual Classification”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pp. 1625–1634. DOI: 10.1109/CVPR.2018.00175.

Fedus, William, Ian Goodfellow, and Andrew M. Dai (2018). "MaskGAN: Better Text Generation via Filling in the blanks". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=ByOExmWAb.

Friedman, James W. (1971). "A Non-cooperative Equilibrium for Supergames". In: *Review of Economic Studies* 38.1, pp. 1–12. URL: https://EconPapers.repec.org/RePEc:oup:restud:v:38:y:1971:i:1:p:1-12..

Fudenberg, Drew and Jean Tirole (1991). *Game Theory*. Translated into Chinesse by Renin University Press, Bejing: China. Cambridge, MA: MIT Press.

Gao, Debin, Michael K. Reiter, and Dawn Song (2008). "BinHunt: Automatically Finding Semantic Differences in Binary Programs". In: *Proceedings of the 10th International Conference on Information and Communications Security*. ICICS '08. Birmingham, UK: Springer-Verlag, pp. 238–255. ISBN: 978-3-540-88624-2. DOI: 10.1007/978-3-540-88625-9_16. URL: http://dx.doi.org/10.1007/978-3-540-88625-9_16.

Garbaczewski, Piotr (Apr. 2006). "Differential Entropy and Dynamics of Uncertainty". In: *Journal of Statistical Physics* 123.2, p. 315. ISSN: 1572-9613. DOI: 10.1007/s10955-006-9058-2. URL: https://doi.org/10.1007/s10955-006-9058-2.

Gascon, Hugo, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck (2013). "Structural Detection of Android Malware Using Embedded Call Graphs". In: *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*. AISec '13. Berlin, Germany: ACM, pp. 45–54. ISBN: 978-1-4503-2488-5.

Gatt, Albert and Emiel Krahmer (Jan. 2018). "Survey of the State of the Art in Natural Language Generation: Core Tasks, Applications and Evaluation". In: *J. Artif. Int. Res.* 61.1, pp. 65–170. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=3241691.3241693.

Gelly, Sylvain and David Silver (July 2011). "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go". In: *Artif. Intell.* 175.11, pp. 1856–1875. ISSN: 0004-3702. DOI: 10.1016/j.artint.2011.03.007. URL: http://dx.doi.org/10.1016/j.artint.2011.03.007.

Ginsberg, Matthew L. (June 2001). "GIB: Imperfect Information in a Computationally Challenging Game". In: *Journal of Artificial Intelligence Research* 14.1, pp. 303–358. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622394.1622405.

Guo, Jiaxian, Sidi Lu, Han Cai, Weinan Zhang, Yong Yu, and Jun Wang (2018). "Long Text Generation via Adversarial Training with Leaked Information". In: *AAAI*, pp. 5141–5148. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16360.

Guo, Yuhong and Dale Schuurmans (2007). "Discriminative Batch Mode Active Learning". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., pp. 593–600. ISBN: 978-1-60560-352-0. URL: http://dl.acm.org/citation.cfm?id=2981562.2981637.

Han, Weijie, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao (2019). "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics". In: *Computers & Security* 83, pp. 208–233. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2019.02.007. URL: http://www.sciencedirect.com/science/article/pii/S016740481831246X.

Harsanyi, John C. (1967). "Games with Incomplete Information Played by "Bayesian" Players, I–III Part I. The Basic Model". In: *Management Science* 14.3, pp. 159–182. DOI: 10.1287/mnsc.14.3.159. eprint: https://doi.org/10.1287/mnsc.14.3.159. URL: https://doi.org/10.1287/mnsc.14.3.159.

Hernandez-Leal, Pablo and Michael Kaisers (2017). “Towards a Fast Detection of Opponents in Repeated Stochastic Games”. In: *Autonomous Agents and Multiagent Systems*. Ed. by Gita Sukthankar and Juan A. Rodriguez-Aguilar. Cham: Springer International Publishing, pp. 239–257. ISBN: 978-3-319-71682-4.

Hiraishi, Kunihiko and Koichi Kobayashi (2014). “Detection of Unusual Human Activities Based on Behavior Modeling”. In: *IFAC Proceedings Volumes* 47.2. 12th IFAC International Workshop on Discrete Event Systems (2014), pp. 182–187. ISSN: 1474-6670. DOI: `https://doi.org/10.3182/20140514-3-FR-4046.00029`. URL: `http://www.sciencedirect.com/science/article/pii/S1474667015374000`.

Ho, Tin Kam (1995). “Random Decision Forests”. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR ’95. Washington, DC, USA: IEEE Computer Society, pp. 278–. ISBN: 0-8186-7128-9. URL: `http://dl.acm.org/citation.cfm?id=844379.844681`.

Jain, Manish, James Pita, Milind Tambe, Fernando Ordóñez, Praveen Paruchuri, and Sarit Kraus (June 2008). “Bayesian Stackelberg Games and Their Application for Security at Los Angeles International Airport”. In: *SIGecom Exch.* 7.2, 10:1–10:3. ISSN: 1551-9031. DOI: `10.1145/1399589.1399599`. URL: `http://doi.acm.org/10.1145/1399589.1399599`.

Jin, Xinyu, Niki Pissinou, Sitthapon Pumpichet, Charles A. Kamhoua, and Kevin A. Kwiat (2013). “Modeling cooperative, selfish and malicious behaviors for Trajectory Privacy Preservation using Bayesian game theory”. In: *38th Annual IEEE Conference on Local Computer Networks, Sydney, Australia, October 21-24, 2013*, pp. 835–842. DOI: `10.1109/LCN.2013.6761339`. URL: `https://doi.org/10.1109/LCN.2013.6761339`.

Jurišić, M., D. Kermek, and M. Konecki (May 2012). “A review of iterated prisoner’s dilemma strategies”. In: *2012 Proceedings of the 35th International Convention MIPRO*, pp. 1093–1097.

Kachitvichyanukul, Voratas and Bruce W. Schmeiser (Feb. 1988). “Binomial Random Variate Generation”. In: *Commun. ACM* 31.2, pp. 216–222. ISSN: 0001-0782. DOI: `10.1145/42372.42381`. URL: `http://doi.acm.org/10.1145/42372.42381`.

Kemeny, J.G. and J.L. Snell (1983). *Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix"*. Undergraduate Texts in Mathematics. Springer New York. ISBN: 9780387901923. URL: `https://books.google.com.sg/books?id=0bTK5uWzbYwC`.

Kocsis, Levente and Csaba Szepesvári (2006). “Bandit Based Monte-carlo Planning”. In: *Proceedings of the 17th European Conference on Machine Learning*. ECML’06. Berlin, Germany: Springer-Verlag, pp. 282–293. ISBN: 978-3-540-45375-8.

Kullback, Solomon and Richard A Leibler (1951). “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1, pp. 79–86.

Lai, T.L and Herbert Robbins (1985). “Asymptotically efficient adaptive allocation rules”. In: *Advances in Applied Mathematics* 6.1, pp. 4–22. ISSN: 0196-8858. DOI: `https://doi.org/10.1016/0196-8858(85)90002-8`. URL: `http://www.sciencedirect.com/science/article/pii/0196885885900028`.

Lakhotia, A., M. Dalla Preda, and R. Giacobazzi (2013). “Fast location of similar code fragments using semantic ’Juice’”. In: *2nd Workshop on Program Protection and Reverse Engineering PPREW 2013*. ACM.

Lee, P.M. (2012). *Bayesian Statistics: An Introduction*. Wiley. ISBN: 9781118359778. URL: `https://books.google.it/books?id=WOWOKqWQwAcC`.

Lin, Chih-Hung, Hsing-Kuo Pao, and Jian-Wei Liao (2018). “Efficient dynamic malware analysis using virtual time control mechanics”. In: *Computers & Security* 73, pp. 359–373. ISSN: 0167-4048. DOI: `https://doi.org/10.1016/j.cose.`

2017.11.010. URL: http://www.sciencedirect.com/science/article/pii/S016740481730247X.

Liu, Xinxin, Kaikai Liu, Linke Guo, Xiaolin Li, and Yuguang Fang (2013). "A game-theoretic approach for achieving k-anonymity in Location Based Services." In: *INFOCOM*. IEEE, pp. 2985–2993. ISBN: 978-1-4673-5944-3. URL: http://dblp.uni-trier.de/db/conf/infocom/infocom2013.html#LiuLGOF13.

Liu, Xuanqing and Cho-Jui Hsieh (2019). "Rob-GAN: Generator, Discriminator, and Adversarial Attacker". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pp. 11234–11243.

Mariconti, Enrico, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini (2017). "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models". In: *NDSS*. The Internet Society.

Marpaung, J. A. P., M. Sain, and Hoon-Jae Lee (Feb. 2012). "Survey on malware evasion techniques: State of the art and challenges". In: *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pp. 744–749.

Martìn, Alejandro, Raúl Lara-Cabrera, and David Camacho (2019). "Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPy-Tool framework and the OmniDroid dataset". In: *Information Fusion* 52, pp. 128–142. DOI: 10.1016/j.inffus.2018.12.006. URL: https://doi.org/10.1016/j.inffus.2018.12.006.

Martìn, Alejandro, Víctor Rodríguez-Fernández, and David Camacho (2018). "CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains". In: *Engineering Applications of Artificial Intelligence* 74, pp. 121–133. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2018.06.006. URL: http://www.sciencedirect.com/science/article/pii/S0952197618301374.

Martìn, Ignacio, José Alberto Hernández, and Sergio de los Santos (2019). "Machine-Learning based analysis and classification of Android malware signatures". In: *Future Generation Computer Systems* 97, pp. 295–305. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.03.006. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18325159.

Mas-Colell, Andreu, Michael D. Whinston, and Jerry R. Green (1995). *Microeconomic Theory*. New York: Oxford University Press.

Mayil, V.valli (May 2012). "Web Navigation Path Pattern Prediction using First Order Markov Model and Depth first Evaluation". In: *International Journal of Computer Applications* 45.16, pp. 26–31.

Meng, Guozhu, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan (2016). "Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbr&#252;cken, Germany: ACM, pp. 306–317. ISBN: 978-1-4503-4390-9.

Mertens, J. -F. and A. Neyman (June 1981). "Stochastic games". In: *International Journal of Game Theory* 10.2, pp. 53–66. ISSN: 1432-1270. DOI: 10.1007/BF01769259. URL: https://doi.org/10.1007/BF01769259.

Mock, William B. T. (2011). "Pareto Optimality". In: *Encyclopedia of Global Justice*. Ed. by Deen K. Chatterjee. Dordrecht: Springer Netherlands, pp. 808–809. ISBN: 978-1-4020-9160-5. DOI: 10.1007/978-1-4020-9160-5_341. URL: https://doi.org/10.1007/978-1-4020-9160-5_341.

Moser, Andreas, Christopher Kruegel, and Engin Kirda (2007). “Exploring Multiple Execution Paths for Malware Analysis”. In: *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. SP ’07. Washington, DC, USA: IEEE Computer Society, pp. 231–245. ISBN: 0-7695-2848-1.

Myerson, Roger B. (Sept. 1997). *Game Theory: Analysis of Conflict*. Harvard University Press.

Nash, J. F. (1950). “Equilibrium Points in N-Person Games”. In: *Proceedings of the National Academy of Sciences of the United States of America* 36.48-49.

Ng, K.W., G.L. Tian, and M.L. Tang (2011). *Dirichlet and Related Distributions: Theory, Methods and Applications*. Wiley Series in Probability and Statistics. Wiley. ISBN: 9781119998419. URL: `https://books.google.it/books?id=k8GS868oyo4C`.

Nissim, Nir, Robert Moskovitch, Lior Rokach, and Yuval Elovici (2014). “Novel active learning methods for enhanced PC malware detection in windows OS.” In: *Expert Syst. Appl.* 41.13, pp. 5843–5857. URL: `http://dblp.uni-trier.de/db/journals/eswa/eswa41.html#NissimMRE14`.

Norris, Donald J. (2017). “Games”. In: *Beginning Artificial Intelligence with the Raspberry Pi*. Berkeley, CA: Apress, pp. 77–110. ISBN: 978-1-4842-2743-5. DOI: `10.1007/978-1-4842-2743-5_4`. URL: `https://doi.org/10.1007/978-1-4842-2743-5_4`.

Nowak, Martin and Karl Sigmund (Aug. 1993). “A Strategy of Win-Stay, Lose-Shift That Outperforms Tit-for-Tat in the Prisoner’s Dilemma Game”. In: *Nature* 364, pp. 56–8.

Osborne, Martin J. and Ariel Rubinstein (July 1994). *A Course in Game Theory*. Cambridge, MA, USA: The MIT Press.

Otte, Stefan, Johannes Kulick, Marc Toussaint, and Oliver Brock (2014). “Entropy-based strategies for physical exploration of the environment’s degrees of freedom”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pp. 615–622. DOI: `10.1109/IROS.2014.6942623`. URL: `https://doi.org/10.1109/IROS.2014.6942623`.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Rajaraman, Anand, Jure Leskovec, and Jeffrey D. Ullman (2014). *Mining Massive Datasets*. URL: `http://infolab.stanford.edu/~ullman/mmds/book.pdf`.

Rieck, Konrad, Philipp Trinius, Carsten Willems, and Thorsten Holz (2011). “Automatic analysis of malware behavior using machine learning”. In: *Journal of Computer Security* 19.4, pp. 639–668. DOI: `10.3233/JCS-2010-0410`. URL: `https://doi.org/10.3233/JCS-2010-0410`.

Samangouei, Pouya, Maya Kabkab, and Rama Chellappa (2018). “Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models”. In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=BkJ3ibb0-`.

Sartea, Riccardo, Georgios Chalkiadakis, Alessandro Farinelli, and Matteo Murari (2020). “Bayesian Active Malware Analysis”. In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS ’20. Accepted for publication. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Sartea, Riccardo and Alessandro Farinelli (2017). "A Monte Carlo Tree Search approach to Active Malware Analysis". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 3831–3837. DOI: `10.24963/ijcai.2017/535`. URL: `https://doi.org/10.24963/ijcai.2017/535`.

Sartea, Riccardo and Alessandro Farinelli (2018). "Detection of Intelligent Agent Behaviors Using Markov Chains". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '18. Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, pp. 2064–2066. URL: `http://dl.acm.org/citation.cfm?id=3237383.3238073`.

Sartea, Riccardo, Alessandro Farinelli, and Matteo Murari (2019). "Agent Behavioral Analysis Based on Absorbing Markov Chains". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '19. Montreal QC, Canada: International Foundation for Autonomous Agents and Multiagent Systems, pp. 647–655. ISBN: 978-1-4503-6309-9. URL: `http://dl.acm.org/citation.cfm?id=3306127.3331752`.

Sartea, Riccardo, Alessandro Farinelli, and Matteo Murari (2020). "SECUR-AMA: Active Malware Analysis Based on Monte Carlo Tree Search for Android Systems". In: *Engineering Applications of Artificial Intelligence* 87, p. 103303. ISSN: 0952-1976. DOI: `https://doi.org/10.1016/j.engappai.2019.103303`. URL: `http://www.sciencedirect.com/science/article/pii/S0952197619302635`.

Sartea, Riccardo, Mila Dalla Preda, Alessandro Farinelli, Roberto Giacobazzi, and Isabella Mastroeni (2016). "Active Android Malware Analysis: An Approach Based on Stochastic Games". In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. SSPREW '16. Los Angeles, California, USA: ACM, 5:1–5:10. ISBN: 978-1-4503-4841-6. DOI: `10.1145/3015135.3015140`. URL: `http://doi.acm.org/10.1145/3015135.3015140`.

Sarukkai, Ramesh R. (2000). "Link Prediction and Path Analysis Using Markov Chains". In: *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*. Amsterdam, The Netherlands: North-Holland Publishing Co., pp. 377–386. URL: `http://dl.acm.org/citation.cfm?id=347319.346322`.

Sharif, Monirul, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee (2008). "Computer Security - ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings". In: ed. by Sushil Jajodia and Javier Lopez. Springer Berlin Heidelberg. Chap. Eureka: A Framework for Enabling Static Malware Analysis, pp. 481–500. ISBN: 978-3-540-88313-5. DOI: `10.1007/978-3-540-88313-5_31`. URL: `http://dx.doi.org/10.1007/978-3-540-88313-5_31`.

Sheppard, Brian (2002). "World-championship-caliber Scrabble". In: *Artificial Intelligence* 134.1, pp. 241–275. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(01)00166-7`. URL: `http://www.sciencedirect.com/science/article/pii/S0004370201001667`.

Sidorov, Grigori, Francisco Velasquez, Efstathios Stamatatos, Alexander Gelbukh, and Liliana Chanona-Hernández (2013). "Syntactic Dependency-Based N-grams as Classification Features". In: *Advances in Computational Intelligence*. Ed. by Ildar Batyrshin and Miguel González Mendoza. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–11.

Sikorski, Michael and Andrew Honig (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA: No Starch Press. ISBN: 9781593272906.

Sobczyk, K. (2001). "INFORMATION DYNAMICS: PREMISES, CHALLENGES AND RESULTS". In: *Mechanical Systems and Signal Processing* 15.3, pp. 475–498. ISSN: 0888-3270. DOI: https://doi.org/10.1006/mssp.2000.1378. URL: http://www.sciencedirect.com/science/article/pii/S0888327000913785.

Suarez-Tangil, Guillermo, Mauro Conti, Juan E. Tapiador, and Pedro Peris-Lopez (2014). "Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models". In: *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I.* Cham: Springer International Publishing, pp. 183–201. ISBN: 978-3-319-11203-9.

Suarez-Tangil, Guillermo, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco Alis (2014). "Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families". In: *Expert Syst. Appl.* 41.4, pp. 1104–1117.

Szita, István and András Lőrincz (2008). "The Many Faces of Optimism: A Unifying Approach". In: *Proceedings of the 25th International Conference on Machine Learning.* ICML '08. Helsinki, Finland: ACM, pp. 1048–1055. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390288. URL: http://doi.acm.org/10.1145/1390156.1390288.

Tambe, Milind (2011). *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned.* 1st. New York, NY, USA: Cambridge University Press. ISBN: 1107096421.

Tarjan, R. (Oct. 1971). "Depth-first search and linear graph algorithms". In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

AV-Test (2019). *Malware Statistics and Trends Report.* https://www.av-test.org/en/statistics/malware/. Independent IT-Security Institute.

Tong, Liang, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik (Aug. 2019). "Improving Robustness of ML Classifiers against Realizable Evasion Attacks Using Conserved Features". In: *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, pp. 285–302. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/tong.

Tong, Simon and Daphne Koller (Mar. 2002). "Support Vector Machine Active Learning with Applications to Text Classification". In: *J. Mach. Learn. Res.* 2, pp. 45–66. ISSN: 1532-4435. DOI: 10.1162/153244302760185243. URL: https://doi.org/10.1162/153244302760185243.

Upchurch, Jason and Xiaobo Zhou (2016). "Malware Provenance: Code Reuse Detection in Malicious Software at Scale". In: *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 1–9. DOI: 10.1109/malware.2016.7888735.

Walenstein, Andrew and Arun Lakhotia (2006). "The Software Similarity Problem in Malware Analysis". In: *Duplication, Redundancy, and Similarity in Software.*

Walenstein, Andrew, Michael Venable, Matthew Hayes, Christopher Thompson, and Arun Lakhotia (2007). "Exploiting Similarity Between Variants to Defeat Malware " Vilo " Method for Comparing and Searching Binary Programs". In:

Wei, Fengguo, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou (2017). "Deep Ground Truth Analysis of Current Android Malware". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17).* Bonn, Germany: Springer, pp. 252–276.

Wei, Kai, Rishabh Iyer, and Jeff Bilmes (2015). “Submodularity in Data Subset Selection and Active Learning”. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France: JMLR.org, pp. 1954–1963. URL: http://dl.acm.org/citation.cfm?id=3045118.3045326.

Whittaker, James A. and Michael G. Thomason (Oct. 1994). “A Markov Chain Model for Statistical Software Testing”. In: *IEEE Trans. Softw. Eng.* 20.10, pp. 812–824. ISSN: 0098-5589. URL: http://dx.doi.org/10.1109/32.328991.

Williamson, Simon A., Pradeep Varakantham, Ong Chen Hui, and Debin Gao (2012). “Active Malware Analysis Using Stochastic Games”. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS ’12. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, pp. 29–36. ISBN: 0-9817381-1-7. URL: http://dl.acm.org/citation.cfm?id=2343576.2343580.

Wressnegger, Christian, Guido Schwenk, Daniel Arp, and Konrad Rieck (2013). “A Close Look on N-grams in Intrusion Detection: Anomaly Detection vs. Classification”. In: *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*. AISec ’13. Berlin, Germany: ACM, pp. 67–76. ISBN: 978-1-4503-2488-5. DOI: 10.1145/2517312.2517316. URL: http://doi.acm.org/10.1145/2517312.2517316.

Xu, Haifeng, Rupert Freeman, Vincent Conitzer, Shaddin Dughmi, and Milind Tambe (2016). “Signaling in Bayesian Stackelberg Games”. In: *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’16. Singapore, Singapore: International Foundation for Autonomous Agents and Multiagent Systems, pp. 150–158. ISBN: 978-1-4503-4239-1. URL: http://dl.acm.org/citation.cfm?id=2936924.2936950.

Xu, Jingjing, Xuancheng Ren, Junyang Lin, and Xu Sun (Oct. 2018). “Diversity-Promoting GAN: A Cross-Entropy Based Generative Adversarial Network for Diversified Text Generation”. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, pp. 3940–3949. DOI: 10.18653/v1/D18-1428. URL: https://www.aclweb.org/anthology/D18-1428.

Yaghmour, Karim (2013). *Embedded Android*. O’Reilly Media.

Yang, Chao, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras (2014). “Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I”. In: ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing. Chap. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications, pp. 163–182. ISBN: 978-3-319-11203-9. DOI: 10.1007/978-3-319-11203-9_10. URL: http://dx.doi.org/10.1007/978-3-319-11203-9_10.

Yin, Li, Li-Guo Huang, Xiu-Li Lin, and Yong-Li Wang (July 2018). “Monotonicity, concavity, and inequalities related to the generalized digamma function”. In: *Advances in Difference Equations* 2018.1, p. 246. ISSN: 1687-1847. DOI: 10.1186/s13662-018-1695-7. URL: https://doi.org/10.1186/s13662-018-1695-7.

Yu, Kai, Jinbo Bi, and Volker Tresp (2006). “Active Learning via Transductive Experimental Design”. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML ’06. Pittsburgh, Pennsylvania, USA: ACM, pp. 1081–1088. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143980. URL: http://doi.acm.org/10.1145/1143844.1143980.

Zhang, Li, Vrizlynn L. L. Thing, and Yao Cheng (2019). “A scalable and extensible framework for android malware detection and family attribution”. In: *Computers & Security* 80, pp. 120–133. DOI: `10.1016/j.cose.2018.10.001`. URL: `https://doi.org/10.1016/j.cose.2018.10.001`.

Zhang, Mu, Yue Duan, Heng Yin, and Zhiruo Zhao (2014). “Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, pp. 1105–1116. ISBN: 978-1-4503-2957-6.

Zhu, Jianhan, Jun Hong, and John G. Hughes (2002). “Using Markov Chains for Link Prediction in Adaptive Web Sites”. In: *Proceedings of the First International Conference on Computing in an Imperfect World*. Soft-Ware 2002. Berlin, Heidelberg: Springer-Verlag, pp. 60–73. ISBN: 3-540-43481-X. URL: `http://dl.acm.org/citation.cfm?id=645974.758446`.

Index