The Extended Maurer Model: Bridging Turing-Reducibility and Measure Theory to Jointly Reason about Malware and its Detection

by

Mohamed Elsayed Abdelhameed Elgamal
B.Sc., Benha University, Egypt, 1996
M.Sc., Cairo University, Egypt, 2004

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering,
University of Victoria,
Victoria, BC, Canada

The Extended Maurer Model:
Bridging Turing-Reducibility and Measure Theory
to
Jointly Reason
about
Malware and its Detection

by

Mohamed Elsayed Abdelhameed Elgamal
B.Sc., Benha University, Egypt, 1996
M.Sc., Cairo University, Egypt, 2004

Supervisory Committee

Dr. Stephen W. Neville, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Fayez Gebali, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Issa Traoré, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Jens Weber, Outside Member
(Department of Computer Science)

Supervisory Committee

Dr. Stephen W. Neville, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Fayez Gebali, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Issa Traoré, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Jens Weber, Outside Member
(Department of Computer Science)

ABSTRACT

An arms-race exists between malware authors and system defenders in which defenders develop new detection approaches only to have the malware authors develop new techniques to bypass them. This motivates the need for a formal framework to jointly reason about malware and its detection. This dissertation presents such a formal framework termed the *extended Maurer model* (EMM) and then applies this framework to develop a game-theoretic model of the malware authors versus system defenders confrontation.

To be inclusive of modern computers and networks, the EMM has been developed by extending to the existing Maurer computer model, a Turing-reducible model of computer operations. The basic components of the Maurer model have been extended

to incorporate the necessary structures to enable the modeling of programs, concurrency, multiple processors, and networks. In particular, we show that the proposed EMM remains a Turing equivalent model which is able to model modern computers, computer networks, as well as complex programs such as modern virtual machines and web browsers.

Through the proposed EMM, we provide formalizations for the violations of the standard security policies. Specifically, we provide the definitions of the violations of confidentiality policies, integrity policies, availability policies, and resource usage policies. Additionally, we also propose formal definitions of a number of common malware classes, including viruses, Trojan horses, spyware, bots, and computer worms. We also show that the proposed EMM is complete in terms of its ability to model all implementable that could exist malware within the context of a given defended environment.

We then use the EMM to evaluate and analyze the resilience of a number of common malware detection approaches. We show that static anti-malware signature scanners can be easily evaded by obfuscation, which is consistent with the results of prior experimental work. Additionally, we also use the EMM to formally show that malware authors can avoid detection by dynamic system call sequence detection approaches, which also agrees with recent experimental work. A measure-theoretic model of the EMM is then developed by which the completeness of the EMM with respect to its ability to model all implementable malware detection approaches is shown.

Finally, using the developed EMM, we provide a game-theoretic model of the confrontation of malware authors and system defenders. Using this game model, under game theory's strict dominance solution concept, we show that rational attackers are always required to develop malware that is able to evade the deployed malware de-

tection solutions. Moreover, we show that the attacker and defender adaptations can be modeled as a sequence of iterative games. Hence, the question can be asked as to the conditions required if such a sequence (or arms-race) is to converge towards a defender advantageous end-game. It is shown via the EMM that, in the general context, this desired situation requires that the next attacker adaptation exists as, at least, a computationally hard problem. If this is not the case, then we show via the EMM's measure theory perspective, that the defender is left needing to track statistically non-stationary attack behaviors. Hence, by standard information theory constructs, past attack histories can be shown to be uninformative with respect to the development of the next to be required adaptation of the deployed defenses.

To our knowledge, this is the first work to: (i) provide a joint model of malware and its detection, (ii) provide a model that is complete with respect to all implementable malware and detection approaches, (iii) provide a formal bridge between Turing-reducibility and measure theory, and (iv) thereby, allow game theory's strict dominance solution concept to be applied to formally reason about the requirements if the malware versus anti-malware arms-race is to converge to a defender advantageous end-game.

Contents

List of Figures

List of Symbols

$\mathcal{E}^+$	The set of all possible EMM benign events	108
$D(.)$	A malware detector	108
$f(.)$	Feature mapping of $D(.)$	109
$\mathcal{X}$	The spatial-temporal feature space	109
$d(.)$	The decision boundary of $D(.)$	111
$D^*(.)$	The ideal detector	112
$\mathbb{D}(.)$	A composite detector	113
$F(.)$	The composite feature mapping of $\mathbb{D}(.)$	114
$\boldsymbol{\omega}$	Classes of events	114
$Pr_{\omega_k}(e)$	The probability of an event e	121
Ω	The sample space	116
$\mathcal{P}(\Omega)$	The power set of Ω	116
$\mathcal{B}$	A class of subsets	116
$\mathcal{F}$	A σ-algebra of subsets	116
$\Re^*$	The set of extended real numbers	116
$\Re^*_+$	The set of non-negative extended real numbers	116
μ	A measure	117
$\langle \Omega, \mathcal{F} \rangle$	A measurable space	117
$\langle \Omega, \mathcal{F}, \mu \rangle$	A measure space	118
i^{-1}	The inverse instruction of i	131
G	A game	147
$\mathcal{N}$	The set of players	147
$\boldsymbol{\Sigma}$	The set of strategy sets	147
$\mathcal{U}(.)$	The set of utility functions	147
$u_j(.)$	The utility function of player j	147
$\boldsymbol{a}$	A strategy profile	147

$\boldsymbol{a}^*$	Nash equilibrium strategy profile	147
a_{-j}	A strategy profile for all players except j	148
$\mathcal{T}$	A transformation	151
$\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$	A dynamical system	152
$\boldsymbol{A}$	The attacker	164
$\boldsymbol{\alpha}$	The attacker's set of attacks	165
$V_{\mathcal{A}}$	The attacker's probing process	165
$u_{\mathcal{A}}(.)$	The utility function of $\mathcal{A}$	167
$\Sigma_{\mathcal{A}}$	The strategy set of $\mathcal{A}$	168
$\mathcal{D}$	The system defender	170
$\mathbb{R}$	The set of responses of $\mathcal{D}$	170
$\Sigma_{\mathcal{D}}$	The set of strategies of $\mathcal{D}$	172
$u_{\mathcal{D}}(.)$	The utility function of $\mathcal{D}$	172
$\{G_k\}_{k=0}^{K}$	A game sequence	177

ACKNOWLEDGEMENTS

All praise is to Allah, the Almighty, who enabled and aided me to complete this dissertation.

I would like to thank the my sincere supporter, my family members. I would like to begin by expressing all appreciation and thanking to my parents, my mother and my deceased father, who always supported and encouraged me allover the course of my life. I would also like to thank my lovely wife, Abeer, and my sons, Serag and Yahya, for their love, and patience. Finally, I would like to thank my parents in law, my sister Sherine and her husband Ehab, and my lovely niece Reham. The love and care of all family members helped me to overcome the troubles and difficulties.

Next, I would like to thank my supervisor, Dr. Stephen Neville for his continuous help, valuable suggestions and huge support. I would also like to thank my supervisory committee members: Dr. Fayez Gebali, Dr. Issa Traoré and Dr. Jens Weber for their valuable discussions.

Many thanks go to my sponsors in Egypt: The Electronics Research Institute (ERI), the Egyptian Government and the Egyptian Bureau of Cultural and Educational Affairs in Canada.

At the end, I would like to express my deep gratitude and love to all my relatives, friends and colleagues in Egypt, Victoria, and Edmonton.

DEDICATION

To my deceased father.

Chapter 1

Introduction and Motivation

1.1 Motivations

Cyber attackers (*e.g.*, individuals, organizations, communities, nation-states, *etc.*) use *malicious software* (simply, *malware*) as one of their main tools to attack targeted computer systems. With the large scale connectivity of today's computers, malware attacks have rapidly increased. For example, Trend Micro has announced an increase in the number of online banking malware infections of about 200% during 2013 than 2012 [2], and Symantec has reported an increase in the number of mobile malware families of about 58% during 2012 [3]. Such trends also exist within the mobile devices' domains as they become the dominant in-use computers [4].

To defend against malware, a large number of malware detection approaches have been proposed, such as those of [5–28]. However, using different evasion techniques (*e.g.*, *obfuscation* [29, 30], *mimicry attacks* [31, 32], *etc.*), malware can be developed to evade current detection approaches [29, 33, 34]. Hence, an arms-race exists in which the defenders develop better detection approaches and malware authors develop

evasion techniques to bypass each new generation of deployed defenses. This arms-race involves the interplay between:

(i) Malware development and obfuscation approaches,

(ii) The analysis and evaluation of malware detection approaches, and

(iii) The overall analysis of the confrontation occurring between the attackers and defenders.

A more detailed discussion of these issues is as follows.

1.1.1 Malware Development Approaches

In the past, the process of developing malware required the deep understanding of both computer assembly language and the intricate working nature of the targeted computer system. However, creating malware is no longer limited to the technically elite as the advent of user-friendly malware developing toolkits has made it possible for lower skilled malware authors with trivial skills to develop novel malware variants by following simple step-by-step process [29, 35, 36]. For example, the *Anna Kournikova* virus author was able to create a world-wide attack infecting hundreds of thousands of systems using just such a toolkit [37]. By using these toolkits, attackers can easily obfuscate malcode and generate large numbers of novel variants structured to evade commercial anti-malware products [29, 33].

1.1.2 Assessing Malware Detection Approaches

Frameworks for the evaluation and analysis of malware detection approaches in order to assess their capabilities and are therefore required. In general, the analysis

and evaluation of malware detection approaches can be done through two main apptoaches: (i) *experimental evaluations*, and (ii) *formal models*.

Experimental evaluation has been the principle approach for the evaluation of malware detection approaches (*e.g.*, [1,29,33,34,38–51]). Various data sets that differ in many aspects (such as, the size of the malware test subset, the size of the benign test subset, *etc.*) have been utilized. This has led to issues in that, in some cases, the reported results were due to artifacts in the used data sets [52, 53]. For example, in [52], Tan *et al.* showed that the recommendation for the system call sequence in the stide anomaly IDS to be of length 6 is due to an artifact in the evaluation data set, whereas in [54], McHugh discussed the artifacts existing in the 1999 evaluation data set proposed by Lincoln Laboratory group for the experimental evaluation of IDSes. In other cases, some approaches have been evaluated using privately held data sets of anti-virus companies [43, 46, 51]. Hence, the reported results typically cannot be independently verified. It can be argued that, to avoid these problems, reference data sets should be created and regularly updated with the newly detected variants. A counter argument though can be made that malware writers will study the characteristics of such reference sets and then seek to design their subsequent malware to deviate from those in these sets (*i.e.*, to bypass detection).

To avoid the limitations of experimental evaluations, formal models can be used to analytically evaluate the capabilities of malware detectors independently of any particular data set. In general, a number of formal models have been proposed either to model malware [55–61]. Or to analyze different aspects of malware and intrusion detection systems [13, 39, 62–64]. Currently, the core limitations of these models are:

(i) **A lack of generality:** Existing intrusion detection models form the main attack detection models and, as such, are not generic models as they have been developed specifically to achieve prescribed modeling objectives. Additionally,

in general, they cannot be used to model malware as they have not been designed for this purpose.

(ii) **Limited expressive capabilities:** Existing models have been developed using standard traditional models of computations (Turing machines, recursive functions, *etc.*) which have been shown to be limited in modeling important aspects of modern malware such as: interactions, concurrency, and non-termination [65,66]. Recent process calculi models (*join*-calculus and κ-calculus) show more expressive capabilities [65,67]. However, they focus on malware modeling and not the modeling of malware detection solutions.

(iii) **A lack of measurable constructs**: Generally, malware detection involves the assessment of measurable information obtained from observing running systems. Current modeling approaches have largely not sought to provide formal models that are inclusive of such measurable sets of run-time information.

1.1.3 Analyzing Attackers-Defenders Confrontations

Arguably, a better understanding of the nature of the attackers versus defenders confrontations could potentially enable the development of more effective anti-malware defenses. Also, analyzing the nature of the attacker's adaptations, strategies, and decisions could potentially enable the development of better countermeasures. Game theory provides a powerful mathematical framework to reason about multi-person competitive decision-making scenarios. Hence, it can be used to formally analyze such confrontations. Particularly, game theory is an effective framework to formally analyze the interactions of rational adversarial decision makers, such as, attackers versus defenders. There exist a number of prior game-theoretic analyses of attackers versus defenders confrontations, such as those of [68–71]. However, these models tend

to focus on analyzing specific system configurations or a certain described attack scenarios (*i.e.*, specific games). Hence, game theory does not appear to have been used to analyze the wider question of when and if a given arms-race is likely to become defender winnable.

1.2 Problem Statement

As discussed in [72], the analysis of malware detection approaches remains an open research area. This dissertation extends the work in this area by proposing a joint analysis framework for reasoning about malware and its detection. In particular, to avoid the limitations of prior experimental based works, the proposed framework is based on formal models, where as per Gordon *et al.* in [73], there is a recognized lack of formal models to evaluate malware detection approaches. The proposed formal framework seeks to avoid the limitations of existing models by providing:

(i) A generic framework that is complete with respect to its ability to model all implementable malware as well as implementable malware detection approaches.

(ii) An *information-centric* model, as detectors must assess measurable information changes within running computers, where the execution of malware generates these changes of interest in the system (or more generally, defended environment).

(iii) A comprehensive framework that is capable of modeling modern computers and networks inclusive of issues such that: concurrency, multicore processors, modern virtual machines (VMs) and browsers, interpreted languages, *etc.*

To develop this framework, an expressive model that is capable of modeling the information changes within modern computers has been developed. More particularly,

the *Maurer model* [74, 75] is used as the basis for this work. The Maurer model is a Turing reducible (or equivalent) model that has the advantage of simplicity and its close resemblance to the functions of modern computers [76]. Moreover, as a set-function model of how instructions executions enact changes to the memory, the Maurer model provides a natural bridge to the information-centric model required to address (ii) above. However, the Maurer model is a basic model in that it does not have the key components that are necessary to represent modern computers, such as, programs, security policies, concurrency, *etc.* In this dissertation, the Maurer model is extended to incorporate these required key structures. The developed model is termed *the extended Maurer model* (EMM) and has the following key features:

- It is able to represent various aspects of modern computer systems, such as programs, multiprocessors, concurrency, the information flows onto and off computer systems, *etc.* Hence, it is able to capture the nature of today's modern complex computing environments.

- It is generic in the sense of its ability to model programs and their executions. Hence, it can model various categories of malware and malware detection systems. Moreover, it is complete in the sense of being able to model all implementable malware and malware detection solutions.

- It clearly defines the various aspects of security in terms of security policy violations where what constitutes malware is defined in terms of violations of these standard security policy definitions.

- As will be shown, it also models a σ-finite information space for formally describing all the operational information that is available about the run-time defended environment that is modeled.

Hence, as will be shown, the EMM allows the analysis of the strategic confrontation of attackers and system defenders to be undertaken. A game-theoretic model is developed to provide a better understanding of the nature of such confrontations. The analysis focuses on the evolution of the confrontation over the time to determine the potential factors that underlie its dynamics in terms of what is required for this arms-race to converge towards a defender winnable (or advantageous) end-game.

1.3 Contributions

The contributions of this dissertation can be summarized as follows:

(1) *Developing the EMM, a generic Turing equivalent formal framework as an augmented version of Maurer's existing computer model (Chapter 3)*: The developed EMM will be shown to be comprehensive in its ability to model modern computers and computer networks. It will also be shown to be able to model complex modern programs, such as virtual machines and web browsers as well as modern computer networks.

(2) *Formalizing the violations of basic security policies as well as the definitions of a number of common malware classes* (Chapter 4): The formal definitions of standard security policies associated with confidentiality, integrity, and availability violation, as well as resource authorization violations will be developed. The EMM will be shown to be inclusive of providing formal definitions for a number of common malware classes. Additionally, the EMM will be shown to be complete in terms of being able to model all implementable malware.

(3) *Evaluating a number of common malware detection approaches* (Chapter 5): The EMM will also be shown to describe a σ-finite measure space. The EMM will be shown to formally model common malware detection approaches. Moreover,

the EMM will be shown to be complete in the sense of being able to model any implementable malware detection approach or composition of approaches.

(4) *Formalizing a game-theoretic model of the confrontation between attackers and system defenders* (Chapter 6): An EMM-based game-theoretic model of the confrontation between attackers and system defenders will be formulated. The model is then used to explore the evolution of this arms-race over time (*i.e.*, as an iterative sequence of games). The analysis of the sequence of games then shows that either the defender must be able to prove that the attackers' next adaptation exists as, at least, a computationally hard problem, or the defender is faced with the problem of needing to track non-stationary attack behaviors (*i.e.*, past information is no longer informative with respect to the problem of how the deployed defenses must be modified or re-tuned).

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces the related work in formal models for malware modeling and the analysis of malware and attack detection approaches. It also provides an overview of the basic components of Maurer model as presented in [74, 75].

Chapter 3 discusses the extensions required to enable Maurer model to model modern malware and malware detection approaches and develops the proposed EMM. In particular, it discusses the evolution of the EMM's memory with time. It also defines the concept of programs, their execution traces, concurrency, and various information sets related to these issues. It formalizes the definition of the set of security policies within the EMM. Finally, Chapter 3 discusses the Turing equivalence of the EMM and discusses the modeling of the computers, computer networks, and

complex programs (*e.g.*, virtual machines and modern browsers) within the proposed model.

Chapter 4 shows the application of the EMM to the modeling of standard security policy violations. It also provides the formal EMM-based definitions for a number of common malware classes. Finally, Chapter 4 shows that the proposed EMM is complete in the sense that it is able to model the execution of all implementable malware within a defined defended environment.

Chapter 5 discusses the application of the EMM to the analysis of malware detection approaches. It provides a formal model for the EMM as a σ-finite measure space, inclusive of the discussion as to why this is a critically important aspect of the models development. It introduces the analysis of a number of static and dynamic malware analysis approaches. Finally, Chapter 5 applies the a measure-theoretic model of the EMM to show that the EMM is complete in the sense that it can model all implementable detection approaches.

Chapter 6 applies the developed EMM to produce a game-theoretic model of the on-going confrontation between the attackers and the system defenders. Game theory's strict dominance solution concept is then applied to show that rational attackers are always formally motivated to develop malware structured to bypass current system defenses. This leads to the overall arms-race being defined in terms of a time evolving sequence of games. This sequence is then analyzed to determine the conditions required if it is to converge to a defender advantageous end-game. The implications of this analysis are to show that either: (i) the defender must formally show that the attackers' next adaptation is, at least, computationally hard to achieve, or (ii) the defender must face the problem of needing to track non-stationary attack behaviors (*i.e.*, past attack information is no longer informative with respect to understanding the next attack).

Finally, Chapter 7 summarizes the contributions of the dissertation and suggests potential directions for future work.

1.5 Summary

In this chapter, the motivations of this dissertation have been discussed. Also, the problem statement has been defined. Additionally, the contributions of this dissertation to the field of computer and information security have been outlined. Finally, the dissertation organization has been previewed.

Chapter 2

Related Work

2.1 Introduction

As discussed in Chapter 1, there is a need to develop a formal framework for the joint analysis of malware and its detection. This chapter provides the literature review of prior approaches in these domains.

In general, formal modeling has been used to model different aspects of computer security, such as viruses[1] and other forms of malware (*e.g.*, [13, 39, 55, 57, 60, 62, 64, 77–79]), various aspects of intrusion detection systems (*e.g.*, [63, 80–89]), and other areas of security (*e.g.*, *access control models* [90]). Since this dissertation is concerned mainly with malware, the primary focus is on existing formal malware and malware detection models. The limitations of these existing models will be highlighted, motivating the development of the new EMM model based on the Maurer computer. This chapter also previews the basic Maurer model constructs and how these need to be extended to model modern malware and its detection solutions.

The remainder of this chapter is organized as follows. Section 2.2 reviews the existing formal models for both malware modeling and the analysis and evaluation

[1] Viruses are defined and discussed in more details in Section 4.2.1.

of malware detection systems. The section also highlights the limitations of existing models thereby motivating the development of the proposed EMM. Section 2.3 provides a detailed overview of the basic building blocks of Maurer model. Section 2.4 discusses the limitations of the basic Maurer model and the nature of the extensions required to enable it to provide a comprehensive model for modern computers and networks. Finally, Section 2.5 summarizes the chapter.

2.2 Existing Formal Models

This section previews a number of existing formal model in malware modeling and malware detection modeling and highlights their limitations. In particular, in Section 2.2.1, existing malware modeling frameworks will be discussed, whereas in Section 2.2.2, existing frameworks for the modeling of malware and attack detection approaches will be discussed. In Section 2.2.3, the limitations of existing formal models will be highlighted. Finally, in Section 2.2.4, the use of Maurer model will be motivated.

2.2.1 Malware Modeling Frameworks

Self-replication is a core aspect in computer virology since it characterizes viruses and worms. In general, as discussed in [91, Section 2.3, pp. 19], self-replication was first discussed by von Neumann in 1948 with the introduction of the theory of cellular automata to study the biological evolution. In the mid 1980s, Cohen developed the first formal model for computer viruses [55, 56]. In Cohen's framework, computers were modeled as Turing Machines (TMs) [92–94], and viruses were modeled as sequences of symbols on the machines' tapes. In particular, Cohen's formalization of computer viruses is introduced in Definition 2.1 as follows.

Definition 2.1 (Cohen's Definition of Computer Viruses)**.** *Let M be a TM and let V be a non-empty set of programs for M which is denoted as the viral set. Then, each $v \in V$ is a sequence of symbols that defines a computer virus and satisfies the following condition: if v exists on the machine's tape at a time instant t, then there should exist a time $t' > t$ and another sequence $v' \in V$ such that v' exists on the machine's tape at t'.*

A major conclusion of Cohen's work was proving that it is undecidable (without execution) as to whether or not a given sequence is in the viral set [55].

Cohen's use of TMs to model viruses was criticized by a number of researchers. In particular, Kauranen *et al.* pointed out that the primary shortcoming of Cohen's model is that traditional TMs do not specify entities corresponding to *programs* [58]. Hence, Kauranen *et al.* suggested the use of *universal Turing machines* (UTMs) to model computers, and accordingly, viruses are considered TMs which write copies of themselves somewhere to the UTMs' tapes. In [59], Jacob *et al.* presented malware model using *interaction machines* (IMs), which are TMs with an added dynamic input/output actions [61]. Jacob *et al.* provided formal definitions of computer viruses as well as formal definitions of interactive and distributed viruses. Finally, Jacob *et al.* proposed an operational malware modeling framework based on interactive languages [59].

In 1988, Adleman used *recursive functions*[2] to model viruses [57]. In this model, Adleman paid attention to the identification and classification of the different categories of viruses with respect to their destructive power. In particular, a virus is defined as a *total recursive*[3] function v that applies to all programs p so that $v(p)$ exhibits viral behaviors such as *injury* (*i.e.*, damage the system by executing its ma-

[2]See [91, Chapter 2] for an introduction to recursive functions.

[3]A function $f(.)$ is called a *total* function if it is defined for all possible input values, and $f(.)$ is a *recursive* function if there is a Turing machine T that computes $f(.)$ [92–94].

licious payload), *infection* (*i.e.*, replicate and infect other programs) and *imitation* (*i.e.*, imitate the host program with no replication or injury). The main advantage of Adleman's model is that it is based on the abstract computability theory allowing the developed definitions to be independent of any specific computational model.

In 2004, Zuo *et al.* extended Adleman's model of computer viruses to include new aspects such as *mutation* and *stealth* [78]. A number of malware modeling frameworks followed the work of Cohen and Adleman that were also based on different types of mathematical machines and automata (*e.g.*, Turing machines, sequential machines, pushdown automata, *etc.*) [13, 39, 59, 62, 64, 77–79].

In 1999, Thimbleby *et al.* introduced a framework for modeling Trojans[4] and computer virus infections [60]. In particular, in this model, the computer is considered to be an array of bits (*e.g.*, RAM, screens, backing store, *etc.*). The exact meaning of bit patterns depend on their location within the array. An instance of this finite array is called a *representation*, and the collection of all possible representations is denoted as R. The users of computers are not concerned with representations, they are instead concerned with the names of the programs. Programs may run and accordingly change the state of the computer. The *meaning* of a program is defined in terms of what the program does when it runs. By applying this model, Thimbleby *et al.* introduced a formal definition for both Trojans and viruses. However, Thimbleby *et al.* did not seek to model other malware categories. In addition, Thimbleby *et al.* did not seek to show the application of their model to address the problem of malware detection.

However, as indicated in [65], the increasing sophistication of recently emerging malware generally reduces the comprehensiveness of these prior models. In particular, complex malware, such as K-ary malicious codes [95] and multiprocess malware [96], generally cannot be formulated within these prior models [65].

[4]Trojans are defined and discussed in more details in Section 4.2.2.

Similarly, as discussed in [60], Thimbleby *et al.* showed the inadequacy of traditional TM models to represent viruses. Specifically, some of the core issues highlighted by Thimbleby *et al.* are:

- Traditional TMs are infinite whereas computers are finite (in terms of memory, processing or other resources), and viruses exist on systems with finite resources.
- Viruses have to enter the system in order to infect it, and this requires the modeling of the interaction within the computer systems. As illustrated in [97], traditional TM models are not sufficiently expressive for systems that interact.
- Viruses are programs that, in addition to having the ability to infect other programs, also have Trojan activities. Hence, the traditional TM equivalent models are insufficient to capture these important details of viruses' behaviors as they do not represent the flow of information onto and off the computers.
- To model infection or replication of programs, the model needs to identify the notion of '*other*' programs. This cannot be modeled within traditional TM as they generally lack the notion of programs.

Additionally, in [75, Section 10], Maurer indicated that the available mathematical machines are not adequate models for modeling modern computers as the majority of these models are either not general enough or they are too general. Moreover, modern computers have several important common features that are not supported in these prior models. For example, the instructions of modern computers have input and output regions [75, Section 2] and the study of these input and output regions can provide more powerful modeling capabilities.

As the Maurer model was designed to address many of the above issues, it was selected as the platform model from which to develop the EMM. Moreover, as the

Maurer model focus on modeling state changes within stored memory, it provides a natural bridge into developing an information-centric model. The Maurer model is formally defined and discussed in detail in Section 2.3.

2.2.2 Malware and Attack Detection Modeling Frameworks

Formal modeling has also been used to model detection systems. In particular, as discussed in [13,39], Christodorescu *et al.* introduced a formalization for semantics-based malware detection which modeled a wide variety of the obfuscation transformations used to develop malware variants. However, the application of this framework was limited to static analysis of obfuscated malware and, hence, this model is not suited to model or analyze dynamic malware detection approaches.

In [62], Filiol *et al.* introduced a statistical testing model of anti-virus detection. Filiol *et al.* were able to reason about anti-virus scanners and presented a statistical variant of Cohen's undecidability result [55]. However, Filiol *et al.* did not discuss how their model could be used to: (1) analyze the potential resiliency of malware detection approaches, or (2) comprehensively model all malware classes.

In [64], Jacob *et al.* developed a formal model for the behavioral detection of malware using *context free grammar* [93]. Additionally, Jacob *et al.* also developed malware detection approach based on their proposed framework. However, the framework is specific to model behavioral-based detection approaches and cannot be used to model static analysis approaches. Also, the developed detection approach yielded low detection rate of only 51% of PE malware which suggested limitations in the applied model [64].

In [84], Gu *et al.* presented an information-theoretic formal framework for analyzing and quantifying the effectiveness of intrusion detection systems. Gu *et al.* started with formally defining a model for IDS, then they analyzed the model via an

information theoretic approach. Additionally, Gu *et al.* proposed a set of information-theoretic metrics to quantitatively measure the effectiveness of an IDS in terms of its feature representation capability, classification information loss, and overall intrusion detection capability. This model though is quite specific to the intrusion detection domain and, as such, cannot be generally applied to model malware. Moreover, the model did not provide structures for measurable information sets.

2.2.3 Limitations of Existing Formal Models

In general, the following observations about the existing formal malware modeling frameworks can be highlighted:

- Most existing formal malware models were developed based on traditional mathematical machines which, as discussed in Section 2.2.1, have a limited ability to capture the functional features of modern malware, such as its interactions with the environment, concurrency, *etc.*

- Generally, these models tend to target specific malware classes and, hence, have not been shown to be comprehensive (or complete), where this is becoming more critical as modern malware instances concurrently incorporate a multiplicity of attack methodologies.

- Existing models were not designed to concurrently address the analysis and evaluation of both malware and its detection solutions.

Whereas, the following insights about the existing formal models for malware detection systems can also be highlighted:

- Formal malware detection models have not been developed to concurrently model malware.

- In general, the focus has been on modeling IDS-style detection approaches and, hence, these approaches have not been shown to be comprehensive of other detection solutions.
- Moreover, typically only specific aspects of IDS systems have been modeled and not even the complete IDS process.
- These detection models have not been shown to map into the measure theory constructs that underlie, for example, probability and statistics theory.

2.2.4 Discussion

As illustrated in the previous section, prior models have not been developed to concurrently address the modeling of malware and its detection approaches. Additionally, the use of traditional mathematical machines to develop these models limits their ability to model modern program and computer constructs. Finally, since malware detection is based on assessing measurable information extracted from the systems, the lack of measurable information constructs prevents these models from providing comprehensive malware detection models. Hence, a comprehensive formal framework to model malware and analyze malware detection approaches is required. Moreover, it should provide additional insights about the operation of different malware classes and variants and how effective proposed or existing detection approaches may be against these classes and variants. The development of such a framework is the objective of this dissertation.

To achieve this objective, the Maurer model [75] has been selected as the base platform from which this framework will be developed for the following reasons. First, the Maurer model is a Turing equivalent model [76] and, hence, it can be used as a general model of computation. Second, it has the advantage of being closer to real computers

than prior traditional mathematical machine based models [98]. Fundamentally, the Maurer model focuses on how a computer's stored memory is changed over time by the execution of instructions. The Maurer model defines instruction executions as the mechanism by which these changes to memory contents occur (*i.e.*, changes to the memory's state). As this dissertation will show, the Maurer model, therefore, provides an information-centric view of the computer's operation where the model's memory defines the information that exists within the computer at any time instant, with the model's instruction set defining how possible changes to this information can occur. Hence, the Maurer model provides a natural bridge from Turing-equivalency into the well developed mathematics of measure theory. Section 2.3 reviews the Maurer model while Chapter 3 details the necessary extensions to the basic Maurer model that are required to enable it to model modern computers and computer networks (*i.e.*, IT environments).

2.3 Maurer model

In [75], Maurer reintroduced a revised version of his original computer model published in [74]. For the completeness of this dissertation, the core modules of Maurer model will be introduced in this section. For the complete details of Maurer model, we refer the reader to [75]. Note that, throughout this dissertation, we will use the terms *Maurer model* and *Maurer computer* exchangeably to denote Maurer definition of computers as introduced in [75].

The remainder of this section is organized as follows. Section 2.3.1 introduces the formal definition of computers as defined by Maurer. Section 2.3.2 then discusses the input and output regions of instructions. Section 2.3.3 defines the concepts of affected and affecting regions. Section 2.3.4 introduces the composition and decom-

positions of instructions. Section 2.3.5 discusses the existence of instructions. Finally, Section 2.3.6 discusses the Maurer model with a control unit.

2.3.1 Maurer Computer

Without loss of generality, the Maurer model models computers in terms of the effects of their instruction executions [75]. The motivation behind developing the model was the perceived inadequacy of the existing mathematical machines to model emerging computer architectures. Fundamentally, the Maurer model focuses on modeling how the information stored in the computer's memory changes over time as a result of the execution of the computer's instructions.

Maurer model begins with the computer's memory, which is represented as a finite set of memory elements that is denoted as M and defined as,

$$M = \{m_k \mid k = 1, 2, \ldots, N_M\}, \tag{2.1}$$

where each memory element m_k is disjoint with any other element (*i.e.*, $\forall k \neq k', m_k \cap m_{k'} = \emptyset$) and N_M is the finite number of memory elements. Importantly, this set denotes the union of all components of the computer that can hold (or store) information (*i.e.*, RAM, CPU registers, hard drives, disk drives, hard coded memory, *etc.*) and not just the computer's main memory.

The possible contents of each memory element is determined by the *base space*, which is denoted as B. In particular, Maurer defined B as the set of values that each memory element can have (*e.g.*, the *bit* is the standard memory element for modern digital computers and its value being either 0 or 1, or alternatively, the *byte* (8 bits) can be considered as the memory element, and hence, its value ranges from 0 to 255). Intuitively, if B has only one element then the memory will have only one fixed state (because all memory elements will be assigned this single value of B).

Nominally, under the model, B should contain at least 2 elements, then $|B| \geq 2$, where $|\cdot|$ denotes the set cardinality. Since digital computers are the focus of this dissertation, then it will be assumed that $|B| = 2$ and $B = \{0, 1\}$.

A *state*, s, of the memory of the computer is defined as an arbitrary map from M into B. Formally,

$$s : M \to B. \tag{2.2}$$

The *finite* set of all possible states of the computer memory M is denoted as $\mathbb{S}$. Finally, Maurer defined an *instruction*, i, as the method of changing from one state to another. Formally, an instruction is defined as a map,

$$i : \mathbb{S} \to \mathbb{S}. \tag{2.3}$$

The set of all instructions of the computer is denoted as $\mathbb{I}$. It should be noted that the use of the term "instructions" in the Maurer model differs from its use within standard programming languages in that the instructions in the Maurer model denote mappings from memory states to new memory states. A more detailed discussion about the semantics of the instructions is provided in Section 2.3.2.1 as this requires the introduction of the concepts of the *input* and *output* regions of instructions. The definition of Maurer computer is formalized in Definition 2.2 as follows.

Definition 2.2 (Maurer Computer, $\mathcal{M}$)**.** *A Maurer computer is denoted as $\mathcal{M}$ and is defined as the tuple $\mathcal{M} = \langle M, B, \mathbb{S}, \mathbb{I} \rangle$ where:*

- *M is a finite set representing the computer's memory,*

- *B is the base set, where generally $|B| \geq 2$,*

- $\mathbb{S}$ *is the set of all possible maps,* $s : M \to B$*, representing the set of all possible states of the memory, and,*

- $\mathbb{I}$ *is the set of all instructions* $i : \mathbb{S} \to \mathbb{S}$ *of the computer*

that satisfies the following two axioms, where $s_j(m)$ *denotes the content of memory element* $m \in M$ *when* M *is in state* s_j*:*

- ***Axiom 1****: (Any recombination of states is a state)*

  If $s_1, s_2 \in \mathbb{S}, M' \subseteq M$ *and* $s_3 : M \to B$ *such that,* $s_3(m) = s_1(m)$ *if* $m \in M'$ *and* $s_3(m) = s_2(m)$ *if* $m \notin M'$*, then* $s_3 \in \mathbb{S}$*;*

- ***Axiom 2****: (Any two states differ only in a finite way)*

  if $s_1, s_2 \in \mathbb{S}$*, then the set* $\{m \in M \mid s_1(m) \neq s_2(m)\}$ *is finite.*

Given $M' \subseteq M$ and $s \in \mathbb{S}$, then "the content of M' during state s" is denoted as $s|M'$. If $M', M'' \subseteq M$ and $s, s' \in \mathbb{S}$, Maurer introduced the following elementary facts about $s|M'$:

1. $\forall m \in M',\ s(m) = s'(m)\ if\ and\ only\ if\ s|M' = s'|M'$.

2. $if\ s|M' = s'|M',\ M'' \subseteq M' \Rightarrow s|M'' = s'|M''$.

3. $if\ s|M' = s'|M'$, then $s|M' \cap M'' = s'|M' \cap M''$ (since $M' \cap M'' \subseteq M'$).

4. $if\ M' = \phi$, then $s|M' = s'|M'$ is always true.

In some cases, the memory M of Maurer computer can be restructured. In particular, Definition 2.3 defines the *memory structure* for Maurer computer as follows.

Definition 2.3. *Let* $\mathcal{M} = \langle M, B, \mathbb{S}, \mathbb{I} \rangle$ *be a Maurer computer and let* $\mathbb{P}$ *be a partition of* M *(*i.e.*, a class of disjoint non-empty subsets of* M *whose union is* M*). Then,* $\mathbb{P}$ *is denoted as a* memory structure *for* $\mathcal{M}$*.*

As discussed in [75, Section 6], such memory restructuring is not a physical process, rather, it is a logical reorientation of the memory view. As will be demonstrated later in Section 3.3, the extension of the memory in the EMM will be based on Definition 2.3.

2.3.2 Input and Output Regions of Instructions

In [75], Maurer introduced the notion of the *input* and *output regions* of an instruction i, which are denoted as $IR(i) \subseteq M$ and $OR(i) \subseteq M$, respectively. In particular, for $s_2 = i(s_1)$ (*i.e.*, s_2 is the state of the computer resulting from the execution of instruction i when the system state is s_1), $OR(i) \subseteq M$ is defined as: *the set of all elements of M that can be changed due to the execution of i* (*i.e.*, the set of all memory elements whose contents before the execution of i are not the same after its execution). Whereas, $IR(i) \subseteq M$ is defined as: *the set of all elements of M that affect $OR(i)$*. The formalizations of $IR(i)$ and $OR(i)$ are presented in Definition 2.4 as follows.

Definition 2.4 (Input and Output Regions of Instructions)**.** *Let $\mathcal{M}$ be a Maurer computer and let $i \in \mathbb{I}$. For $x \in M$, let $s(x)$ be the content of a memory element x at state s and let $i(s(x))$ be its content after executing the instruction i. Then the input region of i, $IR(i) \subseteq M$, and the output region of i, $OR(i) \subseteq M$, are defined as:*

- $OR(i) = \{x \in M : \exists s \in \mathbb{S} \text{ such that } s(x) \neq i(s(x))\}$
- $IR(i) = \{x \in M : \exists s_1, s_2 \in \mathbb{S} \text{ and } y \in OR(i) \text{ such that } s_1(z) = s_2(z) \text{ for all } z \neq x, \text{ and } i(s_1(y)) \neq i(s_2(y))\}$

As indicated in [75, Section 2], defining $IR(i)$ in terms of $OR(i)$ cannot be avoided. Figure 2.1 shows an example of the input and output regions of an instruction i. As

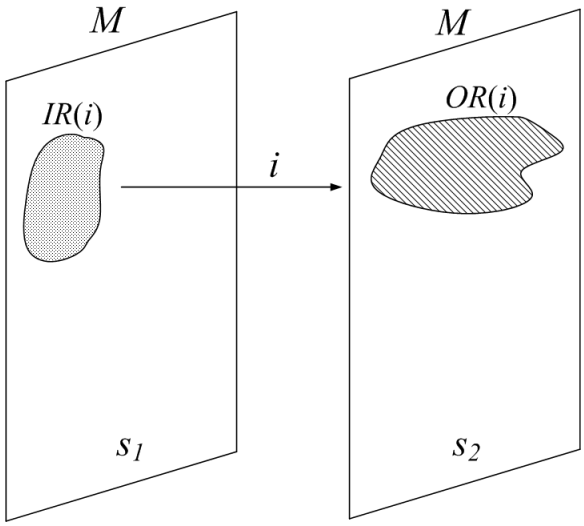


Figure 2.1: Input and output regions of instructions.

shown in the figure, the state of the memory changes from s_1 to s_2 due to the execution of i with $IR(i)$ and $OR(i)$ as indicated.

Maurer also covered the case in which the instruction does not have any input or output regions by the introduction of the *identity instruction*, which will be denoted as i_{id}. The formalization of i_{id} is discussed in Definition 2.5 as follows.

Definition 2.5 (The Identity Instruction, i_{id})**.** *The identity instruction is denoted as* $i_{id} \in \mathbb{I}$ *and has the following properties:*

- $IR(i_{id}) = \phi$, *and*
- $OR(i_{id}) = \phi$.

In fact, the identity instruction is what is commonly known as the *no-operation* or *no-op* instruction, which performs no change to the system. Maurer also introduced Theorem 2.1 for i_{id} as follows.

Theorem 2.1 (A Theorem for i_{id})**.** *For* $i \in \mathbb{I}$, *if* $OR(i) = \phi$, *then* $IR(i) = \phi$, *and* i *is the identity instruction,* i_{id}.

Proof. The proof can be found in [75, Theorem 2.2]. □

Hence, the following corollary can easily be proved.

Corollary 2.1. *$\forall i \in \mathbb{I}, i \neq i_{id}$ we have $OR(i) \neq \emptyset$.*

Proof. It follows directly from Theorem 2.1 that if $OR(i) \neq \emptyset$ then $i \neq i_{id}$. □

2.3.2.1 Discussion

In this subsection, the instructions will be discussed. Without loss of generality, the instructions in Maurer model differ in their nature from those in programming languages. In particular, Maurer model's instructions are defined as general mappings from memory states to other memory states. These mappings are uniquely characterized by both their input and output regions. In particular, any change in these regions means different instructions in the sense of Maurer model as discussed in the following example.

Example 2.1. *Consider the following instructions i_1, i_2, and i_3 with their input and output regions being as indicated in the table.*

Instruction			$IR(.)$	$OR(.)$
i_1:	MOV	R_1, R_2	R_1	R_2
i_2:	MOV	R_1, R_3	R_1	R_3
i_3:	MOV	R_3, R_2	R_3	R_2

In most (unless all) programming languages, the above instructions correspond to the same instruction MOV but with different operands. In Maurer model, since these instructions have different input and/or output memory regions as indicated in the above table, they denote three distinct *Maurer model instructions.* □

As discussed above, Maurer instructions are defined in terms of the input to output memory mappings they produce and, hence, by the underlying mathematical necessities they must be defined in terms of the memory locations where these mappings occur. Hence, standard assembly language mnemonics and, even, higher-level language constructs can be related to classes of composite sets of Maurer instruction executions (*i.e.*, Maurer instructions are loosely analogous to the μ-code instructions that occur within CPU cores, albeit while retaining their memory location dependence). Such instruction compositions are included within the Maurer computer and their details will be discussed in Section 2.3.4. The differences and distinctions between Maurer's use of both the terms "memory" and "instruction" and more standard usages of these terms must be clearly appreciated if the nature of the Maurer model is to be correctly understood. Within the remainder of this work, the terms "memory" and "instruction" solely refer to Maurer's definitions of these terms.

2.3.3 Affected and Affecting Regions

According to Definition 2.4, $OR(i)$ is the set of all memory elements that are affected by the execution of i and $IR(i)$ is defined as the set of all memory elements affect $OR(i)$. This leads to several questions such as: given a specific subset $M' \subseteq IR(i)$, what is the exact output subset region that is affected by M'? Or, given a specific subset $N \subseteq OR(i)$, what is the exact input subset region that affects N? To address these questions, Maurer introduced two substructures in $IR(i)$ and $OR(i)$ referred to as the *affected regions* and the *affecting regions* [75, Definition 7.1]. As shown in Figure 2.2(a), for a region $M' \subseteq IR(i)$, the subset region of $OR(i)$ that is affected by M' under the execution of i is denoted as $AR(M', i) \subseteq OR(i)$. Any change in the contents of M' will affect the contents of $AR(M', i)$ when i is executed. Whereas, as shown in Figure 2.2(b), for a region $N \subseteq OR(i)$, the region in $IR(i)$ that affects N

under i is defined as the *affecting region*, denoted as $RA(N, i) \subseteq IR(i)$. The change in the contents of $RA(N, i)$ will affect the contents of N under the execution of i. The formalizations of the affected and affecting regions are presented in Definition 2.6 as follows.

Definition 2.6 (Affected and Affecting Regions)**.** *Let $\mathcal{M} = \langle M, B, \mathbb{S}, \mathbb{I} \rangle$ be a Maurer computer, and let $i \in \mathbb{I}$, then for a subset $M' \subseteq IR(i)$ and for a subset $N \subseteq OR(i)$:*

- $AR(M', i) = \{x \in OR(i) : \exists s_1, s_2 \in \mathbb{S}$ such that
  $\forall z \in IR(i) \backslash M' \;\; s_1(z) = s_2(z)$ and $i(s_1)(x) \neq i(s_2)(x)\}$
- $RA(N, i) = \{x \in IR(i) : AR(\{x\}, i) \cap N \neq \phi\}$

Where "\" denotes the set difference operation. Maurer also introduced the following three lemmas about the affected and affecting regions.

Lemma 2.1. *Every non-empty subset of $IR(i)$ affects some non-empty subset of $OR(i)$.*

Proof. The proof can be found in [75, Lemma 7.2]. □

Lemma 2.2. *$AR(\emptyset, i) = \emptyset$.*

Proof. The proof can be found in [75, Lemma 7.3]. □

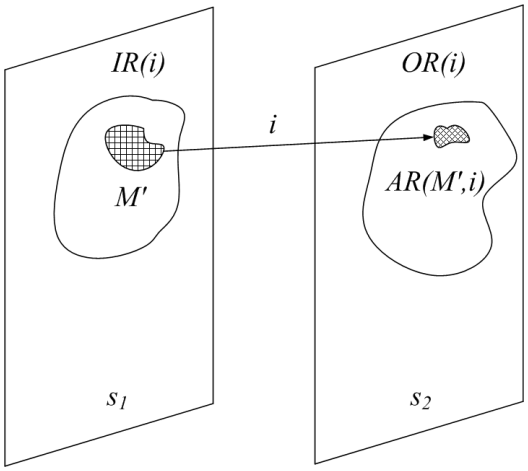
Lemma 2.3. *$RA(\emptyset, i) = \emptyset$.*

Proof. The proof can be found in [75, Lemma 7.4]. □

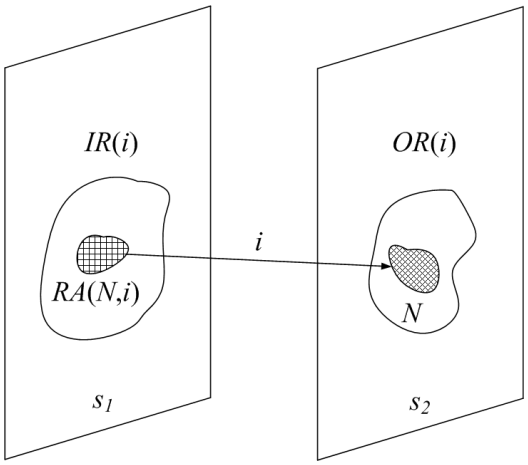
2.3.4 Composition and Decomposition of Instructions

The *composition* of two instructions is also defined in [75]. In particular, if $i_1, i_2 \in \mathbb{I}$ are two instructions on a Maurer computer, then:

$$J = i_1 \circ i_2, \tag{2.4}$$



(a) Affected region $AR(M', i)$ of $M' \subseteq IR(i)$.



(b) Affecting region $RA(N, i)$ of $N \subseteq OR(i)$.

Figure 2.2: Affected and affecting regions relative to the execution of instruction i.

denotes the execution of i_1 followed by the execution of i_2. J can be also expressed as $J = i_2(i_1(s))$ where s is the initial state of the system before executing the two instructions. The composite instruction J also defines a map from $\mathbb{S}$ into $\mathbb{S}$ with its input and output regions are defined in Theorem 2.2 as follows.

Theorem 2.2. *Let $\mathcal{M} = \langle M, B, \mathbb{S}, \mathbb{I} \rangle$ be a Maurer computer, and let $i_1, i_2 \in \mathbb{I}$ be two instructions. Let $J : \mathbb{S} \to \mathbb{S}$ be defined by $J(s) = i_2(i_1(s))$, then:*

1. *$OR(J) \subseteq OR(i_1) \cup OR(i_2)$.*

2. $OR(i_1) \backslash OR(i_2) \subseteq OR(J)$.

3. $IR(J) \subseteq IR(i_1) \cup IR(i_2)$.

Proof. The proof can be found in [75, Theorem 5.1]. □

Theorem 2.2 shows that the output region of a composition of two instructions is a subset of the union of the output regions of the two instructions forming the composition. Similarly, Theorem 2.2 also indicates that the input region of a composition of two instructions is the union of the input region of the two instructions forming the composition. In [75], Theorem 2.2 was extended by the introduction of the following corollaries.

Corollary 2.2. *Under the conditions of Theorem 2.2, if* $IR(i_2) \cap OR(i_1) = \phi$*, then:*

1. $OR(J) = OR(i_1) \cup OR(i_2)$.

2. $IR(i_2) \subseteq IR(J)$.

Proof. The proof can be found in [75, Corollary 5.1]. □

Corollary 2.2 indicates that, if the input region of the second instruction and the output region of the first instruction are disjoint, then: (1) the output region of the composite instruction will equal to the union of the output regions of the two instructions, and (2) the input region of the second instruction is a subset of the input region of the composite instruction.

Corollary 2.3. *Under the conditions of Theorem 2.2, if* $OR(i_1) \cap OR(i_2) = \phi$ *and* $OR(J) = OR(i_1) \cup OR(i_2)$*, then:*

$IR(i_1) \subseteq IR(J)$.

Proof. The proof can be found in [75, Corollary 5.2]. □

Corollary 2.3 indicates that, if the output regions of the two instructions are disjoint and the output region of i_1 and the input region of i_2 are also disjoint, then the input region of i_1 is a subset of the input region of the composite instruction J.

Corollary 2.4. *Under the conditions of Theorem 2.2, if* $IR(i_2) \cap OR(i_1) = \phi$ *and* $OR(i_1) \cap OR(i_2) = \phi$*, then:*

$IR(J) = IR(i_1) \cup IR(i_2)$.

Proof. The proof can be found in [75, Corollary 5.3]. □

Corollary 2.4 indicates that, if the input region of the second instruction and the output region of the first instruction are disjoint and the output regions of the two instructions are also disjoint, then the input region of the composite instruction will equal to the union of the input region of the two instructions.

Corollary 2.5. *Under the conditions of Theorem 2.2, if* $J' = i_1(i_2(s))$*, then* $J = J'$ *if* $OR(i_1) \cap (IR(i_2) \cup OR(i_2)) = \phi$ *and* $OR(i_2) \cap (IR(i_1) \cup OR(i_1)) = \phi$.

Proof. The proof can be found in [75, Corollary 5.4]. □

It should be noted that, in general, $i_1 \circ i_2 \neq i_2 \circ i_1$. Hence, Corollary 2.5 indicates the special case, for which, the order of instructions execution can be changed without affecting the composition (*i.e.*, $i_1 \circ i_2 = i_2 \circ i_1$). In particular, *commuting instructions* are defined as follows.

Definition 2.7. *The two instructions* i_1, i_2 *can commute if and only if:*

1. $OR(i_1) \cap OR(i_2) = \emptyset$,
2. $OR(i_1) \cap [IR(i_1) \cup IR(i_2)] = \emptyset$, *and*
3. $OR(i_2) \cap [IR(i_1) \cup IR(i_2)] = \emptyset$.

Hence, two instructions can commute in two cases: (1) if all their four regions are disjoint, or more generally (2) if the only overlap in their four regions is in their input regions.

In [75], Maurer also showed that the instructions, in general, can also be decomposed into sequences of instructions. In particular, Theorem 2.3 shows that an instruction i can be expressed as a composition of two instructions i_1 and i_2 as follows.

Theorem 2.3 (Decomposition of Instructions)**.** *Let $x \in OR(i) - IR(i)$. Then, i can be written as $i(s) = i_2(i_1(s))$, where $IR(i_1) \subseteq IR(i)$, $IR(i_2) \subseteq IR(i)$, $OR(i_1) = \{x\}$, and $OR(i_2) = OR(i) - \{x\}$.*

Proof. The proof can be found in [75, Theorem 5.2]. □

Note that, by the application of Theorem 2.2 and Theorem 2.3, we can replace composite instruction sequences with other arbitrary equivalent composite instructions sequences (*i.e.*, an instruction can be decomposed into a composite sequence of atomic instructions).

2.3.5 Existence of Instructions

For two arbitrary subset regions of the memory, Maurer showed the existence of the instructions that have these regions as their input and output regions as follows.

Theorem 2.4 (Existence of Instructions)**.** *Let $P, Q \subset M$ in a Maurer computer. Then there exists an instruction i with $IR(i) = P$ and $OR(i) = Q$ if and only if $Q \neq \emptyset$ unless $P = \emptyset$.*

Proof. The proof can be found in [75, Theorem 12.1]. □

The existence of instructions will be also used as a part of the proof of Theorem 5.1 of Chapter 5.

2.3.6 Maurer Computer with a Control Unit

As indicated in [93], one of the most important contributions made by Alan Turing when he introduced the universal Turing machine (UTM) was the idea of pushing the machine controls into the memory in what is now known as the concept of *stored programs*. For Maurer's model, to push the model's controls into the memory, Van Zelst in [99] proposed the introduction of the following two maps:

1. A control unit, C, that is defined as,

$$C : \mathbb{S} \to \mathbb{I}, \tag{2.5}$$

   which is responsible for determining the next instruction to be executed from the current memory state.

2. A memory region denoted as $NI \subset M$ and defined as the *next instruction* subset, that stores the next instruction to be executed after the executing current instruction. Note that, in computer architecture, NI corresponds to the top of the instruction pipeline which has the next instruction to be executed.

3. A map, DEC, that decodes the next instruction from the current state. In particular, DEC is defined as,

$$DEC : \{s|NI\} \to \mathbb{I}. \tag{2.6}$$

   Hence, $C(s) = DEC(s \restriction NI)$.

That is to say, the next instruction to be executed is stored in the NI subset and the control unit C fetches it from its stored location NI while executing current instruction. Maurer computer with a control unit as proposed by Van Zelst is formalized in Definition 2.8 as follows.

Definition 2.8 (Maurer Computer with a Control Unit)**.** *A Maurer computer with a control unit is defined as the tuple* $\mathcal{M}_C = \langle M, B, \mathbb{S}, \mathbb{I}, C \rangle$ *where* $M, B, \mathbb{S},$ *and* $\mathbb{I}$ *are as defined in* Definition 2.2 *and:*

- $C : \mathbb{S} \to \mathbb{I}$ *is the control unit of the computer*
- NI *and* DEC *such that* $DEC : \{S \restriction NI | s \in \mathbb{S}\} \to \mathbb{I}$ *and also specifying that* $C(s) = DEC(s \restriction NI)$ *to ensure that the control unit respects the stored instructions*

By the introduction of C, NI, and DEC to the model, the computer now has the capability to store and execute instructions as per the standard CPU fetch and execute cycle. In addition, both C and NI enable the computer to express the process of sequential execution of instructions in the memory which can be interpreted as a sequential execution of programs. Finally, since each instruction can operate in the whole memory, then instruction execution can potentially change the contents of the set NI. Moreover, these constructs clearly support issues, such as, self-modifying code.

2.4 Discussion

Throughout Section 2.3, the basic modules of Maurer model have been defined. However, several extensions are still required to make the model more suitable to modern computers. In particular, the model should be able to provide the following concepts:

- *Information flow*: A wide variety of malware programs either download malicious code or instructions from remote systems or send private information to remote systems. The Maurer model does not provide a mechanism for the flow of information into or off the system (*i.e.*, external input and output mechanisms should be defined).

- *Multiple control units*: The Maurer model can only be used to model single control unit systems. Hence, it should be extended to enable the modeling of multi-control units systems such as modern multicore and multiprocessor systems so as to support the modeling of concurrency.

- *Programs*: The Maurer model does not have a definition for programs and it should be extended to support this concept.

- *Security policies*: A system's security policy defines the secure states of that system where these are then used to distinguish malware from benign programs. Maurer's model does not include the notion of security policies and it should be extended to capture this concept.

- *Computer networks*: The modeling of computer networks is essential to enable the modeling of specific malware categories (*e.g.*, worms). Maurer model does not contain a definition of networks and needed to be extended to contain these.

The Maurer model will be extended to include the above components, thereby forming the extended Maurer model (EMM), as will be discussed in the next chapter.

2.5 Summary

This chapter provided the literature review for this dissertation has been discussed. In particular, Section 2.2 discussed the existing formal models for both malware modeling and the analysis and evaluation of malware detection system. Whereas Section 2.3 provided a detailed overview of the basic building blocks of Maurer model that will be used to develop the extended Maurer model. Section 2.3 discussed the extensions required for the Maurer computer to make it more suitable to model malware and malware detection approaches which will be introduced in details in the next chapter.

Chapter 3

The Extended Maurer Model (EMM)

3.1 Introduction

As discussed in the previous chapter, some extensions for Maurer model are needed to enable the modeling of modern computers and networks. This chapter introduces these extensions and defines the *extended Maurer model* (EMM). In particular, the proposed EMM differs from the original Maurer model in the following:

(i) **The system memory:** As defined in Section 2.3.1, the size of the memory of the Maurer computer is fixed over time, which does not take into account the dynamic nature of the memories of today's systems (*e.g.*, by plugging or unplugging of USB devices, *etc.*). Moreover, the Maurer computer has no constructs to support the information flow into and out of the system. To handle these issues the memory of the EMM will be made time dependent allowing the memory size to change over time. Additionally, the structures required to support the information flows onto and off the computer will be introduced (Section 3.3).

(ii) **The control unit:** In Definition 2.8, the Maurer computers with control units consider only systems with a single control unit, whereas modern digital computers contain multiple control units. Hence, the control units of the EMM will be extended to comprise a set of control units to enable the modeling of modern computer systems (Section 3.4).

(iii) **The programs:** As shown in Section 2.3, the Maurer model only considers the sequential executions of single instructions (*i.e.*, does not support the concurrent execution of instructions) and does not seek to model the concept of *programs*. Hence, the EMM introduces the definition of programs as *software components* as well as the modeling of concurrent execution of instruction sequences (Section 3.5).

(iv) **The security policies:** As discussed in Section 2.3, the Maurer model does not include a definition for security policies. Since the main objective of this dissertation is to model malware and malware detection systems, the introduction of the concept of security policies in the EMM is required (Section 3.6).

(v) **Modeling computer networks:** As discussed in Section 2.3, the Maurer model does not include a definition for computer networks which is necessary to model specific categories of malware (*e.g.*, worms). The modeling of networks via the EMM will be discussed in this chapter (Section 3.8).

The remainder of this chapter proceeds as follows. Section 3.2 previews the basic modeling assumptions made in this dissertation. Section 3.3 introduces a detailed view of the system's memory M. Section 3.4 discusses the details of extending the notion of a control unit into a set of control units. Section 3.5 introduces the definition of programs (software components) as defined in the EMM together with all other aspects concerning them. Section 3.6 defines the set of system security policies.

Section 3.7 introduces the formal definition of the EMM. Section 3.8 discusses the Turing equivalence of the EMM and discusses also the use of the EMM to model virtual machines, stack, computer networks, *etc.* Finally, Section 3.9 summarizes this chapter.

3.2 Preliminary Assumptions

To simplify the development of the framework and its notation, the following assumptions will be made:

1. The control units are assumed to be implemented in hardware, and for simplicity, they are assumed to be tamper-proof (*i.e.*, the control units are fabricated in integrated circuit chips and their architectures will be assumed unchangeable post-fabrication). This assumption guarantees that the mechanism which is responsible for determining the next instructions from the current state is outside the influence of the attackers (*i.e.*, the mechanism by which the CPUs perform instruction fetches is tamper-proof). The developed EMM can be further extended to include issues such as forged circuitry, *etc.* However, such forms of attacks are not within the scope of this dissertation.

2. The control units within each physical computer are assumed to operate under a single theoretical global system clock, which acts to provide the reference time frame.

3. As per standard computers, the EMM is assumed to be a *causal* system (*i.e.*, the system's past and current states cannot be affected by any of its future states).

4. For simplicity, the instruction outputs are assumed to be available in memory immediately after an instruction's execution. Note that, this assumption can be relaxed by introducing per-instruction timing delays.

It should be noted that, as will be discussed in Section 3.8, Assumption 2 is structured to be consistent with the modeling of computer networks. Hence, these assumptions do not affect the generalizability of the developed EMM.

3.3 The System Memory, M

Modern computers consist of removable components (*e.g.*, USB devices) and permanent components (*e.g.*, CPU registers). Since all of these components are included in the model's memory, the memory must be made time dependent. Hence, in the EMM, the Maurer model's definition of M is extended as follows. At any time instant t, the EMM's memory is denoted as $M(t)$ and is defined as,

$$M(t) = \{m_k | k = 1, \ldots, N_M(t)\}, \tag{3.1}$$

where, as discussed in Section 2.3.1, each memory element m_k is *disjoint* with any other element (*i.e.*, $\forall k \neq k', m_k \cap m_{k'} = \emptyset$) and $N_M(t)$ is the number of elements in existence in the computer at the time instant t. Consequently, $M(t)$ denotes a set whose elements can change over time. For simplicity and as per digital computers, the time t is assumed to be *discrete*.

Based on Definition 2.3, which allows the restructuring of M in Maurer computer, the memory of the EMM is assumed to be structured as follows:

(i) Let $\Theta(t) = \{\theta_j |\ j = 1, \ldots, N_\Theta(t)\}$ be the set of all input devices existing within the computer at the time instant t. For each input device $\theta_j \in \Theta(t)$, define

$M_{\theta_j}(t) \subset M(t)$ to be its associated memory region at t, which is denoted as the *input interface memory region* (simply, *input interface*). Once the input information is written to $M_{\theta_j}(t)$ by the hardware θ_j, this information is assumed to become *immediately* available for the instructions to process. The computer's collection of input interfaces at t is defined as,

$$M_{\Theta}(t) = \bigcup_{\forall \theta_j \in \Theta(t)} M_{\theta_j}(t).$$

For simplicity, it is assumed that all $M_{\theta_j}(t)$ are *disjoint* (*i.e.*, $\forall \theta_j, \theta_k \in \Theta(t)$, $M_{\theta_j} \cap M_{\theta_k} = \emptyset$, $j \neq k$).

(ii) Similarly, define the set of output devices at t as $\Phi(t) = \{\phi_k |\ k = 1, \ldots, N_{\Phi}(t)\}$. Hence, the corresponding collection of *output interfaces* is defined as,

$$M_{\Phi}(t) = \bigcup_{\forall \phi_k \in \Phi(t)} M_{\phi_k}(t),$$

where all $M_{\phi_k}(t)$ are also assumed *disjoint*. Also, for simplicity, it is assumed that once the information is written to an output interface region by an instruction's execution, this information becomes immediately available to the external world. Although not strictly required, for simplicity, it is assumed that $M_{\Theta}(t) \cap M_{\Phi}(t) = \emptyset$.

(iii) The remaining portion of $M(t)$ will be denoted as $M^*(t)$ and will be defined as,

$$M^*(t) = M(t) \backslash [M_{\Theta}(t) \cup M_{\Phi}(t)]. \tag{3.2}$$

$M^*(t)$ denotes the memory regions that are not associated with the information flows to/from the external world (*i.e.*, standard memory), where "\" denotes set difference operation.

Note that, since $M(t)$, $M_\Theta(t)$ and $M_\Phi(t)$ can change with time by their definitions, then $M^*(t)$ can also change with time. For example, turning on the wireless network adapter and plugging in a USB memory drive will result in a change in $M^*(t)$. Now, $M(t)$ on composite can be expressed as,

$$M(t) = M^*(t) \cup M_\Theta(t) \cup M_\Phi(t), \tag{3.3}$$

where, again for simplicity, $M^*(t)$, $M_\Theta(t)$, and $M_\Phi(t)$ are assumed to be disjoint. As $M(t)$ is defined to be inclusive of all the computer's information storage elements, no loss of generality is incurred by this memory mapped view of the computer's input and output devices.

Let $\boldsymbol{T} = [-T_1, T_2]$ be the time interval starting from some $t = T_1$ in the past when the computer is powered on until some typically finite time $t = T_2 < \infty$ into the future, where $T_1, T_2 \geq 0$. For simplicity, it is assumed that $t = 0$ denotes the current time. Hence, $\boldsymbol{T}^- = [-T_1, 0]$ denotes the computer's past history and $\boldsymbol{T}^+ = (0, T_2]$ denotes the computer's future operations. Since t is assumed discrete, then the set of all memory transitions during the time interval $\boldsymbol{T}$, denoted as $\boldsymbol{M}(\boldsymbol{T})$ can be defined as,

$$\boldsymbol{M}(\boldsymbol{T}) = \{M(t)|\ t \in \boldsymbol{T}\}, \tag{3.4}$$

where $|\boldsymbol{M}(\boldsymbol{T})| < \infty$ and $|.|$ denotes set cardinality.

Since the memory of the EMM can change over time, then the set of possible states of the EMM also changes over time. In particular, the state of $M(t)$ at the

time t will be denoted as $s(t) \in \mathbb{S}(t)$, where $\mathbb{S}(t)$ is the set of all possible system states at t. Note that, for $t \neq t'$, if $N_M(t) \neq N_M(t')$ then $M(t) \neq M(t')$ and $\mathbb{S}(t) \neq \mathbb{S}(t')$. Additionally, let $\boldsymbol{S}(\boldsymbol{T})$ be the set of all states of the EMM that occur during the time period $\boldsymbol{T}$, and consequently, $\boldsymbol{S}(\boldsymbol{T})$ can be defined as,

$$\boldsymbol{S}(\boldsymbol{T}) = \{s(t)|\ t \in \boldsymbol{T}\}\,. \tag{3.5}$$

Hence, $\boldsymbol{S}(\boldsymbol{T})$ denotes the set of information that exists and can exist within the computer during $\boldsymbol{T}$. As t is assumed discrete, then $|\boldsymbol{S}(\boldsymbol{T})| < \infty$. It should be noted that, for past and current times the states of the system are known (*i.e.*, $\forall t \in [-T_1, 0], s(t)$ is known and, hence, $\boldsymbol{S}(\boldsymbol{T}^-)$ is known), whereas for future times the states of the system are probabilistic (*i.e.*, $\forall t \in (0, T_2], s(t)$ belongs to a probability distribution over $\mathbb{S}(t)$ and, hence, $\boldsymbol{S}(\boldsymbol{T}^+)$ is probabilistic), as shown in Figure 3.1.

It should be noted that, this probabilistic view of the EMM's future states conforms with those of real world systems where their future states are not known *a priori*. For example, the outputs of programs are not known till they receive their inputs that can vary from one run to the other. Also, other factors (*e.g.*, attacks, faults, *etc.*) can occur which, of course, cannot be exactly predicted by the defender. Hence, although the instruction set of the EMM is deterministic in the sense that for all $i \in \mathbb{I}$ the state of $OR(i)$ is known if the state of $IR(i)$ is known, both the state of $IR(i)$ and the execution time of i are generally not known deterministically leading to the probabilistic view of $\boldsymbol{S}(\boldsymbol{T}^+)$.

Recall that, the state of any $M' \subseteq M$ at t is denoted as $s(t)|M'$. Then the set of states of M' during $\boldsymbol{T}$ is $\boldsymbol{S}(\boldsymbol{T})|M'$. In particular, let the set of computer's input and output interfaces over $\boldsymbol{T}$ be defined as $\boldsymbol{M}_\Theta(\boldsymbol{T}) = \{M_\Theta(t)|\ t \in \boldsymbol{T}\}$ and $\boldsymbol{M}_\Phi(\boldsymbol{T}) = \{M_\Phi(t)|\ t \in \boldsymbol{T}\}$ respectively. Then, their corresponding states during $\boldsymbol{T}$ are $\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}_\Theta$ and $\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}_\Phi$ respectively. Finally, the states of the remainder of

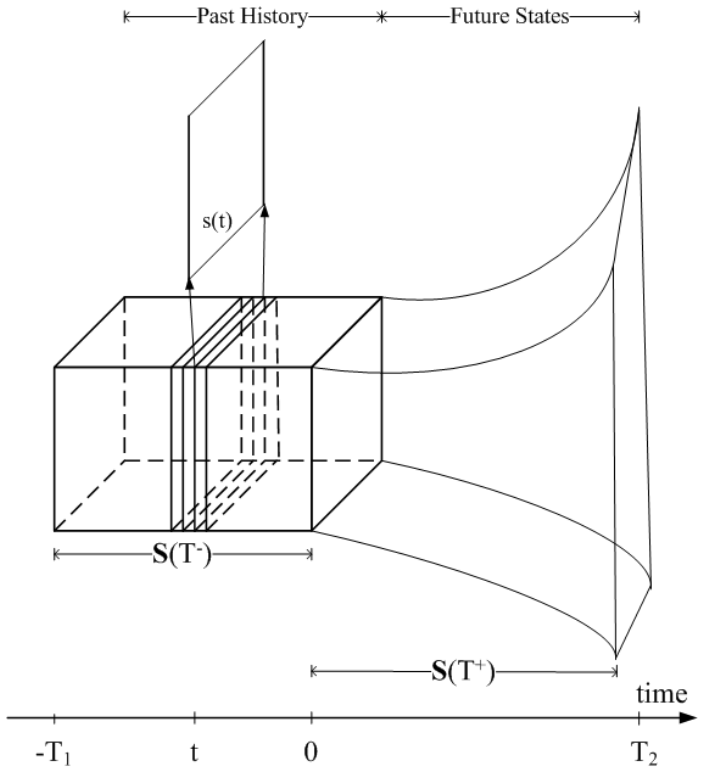


Figure 3.1: The EMM at past, current and future times.

$\boldsymbol{M}(\boldsymbol{T})$, denoted as $\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}^*$, defines all information within the computer over $\boldsymbol{T}$ that were not associated with information flows. Hence, $\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}^*$ is defined as,

$$\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}^* = [\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}] \setminus \left[\boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}_\Theta \bigcup \boldsymbol{S}(\boldsymbol{T})|\boldsymbol{M}_\Phi\right] \tag{3.6}$$

Hence, this extension of the memory now allows the modeling of the dynamic nature of the memory of modern computers as well as the information flows from outside world into and out of the computer. But, as a result, it also requires that $\boldsymbol{S}(\boldsymbol{T}^+)$ be viewed, in general, probabilistically and not deterministically.

3.4 Multiple Control Units

Note that, the Maurer computer with a control unit (as per Definition 2.8) defines only a single control unit for the system, and hence, can only be used to model systems that have single processing units. It only supports the execution of single instructions and does not support the concurrent execution of sets of instructions. However, modern computers have multiple processing units (*e.g.*, multi-processor systems, special purpose processors in interface cards, *etc.*) which execute multiple instructions concurrently to increase processing speeds. To enable the modeling of these more modern systems in the EMM, the single control unit C defined in Definition 2.8 is replaced by a set of $N_{\mathcal{C}}(t)$ control units, denoted as $\mathcal{C}(t)$, and defined as,

$$\mathcal{C}(t) = \{C_k \mid k = 1, 2, \ldots, N_{\mathcal{C}}(t)\}, \tag{3.7}$$

where $N_{\mathcal{C}}(t)$ is the number of control units in the system at the time t and each $C_k \in \mathcal{C}(t)$ is a control unit as defined in Definition 2.8. Additionally, $\mathcal{C}(t)$ includes any and all processing elements that can change the states of $\boldsymbol{M}(\boldsymbol{T})$. Hence, CPUs, GPUs, DMA controllers, *etc.* are all considered as control units (*i.e.*, they are elements in $\mathcal{C}(t)$). Due to this variety of control units, each $C_k \in \mathcal{C}(t)$ could have a different set of instructions $\mathbb{I}_{C_k}$, and consequently, the EMM's complete set of instructions $\mathbb{I}(t)$ is also extended and now defined as,

$$\mathbb{I}(t) = \bigcup_{k=1}^{N_{\mathcal{C}}(t)} \mathbb{I}_{C_k}.$$

Additionally, instead of a single next instruction subset memory region NI associated with a single controller, the memory is extended to include a set $\mathbb{NI}$ of $N_{\mathcal{C}}(t)$ of next instruction subset regions. Each $NI_k \in \mathbb{NI}$ stores the next instruction to be executed by the control unit $C_k \in \mathcal{C}(t)$. Hence, the composite $\mathbb{NI}$ is defined as,

$$\mathbb{NI}(t) = \{NI_k \mid k = 1, 2, \ldots, N_{\mathcal{C}}(t)\},$$

where, for simplicity, these are assumed to be *disjoint*. For notational simplicity, the time notation will be omitted when denoting control units and will be used only when needed contextually (*i.e.*, without ambiguity, $\mathcal{C}(t)$ will be replaced by $\mathcal{C}$, $N_{\mathcal{C}}(t)$ by $N_{\mathcal{C}}$, *etc.*).

As all control units are assumed to operate concurrently, concurrent changes to the memory can occur, whereas the original Maurer model was restricted to single sequential changes. Let i^k denote the execution of an instruction i on control unit $C_k \in \mathcal{C}$. In general, different instructions are assumed to have different execution times (*i.e.*, no atomic execution time for instructions is assumed). Hence, i^k's execution can span over multiple t time slots [100]. Let $\tau(i^k)$ be the time interval needed for the execution of i^k on C_k and let $\min[\tau(i^k)]$ and $\max[\tau(i^k)]$ be, respectively, the time instants of the beginning and the ending of i^k's execution. The i^k induced memory changes can now be viewed as being localized changes to particular *spatial-temporal* regions of $\boldsymbol{S}(\boldsymbol{T})$. The input and output regions of i^k over the time period $\tau(i^k)$ of i^k's execution can be defined respectively as,

$$\begin{aligned} \boldsymbol{IR}(i^k, \tau(i^k)) &= \bigcup_{\forall t \in \tau(i^k)} IR(i^k, t), \\ \boldsymbol{OR}(i^k, \tau(i^k)) &= \bigcup_{\forall t \in \tau(i^k)} OR(i^k, t). \end{aligned} \tag{3.8}$$

Therefore, as defined in Equation (3.8), $\boldsymbol{IR}(i^k, \tau(i^k))$ and $\boldsymbol{OR}(i^k, \tau(i^k))$ are sets of *spatial-temporal* regions within $\boldsymbol{S}(\boldsymbol{T})$ and not just the spatial regions as per the original Maurer model.

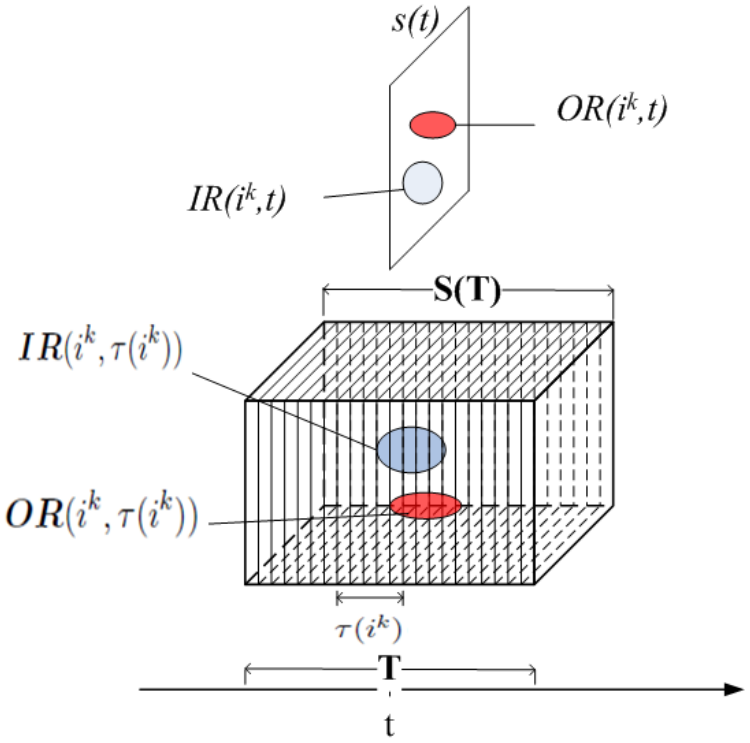


Figure 3.2: The execution of an instruction i^k and the spatial-temporal subspaces representing $\boldsymbol{IR}(i^k)$ and $\boldsymbol{OR}(i^k)$.

Figure 3.2 shows the system over the time period $\boldsymbol{T}$ (*i.e.*, $\boldsymbol{S}(\boldsymbol{T})$). In the figure, a snapshot is taken of at a time instant $t \in \tau(i^k)$ (*i.e.*, at $s(t)$) that shows the instantaneous input and output regions of i^k, $IR(i^k, t)$ and $OR(i^k, t)$ respectively. The figure also shows the spatial-temporal regions representing $\boldsymbol{IR}(i^k, \tau(i^k))$ and $\boldsymbol{OR}(i^k, \tau(i^k))$. As per standard computer architectures, it will be assumed that $\forall t \in \tau(i^k)$, $IR(i^k, t) \cap OR(i^k, t) = \emptyset$ (*i.e.*, $IR(i^k, t), OR(i^k, t)$ are disjoint for all t). Note that, this restriction enforces the EMM's required causality properties. Moreover, due to how Maurer instructions are defined, this is not at odds with standard models of how instructions execute, as illustrated by the following example. Consider the input and output regions of an exchange instruction that swaps the contents of two memory registers. This high-level view of the instruction is represented in terms of Maurer instructions as follows.

Example 3.1. *Consider the instruction:*

$$i: \quad SWAP \quad R_1, R_2$$

*that swaps the contents of registers R_1 and R_2. In real computers, the execution of i occurs over a sequence of steps. Figure 3.3 shows a possible Maurer execution of i. In particular, the figure shows that the execution of i can be considered as the composite execution of two instructions i_1 and i_2. As shown in the figure, the execution of the higher-level instruction i begins by executing the Maurer i_1, which stores the contents of R_1 and R_2 in the temporary locations $temp_1$ and $temp_2$. It is obvious that $IR(i_1) = R_1 \cup R_2$ and $OR(i_1) = temp_1 \cup temp_2$ and, hence, $IR(i_1) \cap OR(i_1) = \emptyset$ as required (*i.e., *$IR(i_1)$ and $OR(i_1)$ are disjoint). The Maurer instruction i_2 is then executed to copy the contents of $temp_1$ and $temp_2$, respectively, into R_1 and R_2, thereby completing the execution of the higher-level instruction i. Clearly, $IR(i_2)$ and $OR(i_2)$ are also disjoint as required under the EMM. Hence, $IR(i,t)$ and $OR(i,t)$ are disjoint for all $t \in \tau$ as a result of the higher-level instruction i's decomposition into its sequence of Maurer instructions. Moreover, instructions are, therefore, the atomic constructs by which state changes to memory occur within the EMM. The closest analogy, albeit a loose analogy, would be to the micro code instruction of modern CPUs.* □

In addition, the next instruction to be executed at C_k is then given by $i^k_{next} = C_k(s(\max[\tau(i^k)]) = DEC_k(s(\max[\tau(i^k)])|NI_k)$, where DEC_k and NI_k respectively denote C_k's next instruction decoder and next instruction subset, where $NI_j \cap NI_k = \emptyset$ for all $j \neq k$.

As discussed throughout this section, the concurrent execution of instructions via multiple control units allows the EMM to model computer systems with multiple

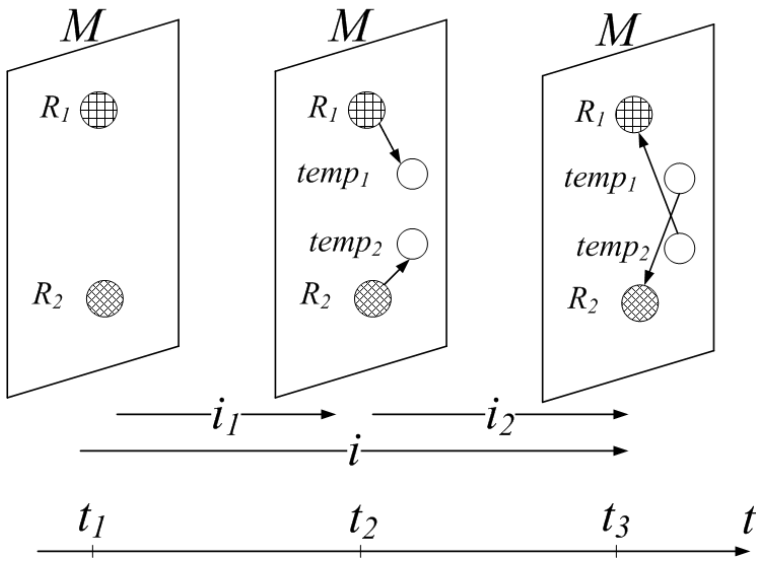


Figure 3.3: The execution of the SWAP instruction.

control units, such as modern multi-processors and multi-cores systems. Additionally, systems with dedicated processing units (*e.g.*, GPUs, *etc.*) can also be modeled via the EMM. Moreover, any hardware entity that enacts memory changes with a computer system under the EMM is a control unit that executes instructions. Hence, the processing done on network cards, GPU accelerators, *etc.* are defined in terms of EMM control units and their actions.

Finally, it should be emphasized that, as discussed in Section 3.2, the EMM must provide a causal system model. Hence, the input region of any instruction i^k that is executed over a time interval $\tau(i^k)$ (*i.e.*, $\boldsymbol{IR}(i^k, \tau(i^k))$) cannot contain regions belong to any future states. Hence, $\boldsymbol{IR}(i^k, \tau(i^k)) \cap [\boldsymbol{S}((\max[\tau(i^k)], T_2])|\boldsymbol{M}] = \emptyset$, as will be discussed later in Section 3.5.4.

3.4.1 Discussion

In this section, the EMM's extensions required to allow for the modeling of concurrent execution of instructions have been introduced via the concept of multiple

control units. For simplicity, the following assumptions are made, where these assumptions agree with those implemented and enforced by standard current memory architectures:

(i) Multiple read operations of the same memory region is allowed. Hence, the input regions of concurrently executing instructions are not assumed to be disjoint (*i.e.*, $IR(i^1, t) \cap IR(i^2, t)$ need not to equal $\emptyset$, where i^1 and i^2 are two concurrently executing instructions).

(ii) At any given time t, only a single write operation to specific memory region is allowed. Hence, the output regions of concurrently executing instructions must be disjoint (*i.e.*, $OR(i^1, t) \cap OR(i^2, t) = \emptyset$).

(iii) Finally, in the case of concurrent read and write operations to the same memory region, it is assumed that the write operation has the higher priority. Hence, the read is assumed to be blocked until the write operation completes.

The above assumptions guarantee that concurrently executing instructions will result in the correct and sound output (*i.e.*, the computer will behave deterministically with respect to individual atomic instruction executions), as discussed in the following example.

Example 3.2. *Consider the concurrent execution of the two instructions i^1 and i^2, respectively, concurrently on control units C_1 and C_2 in the EMM, as shown in Figure 3.4. As shown in the figure, at a time t during the execution, $IR(i^1, t)$ and $IR(i^2, t)$ need not to be disjoint (*i.e.*, $IR(i^1, t)$ and $IR(i^2, t)$ are allowed to intersect). However, it is clear in the figure that $OR(i^1, t) \cap OR(i^2, t) = \emptyset$ and, hence, $OR(i^1, t)$ and $OR(i^2, t)$ must be disjoint.* □

In general, computer memory hardware ensures that concurrent writes cannot occur to yhe same memory location(s).

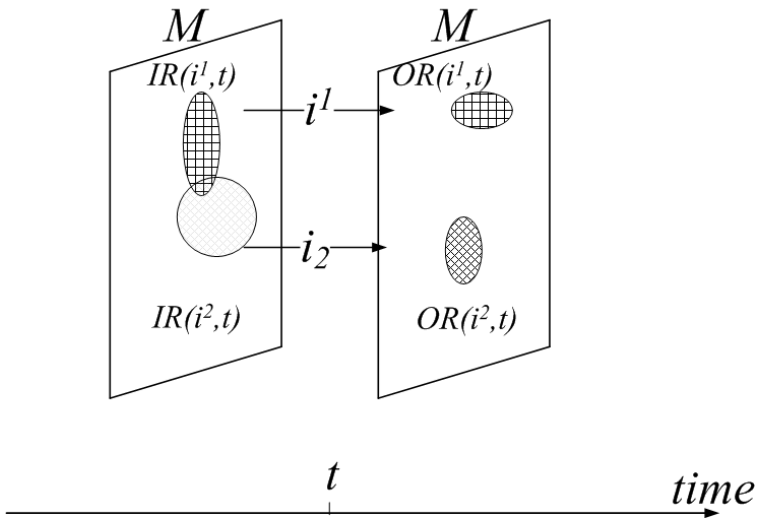


Figure 3.4: The concurrent execution of instructions.

3.5 Software Components

One of the main tasks of the computers is to run *programs*. However, programs have become more complex as software applications have evolved. Programs now span the range from a simple "hello world" executables to programs running within virtual machines. Within the EMM, the standard software engineering concept of *software components* as callable sets of instructions of specific functionalities [101] is used to represent programs. In general, a software component is defined as: "*a block of instructions and internal data that exists as a black box that performs specific input-to-output mappings over given time frames*" [102]. A simple program exists as a software component on its own, whereas, a complex program exists as a potentially concurrently executing set of components.

The rest of this section is organized as follows. Section 3.5.1 extends the definition of instruction compositions introduced in Section 2.3.4 from two instructions to arbitrarily sequences of instructions. Section 3.5.2 defines software components, with Section 3.5.3 defining execution traces and the input and output regions of these

components. Section 3.5.4 then defines the different information sets of interest with respect to software components. Section 3.5.5 introduces the notion of composite software components. Finally, Section 3.5.6 provides a discussion that indicates the modeling of stack in the EMM.

3.5.1 Instruction Composition

Before defining software components, the execution of an *instruction composition* presented in Equation (2.4) will be extended to cover a *sequence* of instruction executions. Let I_J be a *sequence* of instructions that is defined by,

$$I_J = (i_j |\ j = 1, 2, \ldots, N_J), \tag{3.9}$$

where J is an index set on $\mathbb{I}$ of length $N_J > 2$. Let the instructions in I_J be executed sequentially in their indexed order. Note that, as per real world systems, it is possible that the instruction sequence contains instructions that will be executed within different control units. For example, consider a sequence of instructions that includes reading stored information from the disk and emailing it to a remote person. This sequence includes instructions that will be executed by the disk controllers, others that will be executed by the network adapter controller, *etc.* Additionally, by extending the composition of instructions discussed in Section 2.3.4 to instruction sequences, then the execution of this instruction sequence can be modeled as an execution of a *single composite instruction*, that is defined as,

$$I_J = \bigodot_{j \in J} i_j = i_1 \circ i_2 \cdots \circ i_{N_J}. \tag{3.10}$$

The execution of the instruction sequence I_J will create a series of state changes within the system. Without loss of generality, the overall state change can be regarded as a

single change from the initial state of the system, denoted as s, to the system state, denoted as s'. This state change can be expressed as,

$$s' \equiv I_J(s) \equiv i_{N_J}(i_{N_J-1}(\ldots(i_2(i_1(s))))). \tag{3.11}$$

Now consider the execution of the instruction sequence I_J as defined above in a system with multiple control units over a time period $\tau \subseteq \boldsymbol{T}$. Concurrent execution of I_J's independent instructions in different control units, of course, may occur. To capture the timing related information, the *execution trace* of the I_J composition can be defined as,

$$\text{trace}(I_J, \tau) = \{\langle i_j, C_k, \tau(i_j)\rangle \mid i_j \in I_J,\ C_k \in \mathcal{C},\ \tau(i_j) \subseteq \tau\}, \tag{3.12}$$

where $\tau(i_j) \subseteq \tau$ is the time interval over which the instruction $i_j \in I_J$ was executed by $C_k \in \mathcal{C}$. Hence, $\text{trace}(I_J, \tau)$ captures the aspects related to the concurrent execution of the instructions of I_J through the system by specifying the control units that executed the instructions and the time intervals over which those instructions were executed on each of the respective control units. Accordingly, $\text{trace}(I_J, \tau)$ provides a detailed information about the control units that were involved in the execution of the instruction sequence. Moreover, it should be noted that, there is no overt requirement that the control units be synchronized[1].

In addition, the *input* and *output* regions due to the execution of I_J during τ, $\boldsymbol{IR}(I_J, \tau)$ and $\boldsymbol{OR}(I_J, \tau)$ respectively, can be defined as,

[1]Within the EMM, the computer OS can be modeled as a program. Hence, it is the OS's responsibility to ensure the correctness of parallel high-level instruction execution orders via its implementation of standard semaphore (or lock) constructs. Such issues though exist at much higher levels than the Maurer instructions atomic levels.

$$\begin{aligned}\boldsymbol{IR}(I_J,\tau) &= \bigcup_{\substack{\forall i_j \in trace(I_J,\tau) \\ \forall \tau(i_j) \subseteq \tau}} \boldsymbol{IR}(i_j,\tau(i_j)),\\ \boldsymbol{OR}(I_J,\tau) &= \bigcup_{\substack{\forall i_j \in trace(I_J,\tau) \\ \forall \tau(i_j) \subseteq \tau}} \boldsymbol{OR}(i_j,\tau(i_j)),\end{aligned} \tag{3.13}$$

where $\boldsymbol{IR}(i_j,\tau(i_j))$ and $\boldsymbol{OR}(i_j,\tau(i_j))$ are respectively the input and output regions of $i_j \in I_J$ as defined in Equation (3.8).

It should be noted that, trace(I_J,τ) is, by definition, deterministic for past times while it is probabilistic for future times. Then, since $t = 0$ denotes the current time, then all portions of $\boldsymbol{IR}(I_J,\tau)$ and $\boldsymbol{OR}(I_J,\tau)$ within the interval $\boldsymbol{T}^-$ (*i.e.*, the past and current times) exist as subsets of $\boldsymbol{S}(\boldsymbol{T}^-)$, whereas all portions of $\boldsymbol{IR}(I_J,\tau)$ and $\boldsymbol{OR}(I_J,\tau)$ within the interval $\boldsymbol{T}^+$ (*i.e.*, the future times) exist in the probability space $\mathbb{S}(\boldsymbol{T}^+)$. Hence, the computer's operation is knowable when looking into its past history and probabilistic when looking into the future, in keeping with the nature of real-world computer operations. The construct trace(I_J,τ), therefore, defines the spatial-temporal evolution of the subspace of $\boldsymbol{S}(\boldsymbol{T})$ that is associated with I_J's execution that occurs during the executions of I_J's composite instructions, inclusive of all possible concurrency ordering arising from multiple control units being used to execute portions (or all) of the I_J instructions.

3.5.2 The Definition of Software Components

To define software components, the instruction composition defined in Equation (3.9) is used as the building block. Let γ denote a software component that is defined as,

$$\gamma = \left\{ I_{J_q} \mid q = 1,\ldots,Q \right\}. \tag{3.14}$$

Each I_{J_q} is an instruction sequence as defined in Equation (3.9) that denotes the possible input-to-output mappings that can occur by γ's execution. Equation (3.14) defines γ as the set of *all* possible sequences of instructions that can occur as a result of executing the software component γ (*i.e.*, inclusive of error conditions, all possible input states, *etc.*). Of course, in any single instance of γ's execution, only one of these execution sequences will actually be run (*i.e.*, only one set of the possible memory state changes will actually occur each time γ is executed). Exactly which I_{J_q} sequence executes (or runs) for any execution of (or call to) γ depends on its inputs, as discussed in the following example.

Example 3.3. *Let γ be a component (program) that calculates the roots of a quadratic equation. Hence, it can be defined as $\gamma = \{I_{J_1}, I_{J_2}, I_{J_3}\}$, where: I_{J_1} is the sequence of instructions to be executed in the case of real non-equal roots, I_{J_2} is the sequence of instructions to be executed in the case of real equal roots, and, I_{J_3} is the sequence of instructions to be executed in the case of imaginary roots. Clearly, only one of these sequences will be executed depending on the values of the parameters of the quadratic equation that is passed as the input to any given call to γ.* □

Each software component γ must by definition occupy a memory region, which is denoted as M_γ. In real world systems, M_γ corresponds to the memory region allocated to programs (their storage on the disks, allocated instruction and data memory region when they are executed, *etc.*). At any given time t, the contents of M_γ is denoted as $s(t)|M_\gamma \subseteq s(t)|M^*(t)$. Note that, for any two time instants $t \neq t'$, it is possible that $M_\gamma(t) \neq M_\gamma(t')$ (*e.g.*, the component's memory region changes due to the downloading of new modules that changes the memory region required to hold the component) or $s(t)|M_\gamma \neq s(t')|M_\gamma$ (*e.g.*, the contents of the component's memory region changes due to the downloading of a new version). Definition 3.1 defines software components in the EMM.

Definition 3.1. *A software component γ is defined as a set of instruction sequences as shown in Equation* (3.14) *and has a memory region denoted as M_γ.*

Note that, as shown in Definition 3.1, γ does not necessarily define a finite set. Particularly, in cases when $T_2 \to \infty$, issues such as infinite loops would then cause $|\gamma| \to \infty$. More particularly, to be clear, γ defines the set of any and all possible sequences of memory state changes that can occur whenever γ is executed.

3.5.3 The Input and Output Regions of Components

Consider the execution of γ over a time period $\tau \subseteq \boldsymbol{T}$. The *execution trace* of γ will be defined by extending the prior trace construct defined in Equation (3.12) to cover components as follows,

$$\text{trace}(\gamma, \tau) = \{\langle i_j, C_k, \tau(i_j)\rangle \mid i_j \in \gamma,\ C_k \in \mathcal{C},\ \tau(i_j) \subseteq \tau\}. \tag{3.15}$$

In addition, γ's *input* and *output* regions will also be defined by extending Equation (3.13) to components as follows,

$$\begin{aligned} \boldsymbol{IR}(\gamma, \tau) &= \bigcup_{\substack{\forall i_j \in \text{trace}_{(\gamma,\tau)} \\ \forall \tau(i_j) \subseteq \tau}} \boldsymbol{IR}(i_j, \tau(i_j)), \\ \boldsymbol{OR}(\gamma, \tau) &= \bigcup_{\substack{\forall i_j \in \text{trace}_{(\gamma,\tau)} \\ \forall \tau(i_j) \subseteq \tau}} \boldsymbol{OR}(i_j, \tau(i_j)), \end{aligned} \tag{3.16}$$

where $\tau(i_j)$ is the execution period of the instruction $i_j \in \gamma$. Note that, $\boldsymbol{IR}(\gamma, \tau)$ and $\boldsymbol{OR}(\gamma, \tau)$ are also spatial-temporal regions within $\boldsymbol{S}(\boldsymbol{T})$.

Software components typically make extensive use of internal memory (*e.g.*, private methods and variables) that are not accessible by other components. However,

Equation (3.16) denotes *all* spatial-temporal memory regions touched by γ's execution during τ. The component's *internal memory region*, denoted as $\boldsymbol{EMR}(\gamma, \tau)$ is defined as,

$$\begin{aligned} \boldsymbol{EMR}(\gamma, \tau) = \boldsymbol{IR}(\gamma, \tau) \bigcup \boldsymbol{OR}(\gamma, \tau) - \left[\boldsymbol{M}_{\Theta} \bigcup \boldsymbol{M}_{\Phi} \right] \\ - \left[\bigcup_{\forall \gamma' \neq \gamma} \left(\boldsymbol{IR}(\gamma', \tau) \bigcup \boldsymbol{OR}(\gamma', \tau) \right) \right], \end{aligned} \tag{3.17}$$

As defined in Equation (3.17), $\boldsymbol{EMR}(\gamma, \tau)$ denotes all spatial-temporal input and output regions of γ due to its execution during τ that are neither: (i) subsets of the computer's input or output interfaces, nor, (ii) read or written to by *any* other software component γ' that may exist on the computer. Hence, $\boldsymbol{EMR}(\gamma, \tau)$ does not refer to a private memory that is allocated specifically to the program and is not written or read by other programs. Also, $\boldsymbol{EMR}(\gamma, \tau)$ can change from one execution to the other and, moreover, it is fully possible that $\boldsymbol{EMR}(\gamma, \tau) = \emptyset$. Importantly, $\boldsymbol{EMR}(\gamma, \tau)$ denotes all regions that remain private to γ during its execution and not all regions that for security reasons should remain private (*i.e.*, $\boldsymbol{EMR}(\gamma, \tau)$ expressly does not denote a security policy on γ's execution).

Additionally, the *external input region* and *external output region*, denoted as $\boldsymbol{EIR}(\gamma, \tau)$ and $\boldsymbol{EOR}(\gamma, \tau)$ respectively, are defined next as follows,

$$\begin{aligned} \boldsymbol{EIR}(\gamma, \tau) &= \boldsymbol{IR}(\gamma, \tau) \ - \boldsymbol{EMR}(\gamma, \tau), \\ \boldsymbol{EOR}(\gamma, \tau) &= \boldsymbol{OR}(\gamma, \tau) - \boldsymbol{EMR}(\gamma, \tau). \end{aligned} \tag{3.18}$$

Hence, $\boldsymbol{EIR}(\gamma, \tau) \subseteq \boldsymbol{IR}(\gamma, \tau)$ and $\boldsymbol{EOR}(\gamma, \tau) \subseteq \boldsymbol{OR}(\gamma, \tau)$ denote all γ's input and output regions respectively during τ that are not internal regions.

The *affected and affecting regions* in the context of software components are also spatial-temporal regions, which can be defined by extending Definition 2.6 to software components as follows. Let $trace(\gamma, \tau)$ be the execution trace of the software component γ during the time period τ. Consider the two subsets $M' \subseteq \boldsymbol{IR}(\gamma, \tau)$ and $N' \subseteq \boldsymbol{OR}(\gamma, \tau)$. The spatial-temporal regions that are affected by M' and the spatial-temporal regions that affect N', respectively, under the execution of γ during a time period τ, are then denoted as $\boldsymbol{AR}(M', trace(\gamma, \tau))$ and $\boldsymbol{RA}(N', trace(\gamma, \tau))$ respectively, and are defined as,

$$\begin{aligned} \boldsymbol{AR}(M', trace(\gamma, \tau)) &= \bigcup_{\forall M'' \in M'} AR(M'', trace(\gamma, \tau)), \\ \boldsymbol{RA}(N', trace(\gamma, \tau)) &= \bigcup_{\forall N'' \in N'} RA(N'', trace(\gamma, \tau)). \end{aligned} \tag{3.19}$$

3.5.4 The Information Sets of Components

Consider the execution of the software component γ during the time period $\tau \subseteq [-T_1, T_2]$. The set of information available for γ's execution is denoted as $dynamic(\gamma, \tau)$ and will be defined as the tuple,

$$dynamic(\gamma, \tau) = \langle \text{trace}(\gamma, \tau), \boldsymbol{IR}(\gamma, \tau), \boldsymbol{OR}(\gamma, \tau), \boldsymbol{S}(\tau) | \boldsymbol{IR}(\gamma, \tau), \boldsymbol{S}(\tau) | \boldsymbol{OR}(\gamma, \tau) \rangle \,, \tag{3.20}$$

where $dynamic(\gamma, \tau)$ defines the set of complete information contained within the EMM about the execution of γ during the time period $\tau \subset \boldsymbol{T}$. An example of $dynamic(\gamma, \tau)$ is as follows.

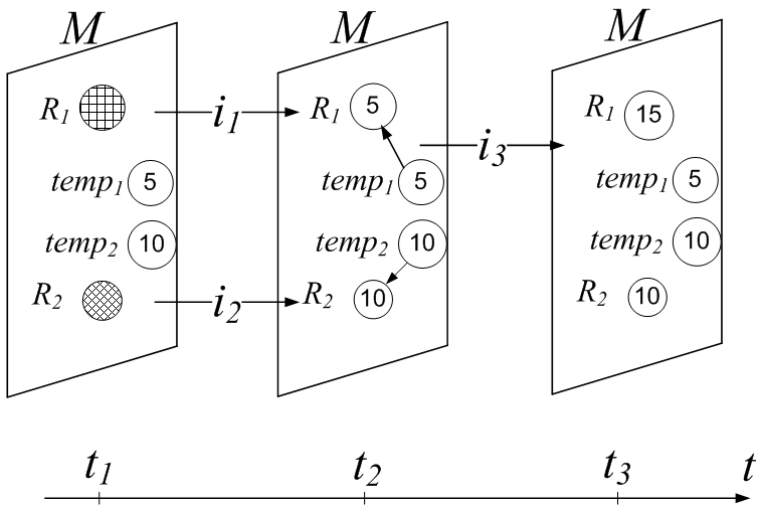


Figure 3.5: The concurrent execution of the instructions of Example 3.4.

Example 3.4. *Consider the software component γ that has the instruction composition $I_J \in \gamma$. Also, consider that I_J is defined as $I_J = (i_1, i_2, i_3)$ where:*

i_1:	*MOV*	$R_1, 5$	*(put the contents of memory region $temp_1$ into R_1).*
i_2:	*MOV*	$R_2, 10$	*(put the contents of memory region $temp_2$ into R_2)*
i_3:	*ADD*	R_1, R_2	*(add the contents of R_2 and R_1 and place the result in R_1)*

Additionally, define $\mathcal{C} = \{C_1, C_2\}$ where C_1, C_2 are two cores in a dual-core system. Consider that I_J is the executed instruction sequence that occurs when γ is executed τ. Since i_1, i_2 are independent, they can be executed concurrently. As shown in Figure 3.5, i_1 and i_2 are independent instructions and can be executed concurrently. Then, the execution of i_3 occurs after the execution of i_1 and i_2. Hence, a possible execution trace of γ can be as follows: $trace(\gamma, \tau) = \{\langle i_1, C_1, \tau_1\rangle, \langle i_2, C_2, \tau_1\rangle, \langle i_3, C_1, \tau_2\rangle\}$, where $\tau_1, \tau_2 \subset \tau$ and $\tau_1 \cap \tau_2 = \emptyset$. Whereas, $\boldsymbol{IR}(\gamma, \tau) = R_1 \cup R_2 \cup temp_1 \cup temp_2$, $\boldsymbol{OR}(\gamma, \tau) = R_1 \cup R_2$, and so on. □

Intuitively, $dynamic(\gamma, \tau) = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ if no instructions in γ have been executed during τ. It should be noted that, Equation (3.20) defines the *full* set of

run-time information regarding γ's execution during τ that is or can be constructed via γ's instruction executions. As $dynamic(\gamma, \tau)$ is defined in terms of instructions, their timings, and their input to output mappings, all possible real-world programming constructs are expressible within this software component notation, albeit from an information-centric viewpoint (*i.e.*, to define γ for an operational computer, one must know all possible paths through a software component that can occur for all possible inputs to that component). Hence, γ is primarily a useful theoretical construct as its precise statement in practice is largely only feasible for denoting simple components. Although it should be noted that knowing the possible paths through software components is a common requirement of software testing regimes.

Also, from the malware detection perspective, the *static information* available about each software component at time $t \in \boldsymbol{T}$ must be defined, where this will be denoted as $static(\gamma, t)$. Note that, $static(\gamma, t)$ is the information that can be extracted from statically analyzing γ when it is not executing. Accordingly, $static(\gamma, t)$ is defined within the EMM as,

$$static(\gamma, t) = s(t)|M_{\gamma}. \tag{3.21}$$

Hence, $static(\gamma, t)$ defines the static information about γ that exists at the time instant t. It should be noted that, for $t \neq t'$, it is possible that $M_{\gamma}(t) \neq M_{\gamma}(t')$ or $static(\gamma, t) \neq static(\gamma, t')$. For example, if γ did not exist on the computer until the time t then, by definition, $static(\gamma, t') = \emptyset$ for all $t' < t$. In general, $static(\gamma, t)$ provides a snapshot of the non-execution measurable information about γ at t, whereas $dynamic(\gamma, \tau)$ provides the run-time measurable information available about γ's execution during $\tau \subset \boldsymbol{T}$.

Note that, for Maurer model, the required causality properties are preserved in a natural way within Definition 2.4 since $IR(i)$ for all instructions $i \in \mathbb{I}$, it the case that,

$$\forall i \in \mathbb{I}, \forall t \in \boldsymbol{T}, s(t)|IR(i) \cap [\boldsymbol{S}((t, T_2])|\boldsymbol{M}] = \emptyset, \tag{3.22}$$

which ensures the causality of the modeled system by ensuring the input regions of instruction cannot come from future states.

Finally, the set of *casual* information regarding γ that is available within the system during $\boldsymbol{T}$ will be denoted as $\boldsymbol{Info}[\gamma, \boldsymbol{T}]$ and will be defined as the tuple,

$$\boldsymbol{Info}[\gamma, \boldsymbol{T}] = \left\langle \bigcup_{\forall t \in \boldsymbol{T}} static(\gamma, t), \bigcup_{\forall \tau \subseteq \boldsymbol{T}} dynamic(\gamma, \tau) \right\rangle. \tag{3.23}$$

Hence, $\boldsymbol{Info}[\gamma, \boldsymbol{T}]$ contains the complete information about all static representations that γ has during $\boldsymbol{T}$ and contains also the complete information about all executions of γ during $\boldsymbol{T}$.

3.5.5 Composite Software Components

To enable the modeling of complex programs, the notion of instruction compositions can be extended as follows. Let $\boldsymbol{\Gamma}(t)$ be the set of all software components that exist in the EMM at t, and defined as,

$$\boldsymbol{\Gamma}(t) = \{\gamma_j \mid j = 1, 2, \ldots, N_{\boldsymbol{\Gamma}}(t)\}, \tag{3.24}$$

where it should be noted that, for $t \neq t'$ it is possible that $\boldsymbol{\Gamma}(t) \neq \boldsymbol{\Gamma}(t')$. Let $P = (p \mid p = 1, 2, \ldots N_P)$ be an index set defined over $\boldsymbol{\Gamma}(t)$. A composite software component can then be denoted as a set γ_P defined as,

$$\gamma_P = \{\gamma_p \mid p \in P\}\,. \tag{3.25}$$

Equation (3.25) introduces the EMM's generalized definition for representing programs as the collection of all possible γ_P instruction compositions that can occur when the program is run for all possible input and output patterns, inclusive of those that are event based and/or require user input/output. By definition, such composite components are inclusive of components that may exist at different times within the EMM. This is notationally important as it, for example, allows for the modeling of executable code that is downloaded at run-time, potentially, on a just-in-time or as-needed basis. This focus on modeling programs in terms of their possible execution traces is necessary to account for the complexities that can arise within modern programing constructs (*i.e.*, most programs are no longer focused on performing simple input to output mapping as per, for example, a "*hello world*" program).

This approach allows complex programs to be defined in terms of their component compositions, under the following extensions of Equations (3.15) to (3.23) as follows.

$$\text{trace}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \text{trace}(\gamma_p, \tau'), \tag{3.26a}$$

$$\boldsymbol{IR}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \boldsymbol{IR}(\gamma_p, \tau'), \tag{3.26b}$$

$$\boldsymbol{OR}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \boldsymbol{OR}(\gamma_p, \tau'), \tag{3.26c}$$

$$\boldsymbol{EMR}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \boldsymbol{EMR}(\gamma_p, \tau'), \tag{3.26d}$$

$$\boldsymbol{EIR}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \boldsymbol{EIR}(\gamma_p, \tau'), \tag{3.26e}$$

$$\boldsymbol{EOR}(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} \boldsymbol{EOR}(\gamma_p, \tau'), \tag{3.26f}$$

$$\boldsymbol{AR}(M', trace(\gamma_P, \tau)) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall M'' \in M' \\ \forall \tau' \subseteq \tau}} AR(M'', trace(\gamma_p, \tau')) \tag{3.26g}$$

$$\boldsymbol{RA}(N', trace(\gamma_P, \tau)) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall M'' \in N' \\ \forall \tau' \subseteq \tau}} RA(N'', trace(\gamma_p, \tau')) \tag{3.26h}$$

$$dynamic(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall \tau' \subseteq \tau}} dynamic(\gamma_p, \tau'), \tag{3.26i}$$

$$static(\gamma_P, \tau) = \bigcup_{\substack{\forall \gamma_p \in \gamma_P \\ \forall t \in \tau}} static(\gamma_p, t), \tag{3.26j}$$

$$\boldsymbol{Info}[\gamma_P, \boldsymbol{T}] = \bigcup_{\forall \gamma_p \in \gamma_P} \boldsymbol{Info}[\gamma_p, \boldsymbol{T}]. \tag{3.26k}$$

3.5.6 Discussion

Since the stack exists in almost all modern computer architectures and it is a source of a number of attacks (*e.g.*, stack overflow, *etc.*) [103], we will discuss the representation of the stack in the EMM. In particular, it should be emphasized that, this work does not propose any approach to protect the stack from malicious attacks as the EMM's focus is to provide a general model of all attacks. The objective of this subsection is

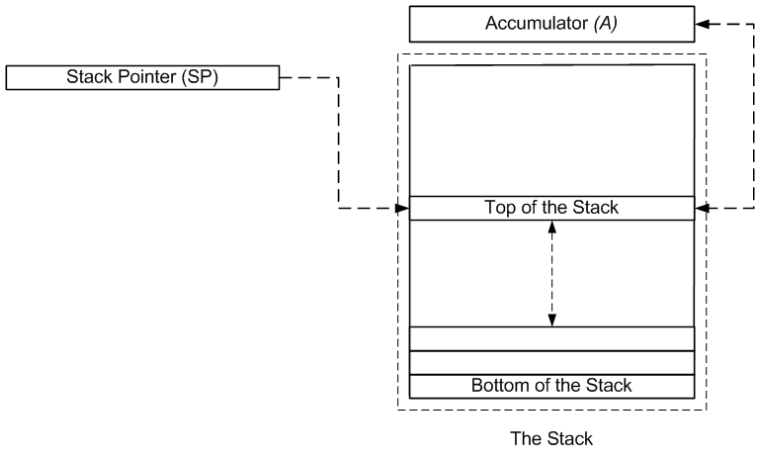


Figure 3.6: The stack architecture.

to show that the stack (as an example of special memory structures) can be modeled within the developed EMM in terms of the developed software component construct.

Figure 3.6 shows an example architecture of the stack as discussed in [100, Chapter 10, pp. 401]. Data items can be added to (or retrieved from) the stack only through its top, which is pointed to by the *stack pointer* (SP). In particular, the stack pointer points to the next empty location at the top of the stack. The data item to be added to (or retrieved from) the stack is stored (or will be stored) in the *accumulator* register A. Adding items to the top of the stack is achieved by executing a special instruction that is commonly denoted as the $PUSH$ instruction. The execution of the $PUSH$ instruction includes:

(i) Copying the contents stored in the accumulator (A) to the top of the stack that is pointed to by the stack pointer (SP), and

(ii) Updating the contents of SP, such that, it points to the next empty location in the top of the stack.

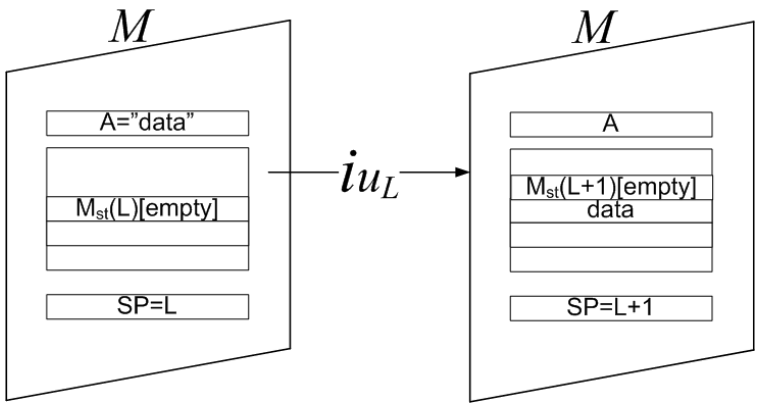


Figure 3.7: Storing data in the stack

Whereas, retrieving items from the top of the stack is achieved by executing another special instruction that is commonly denoted as the POP instruction. The execution of the POP instruction includes:

(i) Updating the contents of SP, such that, it points to the top stored item, and

(ii) Copying the contents of the location that is pointed to by SP into the accumulator register A.

In the EMM, the stack exists as a subset of the memory M. Denote the stack as $M_{st} \subset M$, the stack pointer as $SP \subset M$, and the accumulator as $A \subset M$. The $PUSH$ and POP instructions can also be represented in the EMM. However, as discussed in Section 2.3.2.1, the instructions in the EMM are uniquely characterized by their input and output regions. Hence, each memory location in the stack need to have its own $PUSH$ and POP instructions that store or retrieve data from this particular location. Accordingly, instead of a single $PUSH$ instruction, a set $PUSH = \left\{i_{u_j} | j = 1, 2, \ldots, N_{st}\right\}$ of $PUSH$ instructions will be defined, where $PUSH \subset \mathbb{I}$ and N_{st} is the size of stack (*i.e.*, number of storage units in the stack). Similarly, instead of one POP instruction, a set $POP = \left\{i_{p_j} | j = 1, 2, \ldots, N_{st}\right\}$ of POP instructions will be defined, where $POP \subset \mathbb{I}$, as discussed in Example 3.5.

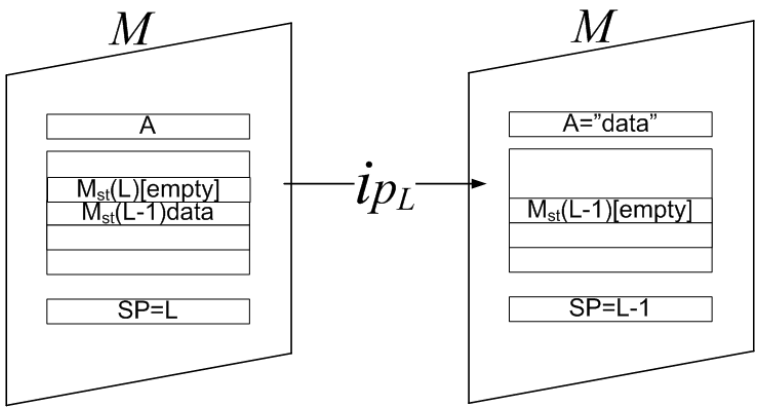


Figure 3.8: Retrieving data from the stack.

Example 3.5. *Consider a stack as discussed above. Let SP point to a location L, where $1 \leq L \leq N_{st}$, then:*

- *To store a new data in the stack, the instruction $i_{u_L} \in PUSH$ will be executed. Additionally, $\boldsymbol{IR}(i_{u_L}) = A \cup SP$ and $\boldsymbol{OR}(i_{u_L}) = M_{st}(L) \cup SP$, where $M_{st}(L)$ is the L location in the stack. Figure 3.7 shows this example of $PUSH$ operation.*

- *To retrieve a stored data from the stack, the instruction $i_{p_L} \in POP$ will be executed, where $\boldsymbol{IR}(i_{p_L}) = M_{st}(L) \cup SP$ and $\boldsymbol{OR}(i_{p_L}) = A \cup SP$. Figure 3.8 shows this example of POP operation.* □

As discussed above, it is clear that the stack can be simply modeled in the EMM taking into account the nature of instructions in the model. It should be noted that, managing $PUSH$ and POP instruction sets is carried out by a software component γ_{stack}, which is a component of the system's OS.

3.6 The System Security Policies

3.6.1 Introduction

Security policies play an important role in computer security research in general and computer virology research in particular. Moreover, with respect to malware, security policies define which programs are to be considered *malware* from the system's perspective. By definition, malware must produce at least one security policy violation (malware is defined as: "*a set of instructions that causes the system's security policy to be violated*" [104, Definition 22.1, pp. 613]). In addition, in many cases, benign component executions could also cause information changes that are similar to what may be considered malware in other contexts. Hence, legitimate applications could be considered malware if security policies are not taken into account. For example, a computer virus could copy code into a target program but so could the program's own update processes. Similarly, a spyware program may send passwords off the computer but so would a standard remote login processes. The distinction between benign components and malware in such cases need to be defined. Within the EMM, this is accomplished by considering whether a perfectly knowledgeable defender would authorize the component to execute given a complete knowledge of its complete information (*i.e.*, given $\boldsymbol{Info}[\gamma_P, \boldsymbol{T}]$, would such a defender authorize or not authorize γ_P's execution).

It should be noted that, defining malware in terms of *malicious intent* is not followed within the EMM. This is because the attacker's *intent* is not a directly defenders' measurable quantity. Instead, intent exists within the mind of the attackers placing it outside of the EMM's information context. The need to quantify security requires to focus on security constructs and models of those aspects that can be directly measured by the defender (*i.e.*, in this case, aspects that are within the EMM

modeled information sets). Security cannot be quantified based on issues which are outside of the defender's context to measure, such as the nature of the human intent which may have given rise to an observed event with the defended environment.

A system's security policies should be specified such that they consider all relevant security aspects, which are *confidentiality*, *integrity*, and *availability* [104, pp. 97]. If the policies do not cover these aspects then the result would be a vulnerable system. In general, the security policies function to partition the set of *states* of the system into a set of *authorized* (secure) states and a set of *unauthorized* (non-secure) states [104, Definition 4.1, pp. 95]. The system is secure if all of the system *state transitions* are only between secure states no non-secure state is never entered, otherwise, the system is vulnerable [104, Definition 4.2, pp. 95]. In general, the entity that is responsible of enforcing the security policies (or a part of the security policy) is denoted as the *security mechanism(s)* [104, Definition 4.7, pp. 98].

3.6.2 Formal Definition

Within the EMM, the defended environments' security mechanisms are defined via the constructs of the system's set of perfect security policies, where these provide the theoretically necessary ground-truth as to whether events are or are not attacks. In this section, the EMM's perfect security policies are formally defined as follows. Without loss of generality, a security policy is denoted as π and is defined in its most basic form as a map,

$$\pi : \boldsymbol{S}(\boldsymbol{T}) \rightarrow \{-1, 1\}. \tag{3.27}$$

Hence, security policies are, in general, defined as the maps from the EMM information space into the set $\{-1, 1\}$. In particular, in the case of software components, security policies can be defined as,

$$\pi : \{\boldsymbol{Info}[\gamma_P, \boldsymbol{T}]\} \rightarrow \{-1, 1\}, \tag{3.28}$$

given that $\boldsymbol{Info}[\gamma_P, \boldsymbol{T}] \subset \boldsymbol{S}(\boldsymbol{T})$. That is to say, security policies are defined as maps from the complete sets of information available over $\boldsymbol{T}$ about the composite component onto the set $\{-1, 1\}$. Moreover, $\pi[\boldsymbol{Info}[\gamma_P, \boldsymbol{T}]] = 1$ denotes that an ideal defender given perfect and complete information about $\boldsymbol{S}(\boldsymbol{T})$ would authorize γ_P, whereas $\pi[\boldsymbol{Info}[\gamma_P, \boldsymbol{T}]] = -1$ denotes that the ideal defender will not authorize γ_P (*i.e.*, given perfect and complete knowledge of γ_P, the defender would denote γ_P as an attack).

It should be noted that, defining policies with respect to component compositions allows the model to be inclusive of modern complex malware for which the overall malicious functions are distributed over many individual components and where the actions of individual components may be denoted as non-malicious nature if they were assessed in isolation (*e.g.*, K-ary malicious codes [95], multiprocess malware [96], *etc.*). The complete set of the system's perfect security policies is then denoted as Π^* and defined in Definition 3.2 as follows.

Definition 3.2. *The set of perfect security policies of the EMM is defined as the set of all policies* $\Pi^* = \{\pi_j \mid j = 1, \ldots, N_{\Pi^*}\}$, *where each* $\pi_j \in \Pi^*$ *is a security policy as defined in Equation* (3.27).

Clearly, as shown in Definition 3.2, the set of perfect security policies Π^* is a theoretical concept in that they are not available to a real-world system defender, *i.e.,* the defender would need to possess formal method proofs of the system's security or have access to all future information about the defended environment. Note that, as will be shown in Definition 4.5, the violation of any π will result in the violation of Π^*. The formal modeling of the violations of confidentiality, integrity and availability

will be discussed in details in Section 4.1. Moreover, this notion of Π^* should not be confused with the notion of a system's operational security policies. Π^* denotes the perfect set of policies that could be instantiated only if the defender has perfect and complete information about all of the defended environment's past and future states. Hence, the Π^* construct solely provides the necessary theoretical ground-truth as to whether or not an observed event is or is not associated with an attack.

3.7 The Extended Maurer Model (EMM)

Finally, after reviewing each of its components separately, the extended Maurer model (EMM), can now be formally defined as:

Definition 3.3 (The extended Maurer Model, EMM)**.** *The EMM is defined as the tuple* $EMM = \langle M, \mathbb{I}, \mathbb{S}, B, \mathcal{C}, \mathbf{\Gamma}, \Pi^* \rangle$ *where:*

1. M *is the finite set of the memory as discussed in Section 3.3.*
2. $\mathbb{I}$ *is the finite set of instructions as discussed in Section 3.3.*
3. $\mathbb{S}$ *is the finite set of states as discussed in Section 3.3.*
4. B *is the base set that as defined in Definition 2.2.*
5. $\mathcal{C}$ *is the set of control units, where* $|\mathcal{C}| \geq 1$*, as defined in Section 3.4.*
6. $\mathbf{\Gamma}$ *is the set of software components that exist on the EMM as discussed in Section 3.5.*
7. Π^* *is the system's set of perfect security policies as defined in Section 3.6.*

3.8 Discussion

Throughout the previous sections, the details and required extensions of the EMM have been discussed and developed. In this section, the Turing equivalence of the EMM will be discussed. Also, the use of the EMM to model important computing aspects will be highlighted. In particular, the use of the EMM to model virtual machines, web browsers, self-modifying code, and computer networks will also be discussed in more details.

3.8.1 Turing Reducibility

Since Maurer model has been shown to be a Turing equivalent model [76], it is important to show that the developed EMM is also retain this Turing equivalency. From Church's thesis[2], showing that the EMM is a Turing equivalent model allows its use as a model of general computations.

The core difference between Maurer computer and the EMM is that in Maurer computer instructions are only allowed to execute sequentially one after another (*i.e.*, one instruction during the time slot), whereas, as discussed in Section 3.4, in the EMM, up to $N_{\mathcal{C}}$ instructions can be executed in parallel during any time slot. By showing that the EMM can be reduced to the original Maurer model, it can be shown that the EMM is also a Turing equivalent model as per the original Maurer model. We will discuss this in the following lemma.

Lemma 3.1. *The EMM is reducible to Maurer computer and, therefore, retains its Turing-equivalency.*

Proof. We will show that the instruction executions in the EMM can be reduced to the instruction executions of the original Maurer model which implies that the

[2]Church's thesis states that: *every effective computation (or algorithm) can be programmed to run on a Turing machine* [105, Section 2.4, pp. 35].

EMM is also a Turing equivalent model. Let the EMM have $1 < N_{\mathcal{C}} < \infty$ control units. Let $i = \left\{ i^k | k = 1, 2, \ldots, N_{\mathcal{C}} \right\}$ be the set of instructions to be executed by the control units during the time instant t with $IR(i^k)$ and $OR(i^k)$ being the input and output regions respectively of each i^k as defined in Equation (3.8). Since the output regions of concurrently executing instructions are assumed disjoint under the EMM as discussed in Section 3.4, then the overall set of memory regions that will be changed due to the execution of these instructions is defined by $\bigcup_{k=1}^{N_{\mathcal{C}}} OR(i^k)$. Additionally, the overall set of memory regions that affect the changed set of memory is $\bigcup_{k=1}^{N_{\mathcal{C}}} IR(i^k)$. Accordingly, without loss of generality, the overall effect of executing $N_{\mathcal{C}}$ concurrent instructions within the EMM is that the state of the EMM is simply changed from one state to another state, and from the definition of instructions in Maurer model introduced in Equation (2.3), this change can be considered to be a result of the execution of a single composite instruction denoted as I. Hence, I can be expressed as the mapping $I : \mathbb{S} \to \mathbb{S}$ with $IR(I) = \bigcup_{k=1}^{N_{\mathcal{C}}} IR(i^k)$ and $OR(I) = \bigcup_{k=1}^{N_{\mathcal{C}}} OR(i^k)$. That is to say, the execution of the parallel instructions within the EMM can be considered as the execution of a single composite instruction I within the original Maurer model occurring over some period of time (*i.e.*, where each instruction $i \in \mathbb{I}$ now executes sequentially). As the parallel instructions executions of the EMM can be reduced to a single composite Maurer instruction's sequential execution over time, then the Turing equivalence property of the Maurer model still holds as any EMM parallel instruction execution can be reduced via the above process to the original Maurer model. □

It should be noted that, the Maurer model does not define the time required for instructions to enact their memory state changes. Hence, within the EMM, these sequences of individual state changes can simply be assumed to occur within one time

slot. Lemma 3.1 shows that the EMM retains the Turing equivalence of the original Maurer model. Hence, the EMM can also serve as a general model for computations.

3.8.2 Modeling Virtual Machines

By Lemma 3.1, the EMM can be used to model general programs and computations. However, there is a set of programs whose modeling needs to be explained in a more details. These programs are the virtual machines (VMs). In general, as shown in Figure 3.9, a VM is a software application that is used to run an unmodified operating system, which is usually denoted as the *guest* OS, over the existing operating system, which is denoted as the *host* OS [106]. Hence, a VM running in the EMM can be considered as a program that is being executed within the model. The most efficient approach to model VMs within the EMM is via a hierarchical structure of the EMMs, where the host computer is represented as first level EMM and each VM is represented by another separate nested EMM model. Moreover, if the set of programs within a VM includes a VM within it, another third level of models would then be needed to model the execution of programs within this VM (*i.e.*, the number of levels of models has no limit). In this hierarchical organization, complex programs such as VMs and modern browsers can efficiently represented within the EMM while still retaining their commonly perceived natures (*i.e.*, VMs see virtualized hardware component as presented to them under software control, the complexities of which are best represented via hierarchical nested EMMs). Clearly, modern multi-tabbed browsers can also be represented by a similar hierarchical EMM approach.

3.8.3 Executing Interpreted Programs

In this subsection, we discuss the execution of the interpreted programs in the EMM. Without loss of generality, there are two general categories of programs:

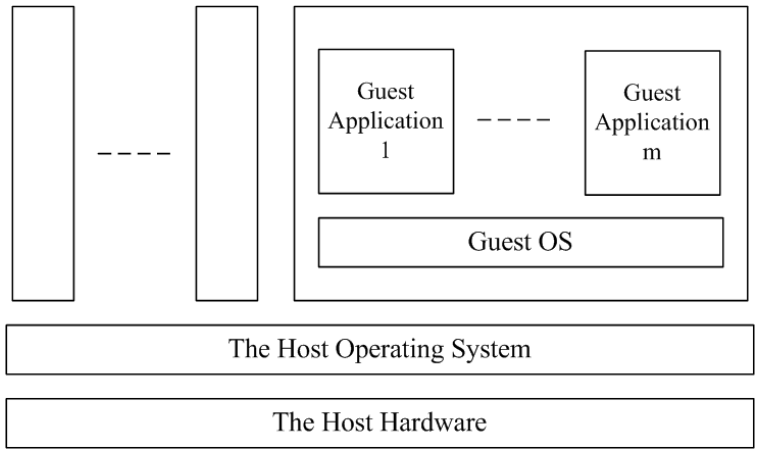


Figure 3.9: The internal view of a computer that has a VM.

- *Compiled programs:* The compiler translates the program's instructions to the machine's instructions that can be executed by the computer [107, Chapter 27, pp. 511]. This translation needed to be done once and then the executable code can run on the machine without further need to the compiler.

- *Interpreted programs:* The interpreter translates the program's instructions in instruction-by-instruction basis during the execution of the program [107, Chapter 27, pp. 511]. Hence, this translation is needed as the program executed within the interpreter.

It should be noted that, both categories have their advantages and disadvantages, *i.e.,* in terms of the execution speed, platform dependency, code protection, *etc.* In this dissertation, the objective is not to compare between them. The objective is to show that the EMM is generic, in that, the execution of the programs belonging to both categories can be represented in the model. In general, the execution of compiled programs in the EMM was described in the preceding sections. Accordingly, this subsection focuses on the execution of interpreted programs.

Without loss of generality, the execution of interpreted programs within the EMM follows the same approach as the execution of the compiled programs with only one

difference. In particular, the interpreter must first translate the higher-level instruction to be executed into the machine code sequence that actually will be executed by the CPU as discussed in the following example.

Example 3.6. *Consider the exchange instruction i discussed in Example 3.1. If i is an instruction in a compiled program, then i will be replaced by the CPU's machine codes for i_1 and i_2 when the program is compiled. Then each time the program is run, i_1 and i_2 will be executed by the system's control units.*

*Now, consider that i is an instruction in an interpreted program. Then, each time to run the program, an instruction i' which corresponds to translating i into its associated set of machine instructions will be executed first. Hence, as per the above, $OR(i')$ will then contain the machine code of i_1 and i_2 (*i.e.*, it is preceded with the translation step). The process performing this translation is the EMM software component $\gamma_{interpreter}$, which is responsible for mapping its higher-level language constructs into control unit instruction sequences for their execution. As shown in Figure 3.10, i' will be executed to translate i into i_1 and i_2. Then, the execution will continue as discussed in Example 3.1. Accordingly, the execution traces of interpreted programs contain more instructions correspond to the translation of the program's instructions into machine code.* □

That is to say, the only different is that the execution of the interpreted instructions takes more time because their execution is under the control and management of the interpreter which is just a form of software component (or collection of components) under the EMM. Other than that, the execution of these instruction follows the same general process as the instruction execution discussed previously.

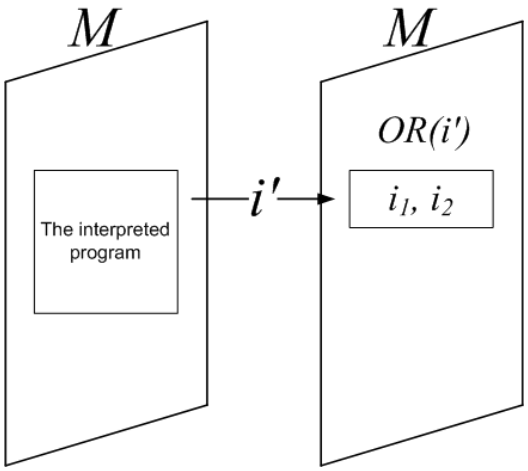


Figure 3.10: Example of execution of an interpreted instruction.

3.8.4 Modeling Self-Modifying Code

Self-modifying code is used to hide the internal constructs of programs to protect, for example, the intellectual property they contain [108]. However, viruses also can use self-modifying code to hide their malicious functions. Within this subsection, we will discuss the modeling of this type of code via the EMM. It should be noted that, the objective of this work is not to propose any approaches or new models for self-modifying code. In particular, the objective of this subsection is to show that this type of code can be represented in the proposed EMM.

Without loss of generality, in the model of the stored-program computers, program's instructions and data are held in a single storage structure [108]. Because of this, the program's code can be treated as data and, hence, can be read and written by the code itself. Accordingly, in the EMM, the execution of self-modifying code instructions simply follows the EMM's standard instruction execution model except that the output regions of the self-modifying code instructions are the memory regions of the component itself, as discussed in the following example.

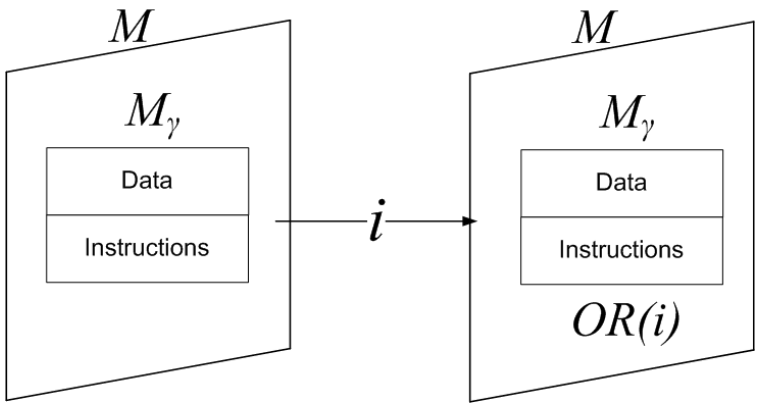


Figure 3.11: The execution of self-modifying code.

Example 3.7. *Consider a software component γ. As discussed in Section 3.5.4, γ is located in the memory at M_γ. Since M_γ contains the data and instructions of γ, then M_γ can be partitioned into two regions: one for the data and the other for the instructions, as shown in Figure 3.11. Hence, in general, if $i \in \gamma$ is a self-modifying code instruction, then $M_\gamma \cap OR(i) \neq \emptyset$, as shown in Figure 3.11. More generally, $M_\gamma \cap OR(\gamma) \neq \emptyset$ (*i.e.*, the component's memory region intersects with the program's output region).*

Accordingly, self-modifying code can be expressed in the EMM as instructions can have output regions that at future times are then also fetched as next instructions.

3.8.5 Modeling Computer Networks

Finally, we discuss the use of the EMM to model computer networks. Consider a network that consists at time t of $1 < N_{net}(t) < \infty$ computers. To model this network, we will first define an EMM (Definition 3.3) for each computer. Then the network can be simply defined as the union over all of these individual EMMs. Hence, the network will be defined as $EMM_{net}(t) = \langle M_{net}(t), \mathbb{I}_{net}(t), \mathbb{S}_{net}(t), B_{net}(t), \mathcal{C}_{net}(t), \mathbf{\Gamma}_{net}(t), \Pi_{net}(t) \rangle$ such that:

- $M_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} M_j$,
- $\mathbb{I}_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} \mathbb{I}_j$,
- $\mathbb{S}_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} \mathbb{S}_j$,
- $B_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} B_j$,
- $\mathcal{C}_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} \mathcal{C}_j$,
- $\mathbf{\Gamma}_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} \mathbf{\Gamma}_j$, and,
- $\Pi^*_{net}(t) = \bigcup_{j=1}^{N_{net}(t)} \Pi^*_j$.

That is to say, the memory of the network model is the union of the memories of all computer models and the same applies for all modules of the model. No assumption is made regarding single system clock as each computer may have its own clock where clock synchronization mechanism may or may not be used to synchronize the clocks of the network nodes (*e.g.*, [109]). Note that, time is still viewed as discrete. Even without clock synchronization, as long as $N_{net}(t) < \infty$, there will still always exist $\epsilon > 0$ such that ϵ is less than the minimum offset between any two clock ticks of any two systems in $N_{net}(t)$. Therefore, ϵ can be used as the theoretical reference time frame for the constructed composite EMM network model. Within this networked EMM view, all entities that can change information within the network are viewed as computers with respect to the EMM. Therefore, routers and other network equipments are viewed as special purpose computers within the EMM as, by definition, they contain EMM's control units.

Clearly, this network based EMM extension can be applied to both wired and wireless networks as well as mixed networks. The only limit in scale is that $|N_{net}(t)| < \infty$ for all $t \in \boldsymbol{T}$. Hence, the EMM can be applied to model modern enterprise and critical infrastructure networks that have wireless elements and sensors that come and go

(*e.g.*, modern smartphones and tablets) as well as stand-alone embedded devices (*i.e.*, the EMM spans the continuum of modern computer network deployment). However, it should be emphasized that, the EMM is limited to only the modeling of digital information and, therefore, it cannot be used for example to model wireless jamming attacks or other attacks that exist within analog domains (*i.e.*, as a core result of the definition of B within the Maurer model).

3.9 Summary

In this chapter, the details of the extensions needed for developing the EMM from the Maurer model have been discussed. In particular, in Section 3.3, the various components of the memory have been discussed. The input and output interfaces were defined to enable the system to model and reason about the information flow to and from the system, which is necessary to model security related concerns. In Section 3.4, the modeling of multiple control systems was discussed, which enables the EMM to model modern multi-processor and multi-core systems. The detailed definitions of programs and their execution have been introduced in Section 3.5. The definitions of perfect security policies have been introduced in Section 3.6. With these extensions, the EMM is able to model different aspects of modern programs and their executions within the a defended environment. In particular, the EMM can be used to capture how programs interact with different memory regions during their executions. Finally, within Section 3.8, the model has been shown to retain the Turing equivalence property of the original Maurer model and has been also shown to be able to model wired and wireless computer networks.

Chapter 4

Modeling Security Policies and Malware

In Chapter 3, the components of the EMM were developed. In this chapter, the EMM will be used to model security policies violations and, therefore, common classes of malware, *i.e.,* where the modeling of the security policy violations becomes the basis for defining malware behaviors within the EMM.

More specifically, within the EMM, malware is modeled as software components whose execution traces cause *integrity*, *availability*, or *confidentially* violations within the given modeled system. In addition, to enable the more general modeling of *bots*, the unauthorized usage of the system resources will also be considered (*i.e.*, assuming that, the bots are structured so as not to cause integrity, confidentiality, or availability violations of systems that they run on and only consumes system's available resources without authorization). These standard security policy violations are innately *information-centric* in that they deal with:

(i) When and how information is passed out from a computer system to the external world,

(ii) How information changes over time,

(iii) The time periods over which modifications to information occur, and,

(iv) Whether system resources (*e.g.*, control units, memory, *etc.*) should be allocated to support particular information processing tasks.

As will be highlighted, the EMM enables the modeling of the above standard security violations viaits theoretical construct of perfect security policies (Π^*).

It should be noted that, a policy-based definitions of malware cannot be avoided. The reason is that, in many cases, the execution of benign components can cause information changes that are *identical* to what may be considered malware in different contexts. In such cases, the distinction between normal components and malware can at least be theoretically defined in terms of whether a *perfectly and completely knowledgeable defender* would authorize the component to execute given full knowledge of its past and future information set. Such perfectly informed authorization policies of course do not and cannot exist in the real-world. But such definitions are of use theoretically to arrive at the necessary formal definitions of what constitutes malware.

The remainder of this chapter is organized as follows. Section 4.1 proposes formal definitions of example violations of the different security policies within the context of the EMM. Section 4.2 introduces example EMM formal definitions for a number of common malware classes via their enacted behaviors. Section 4.3 discusses the completeness of the EMM, where it shows that the model can be used to model any execution of malware. Finally, Section 4.4 summarizes the chapter.

4.1 Modeling Security Policies Violations

In this section, the violation of the standard security policies under the EMM will be formally defined. In particular, the violations of confidentiality, integrity, availability and resource usage policies will be discussed. It should be noted that, we are only concerned with violation of policies with respect to the EMM's modeling of measurable digital information. For example, assume an authorized employee in a company opened a confidential file on the computer and then used a camera to take a photo of its contents as displayed on the screen. This employee then sells this photo to the company's competitors. The employee has clearly violated the company's confidentiality policy. However, the incident is also clearly outside the scope of the EMM's modeled and, hence, measurable information sets. Hence, such attacks are outside the scope of attacks for which the EMM is intended to reason about.

It should be also noted that, the term *policy violation* as used in English is a broad term that includes many scenarios covering a variety of possible formal definitions. Accordingly, the exact nature of how violations of different security policies can occur within tee mathematics of the EMM can vary widely. Hence, we are not trying to propose unique general definitions for each form of policy violation. Instead, our objective is to demonstrate the ability of the EMM to model such violations. For example, as will be seen, Definition 4.1 does not uniquely cover all possible definitions of confidentiality policy violations. Specifically, Definition 4.1 covers only cases where an unauthorized process reads a confidential information. Cases where a user that is authorized to read the information but not authorized to copy it to another location are not covered. Hence, at this stage, we are only seeking to introduce sample behavioral scenarios of how the EMM can model policy violations, whereas Section 4.3 addresses the completeness of the EMM.

4.1.1 Modeling Confidentiality Policies Violations

Confidentiality policies are also known as *Information Flow Policies* [104, pp. 97]. In general, they are used to define the authorized entities (*i.e.*, users, processes, *etc.*) that have the right to access and use confidential information. In addition, confidentiality policies should also clearly define the system states leading to confidential information leaks to the unauthorized entities. As discussed in [104, pp. 97], these states include not only the leakage of the rights to unauthorized parties but also the illicit transmission of confidential information without an accompanying leakage of rights.

Within the EMM, one typical form of violating the confidentiality policies can be described and modeled as follows. The confidentiality policy associated with a set of information can be viewed as having been violated when this information is read by an unauthorized software component. Hence, let $dynamic(\gamma_P, \tau)$ denote the execution of the composite component γ_P during the time period $\tau \subseteq \boldsymbol{T}$ in the EMM. Additionally, let $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ be a memory region that contains a set of information that is confidential with respect to γ_P (*i.e.*, γ_P is unauthorized to read the information). Consequently, γ_P violates the confidentiality of $N(\tau)$ if it partially or totally *reads* the information contained in it (*i.e.*, if and when a subset of $N(\tau)$ becomes a subset of the input region of γ_P) and there did not exist any security policy allowing this read access. Formally, such an EMM confidentiality violation is described by Definition 4.1 as follows.

Definition 4.1 (Confidentiality Violation)**.** *Consider an EMM that has* $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ *that contains confidential information, such that,* $N(\tau)$ *is deemed to be confidential by the defender with respect to any given composite software component* γ_P *during the time period* $\tau \subseteq \boldsymbol{T}$*, where* $N(\tau) \neq \emptyset$*. Then, the execution of* γ_P *during*

the time period τ *(*i.e.*, dynamic*(γ_P, τ)*) violates the confidentiality of* $N(\tau)$*, if the following conditions occur:*

(i) $N(\tau) \cap \boldsymbol{IR}(\gamma_P, \tau) \neq \emptyset$*, and,*

(ii) $\nexists \pi \in \Pi^*$ *such that* $\pi[dynamic(\gamma_P, \tau))] = 1$.

Where Π^* is as defined in Section 3.6. As shown above, Definition 4.1 provides a basic form of the confidentiality policies violations as represented within the EMM. Clearly, other classes of confidentiality violation can be developed similarly. Again, the use of Π^* in such definitions is required as this provides the necessary theoretical ground-truth required to differentiate between attack and normal behavior with both perform Definition 4.1-(i) step.

4.1.2 Modeling Integrity Policies Violations

Without loss of generality, integrity policies are used to clearly identify the entities that are authorized to alter specific elements of stored information. In addition, these policies should also identify the authorized ways with which all authorized entities are allowed to alter the information [104, pp. 97]. Accordingly, the violation of the integrity policies occurs due to the alteration of the information by unauthorized entities or in unauthorized ways.

Within the EMM, a typical form of the violation of an integrity policy is described and modeled as follows. Since the integrity policy of a set of information is violated when it is modified by an unauthorized software component, then again consider a memory subset region $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ where $\tau \subseteq \boldsymbol{T}$ holding information whose integrity is to be protected. Consider also the composite component γ_P which is not authorized to alter the information stored in $N(\tau)$. Then, γ_P clearly violates the integrity of $N(\tau)$ during any execution during a time period τ if its output region

intersects with $N(\tau)$. Formally, the above violation of an integrity policy scenario is described by Definition 4.2 as follows.

Definition 4.2 (Integrity Violation)**.** *Consider an EMM in which* $N(\tau) \subseteq \boldsymbol{M}(\boldsymbol{T})$ *has been denoted as an information set that should not be altered by any composite software component* γ_P*'s execution where* $\tau \subseteq \boldsymbol{T}$ *and* $N(\tau) \neq \emptyset$*. Then, the execution of* γ_P *during* τ *(*i.e.*,* $dynamic(\gamma_P, \tau)$*)* violates the integrity *of the information contained in* $N(\tau)$ *if the following conditions apply:*

(i) $N(\tau) \cap \boldsymbol{OR}(\gamma_P, \tau) \neq \emptyset$*,*

(ii) $\nexists \pi \in \Pi^*$ *such that* $\pi[dynamic(\gamma_P, \tau))] = 1$*.*

Definition 4.2 reflects the ability of the proposed EMM to model this example of violations of the integrity policies, where again other classes of integrity policy violations could also be modeled by the EMM in a similar manner.

4.1.3 Modeling Availability Policies Violations

Without loss of generality, availability policies specify what system resources are to be allocated to entities and processes [104, pp. 97]. A program that acquires a resource during its execution must also release that resource within a specified maximum time after its execution begins, where such time periods are defined explicit or implicit parameters. A subset of these types of policies are the *bounded availability policies*, which specify that if a program acquires a resource during one of its executions, then it must release that resource by *some fixed finite point* later in that execution [110,111].

In the EMM, one typical form of violating the availability policy of a system resource can be described as follows. Let $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ be a memory region that denotes a given system resource. Let N_1 be the content of $N(\tau)$ which denotes that

the resource is acquired during τ and N_2 be its content which denotes that the resource is released during τ. Then, the process of acquiring and releasing the resource can be defined by the map $([\![N]\!] = N_1) \xrightarrow{\tau'} ([\![N]\!] = N_2)$, where $[\![.]\!]$ denotes the contents of a memory region and $\tau' \subset \tau$ is the time period during which the component is authorized to hold the resource. Hence, the execution of a composite component γ_P violates the availability of the resource N during its execution in the time period τ if it acquired the resource and did not release it later during the execution (*i.e.*, did not implement the map at all or in a timely manner). Formally, this form of availability violation can be modeled as described in Definition 4.3 as follows.

Definition 4.3 (Availability Violation)**.** *Let $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ denote a specific resource in an EMM. Let $[\![N]\!] = N_1$ and $[\![N]\!] = N_2$ be the contents of N which denote the acquiring and releasing of N, respectively during τ. Then, the execution of a composite software component γ_P during the time period $\tau \subset \boldsymbol{T}$ (*i.e.*, $dynamic(\gamma_P, \tau)$) violates the availability policy of N if the following conditions apply:*

(i) $\exists t_1 \in \tau$ *such that* $N(t_1) \subseteq OR(dynamic(\gamma_P, t_1))$ *and* $[\![N(t_1)]\!] = N_1$,

(ii) $\nexists t_2 > t_1 \in \tau$ *such that* $N(t_2) \subseteq OR(dynamic(\gamma_P, t_2))$ *and* $[\![N(t_2)]\!] = N_2$, *and,*

(iii) $\nexists \pi \in \Pi^*$ *such that* $\pi[dynamic(\gamma_P, \tau))] = 1$.

As shown in Definition 4.3, the availability policy of the resource will be violated when the composite component acquires the resource during its execution and does not release it in a timing manner. Note that, the violation of bounded availability policies can be trivially defined by restricting the time period $[t_1, t_2]$ during which the component acquires and releases the resource to be larger than the τ' that denotes the upper bound on the time period required to release the resource. Again, other forms of availability policy violations can be modeled within the EMM by similar approaches.

4.1.4 Modeling Resource Usage Policies Violations

In general, the role of the resource usage policies is to specify the software components that are authorized to access specific system resources (*e.g.*, CPUs, network interfaces, memory, *etc.*) [112,113]. Those policies are important as they are necessary to model general *bot* behavior which otherwise does not violate confidentiality, integrity or availability policies. More specifically, it will be assumed that bots do not attack the systems on which they run as they are used by attackers primarily to remotely influence other systems (*e.g.*, *distributed denial of service* (DDoS) attacks, sending *spam* emails, *etc.*) [114, Section 2.1.10, pp. 18]. Hence, the main violation general bot causes to local systems would be a resource usage violation.

Within the EMM, the modeling of a resource usage violation is similar to the modeling of violations of the availability policies. However, in this case just implementing the map by the unauthorized software component is the violation. Hence, a resource will be modeled as a memory subset region $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$. Additionally, the process of acquiring and releasing this resource will be modeled as the mapping $([\![N]\!] = N_1) \xrightarrow{\tau'} ([\![N]\!] = N_2)$. Then, an unauthorized composite component γ_P violates the resource usage policy of $N(\tau)$ if it implements a mapping that denotes an unauthorized usage of $N(\tau)$. Formally, this form of resource usage violation can be described by Definition 4.4 as follows.

Definition 4.4 (Authorized Resource Usage Violation)**.** *Let $N(\tau) \subset \boldsymbol{M}(\boldsymbol{T})$ denote a specific resource in the EMM and let $\tau, \tau' \subseteq \boldsymbol{T}$. Also, let dynamic$(\gamma_P, \tau)$ denote the execution of a composite component γ_P during the time period τ where $\tau' \subseteq \tau$ in which γ_P is unauthorized to access N. Then, γ_P violates the resource usage policy of $N(\tau)$ if the following conditions apply:*

(i) $dynamic(\gamma_P, \tau)$ *implements the mapping* $(\llbracket N \rrbracket = N_1) \xrightarrow{\tau'} (\llbracket N \rrbracket = N_2)$ *(*i.e.*,* $\exists t_1, t_2 \in \tau, t_2 > t_1$ *such that* $N(t_1) \subseteq OR(dynamic(\gamma_P, t_1))$ *where* $\llbracket N(t_1) \rrbracket = N_1$ *and* $N(t_2) \subseteq OR(dynamic(\gamma_P, t_2))$ *where* $\llbracket N(t_2) \rrbracket = N_2$*), and,*

(ii) $\nexists \pi \in \Pi^*$ *such that* $\pi[dynamic(\gamma_P, \tau))] = 1$.

As shown above in Definition 4.4, the unauthorized component violates the resource usage policy of the resource even if it does not violates its availability policy. Note that, the violation as per this definition only occurs once the unauthorized component has acquired the access to the resource. Again, the EMM can be used in a similar manner to define other types (or classes) of resource usage policy violations.

4.1.5 The Consistency of Π^*

In the previous sections, it has been shown that the EMM can be used to define and model various example violations of basic security policies across the domains of confidentiality, integrity, availability, and resource usage violations. As discussed in [110, 111], the violation of any single security policy $\pi \in \Pi^*$ under the EMM is deemed to imply a violation of the system's set of perfect security policies Π^*. Hence, the formalization of this concept is provided in Definition 4.5 as follows.

Definition 4.5. *For the execution of a composite component* γ_P *during a time period* τ *within the EMM, we have that*

$$\Pi^*[dynamic(\gamma_P, \tau)] = \min \{\pi \left(dynamic[\gamma_P, \tau]\right) | \forall \pi \in \Pi^*\}.$$

Accordingly, $\Pi^*[dynamic(\gamma_P, \tau)] = -1$ if there exists any $\pi \in \Pi^*$ such that we have $\pi \left(dynamic[\gamma_P, \tau]\right) = -1$. Hence, Π^* can be seen to provide a mechanism by which a *perfectly informed* defender can denote whether trace-level execution behaviors are or are not authorized on case-by-case basis.

It should be noted that, the approach of defining malware in terms of *malicious intent* is not followed within the definition of Π^*. This is because the attacker's *intent* is not a directly measurable quantity by the defenders. Instead, such intent exists within the mind of the attackers. The desire to formally quantify security requires a need to focus security constructs and models that are restricted to only those aspects which can be measured by the defender (*i.e.*, in this case within the EMM modeled information sets). Clearly, Π^* as defined is useful in formally determining what is and what is not malware (*i.e.*, providing a theoretical ground-truth). Finally, Π^* as defined must also denote an informed choice by the defender, which leads to the following necessary regularity (or consistency) condition formalized in Definition 4.6 as follows.

Definition 4.6 (The consistency of Π^*)**.** *Consider an EMM. Let γ_P and $\gamma_{P'}$ be two composite software components modeled within the EMM. For any two time periods $\tau, \tau' \subset \boldsymbol{T}$ where $dynamic(\gamma_P, \tau), dynamic(\gamma_{P'}, \tau') \neq \emptyset$, if $dynamic(\gamma_P, \tau) = dynamic(\gamma_{P'}, \tau')$, then it* must *be the case for which we have $\Pi^*[dynamic(\gamma_P, \tau)] = \Pi^*[dynamic(\gamma_{P'}, \tau')]$. Similarly, if we have the case for which $\Pi^*[dynamic(\gamma_P, \tau)] \neq \Pi^*[dynamic(\gamma_{P'}, \tau')]$ then it* must *be the case that $dynamic(\gamma_P, \tau) \neq dynamic(\gamma_{P'}, \tau')$.*

As shown in Definition 4.6, under the EMM, Π^* must be, such that, it provides consistent results when given identical information sets. Hence, for example, if $x = y$ then $\Pi^*[x] = \Pi^*[y]$, and if $\Pi^*[x] \neq \Pi^*[y]$ then $x \neq y$ where x, y denoting two arbitrary information sets. Note that, the above definitions for the violations of different security policies highlight the ability of the EMM to reason about computer security. Finally, it should be noted that the EMM can also be used to model *execution monitors* (EMs) which are mechanisms that monitor the execution of programs and enforce security policies [110, 115]. Clearly, these EMs can be modeled within the EMM following the same manner used to model VMs (*i.e.*, via a hierarchical EMM

structure), as was discussed in Chapter 3. What the above sections do not show is that the EMM is capable of modeling any possible implementable security policy definitions, as this property follows directly from the EMM being shown to be complete with respect to its ability to model all implementable malware and malware detection solutions.

4.1.6 Discussion

As discussed in this section, the system's set of perfect security policies Π^* provides a theoretical construct to describe and define various security and malware attacks in terms of the violations of the security policies. In fact, Π^* can be used as an assessment tool to assess the system's theoretical level of security with respect to different types of attacks. Additionally, from practical point of view Π^* most closely describes an *oracle* that gives perfect security decisions over the sets of presented information, while ensuring consistent decisions are given similar information sets.

Finally, known models of security policies could also be represented by the EMM in a similar manner. For example, consider *Bell-LaPadula* model [104, Section 5.2, pp. 124], which corresponds to military-style information classifications and control. In particualr, Bell-LaPadula model defines a set of *security clearances* for the sets of confidential information, where the sets of information is denoted as the *objects*. The security clearances are arranged in a linear ordering, where the higher the security clearance, the more sensitive the information. Bell-LaPadula model also defines clearances for the *subjects* (*i.e.*, the users or processes that access the information). Hence, based on various clearance levels of the subjects and objects, the access to different objects can be granted or denied. Clearly, Bell-LaPadula model can be described within the EMM by specifying the necessary constructs associated with whether a component initiated by a specific user can read specified confidential information

based on the clearances of both the subject (the user) and the object. It should be noted that, in this case, we are defining a system's set of operational security policies Π and not the set of perfect security policies Π^*. Hence, the notion of the perfect set of security policies Π^* can be used as a necessary theoretical construct required to formally reason about operational (or implementable) security policies and their control.

4.2 Malware Modeling

In this section, the use of the EMM to model example classes of common malware categories will be explored. In general, malware is defined as "*any program that has offensive features and/or purposes without the users' awareness, and whose aim is either: (1) to affect the confidentiality, availability and integrity of the system, or (2) to wrongly incriminate the system's users in the realization of a crime or an offense*" [116, Section 34.1, Definition 1, pp. 748]. The first part of the definition refers to the portion of malware that directly attacks the security of the computers upon which it runs (*e.g.*, viruses, Trojan horses, *etc.*). Whereas, the second part refers to the malware that only benefits from using the resources of the hosting computers (without the owner's awareness) to attack external entities (*e.g.*, as in the general case of *bots*). Another definition of malware which ties it directly to the system's security policy is: "*malware is a set of instructions that cause the system's security policy to be violated*" [104, Definition 22.1, pp. 613]. According to this definition, any software component that violates the system's security policy can then be reasonably considered to be malware.

Without loss of generality, there are different categories of malware that vary in their execution behaviors (*e.g.*, *viruses*, *worms*, *Trojan horses*, *spyware*, *etc.*). As

discussed in [55, 57, 60, 78, 117], a number of different formal definitions of some of these malware categories have been proposed. However, these definitions tend to focus on either specific malware categories (in particular, viruses) or are based on specific mathematical machines and, as a result, they are not fully representative of modern malware as discussed earlier in details in Chapter 2.

From the perspective of the EMM, consider the execution of the composite component γ_P during a time period $\tau \subseteq \boldsymbol{T}$. Then, $dynamic(\gamma_P, \tau)$ defines γ_P's generalized and full set of dynamic information. Hence, $dynamic(\gamma_P, \tau)$ denotes the execution of malware if at least one of the following actions occurs:

- $dynamic(\gamma_P, \tau)$ causes confidentiality violation(s) to some information stored within the EMM. A number of malware classes are known to cause confidentiality violations (*e.g.*, *spyware*).

- $dynamic(\gamma_P, \tau)$ causes integrity violation(s) for some information stored within the EMM. A number of malware are known to cause integrity violation (*e.g.*, *viruses*).

- $dynamic(\gamma_P, \tau)$ causes availability violation(s) of some information or resources within the EMM. The malware that causes an availability violations typically consumes the resource of the computer (*e.g.*, *rabbits*) without otherwise violating integrity or confidentiality policies.

- $dynamic(\gamma_P, \tau)$ causes resource usage violation for the system resources (*e.g.*, *bots*) without otherwise violating integrity, confidentiality, or availability policies.

In general, the most commonly agreed on malware classes are viruses, worms, spyware, bots and Trojan horses [91,104,116]. It should be noted that, single malware variants can contain these features from across multiple malware classes. For example,

a replicating virus can carry a spyware payload (*i.e.*, hard boundaries no longer generally exist across these categories as malware instances commonly carry features of multiple categories). Throughout this section, the modeling of malware classes will be based on the key features that denote that class.

Note also that, the malware behavior can therefore vary considerably even within the same malware category. Hence, it should be emphasized that, the objective of this section is not to develop a unique definition of each malware category but to demonstrate the ability of the proposed EMM to model various classes of malware. Hence, the discussions that follow focus on denoting example classes of malware behavior via the EMM's constructs. The following sections introduce the formal modeling of a number of common malware classes to demonstrate the ability of the EMM to model these different malware classes. Again, these definitions should not be taken as the universal and complete definitions for any given malware class, as the multi-class nature of modern malware render such definition untenable.

4.2.1 Modeling Computer Viruses

In general, a computer virus is defined as "*a program that inserts itself into one or more files (programs) and then performs some (possibly null) action*" [104, Definition 22.4, pp. 616]. The infected program is usually denoted as the *host*. In turn, when the host is executed, the infection then spreads further into other files. Viruses are not *self-propagating* malware and they cannot infect files on other computers without active intervention. A virus in its original form before its first infection is denoted as a *germ* [118, Section 2.3.5]. Viruses were the first malware to be formally modeled when Cohen used Turing machines [55]. Generally, the virus infection cycle has the following phases:

(i) *Replication (insertion)*: In this phase, upon the execution of the virus, the virus inserts itself into the target file(s). All viruses implement some replication mechanism which distinguishes them from other malware categories.

(ii) *Activation (execution)*: In this phase, the virus launches its attack by executing its payload. In general, viruses can carry a wide range of payloads which are usually malicious.

Note that, viruses can be classified along many dimensions (*e.g.*, the infection strategies, the payload type, *etc.*) [118], and hence, there exist various definitions of viral behaviors that differ from each other (*e.g.*, boot sector infectors differ from executable infectors, *etc.*). However, all viruses have the distinctive feature of replication. The formalization of an example of a viral replication within the EMM is defined in the next section. This EMM formalization focuses on capturing the replication phase required by a viral component if it is to infect another component.

4.2.1.1 Formal Modeling of Viruses

Consider an EMM and let $dynamic(\gamma_P, \tau)$ denote the execution of a software component γ_P during the time period $\tau \subseteq \boldsymbol{T}$. Then, γ_P behaves like a computer virus if it partially or totally copies itself into another component, $\gamma_{P'} \neq \gamma_P$. Note that, the copied version of the virus may be mutated (*i.e.*, the virus may use obfuscation techniques and replicates an evolved copy). This viral behavior of γ_P is formalized in Definition 4.7 as follows.

Definition 4.7 (Viral Behavior)**.** *Let $dynamic(\gamma_P, \tau)$ denote the execution of γ_P over the time period $\tau \subseteq \boldsymbol{T}$ in an EMM. Let $M_{\gamma_P}(\tau)$ be the memory region of γ_P. Then, $dynamic(\gamma_P, \tau)$ exhibits a viral behavior if the following conditions apply:*

(i) $\exists M_v \neq \emptyset$ such that $M_v \subseteq M_{\gamma_P}(\tau)$ and $M_v \subseteq \boldsymbol{IR}(\gamma_P, \tau)$,

(ii) $\exists M_{v'} \neq \emptyset$ *such that* $M_{v'} \subseteq M_{\gamma_{P'}}(\tau)$ *and* $M_{v'} \subseteq \boldsymbol{OR}(\gamma_P, \tau)$,

(iii) $M_{v'} = \boldsymbol{AR}(M_v, \gamma_P(\tau))$, *and,*

(iv) dynamic(γ_P, τ) *causes a security policy violation under* Π^* *(*i.e.*, the violation of the integrity of* $\gamma_{P'}$*).*

4.2.1.2 Discussion

As shown in Definition 4.7, the execution of γ_P exhibits a viral behavior if there exists a subset memory region within γ_P that is copied into the host $\gamma_{P'}$ in a way that violates the system's security policies. Fundamentally, Definition 4.7 of the viral behavior is a special case of the integrity policy violation introduced in Definition 4.2, as it is the integrity of the host program $\gamma_{P'}$ that has been violated. In particular, Definition 4.2 presents an overview of the integrity violation of the contents of a general memory region by the execution of a software component. Whereas, Definition 4.7 specifies the details of the integrity violation in terms of the alteration of the contents of the memory region(s) associated with $\gamma_{P'}$.

Finally, we should emphasize that, Definition 4.7 defines only an example subset of the viruses which are the executable viruses which infect other executable programs during their execution. For example, the cases of document and macro viruses (*i.e.*, *embedded viruses*[1]) are not covered by the above definition. Again, the objective is to demonstrate the ability of the EMM to formally model viruses not to propose a single complete definition inclusive of all viruses as the later cannot be achieved due to the wide span of features of various virus subcategories. Moreover, as the copied virus could be packed or encrypted, the copied memory region of γ_P need

[1]In these kind of viruses, the viral instructions are included in non-executable files and they infect other files and deliver their payload when the viral instructions are read or executed by the programs that are dedicated to open these infected documents [11, 41, 42].

not to describe instructions (*i.e.*, subsets of $\mathbb{I}$) when it is written to $\gamma_{P'}$. Hence, Definition 4.7 expressly does not define the copied information as being subsets of $\mathbb{I}$.

4.2.2 Modeling Trojan Horses

In general, Trojan horses (or simply, *Trojans*) are computer programs with overt (documented or known) effects and covert (undocumented or unexpected) effects. In particular, a Trojan horse is defined as "*a program which purports to do some benign task, but secretly performs some additional malicious task*" [114, Section 2.1.2, pp. 12]. One of the most common functions of a Trojan horse is to create a backdoor for an attacker to gain access to the system [118, Section 2.3.4]. In this work, we will model this backdoor function as a common example of Trojan behaviors.

In order to be able to perform its malicious functionality, a Trojan horse consists of the following two functional modules: a *server* module and a *client* module [91, Section 4.3.2, pp. 100]. The server module is secretly installed at the victim machine in order to grant the attacker the access to that machine. Whereas the client module is installed in an attacker controlled machine to enable the attacker to communicate with the victim machine and to use its resources (both hardware and software resources) via the established connection to the Trojan server module that is active on the victim machine.

During its operation, a Trojan horse first receives the attacker's commands through an external input interface (typically, through the network interface), then executes the received commands, and sends back the results of the execution through an output interface (typically, the network interface). Hence, the basic operations of the Trojan horse can be summarized in the following steps:

(i) Reading information (received commands) from a memory region that belongs to an input interface,

(ii) Executing these received commands to directly affect the security of the target system, and,

(iii) Writing the results of the commands execution to a memory region that belongs to an output interface.

Note that, more generally, Trojans can also cause confidentiality, integrity, and/or availability violations.

4.2.2.1 Formal Modeling of Trojan Horses

Trojan behaviors under the EMM denote, most generally, the component composition γ_P that accepts commands from the external world and sends information off the computer back into the external world. The formalization of an example Trojan behavior under the EMM is described in Definition 4.8 as follows.

Definition 4.8 (Trojan behavior)**.** *Let* $dynamic(\gamma_P, \tau)$ *denote the execution of the composite software component* γ_P *over the time period* $\tau \subseteq \boldsymbol{T}$ *within an EMM. Let the subset* $\boldsymbol{M}_n^{\Theta}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Theta}(\boldsymbol{T})$ *denote the network input interface during* $\boldsymbol{T}$*. Also, let the subset* $\boldsymbol{M}_n^{\Phi}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Phi}(\boldsymbol{T})$ *denote the network output interface during* $\boldsymbol{T}$*. Then,* γ_P *exhibits the Trojan behavior discussed above during its execution if the following conditions apply:*

(i) $\exists M_c(\tau) \neq \emptyset$ *such that* $M_c(\tau) \subseteq \boldsymbol{EIR}(dynamic(\gamma_P, \tau)) \bigcap \boldsymbol{M}_n^{\Theta}(\boldsymbol{T})$,

(ii) $\exists M_r(\tau) \neq \emptyset$ *such that* $M_r(\tau) \subseteq \boldsymbol{EOR}(dynamic(\gamma_P, \tau)) \bigcap \boldsymbol{M}_n^{\Phi}(\boldsymbol{T})$,

(iii) $M_r(\tau) = \boldsymbol{AR}(M_c(\tau), trace(\gamma_P, \tau))$*, and,*

(iv) $dynamic(\gamma_P, \tau)$ *causes integrity, confidentiality, and/or availability violations under* Π^*.

4.2.2.2 Discussion

Unlike Definition 4.7 of viruses which focuses on capturing the viral replication mechanism, Definition 4.8 of Trojan horses focuses on modeling their functionality. In particular, in Definition 4.8-(i), $M_c(\tau)$ represents the commands that the Trojan receives from the network interface through the backdoor. Whereas, in Definition 4.8-(ii), $M_r(\tau)$ represents the results of the execution of these commands which is then sent via the network interface to the attacker. Consequently, in Definition 4.8-(iii), $M_r(\tau)$ is defined as the affected region of $M_c(\tau)$ under the execution of the Trojan. Finally, in Definition 4.8-(iv), the execution of the Trojan violates at least one of confidentiality, integrity, and/or availability policies depending on its malicious payload. Again, it should be noted that other Trojan behaviors may also occur that are outside of the context of the above specific behavioral definition. Moreover, it should be recalled that under the EMM, it is assumed that output peripherals immediately act on information written to their memory regions. Hence, Definition 4.8-(ii) suffices formally to have the Trojan send the information to the attacker.

4.2.3 Modeling Spyware

Broadly speaking, as discussed in [118, Section 2.4.3], spyware programs collect information from computers and then transmit it to other remote persons through the network without the users' awareness or consent. Spyware silently monitors the behavior of users and, for example, records their web surfing habits, and/or steals their sensitive information (*e.g.*, passwords, bank accounts information, credit card numbers), *etc.* [119]. Hence, by definition, spyware is characterized by violating the user's confidentiality. It should be noted that, as discussed in [120], almost 80% of malware infections is due to spyware.

From a functional perspective, as discussed in [119,120], the distinctive functional characteristics of spyware are:

(i) It collects information about the user or his/her behaviors, and,

(ii) It forwards this information to a third party, possibly after being processed (*e.g.*, encrypted).

4.2.3.1 Formal Modeling of Spyware

A typical example of spyware behavior is when an unauthorized entity accesses sensitive information and then sends it off of the computer through the network to a remote receiver. This behavior is modeled in the EMM as follows. A composite software component γ_P exhibits spyware behavior if, during its execution, it conducts the following two actions:

(i) Unauthorized reading of the information stored in some memory regions (*i.e.*, it violates the confidentiality of some memory regions). That is to say, these memory regions became subsets of the set of input regions of γ_P.

(ii) Unauthorized writing of this information to some output interfaces (*e.g.*, network interface cards, *etc.*) such that it is then sent off the computer through the output devices. That is to say, this accessed information becomes subsets of the set of output regions of γ_P. Note that, the spyware may process the information before sending it off the computer (*e.g.*, by using encryption, compression, *etc.*).

The formalization of this spyware behavior through the EMM is modeled in Definition 4.9 as follows.

Definition 4.9 (Spyware behavior)**.** *Let dynamic*(γ_P, τ) *denote the execution of the composite software component* γ_P *during the time period* $\tau \subseteq \boldsymbol{T}$ *within an EMM.*

Let $M_s(\tau) \neq \emptyset$ be the set of memory regions that are deemed to be confidential by the system defender during τ. Let the subset $\boldsymbol{M}_n^{\Phi}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Phi}(\boldsymbol{T})$ be the subset of the output interfaces that is concerned with the network output interface. Then, γ_P exhibits spyware behavior during its execution if the following conditions apply:

(i) $M_s(\tau) \subseteq \boldsymbol{IR}(\gamma_P, \tau)$,

(ii) $AR(M_s(\tau), trace(\gamma_P, \tau)) \subset \boldsymbol{M}_n^{\Phi}(\boldsymbol{T})$, *and,*

(iii) $dynamic(\gamma_P, \tau)$ *causes confidentiality violations under* Π^*.

4.2.3.2 Discussion

It should be noted that, one important difference between Trojans and spyware is the region of the unauthorized reading. In particular, in the case of a Trojan, the unauthorized reading is through the network interface, through which, the Trojan receives the attacker's commands. Whereas in the case of spyware, the unauthorized reading occurs for confidential information contained within the EMM as indicated in Definition 4.9-(i). This information is then leaked by the spyware to the outside world, as illustrated by Definition 4.9-(ii). In addition, the use of $AR(M_s(\tau), trace(\gamma_P, \tau))$ instead of directly setting $M_s(\tau)$ in Definition 4.9-(ii) allows the spyware definition to include the cases for which the spyware processes the information before leaking it outside the system (*e.g.*, by using encryption, compression, *etc.*). Again, other definitions of spyware behaviors can also be described via the EMM, with the above definition being just one such example.

4.2.4 Modeling Bots

In general, a bot is defined as "*a compromised computer system that is used by the attacker to perform a variety of malicious tasks without the owner's awareness*" [114,

Section 2.1.10, pp. 18]. Bots are also denoted as *zombies*. To control their set of bots, attackers use special software kits that are denoted as *command and control* (C&C) kits [121, 122]. The bots that are under the same C&C form a *botnet*. Typically, the number of bots within a single botnet ranges from thousands to millions (*e.g.*, the size of the *Rustock* botnet has exceeded 1 million bots [123], whereas the size of the *Mariposa* botnet has exceeded 12 million compromised IP addresses [124, 125]).

The most common tasks for bots are in sending *spam* emails and participating in coordinated large-scale *denial-of-service* (DoS) attacks [114, 118]. In general, a bot does not seek to directly impact the security of the system that it runs on. Instead, it uses the system resources (*e.g.*, processors, network resources, *etc.*) to impact the security of another remote system or network [118, 126, 127]. Consequently, bots only generally violate the system's authorized resource usage policies. From the functional perspective, a bot performs three tasks:

(i) It receives information from a remote attacker in an unauthorized way,

(ii) It processes this information in the local machine, and

(iii) It uses the network to carry the required tasks in an unauthorized manner.

4.2.4.1 Formal Modeling of Bots

The bot functionality discussed above is very similar to that of Trojans in that it also receives commands from the attacker through the network interface. Also, the results of executing the attacker's commands is sent through the network interface, however, this is not to the attacker but instead for the purpose of attacking or spaming a third party. However, as indicated above, bots typically would be structured generally to only violate the authorized resource usage polices. Consequently, the bots' formalization within the EMM will be similar to that of Trojans except in the type of their

violated policies. In particular, the formalization of the bot behavior is introduced in Definition 4.10 as follows.

Definition 4.10 (Bot behavior)**.** *Let dynamic*(γ_P, τ) *denote the execution of the composite software component* γ_P *over the time period* $\tau \subseteq \boldsymbol{T}$ *within an EMM. Let the subset* $\boldsymbol{M}_n^{\Theta}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Theta}(\boldsymbol{T})$ *denote the network input interface during* $\boldsymbol{T}$*. Also, let the subset* $\boldsymbol{M}_n^{\Phi}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Phi}(\boldsymbol{T})$ *denote the network output interface during* $\boldsymbol{T}$*. Then,* γ_P *exhibits bot behavior, as discussed above, during its execution if the following conditions apply:*

(i) $\exists M_c(\tau) \neq \emptyset$ *such that* $M_c(\tau) \subseteq \boldsymbol{IR}(\gamma_P, \tau) \bigcap \boldsymbol{M}_n^{\Theta}(\boldsymbol{T})$*,*

(ii) $\exists M_r(\tau) \neq \emptyset$ *such that* $M_r(\tau) \subseteq \boldsymbol{OR}(\gamma_P, \tau) \bigcap \boldsymbol{M}_n^{\Phi}(\boldsymbol{T})$*,*

(iii) $M_r(\tau) = \boldsymbol{AR}(M_c(\tau), trace(\gamma_P, \tau))$*, and,*

(iv) dynamic(γ_P, τ) *causes only authorized resources usage violation and not integrity, confidentiality, and/or availability violations under* Π^**.*

4.2.4.2 Discussion

The difference between Definition 4.10 and Definition 4.8 is in the type of policy violated by the malware. In the case of Definition 4.8 for Trojan behavior, violations to the system's confidentiality, integrity, or availability occur. Whereas in the case of Definition 4.10 for bot behavior, bots potentially only violate the system's resource usage policies. Clearly, other definitions of bot behaviors may also exist where these would be EMM modelable via similar approaches.

4.2.5 Modeling Computer Worms

Without loss of generality, computer worms are defined as: "*programs that copy themselves from one computer to another*" [104, Definition 22.13, pp. 623]. Unlike

viruses, worms are self-propagating malware that can spread themselves (replicate) without the need of any active interaction to reach new targets [104, 114, 128–130]. In addition, worms can launch destructive attacks against other computer systems. In general, as discussed in [128, 130], the cycle of worms infection has the following phases:

(i) *Target finding*: This is the first phase in the life cycle of worms, in which, they search for their potential targets over the network (*i.e.*, by exploiting an operating system or application vulnerability). In general, there are various target finding approaches that worms could implement (*e.g.*, blind scan, hit-list, web search, *etc.*).

(ii) *Transferring* (*replication*): After finding a target, a copy of the worm is then transferred to the target computer through the network. During this and the previous phase, worms are active over the network and, hence, their actions may be detectable by network-based intrusion detection systems.

(iii) *Activation*: In this phase, worms execute their malicious payload on the targeted machines. During this stage, the activities of worms are limited to the local infected computers, and hence, they are only in the detection domain of host-based malware detection systems.

4.2.5.1 Formal Modeling of Worms

Unlike the prior malware behaviors, worm-like behaviors need to be defined within the context of a network, since worms by definition must propagate between computers. Their nature though is similar to viral behaviors except that the copying occurs between the host EMM, and a victim EMM denoted as EMM′ (or a set of

EMMs $EMM_K = \{EMM_k | k = 1, \ldots, K\}$). An example of a worm like behavior is formalized using the EMM in Definition 4.11 as follows.

Definition 4.11 (Worm Behavior)**.** *Let* $dynamic(\gamma_P, \tau)$ *denote the execution of a composite software component* γ_P *during a time period* $\tau \subset \boldsymbol{T}$ *in an EMM. Let* $\boldsymbol{M}_n^{\Phi}(\boldsymbol{T}) \subset \boldsymbol{M}^{\Phi}(\boldsymbol{T})$ *denote the set of memory regions correspond to the network output interface in EMM. Let* EMM' *be another EMM, such that,* $EMM' \neq EMM$*. Let* $\boldsymbol{M}_n'^{\Theta}(\boldsymbol{T}) \subset \boldsymbol{M}'^{\Theta}(\boldsymbol{T})$ *denote the set of memory regions correspond to the network input interface in* EMM'*. Then, the execution of* γ_P*,* $dynamic(\gamma_P, \tau)$*, exhibits a worm behavior if the following conditions apply:*

(i) $\exists M_w(\tau) \subseteq \left[\boldsymbol{OR}(\gamma_P, \tau) \bigcap \boldsymbol{M}_n^{\Phi}(\boldsymbol{T})\right]$,

(ii) $RA(M_w(\tau), trace(\gamma_P, \tau)) \subseteq static(\gamma_P, \tau)$,

(iii) $\exists \tau' \subset \boldsymbol{T}$ *such that* $\min[\tau'] > \max[\tau]$,

(iv) $\exists M_r(\tau') \subseteq \boldsymbol{M}_n'^{\Theta}(\boldsymbol{T})$ *such that* $M_r(\tau') = M_w(\tau)$,

(v) $\exists \gamma_{P'}$ *such that* $\gamma_{P'}$ *is a composite component in* EMM' *and* $static(\gamma_{P'}, \max[\tau']) = M_r(\tau')$*, and,*

(vi) The execution of $\gamma_{P'}$ *in* EMM' *violates its set of policies* Π'^{*}.

4.2.5.2 Discussion

As shown in Definition 4.11, the execution of the worm causes it to copy itself (possibly obfuscated) to the network interface to propagate to the victim node. When it reaches the victim node, the worm becomes a software component within the targeted machine that when executed violates the victim machine's security policies. It should be noted that, Definition 4.11 models the propagation of the worm not the propagation of its payload as the payloads can widely vary. Finally, we emphasize that Definition 4.11

is an example of worm behaviors and there exist other worm behaviors with different propagation details (*e.g.*, bluetooth worms [131], *etc.*). Clearly, the networked EMM model can be used to extend this definition into more complex worm behaviors in which the actions of network-based detection solution could then also be formally reasoned about via the constructed composite networked EMM.

4.3 The Completeness of the EMM

The previous sections have used the EMM to model example security policies' violations and a number of example malware behaviors. In this section, the completeness of the proposed framework with respect to malware modeling is discussed. In particular, the EMM is shown to be able to model *any* malware whose execution within the system(s) it infects produces measurable changes to $\boldsymbol{S}(\boldsymbol{T})$. The completeness of the EMM will be formally developed and proven through Theorem 4.1 as follows.

Theorem 4.1 (The Completeness of the EMM)**.** *The EMM can be used to model* all *malware executions that can occur within an EMM instance and that can be defined in terms of the policy set* Π^*.

Proof. Proof by contradiction. Consider an EMM as defined in Definition 3.3. Consider the execution of γ_P during a time period τ that generates $dynamic(\gamma_P, \tau)$ as its set of dynamic information. By definition, EMM is inclusive of all components that can be modeled, their complete execution traces, *etc.* Hence, if γ_P is malware that cannot be modeled under EMM, then one of the following must have occurred:

(i) $\exists \pi \in \Pi^*$ such that $\pi[\boldsymbol{Info}(\gamma_P, \tau)] = -1$ but $dynamic(\gamma_P, \tau)$ could not be modeled as a malware. Since $\boldsymbol{Info}(\gamma_P, \tau)$ is defined in Equation (3.23) to span all available static and dynamic information about γ_P, then it must be the case that $dynamic(\gamma_P, \tau) = \emptyset$. Clearly, by the consistency principle, if

$dynamic(\gamma_P, \tau) = \emptyset$, then γ_P cannot be malware under the EMM (*i.e.*, γ_P enacts no state changes during its execution). This contradicts the assumption that γ_P behaved as malware within EMM during $\boldsymbol{T}$.

(ii) $\nexists \pi \in \Pi^*$ such that $\pi[\boldsymbol{Info}(\gamma_P, \tau)] = -1$ and $dynamic(\gamma_P, \tau) \neq \emptyset$ and γ_P is known to be malware. As defined in Section 3.6, Π^* is the perfect set of security policies defined over all available static and dynamic information that exists within the EMM about γ_P during the finite period $\boldsymbol{T}$. Hence, a contradiction occurs due to the consistency principle when γ_P is implementable malware yet $\Pi^*[\boldsymbol{Info}(\gamma_P, \tau)] = 1$ (*i.e.*, it cannot be defined as malware even under all of the information modeled in EMM during $\boldsymbol{T}$).

Hence, both cases directly lead to contradictions. □

Consequently, though Section 4.2 has only highlighted the EMM's ability to model specific classes of malware behaviors, Theorem 4.1 shows that any malware executing within the system can be modeled via the EMM assuming the consistency principle holds. In general, it can be concluded that the EMM can be used to model any execution traces that cause changes in the information set of a modeled system (*i.e.*, $\boldsymbol{S}(\boldsymbol{T})$) as a result of its retention of the original Maurer model's Turing-reducibility. The ability of the EMM to model any implementable security policies will be shown in Chapter 5 to arise via the EMM's ability to model any and all decision processes based on its described information sets, *i.e.* fundamentally Π^* describes a two-class decision process of distinguishing between attack and normal events.

4.4 Summary

Throughout this chapter the application of the EMM to model malware behaviors in terms of security policy violations has been demonstrated. In particular, by Sec-

tion 3.6, the framework has been shown to be able to model various common types of security policy violations. Whereas, in Section 4.2, the behaviors of a number of common malware classes have been modeled. In Section 4.3, the completeness of the EMM with respect to these issues has been shown, with Theorem 4.1 showing that the EMM is able to model any malware execution behaviors that enact changes to the EMM's modeled information sets. In the next chapter, the EMM's ability to model various malware detection approaches is formally developed.

Chapter 5

Formal Analysis of Malware Detection Solutions

5.1 Introduction

As discussed in [72], the evaluation and analysis of malware detection approaches remains an active and open area of research. To the best of our knowledge, there are only few attempts to establish formal malware detection evaluation frameworks (*e.g.*, [38]). However, as was discussed in Chapter 2, these frameworks have typically been based on experimental evaluations where the used datasets are subjected to the existence of artifacts as discussed by Tan *et al.* in [52] and McHugh in [54]. In this chapter, a formal framework for the analysis of malware detection using the EMM is developed.

Note that, a comprehensive analysis framework that is able to model all implementable detection approaches should be capable of formally expressing the probabilities and probability distributions associated with events as many detection approaches are based on the statistical analysis of measurable information (*e.g.*, anomaly intru-

sion detection systems, such as, Bro [132]). Hence, this chapter develops a *measure-theoretic* model of the EMM as measure theory underlies the formal definitions of event probabilities and their statistics [133, 134]. In particular, within this chapter, the EMM will be extended such that it also formally defines a σ-finite measure space and, hence, a probability space. Additionally, the analysis of various detection categories with the developed EMM measure-theoretic model will be discussed. The EMM will also be shown to provide a general model of decision-based detection techniques (*i.e.*, the EMM will be shown to provide a complete model in the sense of being able to model all implementable detection approaches).

The remainder of this chapter is organized as follows. Section 5.2 provides a general model for malware detection within the EMM and motivates the need for measure theory. Section 5.3 provides the basic definitions and concepts of measure theory as used in this dissertation. Section 5.4 shows that the EMM can be extended to formally describe a σ-finite measure space. Section 5.5 discusses the use of the developed model in the analysis of anomaly-based and signature-based detection approaches. Section 5.6 discusses the use of the developed model in the analysis of static and dynamic detection approaches. Section 5.7 then shows that all implementable detectors can be modeled by the developed σ-finite measure space EMM model. Finally, Section 5.8 summarizes the chapter.

5.2 A General Model for Malware Detection

In general, as discussed in Section 3.6, the set of perfect security policies Π^* can be used to classify the set of EMM states into the set of *authorized* (secure) states and the set of *unauthorized* (non-secure) states [104, Definition 4.1, pp. 95]. If the EMM describes a sufficiently small number of states (*e.g.*, a small-scale embedded

system), then it becomes feasible to implement a detection system that can monitor all of the system's operational states. However, the scale of modern computers and networks, this is generally untenable. For example, a modern computing cloud comprising $10,000$ servers each containing a Tera byte disk and running approximately 100 VMs would result in an overall EMM system which could generate upwards of 10^{100} possible state changes per second. Hence, developing a detector that can observe and analyze such a large state space is computationally untenable. To overcome such issue, operational malware detection solutions traditionally focus on only measuring selected subsets of the available information sets.

Let $\mathcal{E}$ be the set of all possible events within the EMM (*i.e.*, $\mathcal{E}$ denotes a partitioning over $\boldsymbol{S}(\boldsymbol{T})$). Define $\mathcal{E}$ as $\mathcal{E} = \mathcal{E}^- \cup \mathcal{E}^+$, where $\mathcal{E}^-$ and $\mathcal{E}^+$, respectively, denote the malicious and benign events subsets, and it is assumed that $\mathcal{E}^- \cap \mathcal{E}^+ = \emptyset$ (*i.e.*, $\mathcal{E}$ is properly partitioned into collections of malicious and benign events). Consider a malware detector $D(.)$ that is deployed in the EMM in order to detect the malicious events. To achieve its task, $D(.)$ must measure information that exists and arises within the EMM about those events over time. Hence, $D(.)$ can be defined as a mapping,

$$D : \mathbb{S}(\boldsymbol{T}) \rightarrow [-1, 1], \tag{5.1}$$

where $\mathbb{S}(\boldsymbol{T})$, as defined in Section 3.3, denotes the space of complete information of all possible events that can occur within the EMM during any time period $\boldsymbol{T}$. In particular, for any event $e \in \mathcal{E}$, $D[e] = -1$ means that the detector $D(.)$ has classified e as malicious (*i.e.*, $D(.)$ has assessed that $e \in \mathcal{E}^-$), whereas $D[e] = 1$ means that $D(.)$ has classified e as benign (*i.e.*, $D(.)$ has assessed that $e \in \mathcal{E}^+$). The case where $D[e] \in (-1, 1)$ denotes the degree to which e is believed to be in $\mathcal{E}^-$ or in $\mathcal{E}^+$ from $D(.)$'s perspective. Hence, $D[e] = 0$ denotes the decision boundary that exists

between $\mathcal{E}^-$ and $\mathcal{E}^+$. As shown in Figure 5.1, the operation of $D(.)$ can be more precisely modeled as,

$$D[e] = d\left[f(e)\right], \tag{5.2}$$

where the details of the mappings $f(.)$ and $d(.)$ are as follows. Without loss of generality, the mapping $f(.)$ can be defined as,

$$f : \mathbb{S}(\boldsymbol{T}) \rightarrow \mathcal{X} \cup \{\emptyset\}, \tag{5.3}$$

where $\mathcal{X}$ denotes the spatial-temporal space that represents the abstracted feature (or measurement) space that is extracted from $\mathbb{S}(\boldsymbol{T})$ by $D(.)$. In general, it will be the case that $|\mathcal{X}| \ll |\mathbb{S}(\boldsymbol{T})|$ (*i.e.*, the amount of information represented by the abstracted feature space $\mathcal{X}$ will be much less than complete set of information described by the EMM). In this sense, standard malware detection solutions must be considered to be information lossy solutions within the larger-scale defended environments to which they are typically applied. In pattern recognition terminology, $f(.)$ therefore denotes the pattern classification features that $D(.)$ measures from $\mathbb{S}(\boldsymbol{T})$ in order to classify whether e belongs to $\mathcal{E}^-$ or $\mathcal{E}^+$. The exact nature of these features are within the defenders' purview to select and can span a wide variety of approaches such as *control flow graphs*, *system call sequences*, *etc.* Note that, the feature space $\mathcal{X}$ itself is defined simply as,

$$\mathcal{X} = \Re^n \cup \{\emptyset\}. \tag{5.4}$$

That is to say, $\mathcal{X}$ is a standard n-dimensional euclidean space inclusive of $\emptyset$, in which time is simply one of its n dimensions.

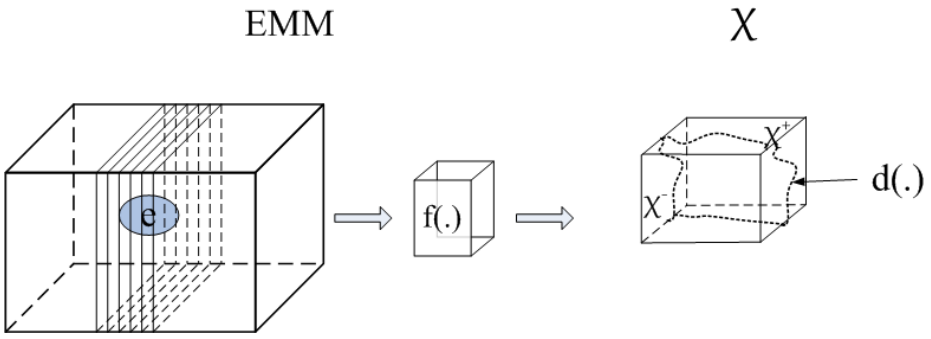


Figure 5.1: Malware detection modeled as a decision problem.

It should be noted that, as indicated in Equation (5.3), there may exist cases whereby $f(e) = \emptyset$, indicating that e is *unobservable* under the selected (or implemented) $f(.)$. More particularly, *unobservable* and *observable* events with respect to a detector $D(.)$ can be formally defined as follows.

Definition 5.1 (Unobservable and observable events under $D(.)$)**.** *For a detector $D(.)$ that uses a feature mapping $f(.)$, an event e within the EMM (*i.e.*, $e \in \mathbb{S}(\boldsymbol{T})$) is* unobservable *with respect to $D(.)$ if and only if for all $t \in \boldsymbol{T}$, it is the case that $f(e) = \emptyset$, otherwise it is* observable*.*

Clearly, the attackers can avoid detection by $D(.)$ if they can craft their malware such that it becomes unobservable. Specifically, they can study or reverse engineer $D(.)$ to determine the information sets that it measures and then develop their new malware or malware variant such they do not include this information. For example, as discussed in [135], if $D(.)$ is designed to only monitor Windows API calls, any malware that only utilizes *native* Windows system calls will then become unobservable with respect to $D(.)$. It should be emphasized that the notion of observability is related to the selection of the feature mapping $f(.)$. A given malware instance can be unobservable with respect to a specific detector $D(.)$ but still be observable with respect to some other detector $D'(.)$. Hence, compositions of malware detectors can

be implemented to reduce the risk posed by unobservable malware. Note that such composition of detectors will be discussed in Section 5.2.1.

As indicated in Equation (5.2), in addition to $f(.)$, $D(.)$ also implements a mapping $d(.)$ that enforces a decision boundary on $\mathcal{X}$. In particular, $d(.)$ can be defined generally as,

$$d : \mathcal{X} \rightarrow [-1, 1]. \tag{5.5}$$

Therefore, $d(.)$ partitions $\mathcal{X}$ into the subspaces $\mathcal{X}^-$ and $\mathcal{X}^+$ such that if $d[x] = -1$ then $x \in \mathcal{X}^-$, and if $d[x] = 1$ then $x \in \mathcal{X}^+$, where for simplicity it is assumed that $\mathcal{X}^- \bigcap \mathcal{X}^+ = \emptyset$. More particularly, if $d\left[f(e)\right] = -1$, then $D(.)$ has denoted that it views e to be malicious (*i.e.*, that $e \in \mathcal{E}^-$), whereas if $d\left[f\left(e\right)\right] = 1$, then $D(.)$ assessed e to be benign (*i.e.*, that $e \in \mathcal{E}^+$). The case where $d\left[f\left(e\right)\right] \in (-1, 1)$ denotes the degree of belief that $D(.)$ has in e being either malware or benign. Hence, $d\left[f\left(e\right)\right] = 0$ denotes the decision boundary between $\mathcal{X}^-$ and $\mathcal{X}^+$. Since typically $f(.)$ will be information-lossy, then $d\left[f\left(e\right)\right] = -1$ or $d\left[f\left(e\right)\right] = 1$ does not guarantee that $e \in \mathbb{S}(\boldsymbol{T})$ is indeed either malware or benign, but only denotes $D(.)$'s assessment of e's perceived maliciousness (*i.e.*, it is possible for $d\left[f(e)\right] = -1$ for $e \in \mathcal{X}^+$ and vice versa).

As per the above, $d(.)$ can be seen to implement a classifier between the classes $\mathcal{X}^-$ and $\mathcal{X}^+$. Hence, the attackers can seek to evade $d(.)$ by crafting their malware in a way that it would be misclassified as being benign and, hence, in $\mathcal{X}^+$ thereby causing *false negatives*. Additionally, $d(.)$ could also wrongly classify benign events as being in $\mathcal{X}^-$ and, hence, as malware thereby generating *false positives*. For example, in [136], Mutz *et al.* proposed a reverse engineering process and tool to analyze signature matching mechanisms in network-based IDS which demonstrated the ability of attackers to craft their attacks such that they generated false negatives. Ideally, malware detectors

should possess low false negative and false positive rates. The notion of the ideal detector can be captured via defining the *soundness* and *completeness* of a detector $D(.)$ as follows.

Definition 5.2 (The soundness and completeness of malware detectors)**.** *The malware detector* $D(.)$ *is* sound *if for all* benign *events* $e \in \mathcal{E}$ *for which* $f(e) \neq \emptyset$ *then* $D[e] = 1$*. Whereas,* $D(.)$ *is* complete *if for all* malware *events* $e \in \mathcal{E}$ *for which* $f(e) \neq \emptyset$ *then* $D[e] = -1$*. That is to say,* $D(.)$ *is sound only if it has a zero false positive rate and complete only if it has a zero false negative rate.*

Ideal malware detectors clearly should be both sound and complete. It should be noted that, from the consistency condition of Π^* discussed in Definition 4.6, with respect to any defined class of malicious events, the EMM guarantees that there exists an ideal detector $D^*(.)$ must theoretically exist that is both sound and complete (*i.e.*, sufficient information always exists within the EMM for $D^*(.)$ to be implementable). Moreover, as the EMM defines all of the measurable information available across the modeled systems, then $D^*(.)$ can be further generalized to also ensure that all $e \in \mathcal{E}^-$ occurring within the EMM are also guaranteed to be observable (*i.e.*, an existence proof could be constructed via the EMM that perfect security against any malware class or classes is always, at least theoretically possible). Hence, when $f(.)$ and $d(.)$ are selected in a way that leads to no false positives, and no false negatives, then $D^*(.)$ is obtained. The definition of $D^*(.)$ in terms of Π^* is formalized as follows.

Definition 5.3 (Ideal detector, $D^*(.)$)**.** *The ideal detector is denoted as* $D^*(.)$*, where* $D^*(.)$ *is defined to be sound and complete. That is to say,* $\forall t \in \boldsymbol{T}$*,* $\forall e \in \mathcal{E}$ *such that* $f(e) \neq \emptyset$*, for* $D^*(.)$*, it is the case that:*

(i) $D^*[e] = 1$ *if and only if* $\Pi^*[e] = 1$*, and,*

(ii) $D^*[e] = -1$ *if and only if* $\Pi^*[e] = -1$*.*

The core problem for the defenders though is to determine the correct measurement features $f(.)$ and decision boundary $d(.)$ by which to achieve $D^*(.)$. Moreover, for any $D(.) \neq D^*(.)$, by definition, the attackers could adapt their attack strategies such that for an event $e \in \mathcal{E}^-$, can be generated such that $D(e) = \emptyset$ or $D(e) \notin \mathcal{X}^+$. Clearly, operational malware detection solutions cannot be shown to be sound and complete except within the contexts of small-scale systems. But the existence of $D^*(.)$ is an important theoretical construct within the EMM.

5.2.1 Compositions of Detectors

Clearly, the compositions of detectors are necessary in any actual deployment. Such compositions can be developed within the EMM as follows. Consider a composite malware detector $\mathbb{D}(.)$ that is composed of a set of N_D detectors each following the definition of $D(.)$ above. Hence,

$$\mathbb{D} = \{D_k | k = 1, 2, \ldots, N_D\}. \tag{5.6}$$

Different compositions of these detectors are clearly possible. For example, $\mathbb{D}(.)$ could be structured as,

$$\mathbb{D}(e) = \min\{D_k(e) | k = 1, 2, \ldots, N_D\}, \tag{5.7}$$

which indicates majority voting is applied over the composition of detectors. Hence, $\mathbb{D}(e) = -1$ if $\exists D_k \in \mathbb{D}$, such that, $D_k(e) = -1$ (*i.e.*, an event is deemed malicious if it is deemed malicious by at least one detector). Alternatively, a weighted detection solution could also be used such that,

$$\mathbb{D}(e) = \sum_{k=1}^{N_D} w_k D_k(e), \tag{5.8}$$

where $0 \leq w_k \leq 1$ and $\sum_{k=1}^{N_D} w_k = 1$. Hence, each w_k in this case would reflect the level of operational value that the defenders place in each detector. Additionally, more complex hierarchical compositions of detectors could also be modeled where, for example, detector $D_j(.)$'s decision is passed onto some next detector $D_k(.)$ for further analysis. Hence, quite generally, the compositions of the detectors can be easily incorporated in the above EMM model, where the selection of any given implementation is at defender's discretion. Finally, the feature mapping $F(.)$ of the composite detector $\mathbb{D}(.)$ can be defined as the set of all feature maps $f(.)$ across the collection of composite detectors as follows.

$$F(.) = \{f_k(.)|k = 1, 2, \cdots, N_D\}. \tag{5.9}$$

5.2.2 Classes of Events

Now assume that the defender has defined the set of classes

$$\boldsymbol{\omega} = \{\omega_1, \omega_2, \ldots, \omega_{N_\omega}\}, \tag{5.10}$$

associated with the events occurring within the EMM. Moreover, assume that each $\omega_k \in \boldsymbol{\omega}$ can be partitioned into the subsets $\boldsymbol{\omega}^-$ and $\boldsymbol{\omega}^+$ denoting, respectively, malicious and benign event classes. Then, from the pattern recognition perspective, the collection of $D_k(.)$'s can be seen as implementing pattern classifiers between these ω_k classes. More particularly, the composite detector $\mathbb{D}(.)$ implements the decision boundary between the classes $\boldsymbol{\omega}^-$ and $\boldsymbol{\omega}^+$. In general, to model both signature (or misuse) and anomaly detection, the probabilities of observing a given event e under each class ω_k must be defined. As probabilities are formally defined in terms of measure theory constructs, it must first be shown that the EMM as structured also

describes a σ-finite measure space as otherwise the required class probabilities cannot be shown to exist.

Accordingly, the subsequent sections provide the necessary measure theory concepts and show how the EMM can be extended to allow it to also formally define the required *σ-finite measure space*. This allows us to formally define the various probability measures that are required, as discussed above. Additionally, this then allows the EMM to be shown to be inclusive of modeling static and dynamic, as well as, anomaly-based and signature-based detection approaches. Moreover, via this measure theory approach, the EMM can then be shown to be complete in the sense of being able to model any implementable malware detection process that makes use of the EMM measurable information.

5.3 Introduction to Measure Theory

For completeness, this section provides the basic definitions and concepts of measure theory that will be used in the development of the EMM's measure-theoretic model. In particular, the concepts of σ-algebras, measurable spaces, measures and probability spaces will be discussed. For more details about measure theory, we refer the reader to measure theory references (*e.g.*, [133,134,137,138]). It should be emphasized that, the majority of the subsections 5.3.1 to 5.3.4 are paraphrased from [133], in part to ensure the accuracy and completeness of the presented standard mathematical definitions. Moreover, it is useful to note that measure theory and its concepts are widely used in other domains to reason about behaviors arising within complex systems, *i.e.,* as per its use in statistical physics [139], dynamical systems theory [140], ergodic theory [140], *etc.* We would argue that measure theory has yet to see significant

use in security research due to a lack of a bridge between it and Turing-equivalence concepts and a lack of familiarity with it within the security community.

5.3.1 Notation

Let Ω be a nonempty set representing a standard sample space. Then, Ω is a *finite set* if it has a finite number of elements (*i.e.*, $|\Omega| < \infty$), otherwise, it is an *infinite set* [133, pp. 391]. The symbols $\cup$ and $\cap$ refer to the standard set *union* and *intersection* operations respectively, whereas the symbols $\setminus$ and $-$ both commonly used to refer to the set *difference* operation and, hence, will be used exchangeably.

In general, for a set $A \subseteq \Omega$, the set $A^c = \Omega \setminus A$ is the *complement* of A (*i.e.*, A^c is the set of all elements of Ω that are not in A). The *power set* of Ω, denoted as $\mathcal{P}(\Omega)$, is the set of all subsets of Ω defined as $\mathcal{P}(\Omega) = \{A : A \subseteq \Omega\}$ where this by definition includes the empty set $\emptyset$ [133, pp. 55]. A set $\mathcal{B}$ whose elements are subsets of the set Ω in called a *class* or a *family* of subsets of Ω (*i.e.*, $\mathcal{B} \subseteq \mathcal{P}(\Omega)$).

Let $\Re$ be the set of *real numbers*. The set of *extended real numbers* is denoted as $\Re^*$ and is defined as $\Re^* = \Re \cup \{-\infty, +\infty\} = [-\infty, +\infty]$ [133, Appendix A, pp. 385]. Henceforth, $\Re^*_+$ will be used to denote the set of non-negative extended real numbers (*i.e.*, $\Re^*_+ = [0, +\infty]$).

5.3.2 σ-Algebras and Measurable Spaces

For any non-empty set Ω, *σ-algebras* are special classes of subsets of Ω with specific properties that are important in measure theory. Particularly, σ-algebras are defined as follows [133, Definition 3.9.1, pp. 80].

Definition 5.4 (σ-algebras)**.** *Let Ω be a non-empty set. The class $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is a σ-algebra (also called σ-field) of subsets of Ω if and only if it satisfies the following properties:*

(i) $\emptyset, \Omega \in \mathcal{F}$,

(ii) $\forall A \in \mathcal{F}, A^c \in \mathcal{F}$, *and,*

(iii) $\forall A_j \in \mathcal{F}, \bigcap_{j=1}^{\infty} A_j \in \mathcal{F}$.

As shown in Definition 5.4, σ-algebras are closed under complement and countable intersection. Additionally, by applying De Morgan's law, it can be shown that σ-algebras are also closed under countable union. The classes $\{\emptyset, \Omega\}$ and $\mathcal{P}(\Omega)$ are obvious examples of σ-algebras of subsets of Ω, where they respectively denote the smallest and largest σ-algebras of subsets of Ω, [141, Section 1.3, pp. 11]. As discussed in [141, Section 1.3, pp. 10], σ-algebras are used to define *measurable spaces* as follows.

Definition 5.5 (Measurable Spaces)**.** *The tuple* $\langle \Omega, \mathcal{F} \rangle$ *of the non-empty set* Ω *and the* σ*-algebra* $\mathcal{F}$ *of subsets of* Ω *is called a* measurable space.

Clearly, not all subsets of Ω are necessarily in $\mathcal{F}$ and, therefore, not every subset of Ω is included in the defined measurable space.

5.3.3 Measures

Let Ω be a non-empty set and $\mathcal{B} \subseteq \mathcal{P}(\Omega)$ be a class of subsets of Ω. A function $\mu : \mathcal{B} \to \Re^*$ is defined as a *set function* [133, Definition 3.3.1, pp. 59]. As discussed in [141, Section 1.6, pp. 19], *measures* are *non-negative set functions* that will be defined as follows.

Definition 5.6 (Measures)**.** *Let* Ω *be a nonempty set and* $\mathcal{B} \subseteq \mathcal{P}(\Omega)$ *be a class of subsets of* Ω. *The* set function $\mu : \mathcal{B} \to \Re^*$ *is a* measure *defined on* $\mathcal{B}$ *if and only if it has the following properties:*

(i) $\forall A \in \mathcal{B}, \mu(A) \in \Re^*_+$,

(ii) $\mu(\emptyset) = 0$, *and,*

(iii) For any countable disjoint *sets* $A_1, A_2, \cdots \in \mathcal{B}$, $\mu(\bigcup_{j=1}^{\infty} A_j) = \sum_{j=1}^{\infty} \mu(A_j)$ *(*i.e., *countable additivity holds).*

5.3.4 Measure Spaces and Probability Spaces

As discussed in [141, Section 1.6, pp. 20], *measure spaces and probability spaces* can be formally defined as follows.

Definition 5.7 (Measure Spaces, σ-finite measure spaces, and Probability Spaces)**.** *Let* Ω *be a nonempty set,* $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ *be a* σ*-algebra of* Ω*, and* $\mu : \mathcal{F} \rightarrow \Re_{+}^{*}$ *be a measure defined on* $\mathcal{F}$*. The tuple* $\langle \Omega, \mathcal{F}, \mu \rangle$ *is called a* measure space *and the elements of* $\mathcal{F}$ *are called* measurable sets *or* events*. A measure space is* finite *if* $\mu(\Omega) < \infty$*. Finally, if* $\mu(\Omega) = 1$*, then the measure* μ*, by definition, will also meet the axioms of probability. Hence, the tuple* $\langle \Omega, \mathcal{F}, \mu \rangle$ *can then also be called a* probability *space.*

Note that, any σ-finite measure space can be converted into a probability space by suitable normalization [141, pp. 20]. Probability spaces play important roles in analyzing statistical experiments. Particularly, for a probability space $\langle \Omega, \mathcal{F}, \mu \rangle$, where Ω is the sample space (*i.e.*, the set of all possible outcomes of an experiment) and $\mathcal{F}$ is the collection of all events, then, for every event $A \in \mathcal{F}$, $\mu(A)$ denotes the probability that event A may occur (*i.e.*, $\forall A \in \mathcal{F}, \mu(A) = Pr(A)$). Hence, an information space must be shown o be at least a σ-finite measure space if it is to be formally shown that event (or class) probabilities can be defined over that space.

5.4 The EMM as a Probability Space

As discussed in Section 5.2, extending the EMM by formally defining a σ-finite measure space over it allows us to define the required probability measures over the various

event classes. It also allows the completeness of the EMM to model various categories of malware detection approaches to be formally shown. Hence, in this section, we will show that the EMM over any finite time period τ defines a σ-finite measure space.

5.4.1 The EMM as a σ-Finite Measure Space

As discussed in Section 3.3, the physical memory M of the EMM was defined as a finite set $M = \{m_k | k = 1, 2, \ldots, N_M\}$ consisting N_M memory elements (bits) (*i.e.*, $N_M < \infty$). These elements are disjoint by the nature of their physical implementation (*i.e.*, $\forall m_j \neq m_k \in M, m_j \cap m_k = \emptyset$). Accordingly, $M(t)$ is therefore a finite set defined by the union of all these disjoint memory elements m_k, and hence, $M(t) = \bigcup_{k=1}^{N_M(t)} m_k$. Each element $m_k \in M$ stores information according to the base set B, where $B = \{0, 1\}$ for digital computers. Accordingly, the set $\mathbb{S}(t)$ of all possible states of the EMM at t is the set of all maps of the form $M(t) \rightarrow B$ by the above is necessarily finite (*i.e.*, $|\mathbb{S}(t)| < \infty$).

Now, consider the set of EMM states during a finite interval $\tau \subseteq \boldsymbol{T}$, denoted as $\mathbb{S}(\tau)$. Lemma 5.1 shows that a measurable space can be defined over $\mathbb{S}(\tau)$.

Lemma 5.1. *A measurable space can be defined over the set of possible EMM states* $\mathbb{S}(\tau)$.

Proof. Denote the set of all possible EMM states during some $\tau \subseteq \boldsymbol{T}$ as $\mathbb{S}(\tau)$. Within the EMM, time is assumed discrete. Therefore, τ as a finite interval as it must be composed of finite number of time slots t. Accordingly, $\mathbb{S}(\tau)$ can be defined as the finite union of all possible states for all $t \in \tau$, which can be expressed as,

$$\mathbb{S}(\tau) = \bigcup_{\forall t \in \tau} \mathbb{S}(t). \tag{5.11}$$

Let $\mathcal{R}(\tau) = \mathcal{P}[\mathbb{S}(\tau)]$ (*i.e.*, $\mathcal{R}(\tau)$ is defined as the power set of $\mathbb{S}(\tau)$). As discussed in Section 5.3.2, since the power set is the largest σ-algebra of subsets of any set and since $\mathcal{R}(\tau)$ is defined as the power set of $\mathbb{S}(\tau)$, then by definition $\mathcal{R}(\tau)$ is a σ-algebra. Hence, as discussed in Definition 5.5, the tuple $\langle \mathbb{S}(\tau), \mathcal{R}(\tau) \rangle$ consists of a set $\mathbb{S}(\tau)$ and a σ-algebra $\mathcal{R}(\tau)$ of subsets of $\mathbb{S}(\tau)$. Hence, $\langle \mathbb{S}(\tau), \mathcal{R}(\tau) \rangle$ is a measurable space, and moreover, it is also a probability space. □

Lemma 5.1 shows that the EMM describes a measurable space over any finite time interval τ and a σ-finite measure space in the limit when τ goes to infinity. This allows to define a non-negative set function μ that satisfies the conditions of measures (Definition 5.6), and hence, defines the EMM as the measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$. Consequently, all events that are defined within this EMM measure space can now be measured. Quite generally, standard Lebesgue measures [133, Chapter 4], which generalizes the notion of cardinality, can therefore be defined over this EMM measurable space. Alternatively, the defenders may define any other non-negative set function μ' to use, as long as this μ' also meets the requirements of being a measure.

Lemma 5.2. *The measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ where μ is Lebesgue measure is σ-finite.*

Proof. Since, by definition, the set $M(t)$ of all memory elements at any time t and the base set B are finite, then the set $\mathbb{S}(t)$ of all possible maps $s : M(t) \to B$ at any discrete t is also finite. Define μ as the standard Lebesgue measure. Since $\mathbb{S}(t)$ is finite, then $\mu[\mathbb{S}(t)] < \infty$. From Equation 5.11, for any $t \neq t' \in \tau$, $\mathbb{S}(t)$ and $\mathbb{S}(t')$ are then temporally disjoint sets within $\mathbb{S}(\tau)$ domain and, hence, $\mathbb{S}(\tau)$ is a finite union of finite disjoint sets. Therefore, $\mu[\mathbb{S}(\tau)] = \sum_{\forall t \in \tau} \mu[\mathbb{S}(t)]$. Since τ is finite, then $\mu[\mathbb{S}(\tau)]$ is a finite sum of finite quantities, and hence, $\mu[\mathbb{S}(\tau)] < \infty$. From Definition 5.7, since $\mu[\mathbb{S}(\tau)] < \infty$, then $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ is a σ-finite measure space. □

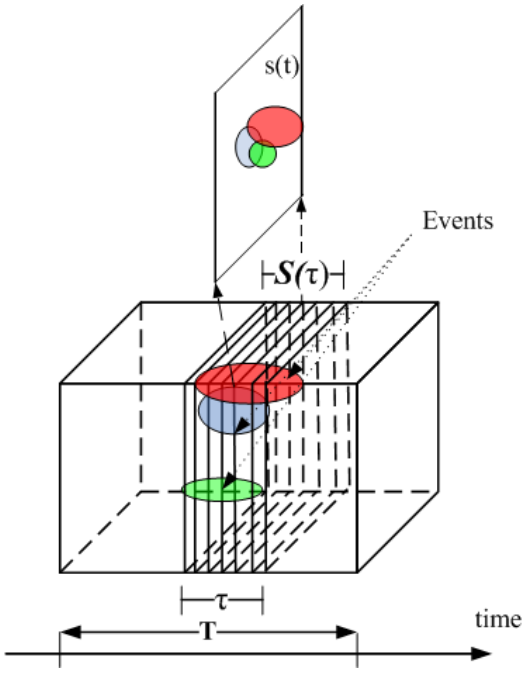


Figure 5.2: EMM events as spatial-temporal objects arising within $\boldsymbol{S}(\boldsymbol{T})$.

Hence, a σ-finite measure space can be defined over the EMM. Additionally, as discussed in Section 5.3.4, $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ can therefore be converted to a probability measure with the proper normalization (*i.e.*, as per the Axioms of Probability [142]). Hence, the probabilities of event classes discussed in Section 5.2.2 (*i.e.*, $Pr_{\omega_k}(e)$) have now been shown to formally exist within the EMM and, hence, can now be measured within the EMM.

5.4.2 Defining Events within the EMM

In this section, the definition of *events* within the context of the EMM will be developed. In general, as discussed in Definition 5.7, the events in the measure space are the measurable sets that are members of the σ-algebra defined by that measure space. Similarly, in the defined EMM finite measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$, *events* are

also the members of the σ-algebra as defined by the EMM (*i.e.*, the events within the EMM defined measure space are the elements of $\mathcal{R}(\tau)$). In particular, within the $\mathbb{S}(\tau)$ space discussed above, all possible spatial-temporal subsets of $\mathbb{S}(\tau)$ represent the possible events as these are members of $\mathcal{R}(\tau)$. Hence, an event formally denotes a collection of states associated with of some collection of memory regions as defined over a given time period. For example, the execution traces of software components, received and sent packets, sequences of users' commands, *etc.* all denote types of events that can be represented as spatial-temporal subsets within $\mathbb{S}(\tau)$. Figure 5.2 shows examples of events in the EMM. In general, events within the EMM would be expected to have sufficiently short and strictly finite time durations such that their durations τ are such that $\tau \ll \boldsymbol{T}$, *i.e.*, as per normal expectations, events occur over far shorted time frames than the full duration of the system's operational time over which security is to be achieved. Moreover, clearly, the above denotes a very general definition of events as, for example, events may be comprised of discontinuous memory regions, and/or discontinuous time periods. Typically, defenders, therefore, would be only interested in specific subsets of this space of all possible events over $\boldsymbol{T}$. However, to define event probabilities, the entire space of the events must exist as a measurable space. Within the EMM, events are therefore formally defined as follows,

Definition 5.8. *An event e within the EMM's σ-finite measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ is defined as a spatial-temporal subspace that occurs over a defined finite time period, and therefore all events, e, exist as elements of the σ-algebra $\mathcal{R}(\tau)$ that is described by the EMM.*

Additionally, the following lemma shows that for all events e within the EMM, their probabilities can also be formally shown to exist.

Lemma 5.3. *For all EMM events $e \in \mathcal{E}$, there exists a mapping $Pr : \mathcal{E} \to [0, 1]$, such that, $Pr(e)$ denotes the probability of the event e.*

Proof. Proof by construction. As discussed above, Lemma 5.1 and Lemma 5.2 showed the existence of a σ-finite measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ over the EMM during τ, for which all events e are members of its σ-algebra $\mathcal{R}(\tau)$. Accordingly, as discussed in Section 5.3.4, $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu \rangle$ can be converted into a probability space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), Pr \rangle$ by suitable normalization. Hence, there exists a mapping $Pr : \mathcal{E} \rightarrow [0, 1]$ such that $Pr(e)$ denotes the probability of the event e for all events e occurring within the σ-algebra. □

Finally, it should be noted that, as defined above, events are restricted to exist within the measurable information sets of the defined EMM. Hence, all incidents that occur outside this defined measure space by definition cannot be measured within or described by information contained in this space. Moreover, it should be noted that the formal existence of event probabilities within the EMM also enables direct linkages between the EMM and formal probability-based definitions of information (*i.e.*, as per information theory [143]). For example, consider the attacks described in *US vs Gorshkov* [144]. In this case, the attackers enacted what would have been considered legitimate user behaviors within each of eBay and PayPal's isolated domains, but which only became observably malicious when viewed jointly across eBay and PayPal's combined information sets. Such attacks would have been undetectable within individual eBay and PayPal EMM models but would become detectable under a joint eBay-PayPal EMM model. Hence, it should be emphasized that, the EMM cannot model attacks (or their detection) which exist outside of the purview of the EMM's modeled information sets.

5.4.3 Discussion

As shown in Section 5.4, the EMM can be modeled as a σ-finite measure space, hence probabilities associated with all events that can occur within the EMM can be defined.

For each event e, its probability over each class $\omega_k \in \omega$, $Pr_{\omega_k}(e)$, can be formally defined. Accordingly, the defenders' observation of a particular event e arising from some class ω_k can then be formally viewed as the observation of a particular instance of a random variable described by $Pr_{\omega_k}(e)$. Hence, events within the same class are not expected to produce identical bit changes within the EMM's memory $\boldsymbol{M}(\boldsymbol{T})$ or even occur over identical time periods. Instead, stochastic variations are expected to occur as described by the event classes underlying $Pr_{\omega_k}(e)$ distributions.

In a slight abuse of notation, e will now be used to denote this random variable described by $Pr_{\omega_k}(e)$. Hence, $f(e)$ now describes a random process. More particularly, the allowable $f(.)$'s will now be restricted to those which preserve the σ-finite measure space characteristics with respect to their generated feature spaces (*i.e.*, for the map $f : \mathbb{S}(\boldsymbol{\tau}) \rightarrow \mathcal{X}$, if $\mathbb{S}(\boldsymbol{\tau})$ defines a σ-finite measure space as per the EMM, then $\mathcal{X}$ must also define a σ-finite measure space). Hence, $\forall e \in \mathcal{E}$ such that $f(e) \neq \emptyset$, if $\mu(e)$ exists and $\mu(e) = Pr(e)$, then $Pr(x)$ also exists for $f(e) = x \in \mathcal{X}$. This notion of retaining measurability across $f(.)$ is captured in the following definition.

Definition 5.9. *If $f(.)$ is to be a feature mapping as defined in Equation* (5.3)*, then $\mathcal{X}$ must define a σ-finite measure space.*

It should be noted that, it is not required that $f(.)$ is measure preserving (*i.e.*, $Pr(e)$ need not necessarily equal $Pr(x)$). Additionally, the above analysis only shows that the required $Pr_{\omega_k}(e)$ exists within the EMM. In particular, it does not specify the analytical forms that these $Pr_{\omega_k}(e)$ or $Pr(x)$ may take. Generally, the analytical forms of the $Pr_{\omega_k}(e)$ or $Pr(x)$ depend on the definitions of the events of interest to the defenders. Moreover, it would not be expected that such events would, for example, meet Central Limit Theorem tenets [145, pp. 621]. Hence, common analytical forms, such as Gaussian distributions, cannot be assumed for the $Pr_{\omega_k}(e)$ or $Pr(x)$ distributions. Specific analytical $Pr_{\omega_k}(e)$ and $Pr(x)$ forms, therefore, can only be obtained

via the detailed analysis of the actual operational system(s) the EMM is being used to model and not via the EMM itself.

5.5 Anomaly and Signature Detection

In general, detection approaches can be classified according to the detection method into *anomaly-based* and *signature-based* [6, 66]. Anomaly-based approaches detect attacks by monitoring the deviations of the system from a collection of its previously learned normal behaviors. Whereas, signature-based approaches use their knowledge about malicious behaviors to directly detect the attacks. In this section, the EMM modeling of anomaly and signature detectors will be discussed.

5.5.1 Modeling Anomaly-based Detection Approaches

Anomaly-based detection approaches use their knowledge about what constitutes normal behaviors to determine the degree of the maliciousness of the analyzed event. Typically, anomaly detectors must be trained before they can be used to detect these malicious behaviors. During this training phase, anomaly-based detection systems analyze the historical data of the past events in order to recognize the patterns associated with normal activities, and thereby, to establish models for the normal behavior. Various approaches can be used to build the normal behavior models (*e.g.*, statistical approaches (*e.g.*, [19, 146]), data mining approaches (*e.g.*, [17, 147]), machine learning approaches [43, 148], *etc.*). After the training phase, the trained detector is then deployed to monitor the system by detecting the deviations from its learned normal behaviors. A core advantage of anomaly-based detection is its ability to detect novel attacks. Whereas, their major disadvantages are its high operational false positive rates and the difficulty of constructing the training models [149]. Numerous anomaly

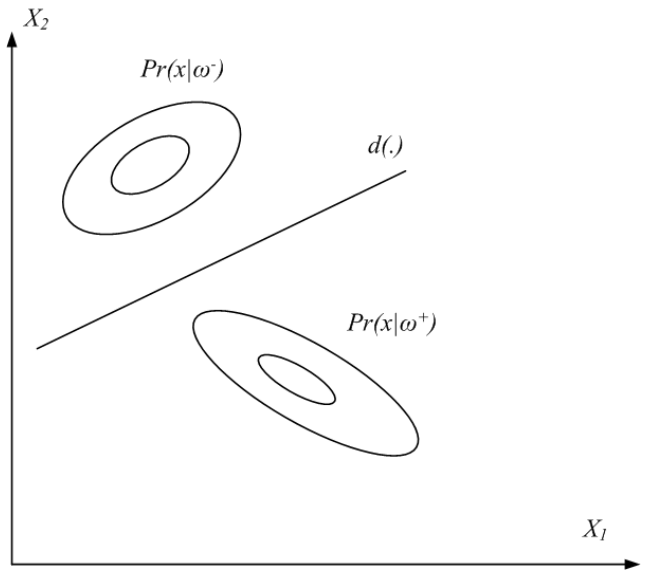


Figure 5.3: Anomaly detection as an EMM decision problem.

detection approaches have been proposed with [17,19,132,146,147,149,150] providing some examples.

The formal EMM modeling of anomaly detection is via the fundamental construct of standard statistical pattern classification problems [145]. Consider an anomaly detector $D_A(.)$. Hence, $D_A(.)$ is used to classify the events into two classes: $\boldsymbol{\omega}^-$ and $\boldsymbol{\omega}^+$ where these represent the malicious and the benign events, respectively. More particularly, $D_A(.)$ uses the probability distributions $Pr(x|\boldsymbol{\omega}^-)$ and $Pr(x|\boldsymbol{\omega}^+)$ of both classes in the classification. As shown in Figure 5.3, $d(.)$ defines a decision boundary between $Pr(x|\boldsymbol{\omega}^-)$ and $Pr(x|\boldsymbol{\omega}^+)$. During the training phase, $D_A(.)$ uses the training data of benign events to obtain (learn) an estimate of $Pr(x|\boldsymbol{\omega}^+)$, denoted as $\widehat{Pr}(x|\boldsymbol{\omega}^+)$. Additionally, $D_A(.)$ also sets (or learns) $d(.)$ such that it achieves a desired false positive rate, *i.e.*, such as via receiver operating characteristic (ROC) curves [145, Section 2.8.3, pp. 49]. Moreover, if the training data also contains ground-truthed malicious events, then $D_A(.)$ can also obtain an estimate of $Pr(x|\boldsymbol{\omega}^-)$, denoted as

$\widehat{Pr}(x|\boldsymbol{\omega}^-)$, and $d(.)$ can then be constructed based on knowledge of both $\widehat{Pr}(x|\boldsymbol{\omega}^+)$ and $\widehat{Pr}(x|\boldsymbol{\omega}^-)$. Furthermore, as per standard statistical pattern recognition theory, $d(.)$ can be constructed through either parametric or non-parametric approaches [145, Chapter 4], [151, Chapter 6]. As the EMM has been shown to define a σ-finite measure space, non-probabilistic decision approaches, such as fuzzy logic, can also be represented within the EMM, provided they operate on measurable information sets.

5.5.2 Signature-based Detection Approaches

Signature-based detection systems use signatures of known malicious behaviors to detect attacks. The core advantage of these approaches is their high accuracy in detecting *known attacks* with very low false positive rates. However, their major disadvantage is their inability to detect novel (previously unseen) attacks [149]. In malware detection, signature-based approaches are the most commonly used methods of detecting malware behaviors (*e.g.*, commercial anti-malware scanners).

The formal EMM modeling of signature-based approaches is similar to that of anomaly-based approaches. However, the applied $d(.)$ is now defined such that it denotes a specific set of defined points in $\mathcal{X}$, or more generally sets, where these points then denote the known malicious event behaviors, as shown in Figure 5.4. Consider the signature-based detector $D_s(.)$. Hence, $D_s(.)$ has a database of known signatures $\boldsymbol{X} = \{x_k | k = 1, 2, \cdots, K\}$ of malware. Accordingly, for the event $e \in \mathcal{E}$, $D_s(.)$ uses its feature mapping $f(.)$ to obtain $f(e) = x$. Finally, if $x \in \boldsymbol{X}$, then the decision boundary $d(.)$ will report e as malware if $x \in \boldsymbol{X}$. Otherwise, x will be denoted as benign (*i.e.*, $d(x) = -1$ if and only if $x \in \boldsymbol{X}$, with $d(x) = 1$ otherwise).

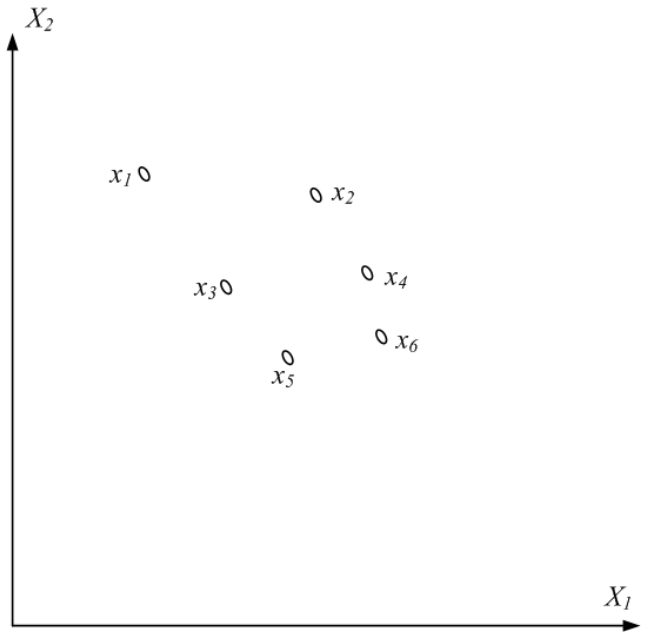


Figure 5.4: Signature detection as an EMM decision problem.

5.5.3 Discussion

In this section, we discussed the modeling of anomaly-based and signature-based detection approaches via the EMM. Without loss of generality, the distinction between both categories is that signatures are specific defined elements of the $\mathcal{X}$ space (*i.e.*, not a space of elements as anomaly-based approaches), where in anomaly detection $d(.)$ is generalized to a more typical decision boundary. Hence, signature detection becomes the redefinition of the decision boundary $d(.)$ to denote only a set of specifically defined points (or more generally regions) in the space $\mathcal{X}$ that represents the known malicious events as shown in Figure 5.4, *i.e.*, within the EMM, signature detection represents a more informed version of anomaly detection as both can be seen to implement different forms of decision functions on $\mathcal{X}$.

5.6 Modeling Static and Dynamic Detection Approaches

In general, malware detection approaches can also be classified according to the way in which detection information is gathered in terms of whether they are *static* and *dynamic* detection approaches [6, 7]. Static approaches attempt to detect malware without executing the analyzed programs, whereas dynamic approaches detect malware during or after the execution of the analyzed programs. The formal EMM modeling of static and dynamic malware analysis approaches is discussed as follows.

5.6.1 Static Detection Approaches

Without loss of generality, static malware detection approaches perform their analysis without executing the analyzed programs. They use the syntactical or structural properties of the programs to reason about their maliciousness. Examples of static approaches are those of [1, 13, 33, 40, 42, 44, 48, 51, 152–160]. Despite the strong belief in the research community that dynamic analysis outperforms static analysis, recent studies showed that the strong detection capabilities of static analysis cannot be ignored [161–166].

In general, the core advantage of static detection is that, it covers the entire code contained in the analyzed sample, and consequently, it can provide additional information about all possible execution traces of the analyzed programs making it more suitable to detect specific malware classes, such as, logic and time bombs. Whereas, the static analysis has the disadvantages of being susceptible to various obfuscation techniques (*e.g.*, packing, encryption, polymorphism) [22, 30, 167–170], and of presenting a known undecidable problem [171].

Within the EMM, to examine whether a composite software component γ_P is malware or not, static analysis approaches extract their information sets from the component's static representation at a given time instant t (*i.e.*, $static(\gamma_P, t)$ as defined in Section 3.5.4). However, static approaches can vary in the subsets of information that they seek to extract from $static(\gamma_P, t)$. Static signature scanners were commonly used anti-malware solutions. In [29], Christodorescu *et al.* experimentally showed that the commercial signature scanners can be easily evaded by using simple obfuscation techniques. In the next subsection, we will use the EMM to provide a formal proof conforming with this result. In particular, we will show in Theorem 5.1 that malware writers can obfuscate the malware signatures by using simple obfuscations to generate novel variants with new signatures that allow the variants to escape detection.

5.6.1.1 The Formal EMM Analysis of Signature Scanners

Signature scanners scan the systems for the signatures of known malware. As discussed in [91, Section 5.2.1, pp. 157], signatures should be able to discriminate between malware classes (*i.e.*, each signature should uniquely characterize a class of malware). Signatures can range from a sequence of instructions, a message displayed by the virus, or even the infection marker itself [91, Section 5.2.1, pp. 158]. Within the EMM, a static signature can be defined as any subset of measurable information that is contained in $static(\gamma_P, t)$. For simplicity but without loss of generality, we will consider the signature to be a sequence of instructions.

Within the EMM, consider a software component γ_P. Consider also an anti-malware scanner denoted as $D_S(.)$. Let $D_S(.)$ have a signature database that will be denoted as Y, where each $y \in Y$ is a signature of each different known malware sample. To decide if γ_P is malware, $D_S(.)$ searches $static(\gamma_P, t)$ and if it finds that there exists

a $y \in Y$, such that, $y \subseteq static(\gamma_P, t)$, then it will report γ_P as malware. That is to say, $D_S[static(\gamma_P, t)] = -1$ if and only if $\exists y \in Y$, such that, $y \subseteq static(\gamma_P, t)$, otherwise $D_S[static(\gamma_P, t)] = 1$.

If the malware writers are able to change the signature of their malware without changing its function, then $D_S(.)$ will no longer be easily able to detect the malware. We will formally show that malware writers are able to obfuscate the instructions of the signature without changing the function of the malware, and hence, will be able to evade the detection. However, before discussing this issue, we must first define the inverse of an instruction as follows.

Definition 5.10 (Inverse Instructions)**.** *Let $\mathcal{M} = \langle M, B, \mathbb{S}, \mathbb{I} \rangle$ be a Maurer computer as defined in Definition 2.2. An instruction $i \in \mathbb{I}$ is called* invertible *if and only if there exists another instruction $i' \in \mathbb{I}$ such that for any two states $s, s' \in \mathbb{S}$, $s \neq s'$, $s' = i(s)$, we have $s = i'(s')$. Hence, i' is called the inverse instruction of i and can be denoted as $i' = i^{-1}$.*

As shown in Definition 5.10, if i^{-1} is immediately executed after the execution of i, the overall effect is that the memory will be restored back to its original state before the execution of i. In other words, the effect of the sequential execution of i and i^{-1} is similar to the effect of the execution of the identity instruction i_{id} (*e.g.*, executing "`add AX,1`" followed by "`sub AX,1`"). Of course, not all instructions are invertible (*e.g.*, the instruction "`del AX`" will delete the contents of `AX` that cannot be restored). Now, Theorem 5.1 shows that malware writers can obfuscate malware and evade signature scanners.

Theorem 5.1. *Signature scanners are not resilient to attacker obfuscations.*

Proof. Proof by construction. We will show that, for a software component γ_P, we can develop an equivalent software components $\gamma_{P'}$ with the same function but

different signature instructions. Hence, attackers can develop new malware variants with the same malicious function, but without the defender known signature sequence that enables the detector to recognize it. Let $D_S(.)$ be a signature scanner with a signature database Y. Let γ_P be malware that has a signature $y \in Y$ (*i.e.*, $D_S(.)$ can detect γ_P by the signature y). Let $\gamma_P = (\dots, i_{k-1}, i_k, i_{k+1}, \dots)$ (*i.e.*, γ_P is defined as a sequence of instructions with i_{k-1}, i_k, and i_{k+1} being sequential instructions in γ_P). The signature y can be either: (i) a sequence of instructions of γ_P, or (ii) a general indexed byte sequence in γ_P. We will prove the theorem for these two cases as follows.

Case (i): The case of y being a sequence of instructions within γ_P. Let y be defined as $y = (i_k, i_{k+1}, \cdots)$ where $y \subset \gamma_P$ (*i.e.*, the signature y is a sequence of instructions that contains i_k, i_{k+1} from γ_P). As discussed in Section 2.3.4, the two instructions $i_k, i_{k+1} \in \gamma_P$ can be replaced by a composite instruction $J = i_k \circ i_{k+1}$ that causes the same state memory changes as that caused by the execution of i_k followed by i_{k+1}, respectively (*i.e.*, i_k, i_{k+1} can be replaced by an equivalent composite instruction J). Next, we need to show that J exists. Since, as discussed in Section 3.4, the input and output regions of instructions are assumed disjoint within the EMM, then the conditions of Theorem 2.4 will be satisfied and, hence, J exists. Now, with the replacement of the instructions of γ_P with composite equivalent ones, we obtained another component $\gamma_{P'}$ that is defined as $\gamma_{P'} = (\dots, k_{k-1}, J, \dots)$, where $\gamma_{P'}$ is equivalent to γ_P (*i.e.*, $\gamma_{P'}$ has exactly the same function as that of γ_P). However, $\gamma_{P'}$ does not contain the instruction sequence $i_k, i_{k+1} \in y$ and, hence, cannot be detected by $D_S(.)$ using the same signature y (*i.e.*, $y \not\subset \gamma_{P'}$).

Case (ii): The case of y being a general indexed byte sequence within γ_P. Let i be an invertible instruction as defined in Definition 5.10. Hence, we can insert an i followed by i^{-1} in γ_P and obtain $\gamma_{P'} = (\dots, i_{k-1}, i, i', i_k, i_{k+1}, \dots)$ where, $\gamma_{P'}$ has

the same function as γ_P (because the insertion of i, i' is the same as inserting the identity instruction i_{id}) but the insertion of i and i' will change the indexes of the bytes sequence of the signature y. Hence, $\gamma_{P'}$ will evade detection by $D(.)$.

As shown in the above two cases, we can easily develop a composite component $\gamma_{P'}$ that is equivalent to γ_P but consists of different instructions, where $\gamma_{P'}$ does not include the instructions of the signature y used by $D_S(.)$ to detect γ_P. □

As discussed above, Theorem 5.1 provides a formal proof for the well-known experimental results reported in the literature that signature scanners can be easily evaded by modifying the signatures' instructions (*e.g.*, [29,33,79]). The theorem also formally shows that signature scanners must necessarily suffer from the lack of resilience against the attackers' adaptations. Additionally, Theorem 5.1 also describes one of the obfuscation techniques that can be used to generate novel malware variants without changing the malicious function. Moreover, by the arbitrarily construction of the required composite instruction J, many obfuscated variants can be obtained.

It should be noted that, the obfuscation discussed in Case (i) in the proof of Theorem 5.1 above is commonly denoted as *equivalent code* obfuscation where the sequences of instructions can be replaced in the programs without changing the function of the code [172]. For example, the assembly instruction `inc AX` which increments the contents of the register `AX` can be replaced by `add AX,1`. The obfuscation discussed in Case (ii) is known as *junk code insertion obfuscation* [172]. Other code obfuscations, such as *no-op insertion obfuscation* where inserting the *no-op* instruction (*i.e.*, i_{id}) will not change the code function [172], can also be included in the above description. Additionally, if i_{y_k} and $i_{y_{k+1}}$ satisfy the conditions of Corollary 2.5, then the order of execution of the two instructions can be exchanged without affecting their composite function, resulting in *code reordering obfuscation* [172]. It should be noted that, Theorem 2.2 and Theorem 2.3 can both be used to construct arbitrary

Algorithm 5.1 Signature extraction algorithm.

```
1: function SIGNATURE(γ_P)
2:     y ← ∅
3:     for all i_k ∈ γ_P do
4:         Replace i_k by any i ∈ 𝕀 where i ≠ i_k to obtain a variant γ_P′
5:         if D_S(γ_P′) == −1 then
6:             y = y + i_k
7:         end if
8:     end for
9:     return y
10: end function
```

sequences of instructions with the same function (*i.e.*, by the arbitrarily composition or decomposition of the instructions).

It should be observe that the proof of Theorem 5.1 relies on the assumption that the attackers have the full knowledge of the defender's known malware signatures, and based on that, they are then able to intelligently perform the required obfuscations necessary to generate the novel variants. Consequently, it can be argued that protecting the signature databases (*e.g.*, by using encryption) should limit the attackers' ability to generate novel obfuscated variants and increase the resilience of the scanners. We disagree with this claim by showing that attackers can easily extract the signatures used by the scanner to detect the malware thereby allowing them to optimize their obfuscation efforts as the exact instructions needed to be obfuscated can be specified by the attackers. In particular, attackers can use black-box analysis techniques to find the signature. For example, in [136], Mutz *et al.* proposed a reverse engineering process and tool for signature matching mechanisms in network-based IDS. This can be generalized under the EMM as follows.

Let $\gamma_P = (i_1, i_2, \ldots, i_N)$ be an instruction sequence that corresponds to malware, where N is the number of instructions constituting γ_P. Algorithm 5.1 demonstrates that the signature y that the signature scanner D_S uses to detect γ_P can be extracted, where $+$ denotes the append operation. In particular, to extract a signature y used

by the signature scanner $D_S(.)$ to detect a malware γ_P, Algorithm 5.1 proceeds as follows. First, it initiate y to $\emptyset$ (*i.e.*, y is initially empty) (Line 2). For each instruction $i_k \in \gamma_P$, Algorithm 5.1 replaces i_k by another arbitrary instruction i to obtain another component $\gamma_{P'}$ (Line 4). Then, $\gamma_{P'}$ will be scanned by $D_S(.)$ and if $D_S(\gamma_{P'}) = 1$ (*i.e.*, $D_S(.)$ recognizes $\gamma_{P'}$ as benign), then i_k is a signature instruction that, when replaced, caused $D_S(.)$ to be evaded (Line 5). Hence, i_k will be added to y (Line 6). Note that, Algorithm 5.1 is of order $O(n)$ (where n is the number of instructions of the malware) and it is similar to those discussed by Filiol in [79] and Christodorescu *et al.* in [29]. Clearly, by implementing a simple algorithm like that of Algorithm 5.1, malware writers can easily extract the signatures used by the scanner to detect the malware, under the proviso that they can know if their malware was detected, *i.e.*, via the use of a watch dog timer or similar constructs within the malware.

5.6.2 Dynamic Detection Approaches

Dynamic malware detection approaches attempt to detect malware during or after the execution of the analyzed programs. The basic principle of dynamic detection is that the behavior of a program can be described by its interactions with its surrounding execution environment [173]. Examples of dynamic approaches are those of [16, 81, 146, 173–176]. The core advantage of dynamic detection is its relative immunity to various binary code obfuscation techniques. Whereas, its core disadvantage is that it only extracts the detection information from the executed traces of the analyzed programs, and hence, it cannot obtain any information from any unexecuted code segments. As discussed in [24], typically only limited subsets of malicious behaviors can be observed within the short time frames of malware executions. To mitigate this effect, approaches exist to explore and analyze all possible execution paths, such as those of [28, 177–179]. However, as discussed in [180, 181], these approaches tend

to offer unacceptably low operational performance. Dynamic analysis can be evaded through special techniques, such as: crafting their malware in a way that it mimics the behavior of the benign programs (*e.g.*, mimicry attacks [31, 32]), delaying the execution of the malicious payloads [27], restricting the execution of the malicious payloads to the existence of specific triggering conditions (*e.g.*, logic and time bombs [91, 104, 114]), *etc.*

Note that, dynamic analysis approaches execute the analyzed components γ_P over a time period $\tau \subseteq \boldsymbol{T}$ and extract their detection information from the sets of dynamic information $dynamic(\gamma_P, \tau)$ as defined in Section 3.5.4. However, dynamic approaches will vary in the subsets of information that they seek to extract from $dynamic(\gamma_P, \tau)$. Dynamic system calls approaches are widely used to analyze malware (*e.g.*, [1, 14, 49, 146, 182–185]). Hence, we will provide a formal analysis of dynamic system call sequence as follows.

5.6.2.1 The Formal Analysis of System Call-based Dynamic Approaches

Generally, system calls are special instructions that enable programs to invoke the OS services. For example, the instruction sequence shown in Figure 5.5 shows an assembly code fragment from the worm variant "*Bagle.J*" quoted from [1], where the instructions i_6, i_{11}, i_{14}, and i_{16} are the system calls, where the *Bagle.J* code serves solely as one such example.

As shown in Figure 5.6, for each system call made by the program, the sequence of system calls inside the window is compared against a database of known malicious sequences to decide whether this sequence is or is not malware. Corollary 5.1 shows via the EMM that these detection solutions can be evaded (or equivalently, call sequences can be obfuscated) as follows.

Corollary 5.1. *System call sequences malware analysis approaches can be evaded.*

I_1:	*push*	*ebp*
I_2:	*mov*	*ebp, esp*
I_3:	*add*	*esp, 0FFFFFFFCh*
I_4:	*push*	*offset aSoftwareDatetime*
I_5:	*push*	*80000001h*
$\boldsymbol{I_6}$:	**call**	**RegDeleteKeyA**
I_7:	*lea*	*eax, [ebp+hKey]*
I_8:	*push*	*eax*
I_9:	*push*	*offset SubKey*
I_{10}:	*push*	*80000001h*
$\boldsymbol{I_{11}}$:	**call**	**RegCreateKeyA**
I_{12}:	*push*	*offset ValueName*
I_{13}:	*push*	*[ebp+hKey]*
$\boldsymbol{I_{14}}$:	**call**	**RegDeleteValueA**
I_{15}:	*push*	*[ebp+hKey]*
$\boldsymbol{I_{16}}$:	**call**	**RegCloseKey**

Figure 5.5: *Bagle.J* code fragment quoted from [1].

Proof. Proof by construction. We will show that, for an EMM software component γ_P, we can develop an equivalent software components $\gamma_{P'}$ with the same function but a different system call instruction sequence and, hence, attackers can develop novel variants with the same malicious function and different system call instructions. Let $D_C(.)$ be a system call sequences malware detection system that utilizes a database Y of known malicious sequences. Let γ_P be malware that is defined as $\gamma_P = (\cdots, i_k, i_{k+1}, \cdots)$, where i_k is a system call instruction. As per the proof of Theorem 5.1 above, there will also be two possible cases to consider. Those cases are as follows.

Case (i): The first case is when the system call instruction can be composed with another instruction. Let $y \in Y$ be the system call sequence used to detect γ_P, hence, y can be defined as $y = (\cdots, i_k, \cdots)$. As discussed in the proof of Theorem 5.1 (page 131), by the composition of the two instructions $i_k, i_{k+1} \in \gamma_P$, a software

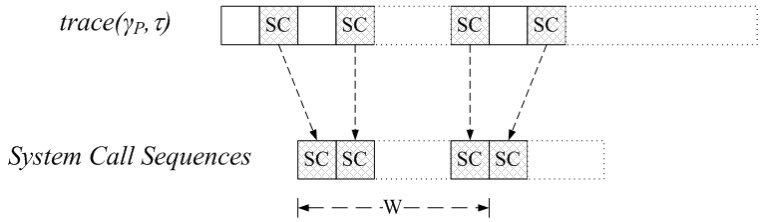


Figure 5.6: System call sequences.

component $\gamma_{P'} = (\cdots, J, \cdots)$ that is equivalent to γ_P can be obtained, where $J = i_k \circ i_{k+1}$.

Case (ii): The second case is when the system call instruction cannot be composed with other instructions. Hence, y can be obfuscated by inserting additional system call instructions into γ_P (*i.e.*, in a similar manner as to the idea of inserting i and i^{-1} in Theorem 5.1's proof), such that, their overall effect is as inserting i_{id} or performing any additional state changes that do not impact the malicious behaviors of γ_P. Hence, a software component $\gamma_{P'}$ that is equivalent to γ_P can be easily developed, such that, $y \not\subset \gamma_{P'}$. □

Corollary 5.1 follows directly from generalizing Theorem 5.1 and shows that, similar to static signature scanners, dynamic system call approaches can also be easily evaded by obfuscating the system call sequences. These results agree with those of the prior experimental work. Particularly, as discussed in [31, 32], *mimicry attacks* are equivalent sequences of system calls with the same malicious function but with different system calls. Additionally, as discussed in [77], *functional polymorphism* can also be used to obtain equivalent code sequences yet with different system calls.

Hence, using the EMM, Corollary 5.1 formally showed that dynamic malware analysis approaches based on system call sequences are not resilient to attacker adaptations as these sequences can be easily obfuscated while retaining their malicious

behavior. Accordingly, the EMM provides a useful model for reasoning about the resilience of these forms of malware detection system.

5.6.3 Discussion

Within the EMM, the distinction between static and dynamic detection approaches is solely in the source of the detection information. In particular, to detect whether a composite software component γ_P is malware or not, static detection approaches extract their information sets from $static(\gamma_P, t)$ as defined in Equation (3.21), whereas dynamic detection approaches extract their detection information from the set of dynamic information $dynamic(\gamma_P, \tau)$ as defined in Equation (3.20). Hence, the core difference is solely in how the feature mapping $f(.)$ is implemented. As discussed in Section 5.4.3, $f(.)$ is defined generally to include any mapping $f : \mathbb{S}(\boldsymbol{T}) \rightarrow \mathcal{X}$ that preserves the EMM measurability within $\mathcal{X}$. Accordingly, the EMM can model both static and dynamic approaches as well as detectors that may combine both static and dynamic information sets. Additionally, in the previous section, the EMM has been shown to be able to model anomaly-based and signature-based detection approaches, where the distinction between them is in solely terms of the way the decision boundary $d(.)$ is defined. Hence, the EMM can cover all combinations of signature-based, anomaly-based, static, and dynamic detection approaches via the proper definitions of $f(.)$ and $d(.)$.

5.7 The Completeness of the EMM with respect to Detection Solutions

In Section 5.6 and Section 5.5, it has been shown that, the standard categories of detection approaches can be modeled via the EMM through its measure-theoretic

properties as developed in this chapter. In this section we generalize this discussion to formally showing that *all* implementable malware detectors can be defined in terms of the EMM, as illustrated by the following theorem.

Theorem 5.2. *The EMM is complete in the sense that it can model all malware detection approaches that extract their detection information from the EMM.*

Proof. Proof by contradiction. Assume that $D(.)$ is a detector that only extracts its information from a system for which an EMM has been constructed but that $D(.)$ itself cannot be modeled by the EMM. By definition, $D(.)$ as a detector, must partition some measure space $\mathcal{X}$ into the subspaces $\mathcal{X}^-$ and $\mathcal{X}^+$ (*i.e.*, it must implement a decision mapping $d(.)$). Therefore, assume an event $e \in \mathcal{E}$ is denoted as attack under $D(.)$. Assume also that there exists some $e' \in \mathcal{E}$ where $e' \neq e$ and e' is benign. Then it should be the case that $D(e) \in \mathcal{X}^-$ and $D(e') \notin \mathcal{X}^-$. Hence, from the EMM's consistency principle, there must exist, at least, a measurable difference in information between e and e' such that $D(e) \in \mathcal{X}^-$ and $D(e') \notin \mathcal{X}^-$. But the existence of this measurable information would allow $D(.)$ to be modeled within the EMM (*i.e.*, with this measurable information, the required $d(.)$ decision function can be constructed). Therefore, it must be the case that $e = e'$ under the EMM. More particularly, as per Section 5.2, a decision function $d(.)$ is constructible between two pattern classes ω^+ and ω^- as long as $Pr_{\omega_1}(.) \neq Pr_{\omega_2}(.)$. By definition, the e and e' must represent different random variables and, by the consistency principle, $Pr_{\omega_1}(.) \neq Pr_{\omega_2}(.)$ if $e \in \omega^+$ and $e' \in \omega^-$. If $D(e) = D(e')$, then given that $D(.)$ knows that $e \neq e'$ whereas as per the above, it must be the case that $e = e'$ under the EMM, it must be the case that $D(.)$ is using information that is not contained within the EMM (*i.e.*, $D(.)$ by definition then cannot be modeled within the EMM). But this contradicts the assumption that $D(.)$ only uses information contained within the EMM to assess the maliciousness of e and e'. □

Hence, as shown in the above theorem, the EMM measure-theoretic model of malware detection is complete in the sense that it can be used to model all types of malware detectors that extract their measurable detection information from the information sets modeled by the EMM (*i.e.*, all detection approaches based on the analysis of measurable information are modeled under the EMM).

5.8 Summary

In the chapter, a general model of the malware detection problem using the EMM has been developed. The EMM has been extended by defining a probability space over it for all events that can occur within its modeled information space. This, in turn, was shown to allow the EMM to be used to provide formal models for the analyses of various categories of malware detection approaches. Particularly, the EMM modeling of the anomaly-based and signature-based approaches was discussed. Additionally, the formal EMM modeling of static and dynamic malware detectors was also discussed. It was then formally shown via the developed EMM that static signature scanners and dynamic system call sequence malware detection solutions can be easily evaded. Finally, the completeness of the EMM model was developed via its measure theory properties and it was shown to be complete in the sense of being able to model all implementable malware detectors that extract their measurable information from the EMM's modeled information, where Chapter 3 showed that this denoted all information that can exist within the modeled defended environment over $\boldsymbol{T}$.

Chapter 6

Game Theoretic Analysis

6.1 Introduction

As discussed in Section 1.1, an arms-race exists in which the system defenders develop better malware detection approaches only to have the attackers develop techniques to bypass each next generation of deployed defenses. This motivates the study of this ongoing confrontation. Arguably, a better understanding of the nature of the confrontation and the analysis of the strategies of the attackers and defenders should enable the development of more effective defenses. More particularly, *game theory* provides the mathematical framework to formally reason about the interactions of rational and intelligent adversarial decision makers. Hence, this chapter exploits the developed EMM to construct a game-theoretic model to analyze attacker-defender confrontations occurring within the context of malware and its detection within the EMM defined defended environments.

Game theory typically focuses on analyzing games to find their corresponding *Nash equilibria* (NE). More particularly, the NE are the set of strategies of the players in the game under which no player is willing to unilaterally change his/her own strategy

assuming the other players' strategies are themselves kept unchanged [186, Chapter 2]. NE are the most widely acceptable solution concept in game theory since it captures the eventual steady-sate solutions to which games eventually must converge [187, Chapter 3]. Another form of optimal solutions exists in cooperative games which is termed the *Pareto optimal solution* (or *Pareto optimality*) [188, Chapter 1, pp. 5]. Pareto optimality differs from NE in that it assumes that the players within a game can coordinate their selection of strategies to play whereas NE presumes that all players' strategies are always arrived at independently. Pareto optimality is typically applied, within social sciences and economics games, where the game should lead to socially desirable outcomes [189, Section 3.3, pp. 57]. As this dissertation is only concerned with attacker-defender confrontations, this work focuses solely on NE and not on Pareto optimal solutions.

It should be noted that, much of the prior work in which game theory has been applied to malware and intrusion detection solutions has tended to focus on the analysis of specific security games (*i.e.*, a specific game is constructed and its NE are computed). This work applies game theory in a different manner in that it uses the EMM to enable the modeling of the sequence of interactions that arise as the attackers and defenders adjust their strategies in light of what they learn about what the other does. More particularly, the EMM provides a complete model of the defended environment in, as has been shown in Chapters 4 and 5, in that it can model all implementable malware and malware detection solutions deployable within the modeled defended environment. Hence, by this EMM facilitated approach, it will be shown that the attacker-defender confrontations can be modeled as a sequence of games that arise as each side adapts in order to maximize their results. Using the dynamical systems theory, this sequence of games can then be analyzed to deter-

mine the conditions required if the sequence is to eventually converge to a defender advantageous (or defender winnable) end-game.

The remainder of this chapter is organized as follows. Section 6.2 provides the background materials for the game theory and dynamical systems concepts that will be used. Section 6.3 discusses the prior work in applying game theory to the analysis of malware and intrusion detection problems. Section 6.4 applies the developed EMM to provide a game theory based model for the attackers-defender interactions. Section 6.5 shows that the developed game evolves over time as an iterative sequence of sub-games. Section 6.6 then analyzes this sequence of games to specify the scenarios under which it could converge. Section 6.7 then studies this sequence in more detail and formally derives the conditions required if the iterative sequence of sub-games is to converge to a defender-advantageous end-game. Section 6.8 provides a discussion of the modeled sequence and the derived analysis results. Finally, Section 6.9 summarizes this chapter.

6.2 Background Material

For the completeness of this dissertation, this section provides a brief overview of the fundamentals of game theory (Section 6.2.1) and dynamical systems theory (Section 6.2.2). For more detailed information about game theory, we refer the reader to [187], [188], [186], and [189]. Whereas, for more details about dynamical systems theory, we refer the reader to [141] and [190].

6.2.1 Principles of Game Theory

Game theory provides a mathematical framework to analyze multi-person decision-making scenarios. It seeks to model the interactions among *rational*[1] decision makers who have to choose actions that might be conflicting. Without loss of generality, a basic assumption in game theory is that decision makers are *rational* and *intelligent*[2]. In game theory, a *game* consists of: (i) a set of decision-makers called the *players*, (ii) a set of *strategy sets* of the players, and (iii) a set of *utility functions* for the players [187, pp. 46]. A player can be a machine, a person, a group of persons, *etc.* In any game, the players are assumed to maximize their payoffs according to their preferences, as expressed mathematically by their utility functions which map the possible consequences of their strategies onto the real numbers.

There are different classifications of games depending on: (i) the nature of the interactions between the players, (ii) the information the players know about each other, and (iii) the strategy sets of the players. If the players are concerned only with maximizing their own payoff, then the game is called a *non-cooperative* game, whereas, if the players form coalitions in order to coordinate their strategies, then the game is called a *cooperative* game [188, Section 1.1]. If the players have the complete knowledge about their own strategies and payoffs as well as the other players' strategies and payoffs, then the game is termed to be one of *complete information*, whereas if at least one player does not have the complete knowledge of the strategies and/or payoffs of any other player, then the game is termed to be one of *incomplete*

[1]The decision-makers are *rational* if they take decisions that consistently maximize their own utilities [187, pp. 2].

[2]The decision-makers are *intelligent* if they know everything about the game and can make all inferences about the situations of the game [187, pp. 4]. Hence, if a theory that describes the behavior of intelligent players in some game has been developed and it is believed that this game is correct, then it should be assumed that each player in the game will understand this theory and any subsequent predictions derivable from it [187, pp. 4].

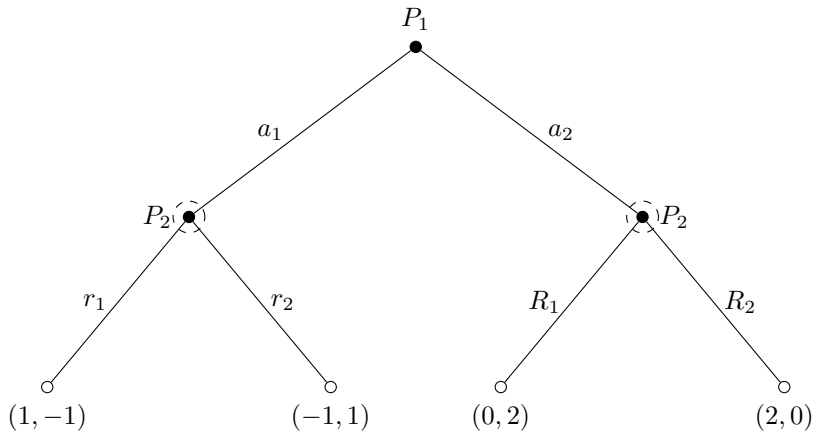


Figure 6.1: An example of an extensive form 2-player game.

information [187, pp. 67]. Finally, if all strategy sets of the players are finite, then the game is termed *finite*, otherwise, it is termed *infinite* [187, pp. 140].

There are two different forms for games according to the nature of the decision making process. If all players decide their strategies simultaneously or independently without having any information about the selected strategies of any other player, then the game is called a *normal* or a *matrix* form game [189, Chapter 3]. Whereas if the players decide their strategies at different times or any of them has access to information about some of the strategies selected by any of the other players prior to selecting their decisions, then the game is called an *extensive* or a *tree* form game [189, Chapter 7]. Extensive form games are more suitable to capture the dynamics of potential information exchange among the players. Figure 6.1 shows an example of an extensive form game that has two players P_1 and P_2, where P_1 has two strategies ($\{a_1, a_2\}$) and P_2 has four strategies ($\{r_1, r_2, R_1, R_2\}$). As shown in the figure, P_1 is assumed to act first and then P_2 acts after observing the actions of P_1. Also, P_2 is assumed to perfectly observe P_1's actions and he/she can discriminate whether P_1 used actions a_1 or a_2. This is represented by the dashed circles around P_2's nodes, which are called the *information sets* of P_2 (*i.e.*, P_2 has two information sets). The

ordered pairs at the terminal nodes represent the payoffs of P_1 and P_2, respectively, according to their different selections of strategies.

6.2.1.1 Formal Definition of Games

Without loss of generality the *game* will be denoted as G and will be defined as follows [187, Chapter 2].

Definition 6.1 (Definition of games)**.** *A* game, *G, is defined as the tuple $G = \langle \mathcal{N}, \mathbf{\Sigma}, \mathcal{U}(.) \rangle$, where:*

- *$\mathcal{N}$ is a set of $N \geq 2$* players *(*e.g.*, attackers and network defenders, malware and malware detection systems,* etc.*). Players are the basic entities of the game.*
- *$\mathbf{\Sigma} = \{\Sigma_1, \Sigma_2, \cdots, \Sigma_N\}$ is a set of N* strategy sets *of the players. In particular, for $j = 1, 2, \cdots, N$, each $\Sigma_j \in \mathbf{\Sigma}$ is a nonempty set of the strategies that player j can take in the game. A* strategy profile *is $\boldsymbol{a} = (a_1, a_2, \cdots, a_N) \in \times_{j=1}^{N} \Sigma_j$, where $\forall j = 1, 2, \cdots, N$, it is the case that $a_j \in \Sigma_j$.*
- *$\mathcal{U}(.)$ is the set of N* utility functions *(or* payoff functions*) of the players. Specifically, $\forall j = 1, 2, \cdots, N$, it is the case that $u_j(.) \in \mathcal{U}(.)$ where $u_j(.)$ denotes the utility function of player j that assigns a payoff value for each combination of the players' strategies. Hence, $u_j(.)$ is a mapping $u_j : \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_N \rightarrow \Re$, where $\Re$ is the set of real numbers.*

6.2.1.2 Nash Equilibrium

As discussed in [186, Chapter 2], *Nash equilibria* (NE) are the most commonly used solution concept in game theory as it captures the eventual steady-state of the play of a strategic game in which each player acts rationally. Let $\boldsymbol{a}^* = (a_1^*, a_2^*, \cdots, a_N^*)$ be

a strategy profile. Let a_{-j} denote the profile of strategies of all players except player j. Hence, NE is defined as follows.

Definition 6.2 (Nash Equilibrium)**.** *Nash Equilibrium is a set of strategies under which no player is willing to unilaterally change his/her own strategy if other players' strategies are kept unchanged* [186, Chapter 2]. *Formally,* $\boldsymbol{a}^*$ *is an NE if and only if,*

$$\forall a_j \in \Sigma_j, \quad u_j(a_j^*) \geq u_j(a_j, a_{-j}^*), \quad \forall j = 1, 2, \cdots, N, \tag{6.1}$$

where $u_j(a_j^*)$ *is the utility of* j *when all players in the game use their NE strategies and* $u_j(a_j, a_{-j}^*)$ *is the utility when player* j *uses different strategies than the NE strategies while all other players use their NE strategies.*

In general, as discussed in [187, Theorem 3.1, pp. 95], there exists at least one NE in *mixed strategies*[3] for finite strategic form games, where as discussed in [187, Section 3.12, pp. 136], the proof of this theorem relies on the *fixed point theorem* of Kakutani. As will be discussed in details in Section 6.4, under the EMM, the attacker-defender confrontation can be modeled as an extensive form, finite, two-person, non-zero sum, non-cooperative game, hence, it is important to show that NE exists for this game. Before discussing the existence of NE, we will first define the notions of *behavioral strategies* and *perfect recall* for extensive form games as follows.

Definition 6.3 (Behavioral Strategies)**.** Behavioral strategies *in extensive form games specify an independent probability distribution over the strategies of the players at each information set [189, Definition 7.6, pp. 140].*

Definition 6.4 (Perfect Recall)**.** *A* perfect recall *in extensive form games is the assumption that, whenever a player acts, the player can perfectly remember all previous strategies made in the game.*

[3] A mixed strategy of a player is a set of probability distributions over the player's action set [186, Chapter 3, pp. 32].

Without loss of generality, as shown in Theorem 6.1, NE exists for such extensive form games. Particularly, when the game is repeated for a sufficiently large number of times, the players are guaranteed to converge to using only their NE strategies.

Theorem 6.1. *For any extensive form game with perfect recall, NE exists in behavioral strategies.*

Proof. The proof can be found in [187, Theorem 4.3, pp. 162]. □

As discussed in [187, Section 4.2], the proof of Theorem 6.1 of the existence of NE in extensive form games is structured based on the existence of NE in strategic form games, where this can be proven to hold via the application of Kakutani's fixed point theorem [187, Section 3.12, pp. 136]. As players repeatedly play a game for a sufficiently large number of times, then the players will necessarily converge to only using their NE strategies. Hence, Theorem 6.1 guarantees the existence of NE for the forms of games of interest in this work to model attacker-defender confrontations. Moreover, these NE denote the points of interest to study within such game as these are the points to which the arms-race must over time converge to. Note that, as discussed in [187, Section 3.4, pp. 106], rational and intelligent players will only use their NE strategies in the game (or converge to use their NE strategies) since if they were use different strategies, then they would notice that they could unilaterally change their played strategies to NE strategies and, thereby, obtain improved payoffs. This allows game theory-based analyses to focus on assessing the characteristics of the NE of the games under study to the exclusion of the remainder of the game's strategy space.

6.2.1.3 Strategy Domination

The concept of domination in player strategies is an important concept in game theory as it allows the players to reduce the set of strategies they need to consider

by eliminating any *strictly dominated strategies*. In particular, as discussed in [187, Section 2.5, pp. 57], a strictly dominated strategy for the player j in the game by definition can never be j's best response no matter what j may believe or know about the other players' strategies. In particular, dominated strategies are formally defined as follows.

Definition 6.5 (Dominated Strategies)**.** *Let $a_k, a_{k'}$ be two strategies for the player j in a game G. The strategy a_k is termed* dominated *by $a_{k'}$ for player j in a state of the game [187, Section 2.5], if and only if,*

$$u_j(a_{k'}, a_{-j}) \; > \; u_j(a_k, a_{-j}). \tag{6.2}$$

Additionally, *strictly dominated strategies* are defined as follows.

Definition 6.6 (Strictly Dominated Strategies)**.** *If a strategy a_k of player j is a dominated strategy in all states of the game G, then a_k is termed a* strictly dominated strategy *for the player j in the game G [189, Chapter 4].*

As shown in Equation (6.2), j is always better off playing $a_{k'}$ than playing a_k (*i.e.*, by playing any other $a_{k'} \neq a_k$, player j will receive a higher payoff irrespective of the state of the game G). Hence, rational players will never use strictly dominated strategies as other strategies yielding higher payoffs will always exist [189, Claim 4.1, pp. 60]. Consequently, strictly dominated strategies can never, by definition, be within the NE of the game G because the rational player can and would always elect to unilaterally change his/her own strategy to obtain a higher payoff, which contradicts the definition of NE.

Hence, strictly dominated strategies can be eliminated without affecting the structure of the game. Particularly, the *iterated elimination of dominated strategies* is a weak solution concept for games in that it iteratively removes dominated strategies

as these cannot be within the games NE[4] [189, Section 4.2]. More particularly, eliminating strictly dominated strategies does not affect the analysis of the game as such strategies will never by used by rational and intelligent players [187, Section 2.5, pp. 58]. As will be seen in Section 6.4, this solution concept can be applied to simply analyze the forms of attackers-defender confrontations of interest in this work.

6.2.2 Dynamical Systems Theory

In this section, the concept of *dynamical systems* and *dynamical systems theory* will be discussed. This is important since, as will be shown in Section 6.6, we will use game theory in combination with dynamical systems theory to reason about how the attackers-defender confrontations evolve over time. In particular, the reasoning about how the event probabilities of Chapter 5 may or may not change over time as a result of the attackers' adaptation requires the application of dynamical systems theory constructs. It should be emphasized that, the majority of this subsection is paraphrased from [141], in part to ensure the accuracy and completeness of the presented definitions.

Let $\langle \Omega, \mathcal{F}, \mu \rangle$ be a measure space, where Ω is the standard sample space denoting all possible events, $\mathcal{F}$ is a σ-algebra of subsets of Ω, and μ is a measure defined on $\mathcal{F}$ as discussed in Section 5.3. Without loss of generality, a dynamical system is defined as a measure space $\langle \Omega, \mathcal{F}, \mu \rangle$ equipped with a *measurable transformation* $\mathcal{T} : \Omega \to \Omega$ [141, Section 2.3, pp. 42]. We will begin by introducing *transformations*, *measurable transformations*, and *measure preserving transformations* in Definition 6.7 which will then be used to define dynamical systems in Definition 6.8.

[4]The term "*weak solution concept*" is used as iterative elimination of dominated strategies removes strategies that cannot be part of the NE as opposed to directly seeking to solve for the NE.

Definition 6.7 (Transformations, Measurable Transformations, and Measure Preserving Transformations)**.** *Let $\langle \Omega, \mathcal{F}, \mu \rangle$ be a probability space, then:*

(i) *A* transformation $\mathcal{T}$ *is defined as a map* $\mathcal{T} : \Omega \to \Omega$.

(ii) $\mathcal{T}$ *is a* measurable transformation *if and only if* $\forall A \in \mathcal{F}$ *we have* $\mathcal{T}^{-1}(A) \in \mathcal{F}$, *where* $\mathcal{T}^{-1}$ *is the inverse of* $\mathcal{T}$ *[190, Definition 1.1, pp. 19].*

(iii) $\mathcal{T}$ *is a* measure preserving transformation *if and only if* $\mathcal{T}$ *is a measurable transformation, such that,* $\forall A \in \mathcal{F}$, $\mu[\mathcal{T}^{-1}(A)] = \mu[A]$ *[190, Definition 1.1, pp. 19].*

It should be noted that, as discussed in [141, Chapter 2, pp. 35], a typical transformation $\mathcal{T}$ of interest in engineering domains is the one that corresponds to the *time shift operator*. Since we are seeking to analyze the attacker-defender interactions over time, then the transformation $\mathcal{T}$ of interest in this work is also this time shift operator. Next, *dynamical systems* and *stationary dynamical systems* can now be defined as follows.

Definition 6.8. *Let $\langle \Omega, \mathcal{F}, \mu \rangle$ be a measure space and $\mathcal{T} : \Omega \to \Omega$ be the time shift operator, then:*

(i) *If* $\mathcal{T}$ *is a* measurable transformation*, then the tuple* $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$ *is called a* dynamical system *[141, Section 2.3, pp. 42].*

(ii) *Furthermore, if* $\mathcal{T}$ *is a* measure preserving transformation*,* one-to-one[5]*, and* onto[6]*, then the dynamical system* $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$ *is a statistically* stationary dynamical system *[192, Chapter 9, pp. 429], [141, Section 5.9, pp. 152].*

[5]A map $\mathcal{T} : \Omega \to \Omega$ is one-to-one when no two distinct inputs give the same output [191, pp. 37]. Hence, $\forall x_1, x_2 \in \Omega$, if $\mathcal{T}(x_1) = \mathcal{T}(x_2)$, then it must be the case that $x_1 = x_2$.

[6]A map $\mathcal{T} : \Omega \to \Omega$ is onto (or surjective) if the codomain is the range of $\mathcal{T}$ [191, pp. 39].

As discussed in [141, Section 2.3, pp. 42], the name dynamical system comes from the focus on the long-term dynamics (or dynamical behavior) of the repeated application of the transformation $\mathcal{T}$ on the underlying measure space. As shown in Definition 6.8 above, the probabilities of the events remain unchanged over time within stationary dynamical systems, which is a key concern if decision functions are to be constructed based these event probabilities.

Now, $\forall A \in \mathcal{F}$ in the dynamical system $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$, the sets $\mathcal{T}(A)$ are the sets that are produced after the application of $\mathcal{T}$. Additionally, $\mathcal{T}^2(A) = \mathcal{T}[\mathcal{T}(A)]$. In general, for any $j \in \{1, 2, \cdots\}$, $\mathcal{T}^{j+1}(A) = \mathcal{T}[\mathcal{T}^j(A)]$. Next, we introduce the important definition of *weakly wandering sets* for measurable transformations as follows.

Definition 6.9. *Let $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$ be a dynamical system. A measurable set $W \in \mathcal{F}$ of positive measure is called a* weakly wandering set *for the measurable transformation $\mathcal{T}$ if there exists a countable sequence $n \to \infty$, such that, $\mathcal{T}^k(W) \cap \mathcal{T}^{k'}(W) = \emptyset$ for all $k \neq k'$ where $k, k' \in n$ [193, pp. 330]. To be measure-preserving, a transformation cannot give rise to weakly wandering sets [193, pp. 330].*

Hence, from Definition 6.9, it is sufficient to show that there exists weakly wandering sets for a transformation $\mathcal{T}$ to prove that $\mathcal{T}$ is not measure preserving. Additionally, since the stationarity of a dynamical system requires $\mathcal{T}$ to be measure preserving, then the existence of weakly wandering sets suffices to prove that the dynamical system described by $\mathcal{T}$ is non-stationary (*i.e.*, that the associated event probabilities are themselves time-dependent quantities). In Section 6.7.2, we will show that weakly wandering sets exist with the iterative sequence of sub-games that describes the attacker-defender malware arms-race. Hence, any malware measurement features that span these wandering sets must describe (or denote) non-stationary random processes. This observation is important as it formally states that the defenders must track non-stationary attack behaviors if they are to successfully defend against

the next attacker generated malware variant. Hence, by definition, any and all past information the defender may have is not sufficient to determine how the defenses should be improved. This is important as the EMM's events formally described by (or as) random variables with the detection measurement features then describing random processes.

6.2.3 Random Variables and Random Processes

In this section, the concepts of *random variables* and *random processes* from a *dynamical systems* perspective will be discussed.

6.2.3.1 Random Variables

A *random variable*, X, is commonly defined as: *a real function of the elements of a sample space* Ω [194, Chapter 2, pp. 41]. Let $\langle \Omega, \mathcal{F} \rangle$ be a measurable space, where $\mathcal{F}$ is a σ-algebra of subsets of Ω. As discussed in [141, Section 2.1], a random variable is called a *measurable function* that is defined as follows.

Definition 6.10. *A* measurable function $g(.)$ *defined on* $\langle \Omega, \mathcal{F} \rangle$ *is the mapping* $g : \Omega \to \Omega$ *with the property that,*

$$if\ A \in \mathcal{F},\ then\ \ g^{-1}(A) \in \mathcal{F}. \tag{6.3}$$

Hence, as discussed in [141, pp. 36], a measurable function is just a mapping with the property that inverse images of the output events are also events in the defined measurable space.

6.2.3.2 Random Processes

As discussed in [142, Chapter 2, pp. 49], a *random process*, $X(t)$, assigns a random function of time as the outcome of a random experiment. By considering $\mathcal{T}$ to be time shift operations, we can define random processes from the perspective of dynamical systems as follows [141, Section 2.3, pp. 40].

Definition 6.11. *Let $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$ be a dynamical system where $\mathcal{T}$ is a time shift. As discussed in [141, Section 2.3], a random process $\{X_n | n \in \mathcal{I}\}$, where $\mathcal{I}$ is an index set, is defined as the dynamical system $\langle \Omega, \mathcal{F}, \mu, \mathcal{T} \rangle$ together with a measurable function $g(.)$. Hence, $\forall A \in \mathcal{F}$, $\forall n \in \mathcal{I}$,*

$$X_n(A) = g[\mathcal{T}^n(A)] \tag{6.4}$$

Additionally, a statistically stationary[7] random process can also be defined as follows.

Definition 6.12 (Stationary Random Processes)**.** *A random process is statistically* stationary *if and only if its corresponding dynamical system is stationary with respect to the shift operator $\mathcal{T}$ [141, Section 5.9, pp. 152]. A stationary random process is the one for which the probability of any event is the same regardless of whenever in time the event occurs [141, Section 5.9, pp. 152].*

As shown in Definition 6.12 above, the statistical stationarity of a random process requires the stationarity of its underlying dynamical system, which also requires the transformation $\mathcal{T}$ to be a measure preserving transformation, as per Definition 6.8. Hence, of concern in this work is to assess whether or not the defender's measurement features $F(.)$ under the EMM are measure invariant with respect to the attackers'

[7]Stationarity as a term is assumed to include *quasi-stationary processes*, where for the purpose of this dissertation, a quasi-stationary random process will be defined as the cases where a known model can be applied to reduce a non-stationary process to a stationary process.

adaptations, as if they are not, then $\exists e \in \mathcal{E}^{-}$ such that $Pr(e)$ is a time varying function. Hence, the decision boundary $d(.)$ is also time varying and the way it should b adjusted cannot be learned from $\boldsymbol{S}(\boldsymbol{T}^{-})$.

6.3 Related Work

In this section, a summary of some of the prior works in applying game theory to intrusion and malware detection problems will be discussed. As many works exist in this domain, only a summary of some of the key works is provided.

In [68], Alpcan *et al.* investigated the application of game theoretic concepts to develop a formal decision and control framework for the analysis and decision processes involved in network intrusion detection. Alpcan *et al.* developed a security attack game that modeled and analyzed both attacker and IDS behaviors within a two-person, nonzero-sum, non-cooperative game with dynamic information. Finally, Alpcan *et al.* obtained this game's NE solutions in closed form and provided two illustrative examples.

In [195], Agah *et al.* proposed a game theoretic framework for defending nodes in a sensor network. Agah *et al.* formulated the attack-defense problem as a two-player, non zero-sum, non-cooperative game between an attacker and a sensor network. Agah *et al.* analytically determined the NE of the developed game. Additionally, Agah *et al.* evaluated the performance of the game and concluded that the proposed game framework had good performance with respect to defending the sensor network.

In [196], Chen *et al.* targeted the intrusion detection problem in heterogeneous networks consisting of nodes with different non-correlated security assets. In particular, the main objectives of Chen *et al.* were to find the expected behaviors of rational attackers and the optimal strategy of the defending IDSes. Accordingly, Chen *et al.*

modeled the network intrusion detection as a non-cooperative game and analyzed its NE. Additionally, Chen *et al.* also provided an evaluation for the proposed game via simulations that validated of the analytical results.

In [197], Lu *et al.* modeled active cyber-defenses taking into account strategic attackers and/or strategic defenders. Lu *et al.* investigated infinite-time horizon optimal control and fast optimal control strategies for strategic defenders against non-strategic attackers and discussed the resulting NE.

In [198], Lye *et al.* presented a game-theoretic method for analyzing the security of computer networks. Lye *et al.* modeled the interactions between the attacker and the network administrator as a two-player stochastic game. Lye *et al.* computed the NE strategies for the players (attacker and administrator). Finally, based on the obtained results, Lye *et al.* discussed possible ways that could be used to enhance the security of the analyzed network.

In [199], Nochenson *et al.* used agent-based simulation to determine the appropriate strategies for attackers and defenders within a simple network security game. In the game, attackers and defenders were modeled as strategic entities in that the attackers were assumed to seek to maximize the amount of damage they caused and the defenders were assumed to seek to minimize their losses subject to some cost constraints. Through simulation, Nochenson *et al.* derived the NE strategies for the players under a variety of cost conditions with the goal to inform network administrators about possible attacker behaviors and their mitigations.

In [200], Alpcan *et al.* investigated the security aspects of vehicular networks (VANETs) within a game-theoretic framework where the defensive measures were optimized with respect to the threats posed by attackers. Alpcan *et al.* discussed the optimal deployment of traffic control and security infrastructure in both static and

dynamic cases. In this work, Alpcan *et al.* also studied multiple types of security games under a variety of information availability assumptions for the players.

In [201], Zhu *et al.* proposed a model based on trust management by using game theory for peers seeking to collaborate truthfully in an intrusion detection network (IDN) environment. Zhu *et al.* showed the existence and uniqueness of a NE under which such peers can communicate within an incentive compatible manner. Additionally, Zhu *et al.* developed an iterative algorithm to assess the game's NE.

In [202], Theodorakopoulos *et al.* combined an epidemic model for malware propagation in a network by a game-theoretic model of the users' decisions. In this model, users were allowed to dynamically change their decisions in order to maximize their perceived utility. In this work, Theodorakopoulos *et al.* also studied the NE and their dependence on the speed of the learning process through which the users come to understand the state of the network. Theodorakopoulos *et al.* showed that faster learning processes corresponded to higher total network costs at the game's NE.

In [203], Zhu *et al.* developed a nonzero-sum stochastic game model to assess the interactions among detection systems in networks as well as the interactions that occur against network intruders. Zhu *et al.* showed the existence of NE of the game and discussed a method for attaining the NE. Zhu *et al.* also proposed the notion of security capacity as the largest achievable payoff to an agent at an NE and discussed the mathematical approach to characterize such equilibria.

In [70, Chapter 3], Gueye introduced a game model to study the interactions between an intelligent virus and an intrusion detection system where the virus is attempting to infect as many computers as possible in the network. Gueye analyzed the interactions using a zero-sum Bayesian game model. Additionally, Gueye applied a Markov chain model to compute the NE of this game and analyze NE's characteristics.

In [204], Alpcan *et al.* developed a two-player zero-sum stochastic security game to model the interactions between malicious attackers and IDSes. Alpcan *et al.* modeled the operation of a sensor network that is observing and reporting the attack information to the IDS in terms of a finite Markov chain. Alpcan *et al.* analyzed the outcomes and evolution of a numerical example of this game for various game parameters. Additionally, Alpcan *et al.* also studied the cases of limited information where the players optimize their strategies either off-line or on-line depending on the type of available information, again using methods based on Markov decision processes.

In [71], Bensoussan *et al.* developed a game-theoretic approach for finding the optimal strategies in a botnet defense model. In particular, Bensoussan *et al.* used a differential game model to analyze the interactions between the botnet herder and the network defender group and derived two closed-loop NE solutions for the developed game.

In [69], Schmidt *et al.* proposed a game-theoretic approach for the malware filtering and detector placement problems in network security, with their main objective being to develop optimal detector algorithms that took into account the possible attacker strategies and actions. Assuming rational and intelligent attackers, Schmidt *et al.* developed a two-person, zero-sum, non-cooperative Markov security game model as the basis for modeling the interactions between the attackers who generate malware traffic on a network and the corresponding intrusion detection systems (IDSes). Using this model, Schmidt *et al.* determined the optimal strategies for both players. In addition, Schmidt *et al.* also tested these optimal strategies in agent-based network simulation environment. Finally, from the simulation results, Schmidt *et al.* provided useful insights for optimally deploying malware detectors in a network environment under the assumptions of their developed games.

In [205], Li *et al.* proposed a dynamic Bayesian signaling game to model and analyze the strategy profiles for regular and malicious nodes within mobile ad hoc networks. In this model, regular nodes consistently updated their beliefs based on the opponents' behavior, whereas malicious nodes evaluate their risk of being caught in order to decide when to flee. Additionally, some possible countermeasures for regular nodes to impact these malicious nodes' decisions were presented. Li *et al.* also provided an analysis and simulation for the NE strategy profiles.

In [206], Patcha *et al.* proposed a game-theoretic model to analyze intrusion detection approaches within mobile ad hoc networks. In particular, Patcha *et al.* modeled the interactions between the nodes of an ad hoc network as a two player multi-stage dynamic non-cooperative game. Additionally, Patcha *et al.* also discussed the NE of the resulting game.

In [207], Liu *et al.* proposed a game-theoretic framework to analyze the interactions between pairs of attacking and defending nodes within wireless ad hoc networks also using a Bayesian game model. Liu *et al.* analyzed the NE for the resulting game in both static and dynamic scenarios. Liu *et al.* highlighted that the developed dynamic Bayesian game model was more realistic, as it allowed the defenders to consistently update their beliefs in terms of the opponent's maliciousness as the game evolved.

In [208], Alpcan *et al.* developed a game-theoretic analysis for intrusion detection approaches in access control systems. Alpcan *et al.* investigated the security game that occurs between the attacker and the IDS both in finite and continuous-kernel versions, where in the latter case the players are associated with specific cost functions. The developed model also captured the scenario in which distributed virtual sensor networks are based on software agents with imperfect detection capabilities. Alpcan *et al.* then extended this model to take the dynamic characteristics of the sensor network

into account. The properties of the resulting dynamic system and repeated games between the players were discussed both analytically and via simulation. Alpcan *et al.* also discussed the existence of a unique NE of the game.

Finally, in [209], Khouzani *et al.* proposed a game-theoretic framework to model the strategic confrontations that occur between malware and the network defensive group within wireless networks. Specifically, Khouzani *et al.* modeled the confrontation as a zero-sum dynamic game and investigated its structural properties. Khouzani *et al.* also derived the saddle-point strategies of the game and showed that these strategies are simple threshold-based policies and, therefore, Khouzani *et al.* concluded that a robust dynamic defense is viable.

6.3.1 Discussion

As per the above, the prior works in which game theory has been applied to malware and intrusion detection have tended to focus on the analysis of specific security games (*i.e.*, were such works then focus on the derivation of that given game's NE). More particularly, within these prior works, the analyzed games have been structured such that:

(i) The attacker's strategy set $\Sigma_{\mathcal{A}}$ is assumed fixed throughout the game.

(ii) The defender's strategy set $\Sigma_{\mathcal{D}}$ is assumed fixed throughout the game.

(iii) The utility function $\mathcal{U}(.)$ is assumed fixed throughout the game.

(iv) As a result, the game being analyzed is shown to admit NE which are then solved for.

Clearly, if $\Sigma_{\mathcal{A}}$, $\Sigma_{\mathcal{D}}$ and $\mathcal{U}(.)$ are allowed to change, for example as a result of the attacker and defender's expected intelligent and rational adaptation, then the NE

of the analyzed game will also, almost surely, change. It is this issue of how the NE change under such adaptation that is of interest in this dissertation and which has not been previously addressed. In these prior works, in particularly, during the stages of the arms-race between the attackers and system defenders, the players are involved in playing a game G. Certain sets of actions and, therefore, the NE of this game G will be advantageous to one of the players (*i.e.*, the one who benefits most from G's current configuration). The other player therefore is motivated to adapt thereby producing new actions (or strategies) that enable him/her to change the NE to his/her favor. Since G is defined in terms of the players' action sets, this process of adaptively developing new actions (or strategies) results in the transition to a new game G' that is then the game being played by the players. Hence, the game G necessarily evolves into a new game G' with different action sets and, hence, different NE (*i.e.*, as a result of the player intentional changes to $\Sigma_{\mathcal{A}}$, $\Sigma_{\mathcal{D}}$, or $\mathcal{U}(.)$). Moreover, the iterative adaptation of the players' action sets gives rise to a sequence of games resulting between the players. In this work, this game sequence will be analyzed to investigate the conditions under which it would converge towards a defender's advantageous end-game, as these are the conditions of interest if cyber-security is to be achieved.

More specifically, the remainder of this chapter applies the developed EMM to enable the modeling of this iterative sequence of sub-game interactions that arise as the attackers and defenders intelligently adapt their strategies in light of what they learn about both the defended environment itself and what the other does. By this EMM-based approach, we will formally show that the attacker-defender confrontations can be modeled as an iterative sequence of sub-games that arises as each side adapts to the other's actions. We will then analyze this sequence to derive the conditions that are required if it is to converge to the desired defender advantageous

(or defender winnable) end-game, where these conditions are shown to directly depend on the measure invariance of the defender's deployed measurement features $F(.)$ (*i.e.*, the derivation follows from the application of the dynamical system theory of Section 6.2.2).

6.4 The Attackers-Defender Game

In this section, a game-theoretic model for the attackers-defender interactions will be developed. This game will be denoted as G. However, before delving into the definitions of the various components of G, we will start by discussing the environment over which G is played (*i.e.*, the defended environment). Consider the computer systems run by an organization. Such systems could span a single server, a network of computers (including mobile nodes), *etc.* This system defines the defended environment that should be protected by the defender from malware attacks. For such a system, as discussed in Chapter 3, irrespective of scale, an EMM can be constructed. Additionally, as discussed in Chapters 4 and 5, this EMM can be used to model all malware and malware detection approaches that are implementable within the given EMM modeled environment. Hence, without loss of generality, the defended environment over which G is played will be modeled as in terms of an EMM as defined in Section 3.7.

As discussed in Section 5.2, if the size of the set of all possible states $|\mathbb{S}(t)|$ of the EMM is sufficiently small (*e.g.*, as per small-scale embedded systems, *etc.*), then the set $\mathcal{E} = \mathcal{E}^{-} \cup \mathcal{E}^{+}$ of all possible events within the EMM will also be sufficiently small to be perfectly knowable to the defender. Accordingly, formal methods can then be used to ensure the security of the EMM [104, Chapter 20]. However, for many of today's systems, $|\mathbb{S}(t)|$ would be very large and, consequently, formal methods are intractable

to apply (*i.e.*, state space can easily exceed 10^{100} possible state changes per second). The EMM is assumed to be subjected to malware attacks by an unknown set of potentially collaborating attackers and is defended from these attacks by a defender.

In general, as discussed in Section 6.2.1.1, G can be defined as the tuple $G = \langle \mathcal{N}, \mathbf{\Sigma}, \mathcal{U} \rangle$, where $\mathcal{N}$ is the set of players, $\mathbf{\Sigma}$ is the space of the action sets of all of the players, and $\mathcal{U}$ is the set of utility functions for all of the players. Hence, to model the attackers-defender confrontations as a game, we need to formally specify each of these different components.

6.4.1 The Attackers

We will begin by discussing the attacker side of the game. In real world, computer systems are subjected to malware attacks launched by a diverse set of attackers, where these attackers can have different objectives, attack tools, *etc.* Accordingly, in this work, we consider that the defended EMM is subjected to malware attacks conducted by a set $\boldsymbol{A}$ of N_a attackers defined as,

$$\boldsymbol{A} = \{A_k | k = 1, 2, \cdots, N_a\}. \tag{6.5}$$

Each $A_k \in \boldsymbol{A}$ is assumed to have his/her own set of attacks that can be used against the EMM. These attack sets can differ across the A_k. It should be noted that, specifying the attacker who launches an attack is not of interest in this work. In particular, we are interested in the fact that it is the developed EMM that is under attack, and not in determining who may have launched the attack. Hence, from the defender's perspective, without a loss of generality this set of attackers can be modeled as a single attacker that attacks the system via the composite set of malware attacks held by the A_k. Therefore, within G, the set of attackers defined in Equation (6.5) will be modeled as a single player, denoted as $\mathcal{A}$, with his/her attack set being the

union of all of the attack sets of all attackers $A_k \in \boldsymbol{A}$. It should be noted that, this approach includes all potentially coordinated actions that can possibly occur between sets of attackers. Also, $\mathcal{A}$'s set of strategies in general will be larger than $\bigcup_{k=1}^{N_a} A_k$'s strategy sets as a result. Hence, we assume that $\mathcal{A}$ has a finite set $\boldsymbol{\alpha}$ consisting of $N_{\mathcal{A}}$ unique malware attacks, that is formally defined as,

$$\boldsymbol{\alpha} = \{\alpha_k | k = 1, 2, \ldots, N_{\mathcal{A}}\}. \tag{6.6}$$

To attack the EMM, $\mathcal{A}$ is assumed to employ a sequence of attacks $\alpha_k \in \boldsymbol{\alpha}$. Without loss of generality, we do not consider that $\mathcal{A}$ selects these attacks randomly from $\boldsymbol{\alpha}$ as such approaches do not require game theory to analyze. Instead, we consider that $\mathcal{A}$ is intelligent and rational and selects the attacks that maximize his/her utility function $u_{\mathcal{A}}(.)$. Moreover, to launch successful attacks, $\mathcal{A}$ must first gain sufficient information about the targeted system. Hence, it is assumed that $\mathcal{A}$ probes the EMM in an attempt(s) to find vulnerabilities that can be used to attack the system. This probing process is denoted as $V_{\mathcal{A}}$ and is defined as the map

$$V_{\mathcal{A}} : \boldsymbol{S}(\boldsymbol{T}^-) \rightarrow \left\{\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)\right\}, \tag{6.7}$$

where $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t) \subset \widehat{\boldsymbol{Info}}(\boldsymbol{T^-})$ is the set of knowledge $\mathcal{A}$ has gained about the nature of the EMM and its states. For $\mathcal{A}$, $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ is such that $t \in \mathcal{T}^-$ as $\mathcal{A}$ can only gain their knowledge from the EMM's past states and not its future ones. Hence, $V_{\mathcal{A}}$ is a map from the set of past states of the EMM to $\mathcal{A}$'s partial knowledge of these states. Clearly, $\left\{\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)\right\}$ would differ from one attacker to another in ways that would reflect each attackers knowledge and skills. It should be noted that, $\mathcal{A}$ applies $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ to assess the EMM's vulnerabilities, where in the ideal defensive case these would be the empty set.

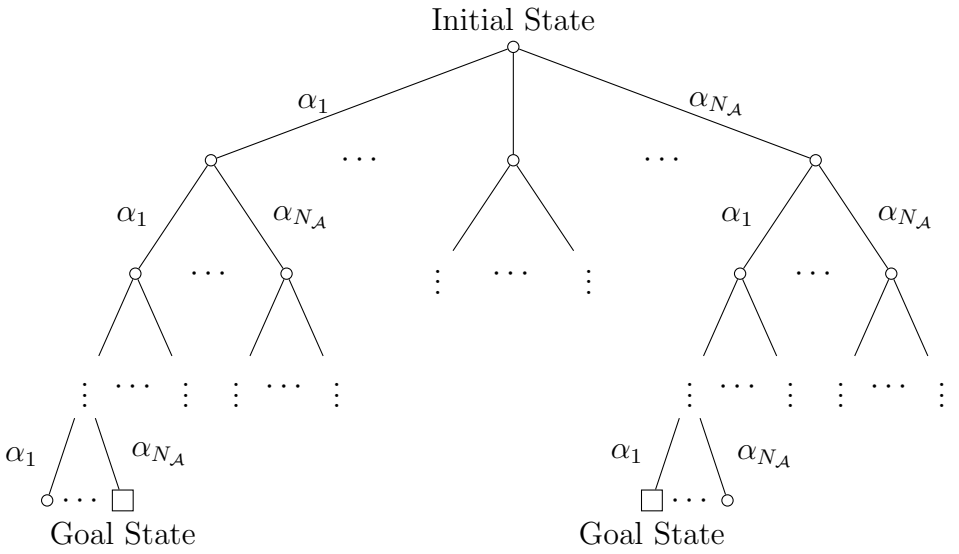


Figure 6.2: The state space search performed by $\mathcal{A}$.

Hence, we consider that $\mathcal{A}$ develops an estimated $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ about the state of the EMM to be attacked[8]. This agrees with what occurs in the real-world where attackers gather information about the targeted systems prior to conducting their attacks [210]. Based on this $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$, $\mathcal{A}$ can be modeled as performing a *state space search*[9] to determine an attack sequence to achieve their goals. Additionally, we consider that $\mathcal{A}$ selects the attacks by searching his/her attack state space, as shown in Figure 6.2, where the *branching factor*[10] of this state space tree is equal to $|\boldsymbol{\alpha}|$. The nodes in Figure 6.2 represent the estimated states $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ of the EMM as estimated by $\mathcal{A}$ after α_k has been enacted whereas the edges represent the enacting of an α_k attack by $\mathcal{A}$. As indicated in the figure, $\mathcal{A}$ generally has multiple possible attack paths at each node.

[8] $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ follows the generally used signal processing convention of '$\widehat{\ }$' denoting an estimate to. hence, $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$ is $\mathcal{A}$'s estimate of $\boldsymbol{Info}(t)$ for $t \in \boldsymbol{T}^-$.

[9] The state space is the set of all states reachable from the initial state [211, Chapter 3, pp. 62].

[10] The branching factor is the maximum number of successors of any node [211, Chapter 3, pp. 72]

More specifically, we consider that $\mathcal{A}$ selects the attacks via a standard state space search methodology. In particular, the A^* optimal search method can be assumed, as the A^* search is widely known to guarantee the finding of the optimal path in the minimum amount of work, provided an admissible heuristic $\widehat{h}(.)$ is used [211, Chapter 4, pp. 97]. With this context, the required heuristic $\widehat{h}(.)$ can be defined as $\widehat{h}(\alpha_k) = u_{\mathcal{A}}(\alpha_k, t)$. Accordingly, based on $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$, $\mathcal{A}$ will begin from a possible set $\mathcal{S} = \{S_j | j = 1, 2, \cdots, N_s\}$ of start nodes representing the EMM's states at the time when $\mathcal{A}$ initially attacks, and a set $\mathcal{G} = \{g_k | k = 1, 2, \cdots, N_g\}$ of goal nodes, that represent the target EMM states that $\mathcal{A}$ wants to reach through conducting his/her attacks. Hence, $\mathcal{A}$ searches this attack state space for a path P between any node $S_j \in \mathcal{S}$ to any node $g_k \in \mathcal{G}$.[11] Such a path P then represents the attacks that should be launched by $\mathcal{A}$ to achieve the desired objectives. By considering this way of attack selection, $\mathcal{A}$ is optimally selecting the attacks while minimizing the number of conducted attacks (*i.e.*, this is the best $\mathcal{A}$ can do to attack the EMM). Note that, the only required conditions on $u_{\mathcal{A}}(.)$ are,

$$u_{\mathcal{A}}(.) \geq 0, \text{ and that, } \widehat{h}(n) \leq h(n). \tag{6.8}$$

Hence, we assume that there is no negative penalty on $\mathcal{A}$ for attacking the EMM (*e.g.*, we assume that the attacker will not be arrested, *etc.*). This assumption is reasonable as real-world attackers could, for example, be attacking from other countries and, hence, may be out of reach of the local law enforcement and its jurisdiction. For example, in the case of *US vs Gorshkov* [144], the attackers launched their attacks from Russia to target eBay and PayPal's systems located in USA. Moreover, $u_{\mathcal{A}}(.) \geq 0$

[11] Note that, by standard state space search conventions, all nodes $g_k \in \mathcal{G}$ are assumed to be equivalent from $\mathcal{A}$'s perspective.

is consistent with our assumption that we are solely focused on analyzing the defenses with respect to the information sets as they are modeled within the EMM.

Finally, we will summarize the components of the attacking side of G as follows. The set of attackers will be modeled as a single player $\mathcal{A}$. The set $\Sigma_{\mathcal{A}}$ of actions of $\mathcal{A}$ will be $\boldsymbol{\alpha}$ as defined in Equation (6.6). Hence, $\Sigma_{\mathcal{A}}$ is defined as,

$$\Sigma_{\mathcal{A}} = \boldsymbol{\alpha} = \{\alpha_k | k = 0, 1, 2, \ldots, N_{\mathcal{A}}\}, \tag{6.9}$$

where the *no attack* action (*i.e.*, $\alpha_0 = \emptyset$) is included within $\Sigma_{\mathcal{A}}$. The utility function $u_{\mathcal{A}}(.)$ of $\mathcal{A}$ defines the heuristic function $\widehat{h}(.)$ used with the A^* search (*i.e.*, $u_{\mathcal{A}}(.) = \widehat{h}(.)$) by which $\mathcal{A}$ selects the next attack to enact. It should be noted that $\mathcal{A}$ is assumed to engage in this A^* search process by model their iterative enacting of attacks against the defended environment modeled by the EMM. Hence, $\mathcal{A}$ is assumed to be able in practice to return to past nodes as represented in the state space search as required for A^* search and bound processes.

More particularly, it is useful to look at this state space search process in more detail and relate it to the actions of real attackers against real defended environments. When an attacker begins to attack a real defended environment, this environment will be in some state $\boldsymbol{Info}(t)$, as per the EMM model of that environment at time t. Excluding trivial environments (*i.e.*, focusing on larger-scale IT environments), then the attacker can only have partial knowledge of $\boldsymbol{Info}(t)$, where it is this attacker's partial knowledge that is denoted by $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t)$. Moreover, with respect to the state space search process described above, this $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t) = S_j$ where $S_j \in \mathcal{S}$ and, hence, it denotes the attackers initial knowledge about the specific defended environment that he/she is seeking to attack but before he/she has actual enacted any attacks. The attacker then enacts an attack $\alpha_k \in \boldsymbol{\alpha}$ against the defended environment, as selected via the A^* search process.

The effect of having conducted the attack α_k is then what moves the attacker from the start node S_j to some new node n within the state space graph of Figure6.2 (*i.e.*, α_k provides the edge that exists between S_j and this new node n). By having enacted attack α_k, the attacker by definition gains additional knowledge about the defended environment under attack (*e.g.*, at a minimum if α_k fails then the attacker knows that the defended environment is not vulnerable to α_k, whereas if its successes the attacker now knows the vulnerability exists and is exploitable). Hence, the attackers new knowledge about the defended environment, post-performing α_k, can be denoted as $\widehat{\boldsymbol{Info}}_{\mathcal{A}}(t|n)$.

Clearly, node n may or may not be on the optimal path P to a goal node. Hence, the attacker can then again apply the A^* search process to enact the next attack $\alpha_{k'}$ in order to progress to the next node in the search tree n'. By definition, through this iterative node-by-node application of the A^* search process, the attacker will discover the shortest path P that exists between any start node in $\mathcal{S}$ and any goal node in $\mathcal{G}$ with the least amount of steps (*i.e.*, with enacting the least number of attacks α_k). Once, such a path P is found, the attacker can of course simply re-use this optimal path to obtain their desired objectives, until such time as the defender changes the defenses (as described in Section 6.4.2). Hence, the described A^* search processes exists as the mechanism by which the modeled game theory attackers $\mathcal{A}$ perform their intelligent and rational assessment of the their strategy space $\Sigma_{\mathcal{A}}$.

Moreover, any real-world attacker who does not perform this type of A^* search, by definition, must either enact more attacks than $\mathcal{A}$ does in finding an optimal path P or already have prior knowledge of the sequence of attacks that P describes. Hence, the way in which $\mathcal{A}$ is being considered denotes, from the defender's perspective, the worst-case attacker excluding only those already know how to efficiently and effectively beat the deployed defenses.

Further, as per above, by modeling $\mathcal{A}$ as possessing the composite knowledge of all real-world attackers who may be attacking an EMM modeled defended environment, the considered worst-case attacker $\mathcal{A}$ more closely models the types of attackers of concern when skilled, motivated, and collaborating nation-state adversaries must be considered. Other attacker models can, of course, be developed who possess lower assumed skill sets than the described $\mathcal{A}$, but the focus of the analysis is solely on this form of worst-case attacker and not on less skilled and/or less collaborative attackers.

6.4.2 The Defender

In this subsection, the defending side of G is described. Without loss of generality, we assume that the EMM is to be defended by a single defender. This assumption is reasonable as real-world computer systems are under the management and control of typically single entities (*e.g.*, system administration, IT departments, *etc.*). Accordingly, we will denote the defender as $\mathcal{D}$, where $\mathcal{D}$ is a single player in G. Moreover, as with the case of the attackers, $\mathcal{D}$ can be easily extended in order to model a collection of collaborating defenders acting on the same defended environment.

Since the objective of $\mathcal{D}$ is to protect the EMM from the malware attacks conducted by $\mathcal{A}$, we consider that $\mathcal{D}$ has deployed a composite set of malware detection systems $\mathbb{D}(.)$, as per Section 5.2.1. More particularly, we assume that the composite detector $\mathbb{D}(.)$ is composed of standard collection of the types of detection systems in existence to protect against malware. For example, $\mathbb{D}(.)$ may include host-based malware detection systems, network-based IDS, *etc.*

Additionally, we also consider that $\mathcal{D}$ has a set $\mathbb{R}$ of $N_{\mathbb{R}}$ event responses (automated and/or human-centric), defined individually as,

$$\mathbb{R} = \{r_j | j = 0, 1, 2, \ldots, N_{\mathbb{R}}\}. \tag{6.10}$$

In particular, these responses represent the defensive actions (or procedures) that are invoked when malware is detected by $\mathbb{D}(.)$, where $r_0 = \emptyset$ denotes the *null response* (or no action) response. For example, in real world such responses span the range from doing nothing (*e.g.*, in the case of a received spam email message) to actively removing the infections, updating signature databases, or patching vulnerabilities on infected machines (*e.g.*, in the case of worm infections), *etc.* We consider that $\mathcal{D}$'s responses focus only on the attacks that are expected to cause damages that exceed some predefined value $\zeta_{\mathcal{D}} > 0$ and, therefore, there is no responses for attacks that are expected to cause losses below $\zeta_{\mathcal{D}}$. For example, as discussed above, $\mathcal{D}$ can select to do nothing to respond to a spam email message. The value of $\zeta_{\mathcal{D}}$ is solely within the defender purview to set and, as such, it denotes the subset of events $e \in \mathcal{E}^{-}$ for which $\mathcal{D}$ is seeking to actively defend against.

Additionally, we assume that an ideal responses $r^*(.)$ exists within $\mathbb{R}$ for each detectable attack. More particularly, these ideal responses are defined such that, $\forall \alpha_k \in \Sigma_{\mathcal{A}}$ for which $\mathbb{D}(\alpha) = -1$ and $cost(\alpha) > \zeta_{\mathcal{D}}$, $\exists r^*(\alpha_k) \in \mathbb{R}$, such that,

$$u_{\mathcal{A}}[(\alpha_k, t)|(r^*(\alpha_k), t)] = 0. \tag{6.11}$$

As shown by Equation (6.11), $\mathcal{A}$ gets zero utility when $\mathcal{D}$ properly react to α_k using r^*. Hence, $r^*(\alpha_k)$ defines the best possible response by $\mathcal{D}$ against the attack α_k. Specifically, the existence of $r^*(\alpha_k)$ for a specific attack α_k corresponds, within Section 6.4.1's state space search, to the defender's elimination of the edges that represent α_k in $\mathcal{A}$'s state space search. For simplicity but without loss of generality, the idealization will be assumed that $r^*(e)$ is enacted instantaneously once $\mathbb{D}(e) = -1$.

Similarly, it is also assumed that $\mathcal{A}$ will have non-zero utility if $\mathcal{D}$ improperly responds to the attack. That is to say, considering α_k and $r^*(\alpha_k)$ as discussed in Equation (6.11), then $\forall r_j \in \mathbb{R}$ where $r_j(\alpha_k) \neq r^*(\alpha_k)$ we have that $u_{\mathcal{A}}[(\alpha_k, t)|(r_j(\alpha_k), t)] >$

0. Since $\mathbb{R}$ denotes the set of all possible actions available to $\mathcal{D}$ to address the attacks conducted by $\mathcal{A}$, then $\mathbb{R}$ also denotes the set of actions of $\mathcal{D}$ within the game G. More particularly, the set $\Sigma_{\mathcal{D}}$ of actions of $\mathcal{D}$ in the game G can now be defined via Equation (6.10) as,

$$\Sigma_{\mathcal{D}} = \mathbb{R} = \{r_j | j = 0, 1, 2, \ldots, N_{\mathbb{R}}\}, \tag{6.12}$$

where again $r_0 = \emptyset$ denotes the *no response* action. Finally, we define the utility function $u_{\mathcal{D}}(.)$ for the defender $\mathcal{D}$. Without loss of generality, for any $e \in \mathcal{E}$ and $r^*(\alpha_k)$ as discussed in Equation (6.11), $u_{\mathcal{D}}(.)$ can be defined as,

$$u_{\mathcal{D}}[r(\alpha), t] = \begin{cases} \leq 0 & \text{if } r \neq r^*(\alpha_k), \\ > 0 & \text{if } r = r^*(\alpha_k). \end{cases} \tag{6.13}$$

Hence, $\mathcal{D}$ can receive negative utilities if $\mathcal{D}$ improperly responds to $\mathcal{A}$'s attacks, whereas $\mathcal{D}$ receives positive utility when the attacks are detected and the best responses are enacted or if $\mathcal{A}$'s attacks go undetected. In real world, this negative utility corresponds to, for example, the losses incurred in the system due to failing to completely wipe out malware infection from all infected machines, *etc.* Accordingly, $\mathcal{D}$ is necessarily motivated to properly respond to the malware attacks, where this is based on the nature of the deployed $\mathbb{D}(.)$ also detecting the attacks (*i.e.*, $r^*(\alpha_k)$ cannot be enacted, by definition, if α_k is not detected by $\mathbb{D}(.)$).

It should be noted that, quite generally, the best responses $r^*(.)$ denote $\mathcal{D}$'s currently best known responses and, hence, are based on $\mathcal{D}$'s current estimate $\widehat{\boldsymbol{Info}}_{\mathcal{D}}(t)$ about (or knowledge of) $\boldsymbol{Info}(t)$. As $\mathcal{D}$'s estimate $\widehat{\boldsymbol{Info}}_{\mathcal{D}}(t)$ improves over time then $\mathcal{D}$ may be able to develop better best responses. Hence, the notion of these best responses should not be confused with the notion of the optimal theoretical re-

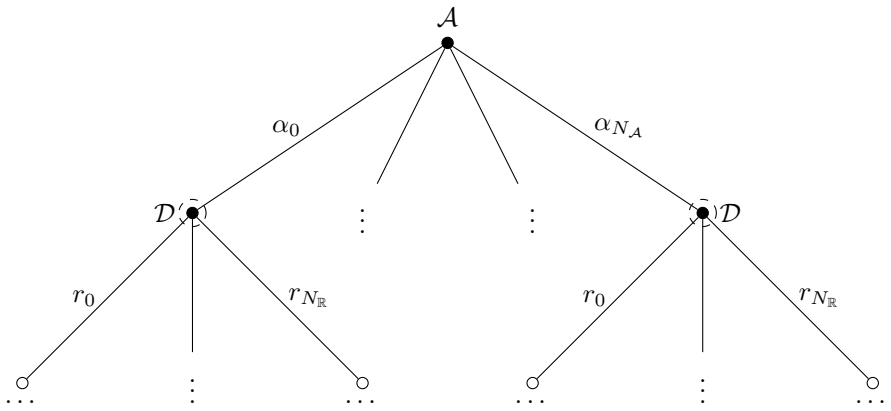


Figure 6.3: The malware arms-race game G as an extensive form game.

sponses that may be possible against $\mathcal{A}$'s attacks. The $r^*(.)$ are expressly restricted to denoting only the best responses that $\mathcal{D}$ currently knows based on his/her current $\widehat{\boldsymbol{Info}}_\mathcal{D}(t)$ estimate.

6.4.3 Defining the Game G

The malware arms-race game G can now be defined as the tuple $G = \langle \mathcal{N}, \Sigma, \mathcal{U} \rangle$, where:

- The set of players of G is defined as $\mathcal{N} = \{\mathcal{D}, \mathcal{A}\}$.
- The set of actions of the players is defined as $\boldsymbol{\Sigma} = \{\Sigma_\mathcal{D}, \Sigma_\mathcal{A}\}$, as discussed in Equations (6.9) and (6.12), therefore the space of the complete strategies is $\Sigma_\mathcal{D} \times \Sigma_\mathcal{A}$.
- The set of utility functions of G is defined as $\mathcal{U} = \langle u_\mathcal{D}(.), u_\mathcal{A}(.) \rangle$, as defined in Equations (6.8) and (6.13) above.

In real world, attackers launch their attacks and then the defenders respond to the attacks that they were able to detect. Accordingly, both players do not act simultaneously and, hence, G is best described as an extensive form game as shown in

Figure 6.3, where $\mathcal{A}$ attacks and $\mathcal{D}$ responds. That is to say, G can be idealized as a two-person, non-zero sum, finite, extensive form, non-cooperative game of incomplete information. Both $\mathcal{D}$ and $\mathcal{A}$ are assumed to engage in repeatedly playing G over a some finite time period $\tau \subset \boldsymbol{T}$. It should be noted that, as discussed in Theorem 6.1, assuming perfect recall, NE will, by definition, exist in behavioral strategies for G.

More particularly, when G is repeatedly played by $\mathcal{A}$ and $\mathcal{D}$, then two possible outcomes exist:

(i) An attack path P from a start node $S_j \in \mathcal{S}$ to a goal node $g_k \in \mathcal{G}$ exists. In this case, since $\mathcal{A}$ is assumed to be using an A^* search, then $\mathcal{A}$ is guaranteed to find P [211, Chapter 4, pp. 97]. Clearly, $\mathcal{A}$ will win G in this case (*i.e.*, the NE of G must be to $\mathcal{A}$'s advantage).

(ii) Or, the deployed defenses $\mathbb{D}(.)$ are, such that, an attack path P does not exist as a result of enacted best responses. Clearly, in this case, $\mathcal{D}$ will win G (*i.e.*, the NE of G must be to $\mathcal{D}$'s advantage).

In the next section it is discussed how G itself must be constructed to evolve over time as a result of $\mathcal{A}$ and $\mathcal{D}$'s intelligent adaptations as they learn about what the other is doing. More particularly, we will show that G's evolution over time can be modeled as a sequence of sub-games.

6.5 The Evolution of G over Time

In the previous section, we formalized the attackers-defender ongoing confrontation as a game. We showed that wining the game will be determined according to whether some path P exists from a start node to a goal node within $\mathcal{A}$'s state space search. In this section, we will show that the on-going adaptations of $\mathcal{A}$ and $\mathcal{D}$'s action sets lead to the evolution of the game into a sequence of sub-games.

Consider that the defended EMM is in some initial state. In real world, this initial state corresponds to the state of the system when it is first installed and starts to operate. This initial state with respect to the analyzed game G represents some initial game G_0 given by $G_0 = \langle\langle \mathcal{A}, \mathcal{D}\rangle, \langle \Sigma_{\mathcal{A}}, \Sigma_{\mathcal{D}}\rangle, \langle u_{\mathcal{A}}(.), u_{\mathcal{D}}(.)\rangle\rangle$. In securing the EMM, we assume that $\mathcal{D}$ follows industry's best practices and, hence, $\mathcal{D}$ will update and deploy an EMM composite detector solution $\mathbb{D}(.)$. As part of this process, $\mathcal{D}$ will also update his/her set of responses and best responses from $\Sigma_{\mathcal{D}}$ into $\Sigma'_{\mathcal{D}}$. As discussed in Section 6.2.1.1, since the utility functions are defined in terms of the players' action sets, then changing $\Sigma_{\mathcal{D}}$ into this new $\Sigma'_{\mathcal{D}}$ will also cause $u_{\mathcal{D}}(.)$ to change into the new utility function $u'_{\mathcal{D}}(.)$. This change in the action sets and utility functions of the players therefore causes the game $G_0 = \langle\langle \mathcal{A}, \mathcal{D}\rangle, \langle \Sigma_{\mathcal{A}}, \Sigma_{\mathcal{D}}\rangle, \langle u_{\mathcal{A}}(.), u_{\mathcal{D}}(.)\rangle\rangle$ to be changed into the new game $G_1 = \langle\langle \mathcal{A}, \mathcal{D}\rangle, \langle \Sigma_{\mathcal{A}}, \Sigma^1_{\mathcal{D}}\rangle, \langle u_{\mathcal{A}}(.), u^1_{\mathcal{D}}(.)\rangle\rangle$, or,

$$G_0 \underset{\mathcal{D}}{\longrightarrow} G_1,$$

where this change in the game is directly caused by $\mathcal{D}$'s actions as discussed above. Therefore, $\mathcal{A}$ and $\mathcal{D}$ will, by definition, be playing sub-game G_1 as $\mathcal{D}$'s actions have caused the transition out of G_0.

Now, assume that after engaging over G_1 for a while, $\mathcal{A}$ decides to update his/her attack set $\Sigma_{\mathcal{A}}$ by:

(i) Obfuscating one or more known attacks α_k. More particularly, an obfuscation $O(.)$ is defined as a map $O : \mathcal{E} \rightarrow \mathcal{E}$ in which obfuscated attacks are defined as follows.

Definition 6.13 (Obfuscated Attacks)**.** *For any $\alpha_k \in \boldsymbol{\alpha}$, an obfuscation $O : \mathcal{E} \rightarrow \mathcal{E}$ under the EMM is the construction of a new attack $\tilde{\alpha}_k = O(\alpha_k)$ such*

that if $\mathbb{D}(\alpha_k) = -1$ *then* $\mathbb{D}(\tilde{\alpha}_k) \neq -1$ *and if* α_k *provides an edge between nodes* n *and* n' *in* $\mathcal{A}$*'s state space search then* $\tilde{\alpha}_k$ *provides the same edge.*

Hence, $\mathcal{A}$ may elect to develop a set of obfuscated attacks $\tilde{\boldsymbol{\alpha}}$, such that,

$$\forall \tilde{\alpha}_k \in \tilde{\boldsymbol{\alpha}},\ F^1(\tilde{\alpha}_k) = \emptyset,\ \text{or}\ \mathbb{D}^1(\tilde{\alpha}_k) \neq -1.$$

(ii) Creating a new attack or a set of new attacks $\{\alpha'_k | k = 1, 2, \ldots, N_{\alpha_{new}}\}$, such that,

$$\forall \alpha'_k, F^1(\alpha'_k) = \emptyset\ or\ \mathbb{D}^1(\alpha'_k) \neq -1,$$

where $F^1(.)$ and $\mathbb{D}^1(.)$ denote the measurement features and detection solutions that exist in game G_1.

In particular, we denote any such set of these newly obfuscated and/or created attacks as $\boldsymbol{\alpha}_{new}$. The addition of $\boldsymbol{\alpha}_{new}$ to $\mathcal{A}$'s action set therefore creates new edges that are exploitable in $\mathcal{A}$'s state space process. The overall effect of creating these $\boldsymbol{\alpha}_{new}$ is that the $\mathcal{A}$'s action set $\boldsymbol{\alpha}$ now be changing into a new $\boldsymbol{\alpha}^1$ that is defined by,

$$\boldsymbol{\alpha}^1 = \boldsymbol{\alpha} \cup \boldsymbol{\alpha}_{new}. \tag{6.14}$$

The change of $\boldsymbol{\alpha}$ into $\boldsymbol{\alpha}^1$ changes $\Sigma_{\mathcal{A}}$ into $\Sigma^1_{\mathcal{A}}$, which also causes $u_{\mathcal{A}}(.)$ to change into $u^1_{\mathcal{A}}(.)$. Hence, the change in the action sets and utility functions by $\mathcal{A}$ while playing G_1 will cause $G_1 = \langle \langle \mathcal{A}, \mathcal{D} \rangle, \langle \Sigma_{\mathcal{A}}, \Sigma^1_{\mathcal{D}} \rangle, \langle u_{\mathcal{A}}(.), u^1_{\mathcal{D}}(.) \rangle \rangle$ to change into the new game G_2 that is defined as $G_2 = \langle \langle \mathcal{A}, \mathcal{D} \rangle, \langle \Sigma^1_{\mathcal{A}}, \Sigma^1_{\mathcal{D}} \rangle, \langle u^1_{\mathcal{A}}(.), u^1_{\mathcal{D}}(.) \rangle \rangle$, or:

$$G_1 \underset{\mathcal{A}}{\longrightarrow} G_2,$$

where this transition from G_1 to G_2 is caused by $\mathcal{A}$'s desire to beat the deployed defense $\mathbb{D}^1(.)$ in order to achieve the attack goals. The players $\mathcal{A}$ and $\mathcal{D}$ will now be engaged in playing G_2. From G_2, $\mathcal{A}$ can now either achieve a path to a goal node, or can continue to perform attack obfuscations or develop new attacks until such a path is created. Similarly, $\mathcal{D}$ can continue to improve and adapt the deployed defenses in light of applying improved best practices, applying security patches, *etc.* Moreover, by this process, the attackers-defender on-going confrontations can then be modeled as the sequence of sub-games that occur as both players continue to adapt and is given by,

$$G_0 \underset{\mathcal{D}}{\longrightarrow} G_1 \underset{\mathcal{A}}{\longrightarrow} G_2 \underset{\mathcal{D}}{\longrightarrow} G_3 \underset{\mathcal{A}}{\longrightarrow} \cdots \tag{6.15}$$

where importantly each game in this sequence will, by definition, have its own NE.

In this section, we showed that the players in the overall arms-race game are motivated to update their action sets in order to arrive at a new game G_k that has advantageous NE. these adaptations, therefore, produce a sequence of games $\{G_k\}_{k=0}^{K}$. In the next section, we will investigate the conditions required if this sub-game sequence is to converge to a defender's advantageous end-game.

6.6 The Game Sequence $\{G_k\}_{k=0}^{K}$

As discussed in the previous section, the interactions between $\mathcal{D}$ and $\mathcal{A}$ results in a sequence of sub-games as shown in Equation (6.15). As discussed above, we consider that $\mathcal{D}$ and $\mathcal{A}$ can include the null or $\emptyset$ action in their set of adaptation approaches used to update their action sets $\Sigma_{\mathcal{D}}$ and $\Sigma_{\mathcal{A}}$, respectively. More specifically, if $\mathcal{A}$ uses $\emptyset$ to update $\Sigma_{\mathcal{A}}$, then the resulting change in $\Sigma_{\mathcal{A}}$ can be expressed as,

$$\Sigma_{\mathcal{A}} \underset{\emptyset}{\longrightarrow} \Sigma_{\mathcal{A}}.$$

Hence, $\Sigma_{\mathcal{A}}$ remains unchanged if updated with the null action $\emptyset$. Similarly, the change in $\Sigma_{\mathcal{D}}$ due to applying the defender's null action $\emptyset$ can be expressed as,

$$\Sigma_{\mathcal{D}} \underset{\emptyset}{\longrightarrow} \Sigma_{\mathcal{D}}.$$

By including the null actions within both $\mathcal{D}$ and $\mathcal{A}$ action sets, the defender and attackers adaptations can be assumed to occur iteratively without in turn a loss of generality. Hence, the sequence,

$$G_0 \underset{\mathcal{D}}{\longrightarrow} G_1 \underset{\mathcal{A}}{\longrightarrow} \cdots \underset{\mathcal{D}}{\longrightarrow} G_k \underset{\mathcal{A}}{\longrightarrow} G_{k+1} \underset{\mathcal{D}}{\longrightarrow} \cdots \tag{6.16}$$

can be considered as the general form of the described sub-game in which the defender performs every adaptation even indexed transitions and the attacker performs all odd indexed transitions.

Denote the full game sequence as $\{G_k\}_{k=0}^{K}$. Then the important issue to be addressed is whether or not $\{G_k\}_{k=0}^{K}$ converges into a defender advantageous end-game. That is to say, does a final game $G_k = G^*$ in which $\mathcal{A}$ can no longer update their action set exist. Clearly, under the EMM, such an end-game can exist, as this end-game would arise when $\mathbb{D}_k(.) = \mathbb{D}^*(.)$ as defined in Section 5.2. Hence, within G^*, the following conditions should be satisfied:

(i) $\mathbb{D}(\alpha|G^*) = -1$ for all $\alpha \in \Sigma_{\mathcal{A}}$, where $\mathbb{D}(\alpha|G^*)$ denotes the composite detector $\mathbb{D}(.)$ applied to any attack α available to $\mathcal{A}$ in the sub-game G^*, and

(ii) There does not exist an $\tilde{\alpha}_k = O(\alpha_k)$ for all $\alpha_k \in \boldsymbol{\alpha}$ in sub-game G^* such that $\mathbb{D}(\alpha_k) = -1$ and $\mathbb{D}(\tilde{\alpha}_k) \neq -1$ and there does not exist any $\alpha_j \in \mathcal{E}^-$ for which $\alpha_j \notin \boldsymbol{\alpha}$ and $\mathbb{D}(\alpha_j) \neq -1$ or $F(\alpha_j) = \emptyset$ which $\mathcal{A}$ can successfully create.

Moreover, within the following analyses, the assumption is made that $\mathbb{D}(.)$ is such that it only improves over time. Hence, it is assumed that if $e \in \mathcal{E}^-$ is such that for some $\mathbb{D}^k(e) = -1$, then for all $k' > k$ we also have that $\mathbb{D}^{k'}(e) = -1$, where $\mathbb{D}^k(.)$ denotes the composite detectors $\mathbb{D}(.)$ in use during the sub-game G_k.

6.7 The Convergence of the Game Sequence

To discuss the issue of the convergence of $\{G_k\}_{k=0}^K$ in more details, we need to consider the two possible cases for the EMM. From the point of view of the EMM's memory $M(t)$ either:

(i) $M(t)$ is static (*i.e.*, M does not change over time). Hence, $\forall t, t' \in \boldsymbol{T}$ and $t \neq t'$, we have $M(t) = M(t')$. This case represents an idealization of the real-world where the set of all possible EMM events remain fixed for all time.

(ii) Or, $M(t)$ is allowed to vary with time. Hence, $\exists t, t' \in \boldsymbol{T}$, such that, $M(t) \neq M(t')$. More particularly, this case more accurately models real-world defended environments in which systems come and go, *etc.*

In the following subsections these two cases are discussed in more details with respect to how they impact $\{G_k\}_{k=0}^K$ convergence.

6.7.1 Static $M(t)$ and Convergence of $\{G_k\}_{k=0}^{K}$

In this subsection, we discuss the convergence of $\{G_k\}_{k=0}^{K}$ in the case of static $M(t)$. In particular, the following lemma shows that $\{G_k\}_{k=0}^{K}$ must converge for some sufficient large $k < \infty$ to a defender's advantageous end-game.

Lemma 6.1. *For sufficiently large $k < \infty$, $\{G_k\}_{k=0}^{K}$ must converge to a defender's advantageous end-game.*

Proof. Proof by construction. If $M(t) = M(t')$ for all $t \neq t' \in \boldsymbol{T}$, then the set of all possible states $\mathbb{S}(t)$ will be fixed for all $t \in \boldsymbol{T}$. Hence, even in the limit as $\boldsymbol{T} \rightarrow \infty$, the set of all possible events $\mathcal{E} = \mathcal{E}^- \cup \mathcal{E}^+$ associated with $\boldsymbol{S}(\boldsymbol{T})$ will also be fixed. Since, by definition, $M(t)$ is finite, then $\mathbb{S}(t)$ and $\mathcal{E}$ are also necessarily finite. Hence, $\mathcal{E}^-$ is finite and fixed over time. Consider that $\mathbb{D}(.)$ implements a decision boundary that properly partition $\mathcal{E}^-$ into $\mathcal{E}^-_{detectable}$ and $\mathcal{E}^-_{undetectable}$, such that, $\mathbb{D}(e) = -1$ if $e \in \mathcal{E}^-_{detectable}$ and $\mathbb{D}(e) \neq -1$ if $e \in \mathcal{E}^-_{undetectable}$. As k is increasing and under the assumption that $\mathbb{D}(.)$ is only ever improving, each defender adaptation $G_k \underset{\mathcal{D}}{\longrightarrow} G_{k+1}$ can only cause $\mathcal{E}^-_{detectable}$ to grow and $\mathcal{E}^-_{undetectable}$ to decrease. Hence, $\mathbb{D}(.)$ can only move in the direction of $\mathbb{D}^*(.)$ via the defender's efforts to improve the defenses. But, for the attackers $\mathcal{A}$, there must then exist some $G_{k'}$ for which $G_{k'} \underset{\mathcal{A}}{\longrightarrow} G_{k'}$, which means there must exist some k' in the game sequence after which the attackers has exhausted their ability, within a finite EMM, to obfuscate their existing attacks or create new ones. In the defender's worst case, the attackers would be able to exploit all of $\mathcal{E}^-$ and, hence $\boldsymbol{\alpha} = \mathcal{E}^-$. However, as the defender improves $\mathbb{D}(.)$, it is the case that $\mathbb{D}(.) \underset{\mathcal{D}}{\longrightarrow} \mathbb{D}^*(.)$, $\mathcal{E}^-_{detectable} \longrightarrow \mathcal{E}^-$, and $\mathcal{E}^-_{undetectable} \longrightarrow \emptyset$. Hence, in a finite $\boldsymbol{S}(\boldsymbol{T})$ space, the attackers must eventually run out of adaptations at some $G_{k'}$ but the defender will still be able to adapt to $G_{k'} \underset{\mathcal{D}}{\longrightarrow} G_{k'+1}$, then $\{G_k\}_{k=0}^{K}$ must eventually

converge to a defender advantageous end-game (*i.e.*, there must exist a $k \to \infty$ such that $\{G_k\}_{k=0}^{K}$ converges to a G_k possessing defender advantageous NE). □

However, this addresses the question of whether $\{G_k\}_{k=0}^{K}$ eventually converges to a defender's advantageous end-game not whether this occurs within a reasonable time frame. The following lemma addresses this issue.

Lemma 6.2. *$\{G_k\}_{k=0}^{K}$ can converge in reasonable time frames if and only if the creation of new attacks exists, at a minimum, as a computationally hard problem for $\mathcal{A}$.*

Proof. Proof by construction. In order for $\{G_k\}_{k=0}^{K}$ to converge into a defender's advantageous end-game, the convergence conditions discussed above must be satisfied. Assume that we are in the game G_k. Let $\mathbb{D}(\alpha|G_k) = -1$ for all $\alpha \in \Sigma_{\mathcal{A}}$ (*i.e.*, G_k is $\mathcal{D}$ winnable in a NE sense). If $\mathcal{D}$ can show that $\mathcal{A}$ cannot adapt such that $G_k \underset{\mathcal{A}}{\longrightarrow} G_{k+1}$, then by definition, both convergence conditions would be satisfied. Let $\boldsymbol{\alpha}^k$ denote the set of attacks available to $\mathcal{A}$ in sub-game G_k. Clearly, if $\mathcal{D}$ can prove that extending $\boldsymbol{\alpha}^k$ constitutes a computationally hard or impossible attacker problem, then the defender can reasonably claim that $G_k \underset{\mathcal{A}}{\longrightarrow} G_k$. Whereas if the defender does not possess such a proof, then $\mathcal{D}$ must assume that $G_k \underset{\mathcal{A}}{\longrightarrow} G_{k+1}$ is possible and the defender's claim that $\{G_k\}_{k=0}^{K}$ converges no longer holds. □

As shown in Lemma 6.2, if $\mathcal{D}$ can show (*i.e.*, formally prove) that the transition to the next game G_{k+1} is at least computationally hard for $\mathcal{A}$ to accomplish (*i.e.*, an NP-complete problem, *etc.*), then $\mathcal{D}$ can reasonably claim that,

$$G_0 \longrightarrow G_1 \longrightarrow G_2 \longrightarrow \cdots \longrightarrow G^*,$$

i.e., $\mathcal{D}$ can reasonably claim that $\{G_k\}_{k=0}^{K}$ converges into a final $\mathcal{D}$'s advantageous end-game G^* with some reasonable number K of adaptation steps. Hence, we can

conclude from Lemma 6.2 that $\mathcal{D}$'s goal should be to update $\mathbb{D}(.)$ in manners that makes the next adaptations increasingly harder for $\mathcal{A}$ to do. It should be noted though that arriving at a G_k for which the defender $\mathcal{D}$ knows that $\mathcal{A}$ cannot adapt away from is equivalent to stating that $\mathcal{D}$ possesses a formal method proof of the defended environment's security.

6.7.2 Dynamic $M(t)$ and Convergence of $\{G_k\}_{k=0}^{K}$

Now, assume that $M(t)$ is not fixed over $\boldsymbol{T}$. As discussed above, this case more closely models real-world computer systems in which hardware updates occur and machines come and go. Accordingly, the set of all possible events $\mathcal{E}(t) = \mathcal{E}^{-}(t) \cup \mathcal{E}^{+}(t)$ also change over time as $\boldsymbol{S}(\boldsymbol{T})$ is no longer fixed. Moreover, due to the changing of $M(t)$, $\mathcal{E}$ must now be modeled in terms of a dynamical system. In particular, in Section 5.4, we showed that the EMM over a finite time period τ defines a measure space $\langle \mathbb{S}(\tau), \mathcal{R}(\tau), \mu' \rangle$. Additionally, in Section 5.4.3, we discussed that the only allowable feature maps $F(.)$ are those which preserve the σ-finite measure space characteristics of the EMM and, hence, the generated feature space $\mathcal{X}$ must also define a σ-finite measure space. Denote this feature measurement space as $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu \rangle$, where $\mathcal{P}(\mathcal{X})$ is power set of $\mathcal{X}$ which is a σ-algebra by definition, as discussed in Section 5.3.2.

Consider the EMM during any finite time period $\tau \subset \boldsymbol{T}$ over which the sequence of games $\{G_k\}_{k=0}^{K}$ is being played. The transition from any G_k to the next game G_{k+1} can be viewed as the action of a time shift operator $\mathcal{T}$ over the event spaces $\mathcal{E}_k$ and $\mathcal{E}_{k+1}$ that are associated with sub-games G_k and G_{k+1} respectively. Accordingly, as discussed by Section 6.2.2, the tuple $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$ will formally define a dynamical system. The convergence of $\{G_k\}_{k=0}^{K}$ can now be described in terms of the measure invariance of $F^k(.)$ over $\mathcal{T}$ in the limit as K goes to infinity. More particularly, the suite of defender's measurement features $F(.)$ can be viewed as generating random

processes $F(e)$ within their respective measurement spaces. If these are stationary random processes then clearly, by definition, $\mathcal{D}$ can apply anomaly detection to arrive at a $\mathbb{D}(.)$ such that $\forall \alpha_k \in \boldsymbol{\alpha}$, $\mathbb{D}(\alpha_k) = -1$. If $\mathcal{D}$ can also prove that $\mathcal{A}$'s adaptation problem is at least computationally hard, then $\{G_k\}_{k=0}^{K}$ must converge. Hence, in this case, the assessment of whether or not $\{G_k\}_{k=0}^{K}$ converges hinges on whether or not the $F(e)$ are stationary random processes. This in turn hinges on whether the $F(e)$ are measure invariant, or equivalently, whether or not the sequence $\{G_k\}_{k=0}^{K}$ denoted by the mapping $\mathcal{T} : \mathcal{E} \to \mathcal{E}$ denotes a stationary dynamical system. In Lemma 6.3, it is shown that $\mathcal{T}$ is not, in general, a measure preserving transformation and, hence, $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$ is not, in general, a stationary dynamical system. Therefore, $\{G_k\}_{k=0}^{K}$ cannot be, in general, assumed to converge to a defender advantageous end-game.

Lemma 6.3. *The dynamical system $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$ defined by the EMM is not a stationary dynamical system unless $\mathcal{A}$'s next adaptation problem is known by $\mathcal{D}$ to exist as at least a computationally hard problem.*

Proof. Proof by construction. Consider the EMM during any two consecutive games G_k and G_{k+1}, where the EMM defines the dynamical system $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$. Furthermore, assume there exists a set of dominated attackers' strategies $\boldsymbol{\alpha}_{dom}^{k}$ in G_k as well as the non-dominated strategies $\boldsymbol{\alpha}_{nd}^{k} = \boldsymbol{\alpha}^{k} \backslash \boldsymbol{\alpha}_{dom}^{k} \neq \emptyset$. Denote $\boldsymbol{\alpha}_{dom}^{k}$ as W^k. Now assume the game G_k transitions to a new game G_{k+1} as $G_k \underset{\mathcal{D}}{\longrightarrow} G_{k+1}$, where G_{k+1} now has a set of dominated attack strategies W^{k+1}. Clearly, $W^k \cap W^{k+1} = \emptyset$ as these dominated attacks denote the portions of the attack space that $\mathcal{A}$ discards as a result of each new defender adaptation. Moreover, it is clear that $W^{k+1} = \mathcal{T}(W^k)$, where W^k can be rewritten beginning from $k = 1$ as $W^k = \mathcal{T}^k(W^1)$. Clearly, in the limit as $k \to \infty$, $W^1 \cap \mathcal{T}^k(W^1) = \emptyset$ and, therefore, the W^k denotes, by Definition 6.9 in Section 6.2.2, a wandering set. Moreover, as the $\alpha_k \in W^k$ are useful attacks within G^k, then by definition they are also of non-zero measure (*i.e.*, $\forall \alpha_k \in W^k$, $\mu(\alpha_k) > 0$).

Therefore, the W^k form a wandering set of non-zero measure. Hence, any measurement features $F(.)$ spanning these wandering sets, by definition, cannot be measure invariant. Similarly, the attackers' adaptation can create other wandering sets of non-zero measure through the creation of the new attacks that are iteratively added at each attacker adaptation stage. These W^k can be redefined as the $\boldsymbol{\alpha_{new}}$ added into game G_k. Hence, by definition, $W^k = \mathcal{T}^k(W^1)$ and again, $W^k \cap W^1 = \emptyset$ for all k in limit $k \to \infty$ and $\mu(\alpha_k) > 0$ for all $\alpha_k \in \boldsymbol{\alpha}_{new}$ in each G_k. Hence, such new attacker adaptations also cause the $F(.)$ to fail to be measure invariant. Hence, unless the defender can show that such obfuscations and/or new attack creation cannot occur, then the defender cannot claim that $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$ is a stationary dynamical system as required if $\{G_k\}_{k=0}^{K}$ is to converge to a defender advantageous end-game. □

As discussed in the above lemma, since the dynamical system $\langle \mathcal{X}, \mathcal{P}(\mathcal{X}), \mu, \mathcal{T} \rangle$ cannot be considered to be a stationary dynamical system. The probabilities associated with α_k change over time within the defined EMM as a result of the attackers adaptations. Consequently, the attacker's behavior under the defender's measurement feature $F(.)$ constitutes a non-stationary process and, hence, $\mathcal{D}$ is required to track a non-stationary attack behaviors in order to construct the next required $\mathbb{D}(.)$ (*i.e.*, $Pr_{\omega^-}(.)$ changes over time in manners that increases its overlaps with $Pr_{\omega^+}(.)$ as a result of the attacker's need not to use dominated strategies). Figures 6.4 and fig:ch6:weaklywandersetnewattacks show the weakly wandering set resulting from both types of $\mathcal{A}$'s adaptations as described in Lemma 6.3. Moreover, by information theory, the time dependence of $Pr_{\omega^-}(.)$ means that the past histories available via $\widehat{\boldsymbol{Info}}(\boldsymbol{T}^-)$ cannot be used to construct the next defender adaptation to $\mathbb{D}(.)$.

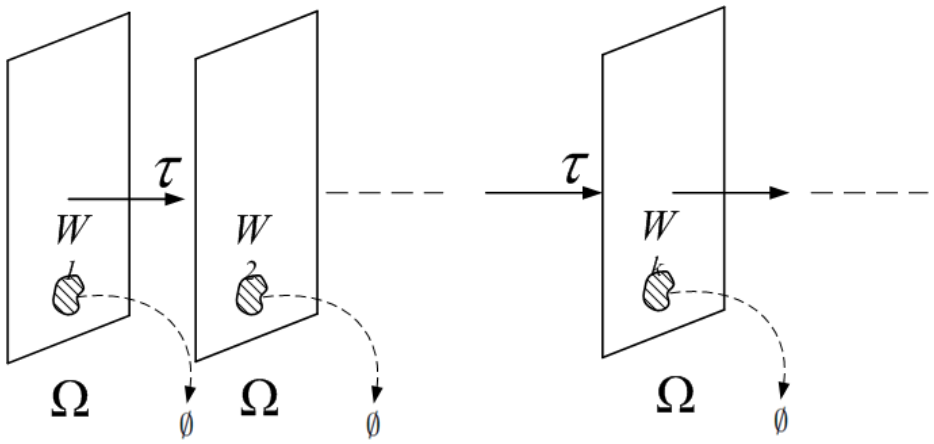


Figure 6.4: The weakly wandering set created by $\mathcal{A}$'s dominated attacks.

6.8 Discussion

In the previous sections, the attackers-defender on-going confrontation has been formalized as a game G. Then, we showed that the on-going adaptations of the players causes G to evolve into an iterative sequence of sub-games. We then discussed this game sequence and analyzed its potential convergence to a defender's advantageous end-game.

As shown by Lemma 6.1, if the EMM's memory is static, then the game sequence must eventually converge into a $\mathcal{D}$'s advantage as the on-going updates made by $\mathcal{D}$ to $\mathbb{D}(.)$ will eventually cause $\mathcal{A}$ to run out of attack adaptations. Additionally, as discussed in Lemma 6.2, if the sequence is to converge within reasonable time frames, then $\mathcal{D}$ must update $\mathbb{D}(.)$ in ways that can be formally make $\mathcal{A}$'s next adaptations harder to accomplish. Otherwise, $\mathcal{A}$ will continue to update their available attack set thereby not allowing the sub-game sequence to converge in a defender timing manner.

Whereas, if the EMM's memory is dynamic, Lemma 6.3 showed that the defender's measurement features $F(.)$ under the EMM cannot be considered to be measure in-

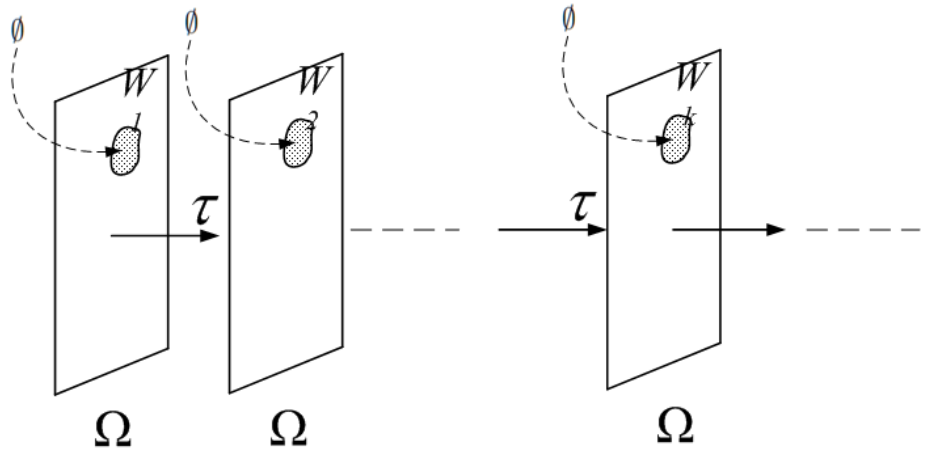


Figure 6.5: The weakly wandering set created by $\mathcal{A}$'s new attacks.

variant processes unless the defender can prove that the attackers' adaptation problem is at least computationally hard. Hence, $\mathcal{D}$ is left to track a non-stationary attack behaviors (or $e \in \mathcal{E}^-$ events) in the construction of the next required $\mathbb{D}(.)$ (*i.e.*, attacks $e \in \mathcal{E}^-$ for which $\mathbb{D}(e) = -1$ can be adapted by $\mathcal{A}$ to become events for which $\mathbb{D}(e) \neq -1$ solely on the basis of the attackers actions). Moreover, from information theory, it cannot be simply assumed that the required new $\mathbb{D}'(.)$ can be trained (*i.e.*, via machine learning approaches), such that, $\mathbb{D}'(e) = -1$, as the non-stationarity of $F(.)$ directly impose the constraint that past histories of these events e are no longer informative as to how $Pr[F(e)]$ changes over time. That is to say, $Pr[F(e)]$ becomes a time-dependent probability distribution, which changes as a direct result of the attackers' adaptations. Hence, $\mathbb{D}'(.)$ cannot be trained by simply training over past attack histories, as required by machine learning solutions.

6.9 Summary

In this chapter, an EMM-based game-theoretic model of the on-going confrontations between the attackers and system defenders has been formulated. The model has been used to explore the evolution of the resulting sequence of sub-games over time. The analysis showed that the attackers are always motivated to develop, if possible, malware that structured to bypass current system defenses, where this formally arises via game theory's strict dominance solution concept. This leads to the overall arms race being defined in terms of a sequence of sub-games that evolve over time. This sub-games sequence was then analyzed to determine the conditions required if it is to converge to a defender advantageous end-game. The implications of this analysis lead to the conclusions that either: (i) the defender must formally show that the attackers' next adaptation is, at least, computationally hard to achieve, or (ii) the defender must face the problem of needing to track non-stationary attack behaviors (*i.e.*, past attack information is not informative with respect to the necessary re-tuning of deployed defenses) as required to respond to the attacker's adaptations.

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions of this dissertation and outlines the contributions of the work to the field of computer security research. Additionally, the chapter provides some suggestions as to possible areas for future research.

7.1 Conclusions

The existing arms-race between malware authors and system defenders has motivated the need for formal frameworks to evaluate malware detection approaches. Hence, this dissertation has sought to address the open research area of the evaluation and analysis of malware detection approaches by developing and applying the EMM as an integrated formal framework for jointly modeling malware and its detection. To avoid the limited expressive capabilities of prior mathematical machines, the EMM has been developed as an extension to the existing Maurer model, a Turing equivalent model which possesses a close resemblance to real computers. This dissertation showed that the proposed EMM remains a Turing equivalent model which is able to model modern computer constructs and computer networks, as well as more complex programs such as virtual machines and modern multi-tabbed web browsers (Chapter 3).

Through the proposed EMM, we provided formalizations for the violations of the standard security policies (Chapter 4). Specifically, we provided the definitions for violations of confidentiality policies, integrity policies, availability policies, as well as resource usage policies. Additionally, we also provided formal definitions of a number of common malware classes, including viruses, Trojan horses, spyware, bots, and computer worms based on their common behaviors. It was shown that the proposed EMM is complete in terms of its ability to model all implementable malware within the context of a given EMM modeled defended environment.

The developed EMM was then used to evaluate and analyze the resilience of standard malware detection approaches (Chapter 5). We showed that static anti-malware signature scanners can be easily evaded by obfuscation, a result that is consistent with prior experimental work. Additionally, we also used the EMM to formally show that malware authors can avoid detection by dynamic system call sequence detection approaches, which also agrees with recent experimental work. A measure-theoretic model of the EMM was then developed through which the completeness of the EMM with respect to its ability to model all implementable malware detection approaches within an EMM modeled defended environment was shown.

Finally, using the developed EMM, a game-theoretic model was developed for the on-going confrontations between the attackers and the system defenders (Chapter 6). Using this game model, by applying game theory's strict dominance solution concept, it was shown that rational attackers are always required to develop malware that is designed to evade the currently deployed malware detection solutions. Additionally, we showed that the attacker and defender adaptations can be modeled as an iterative sequence of sub-games. Therefore, the question can be asked as to the conditions required if such a sequence (or arms-race) is to converge towards a defender advantageous end-game. It was shown via the EMM that, in the general context, this defender

desirable situation requires that the attacker's next attack adaptation problem exists as, at least, a computationally hard problem. If this is not the case, then we showed via modeling the EMM as a dynamical system model that the defender is left needing to track statistically non-stationary attack behaviors. Hence, by standard information theory constructs, past attack histories can be shown to be uninformative with respect to the defender's required development of the next defensive adaptations.

The major contributions of this dissertation to the field of computer security can be summarized as:

- The development of the EMM as a novel generic formal framework for the joint modeling and evaluation of malware and malware detection approaches which bridges Turing-reducible models and formal measure theory constructs.

- The EMM-based formalization of the violations of common security policies and a number of common malware classes based on their exhibited behaviors.

- The application of the proposed EMM to evaluate common malware detection approaches.

- The verification that the proposed EMM is complete in the sense that it can be used to both model all malware and all malware detection approaches that are implementable within a given modeled defended environment.

- The formalization and analysis of an EMM-based game-theoretic model for describing the time-evolution of on-going confrontations between the attackers and system defenders within given defended environments.

- The development of a formal proof via the EMM for the defender's need to track statistically non-stationary attack behaviors when needing to defend non-trivial IT environments from attacks by intelligent and rational adversaries.

The major significance of these contributions is that, prior to this work, no generic formal framework existed that allowed for the joint reasoning about malware and its detection approaches. In particular, existing formal frameworks have been developed for specific modeling objectives and have not been designed to be complete and comprehensive generic frameworks. As shown throughout this dissertation, the proposed EMM framework can be used to model a wide variety of issues, such as, the violations of common security policies and the behavior of common malware classes. The proposed EMM also provides an important bridge between Turing-reducible models with measure theory constructs. Moreover, prior to this work, the game-theoretic modeling of attackers and system defenders has sought to model specific games under specified network settings and system configurations (*i.e.*, specific security games) and has not sought to model the wider strategic confrontation or how such games evolve over time as the attackers and defenders necessarily adapt to what the other is doing.

7.2 Future Work

The theoretical areas which were touched on in this work and which should be areas of future research include but are not limited to:

- The formal modeling of malware: Formally defining other malware classes, such as, *adware*, *rabbits*, *etc* [114], through the use of the proposed EMM can be further explored. More generally, formal definitions of security attacks occurring within enterprise-scale environments could be explored via the EMM.

- The formal evaluation of malware detection approaches: Using the EMM as an analytical tool to evaluate the detection capabilities of potential malware detection approaches versus intelligent adversaries can be further investigated.

- Additionally, using the EMM to evaluate the resiliency of various existing malware detection and intrusion detection approaches could be explored.

Moreover, the EMM provides a formal framework that can be used for exploring other relevant security issues, such as:

- Assessing the probability of developing resilient execution monitors (EMs) [110] which monitor the execution of programs to detect if whether or not they violate the system's security policies. These mechanisms of enforcing security policies can benefit from the formal definitions of malware and the violations of security policies,as well as from the EMM's ability to incorporate issues arising from the presence of intelligent and rational adversaries.

- Investigating the application of the EMM to provide a formal modeling framework for malware and other attacks within *clouds*. In particular, with the possible continuous growth of cloud computing, a better understanding of security and attack mechanisms within clouds is needed. The scale-independent nature of the EMM combined with its completeness properties may enable it to provide a new analysis tool in this domain.

- Assessing the computational complexity of the attackers' adaptation process under various scenarios and defenses could also be explored via the EMM. This may support to develop more resilient security defenses.

- Finally, the EMM could be applied to assess new approaches for developing quantitative cyber-security risk metrics, which is the the third of the four grand challenges in trustworthy computing [212].

It should be noted that the EMM is designed and intended to provide a theoretical framework within which to reason about security issues in light of intelligent and

capable adversaries. As such, its purpose is not to provide direct contributions to day-to-day operational security issues, although it can be used to provide insights into how such issues may evolve over time.

Bibliography

[1] M. Christodorescu, C. Kruegel, and S. Jha, "Mining Specifications of Malicious Behavior," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, (New York, NY, USA), pp. 5–14, ACM Press, 2007.

[2] Trend Micro, "TrendLabs 2013 Annual Securiy Roundup." `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-cashing-in-on-digital-information.pdf`, 2014. Last accessed: 27-June-2014.

[3] Symantec, "Symantec Internet Security Threat Report, Volume 18, Trends for 2012." `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf`, April 2013. Last accessed: 23-April-2013.

[4] R. Godwin-Jones, "Emerging Technologies Mobile-Computing Trends: Lighter, Faster, Smarter," *Language Learning & Technology*, vol. 12, pp. 3–9, November 2009.

[5] J. Shin and D. Spears, "The Basic Building Blocks of Malware," tech. rep., University of Wyoming, 2006.

[6] W. Cui, *Automated Malware Detection by Inferring Intent*. PhD thesis, Electrical Engineering and Computer Science, University of California at Berkeley, September 2006.

[7] N. Idika and A. Mathur, "A Survey of Malware Detection Techniques," tech. rep., Department of Computer Science, Purdue University, West Lafayette, IN, USA, 2007.

[8] H. Inoue, *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, University of New Mexico, December 2005.

[9] S. Hofmeyr, *An Immunological Model of Distributed Detection and Its Application to Computer Security*. PhD thesis, University of New Mexico, May 1999.

[10] M. Marsono, *Towards Improving E-mail Content Classification for Spam Control: Architecture, Abstraction, and Strategies*. PhD thesis, University of Victoria, August 2007.

[11] S. Stolfo, K. Wang, and W. Li, *Towards Stealthy Malware Detection*. Advances in Information Security, Springer US, 2007.

[12] F. Song and T. Touili, "Efficient malware detection using model-checking," in *Proceedings of the 18th International Symposium on Formal Methods (FM 2012)*, pp. 418–433, August 2012.

[13] M. Preda, M. Christodorescu, S. Jha, and S. Debray, "A Semantics-Based Approach to Malware Detection," *Journal of SIGPLAN Notices*, vol. 42, no. 1, pp. 377–388, 2007.

[14] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, 2006.

[15] H. Inoue and S. Forrest, "Anomaly Intrusion Detection in Dynamic Execution Environments," in *Proceedings of the 2002 Workshop on New Security Paradigms, NSPW '02*, (Virginia Beach, Virginia, USA), pp. 52–60, 2002.

[16] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy.*, (Oakland, CA, USA), pp. 133–145, 1999.

[17] W. Lee and S. Stolfo, "Data Mining Approaches for Intrusion Detection," in *Proceedings of the 7th USENIX Security Symposium*, (San Antonio, TX), pp. 6–22, 1998.

[18] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen, "HoneyStat: LocalWorm Detection Using Honeypots," in *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, pp. 39–58, September 2004.

[19] K. Wang and S. Stolfo, "Anomalous Payload-based Network Intrusion Detection," in *Proceedings of the Eighth International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, pp. 203–222, 2004.

[20] S. Ortolani, C. Giuffrida, and B. Crispo, "Bait Your Hook: A Novel Detection Technique for Keyloggers," in *Proceedings of 2010 Recent Advances in Intrusion Detection (RAID 2010)*, pp. 198–217, September 2010.

[21] A. Srivastava and J. Giffin, "Automatic Discovery of Parasitic Malware," in *Proceedings of 2010 Recent Advances in Intrusion Detection (RAID 2010)*, pp. 97–117, September 2010.

[22] M. Shankarapani, S. Ramamoorthy, R. Movva, and S. Mukkamala, "Malware Detection Using Assembly and API Call Sequences," *Journal of Computer Virology*, pp. 1–13, 2010.

[23] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using System-Centric Models for Malware Protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, (New York, NY, USA), pp. 399–412, ACM, October 2010.

[24] P. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying Dormant Functionality in Malware Programs," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, (Washington, DC, USA), pp. 61–76, IEEE Computer Society, May 2010.

[25] Z. Lin, X. Zhang, and D. Xu, "Automatic Reverse Engineering of Data Structures from Binary Execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, March 2010.

[26] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS 2011)*, February 2011.

[27] C. Kolbitsch, E. Kirda, and C. Kruegel, "The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, (New York, NY, USA), ACM, October 2011.

[28] M. Lindorfer, C. Kolbitsch, and P. Comparetti, "Detecting Environment-Sensitive Malware," in *Proceedings of the 14th International Symposium of Recent Advances in Intrusion Detection (RAID 2011)*, (Menlo Park, CA, USA), September 2011.

[29] M. Christodorescu and S. Jha, "Testing Malware Detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, (Boston, Massachusetts, USA), pp. 34–44, ACM Press, 2004.

[30] N. Kuzurin, A. Shokurov, N. Varnovsky, and V. Zakharov, *Information Security*, ch. On the Concept of Software Obfuscation in Computer Security, pp. 281–298. Lecture Notes in Computer Science, Springer Berlin/Heidelberg, September 2007.

[31] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating Mimicry Attacks Using Static Binary Analysis," in *Proceedings of the 14th conference on USENIX Security Symposium (SSYM 2005)*, (Berkeley, CA, USA), pp. 11–36, USENIX Association, August 2005.

[32] D. Wagner and P. Soto, "Mimicry Attacks on Host-based Intrusion Detection Systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, (New York, NY, USA), pp. 255–264, ACM, November 2002.

[33] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static Analyzer of Vicious Executables (SAVE)," in *Proceedings of 20th Annual Computer Security Applications Conference*, pp. 326–334, 2004.

[34] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2007)*, (Los Alamitos, CA, USA), pp. 421–430, IEEE Computer Society, December 2007.

[35] M. Riccardi, R. Di Pietro, M. Palanques, and J. Vila, "Titans' Revenge: Detecting Zeus via its Own Flaws," *Journal of Computer Networks*, vol. 57, no. 2, pp. 422–435, 2013.

[36] P. Sinha, A. Boukhtouta, V. Belarde, and M. Debbabi, "Insights from the Analysis of the Mariposa Botnet," in *Proceedings of 2010 Fifth International Conference on Risks and Security of Internet and Systems (CRiSIS)*, October 2010.

[37] T. Chen, "Trends in Viruses and Worms," *The Internet Protocol Journal*, vol. 6, pp. 23–33, September 2003.

[38] S. Josse, "How to Assess the Effectiveness of your Anti-virus?," *Journal of Computer Virology*, vol. 2, pp. 51–65, August 2006.

[39] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-Aware Malware Detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pp. 32–46, 2005.

[40] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM 2003)*, (Berkeley, CA, USA), pp. 1–18, USENIX Association, 2003.

[41] S. Stolfo, K. Wang, and W. Li, "Fileprint Analysis for Malware Detection," tech. rep., Columbia University, June 2005.

[42] M. Shafiq, S. Khayam, and M. Farooq, "Embedded Malware Detection Using Markov n-Grams," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)*, pp. 88–107, July 2008.

[43] J. Kolter and M. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *The Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[44] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime," in *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection (RAID 2009)*, (France), Springer, September 2009.

[45] I. Yoo and U. Ultes-Nitsche, "Non-Signature Based Virus Detection: Towards Establishing Unknown Virus Detection Technique Using SOM," *Journal of Computer Virology*, vol. 2, pp. 163–186, December 2006.

[46] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An Intelligent PE-Malware Detection System Based on Association Mining," *Journal of Computer Virology*, February 2008.

[47] Y. Zhou and M. Inge, "Malware Detection Using Adaptive Data Compression," in *Proceedings of the 1st ACM Workshop on Security and Artificial Intelligence (AISec 2008)*, (New York, NY, USA), pp. 53–60, ACM, October 2008.

[48] M. Shafiq, S. Tabish, and M. Farooq, "PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables," in *Proceedings of 2009 Virus Bulletin Conference (VB '09)*, September 2009.

[49] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th USENIX Security Symposium*, (Berkeley, CA, USA), pp. 351–366, USENIX Association, 2009.

[50] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, February 2009.

[51] X. Hu, T. Chiueh, and K. Shin, "Large-Scale Malware Indexing Using Function-Call Graphs," in *Proceedings of ACM Conference on Computer and Communications Security (CCS'09)*, pp. 611–620, November 2009.

[52] K. Tan and R. Maxion, ""Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 188–202, 2002.

[53] P. Li, L. Liu, D. Gao, and M. Reiter, "On Challenges in Evaluating Malware Clustering," in *Proceedings of 2010 Recent Advances in Intrusion Detection (RAID 2010)*, pp. 238–255, September 2010.

[54] J. McHugh, "Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory," *ACM Transactions on Information System Security*, vol. 3, pp. 262–294, November 2000.

[55] F. Cohen, *Computational Aspects of Computer Viruses*, pp. 324–355. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[56] F. Cohen, "Computer Viruses: Theory and Experiments," *Computers and Security*, vol. 6, pp. 22–35, February 1987.

[57] L. Adleman, "An Abstract Theory of Computer Viruses," in *Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO 1988)*, (London, UK), pp. 354–374, Springer-Verlag, 1990.

[58] K. Kauranen and E. Mäkinen, "A Note on Cohen's Formal Model for Computer Viruses," *ACM SIGSAC Review*, vol. 8, no. 2, pp. 40–43, 1990.

[59] G. Jacob, E. Filiol, and H. Debar, "Malware as Interaction Machines: a New Framework for Behavior Modeling," *Journal of Computer Virology*, vol. 4, pp. 235–250, August 2008.

[60] H. Thimbleby, S. Anderson, and P. Cairns, "A Framework for Modelling Trojans and Computer Virus Infection," *The Computer Journal*, vol. 41, no. 7, pp. 444–458, 1999.

[61] P. Wegner, "Interaction as a basis for empirical computer science," *ACM Computing Surveys*, vol. 27, pp. 45–48, March 1995.

[62] E. Filiol and S. Josse, "A Statistical Model for Undecidable Viral Detection," *Journal of Computer Virology*, vol. 3, pp. 65–74, June 2007.

[63] M. Sharif, K. Singh, J. Giffin, and W. Lee, "Understanding Precision in Host Based Intrusion Detection: Formal Analysis and Practical Models," in *Proceedings of the 10th International Symposium of Recent Advances in Intrusion Detection (RAID 2007)*, (Gold Goast, Queensland, Australia), pp. 42–62, September 2007.

[64] G. Jacob, H. Debar, and E. Filiol, "Malware Behavioral Detection by Attribute-Automata Using Abstraction from Platform and Language," in *Proceedings of the 12th International Workshop on Recent Advances in Intrusion Detection (RAID 2009)*, pp. 81–100, Springer Berlin/Heidelberg, September 2009.

[65] G. Jacob, E. Filiol, and H. Debar, "Formalization of Viruses and Malware Through Process Algebras," in *Proceedings of the 2010 International Conference on Availability, Reliability and Security*, pp. 597–602, February 2010.

[66] G. Jacob, *Malware Behavioral Models: Bridging Abstract and Operational Virology*. PhD thesis, Université de Rennes, December 2009.

[67] M. Preda and C. D. Giusto, "Hunting Distributed Malware with the κ-Calculus," in *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory (FCT 2011)*, August 2011.

[68] T. Alpcan and T. Başar, "A Game Theoretic Approach to Decision and Analysis in Network Intrusion Detection," in *Proceedings of the 42nd IEEE Conference on Decision and Control*, (Maui, HI), December 2003.

[69] S. Schmidt, T. Alpcan, S. Albayrak, T. Başar, and A. Muller, "A Malware Detector Placement Game for Intrusion Detection," in *Proceedings of the 2nd International Workshop on Critical Information Infrastructures Security (CRITIS 2007)*, (Malaga, Spain), 2007.

[70] A. Gueye, *A Game Theoretical Approach to Communication Security*. PhD thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, March 2011.

[71] A. Bensoussan, M. Kantarcioglu, and S. Hoe, "A Game-Theoretical Approach for Finding Optimal Strategies in a Botnet Defense Model," in *Proceedings of the First International Conference on Decision and Game Theory for Security (GameSec 2010)*, (Berlin, Heidelberg), pp. 135–148, Springer-Verlag, November 2010.

[72] E. Filiol, M. Helenius, and S. Zanero, "Open Problems in Computer Virology," *Journal of Computer Virology*, vol. 1, pp. 55–66, 2006.

[73] S. Gordon and R. Ford, "Real World Anti-Virus Product Reviews and Evaluations, the Current State of Affairs," in *Proceedings of the 19th National Information Systems Security Conference (NISSC 1996)*, October 1996.

[74] W. Maurer, "A Theory of Computer Instructions," *Journal of the ACM (JACM)*, vol. 13, pp. 226–235, April 1966.

[75] W. Maurer, "A Theory of Computer Instructions," *Science of Computer Programming*, vol. 60, no. 3, pp. 244–273, 2006.

[76] J. Bergstra and C. Middelburg, "Simulating Turing Machines on Maurer Machines," *Journal of Applied Logic*, vol. 6, pp. 1–23, March 2008.

[77] G. Jacob, E. Filiol, and H. Debar, "Functional Polymorphic Engines: Formalisation, Implementation and Use Cases," *Journal of Computer Virology*, vol. 5, pp. 247–261, August 2009.

[78] Z. Zuo and M. Zhou, "Some Further Theoretical Results about Computer Viruses," *The Computer Journal*, vol. 47, no. 6, pp. 627–633, 2004.

[79] E. Filiol, "Malware Pattern Scanning Schemes Secure Against Black-Box Analysis," *Journal of Computer Virology*, vol. 2, pp. 35–50, August 2006.

[80] F. Esponda, S. Forrest, and P. Helman, "A Formal Framework for Positive and Negative Detection Schemes," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 34, pp. 357–373, February 2004.

[81] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, "Formalizing Sensitivity in Static Analysis for Intrusion Detection," in *Proceedings of 2004 IEEE Symposium on Security and Privacy (SP 2004)*, May 2004.

[82] T. Song, *Formal Reasoning about Intrusion Detection Systems*. PhD thesis, Computer Science, University of California at Davis, 2007.

[83] P. Fogla and W. Lee, "Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, (New York, NY, USA), pp. 59–68, ACM, 2006.

[84] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skoric, "Towards an Information-Theoretic Framework for Analyzing Intrusion Detection Systems," in *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS 2006)*, pp. 527–546, September 2006.

[85] W. Lee and D. Xiang, "Information Theoretic Measures for Anomaly Detection," in *Proceedings of 2001 IEEE Symposium on Security and Privacy (SP 2001)*, pp. 130–143, May 2001.

[86] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skoric, "Measuring Intrusion Detection Capability: An Information Theoretic Approach," in *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2006)*, (New York, NY, USA), pp. 90–101, ACM, 2006.

[87] S. Camtepe and B. Yener, "Modeling and Detection of Complex Attacks," in *Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks (SecureComm 2007)*, September 2007.

[88] S. Upadhyaya, R. Chinchani, and K. Kwiat, "An Analytical Framework for Reasoning about Intrusions," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS 2001)*, pp. 99–108, October 2001.

[89] A. Cárdenas, J. Baras, and K. Seamon, "A Framework for the Evaluation of Intrusion Detection Systems," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP 2006)*, (Washington, DC, USA), pp. 63–77, IEEE Computer Society, May 2006.

[90] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca, "A Logical Framework for Reasoning About Access Control Models," in *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT 2001)*, (New York, NY, USA), pp. 41–52, ACM, 2001.

[91] E. Filiol, *Computer Viruses: From Theory to Applications*. Springer Paris, 2006.

[92] H. Lewis and C. Papadimitriou, *Elements of the Theory of Computation*. Prentice Hall, Inc., second ed., 1998.

[93] J. Martin, *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Inc., second ed., 1997.

[94] P. Linz, *An Introduction to Formal Languages and Automata*. D. C. Heath and Company, 1990.

[95] E. Filiol, "Formalisation and Implementation Aspects of K-ary (Malicious) Codes," *Journal of Computer Virology*, vol. 3, pp. 75–86, 2007.

[96] M. Ramilli, M. Bishop, and S. Sun, "Multiprocess Malware," in *Proceedings of the 6th International Conference on Malicious and Unwanted Software*, October 2011.

[97] P. Wegner, "Why Interaction is More Powerful Than Algorithms," *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, 1997.

[98] J. Bergstra and C. Middelburg, "Maurer Computers with Single-Thread Control," *Fundamenta Informaticae*, vol. 80, pp. 333–362, April 2008.

[99] L. van Zelst, "Modeling Computer Viruses," Master's thesis, Universiteit van Amsterdam, July 2008.

[100] I. Tomek, ed., *The Foundations of Computer Architecture and Organization*. W. H. Freeman and Company, 1990.

[101] D. Hamlet, *Composing Software Components*. Springer, 2010.

[102] P. Cox and B. Song, "A Formal Model for Component-Based Software," in *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC 2001)*, (Washington, DC, USA), IEEE Computer Society, 2001.

[103] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, vol. 2, pp. 119–129, 2000.

[104] M. Bishop, *Computer Security: Art and Science*. Addison Wesley Publishing Company, 2002.

[105] S. Homer and A. Selman, *Computability and Complexity Theory*. Springer, 2011.

[106] G. Popek and R. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, pp. 412–421, July 1974.

[107] T. Budd, *An Introduction to Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2002.

[108] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," in *Information Hiding*, vol. 4437 of *Lecture Notes in Computer Science*, pp. 232–248, Springer Berlin Heidelberg, 2007.

[109] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[110] F. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[111] L. Bauer, J. Ligatti, and D. Walker, "More Enforceable Security Policies," in *Foundations of Computer Security Workshop*, July 2002.

[112] A. Mok and W. Yu, "TINMAN: A Resource Bound Security Checking System for Mobile Code," in *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, (London, UK, UK), pp. 178–193, Springer-Verlag, 2002.

[113] J. Yi, H. Woo, J. Browne, A. Mok, F. Xie, E. Atkins, and C. Lee, "Incorporating Resource Safety Verification to Executable Model-based Development for Embedded Systems," in *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 137–146, april 2008.

[114] J. Aycock, *Computer Viruses and Malware (Advances in Information Security)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[115] J. Ligatti, *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.

[116] P. Stavroulakis and M. Stamp, eds., *Handbook of Information and Communication Security*. Springer Berlin Heidelberg, 2010.

[117] H. Hongjun, C. Sihua, L. Li, F. Tao, P. Li, and Z. Zhiji, "A New Definition of Virus," *Wuhan University Journal of Natural Science*, vol. 11, no. 6, pp. 1697–1700, 2006.

[118] P. Szor, *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, first ed., 2005.

[119] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic Spyware Analysis," in *Proceedings of 2007 USENIX Annual Technical Conference*, (Berkeley, CA, USA), USENIX Association, June 2007.

[120] A. Stamminger, C. Kruegel, G. Vigna, and E. Kirda, "Automated Spyware Collection and Analysis," in *Proceedings of 2009 Information Security Conference (ISC 2009)*, September 2009.

[121] G. G. ad J. Zhang and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*, August 2008.

[122] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "Jackstraws: Picking command and control connections from bot traffic," in *Proceedings of the 20th USENIX Conference on Security*, (Berkeley, CA, USA), USENIX Association, August 2011.

[123] M. Fossi, "Symantec Internet Security Threat Report, Volume 16, Trends for 2010." `https://www4.symantec.com/mktginfo/downloads/21182883_GA_REPORT_ISTR_Main-Report_04-11_HI-RES.pdf`, April 2011. Last accessed: 8-April-2011.

[124] PandaLabs, "Pandalabs Quartely Report (January-March 2010)." `http://press.pandasecurity.com/wp-content/uploads/2010/05/PandaLabs-Quaterly-Report.pdf`, April 2010. Last accessed: 27-June-2014.

[125] McAfee Labs, "McAfee Threats Report: First Quarter 2012." `http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q1-2012.pdf`, April 2012. Last accessed: 27-June-2014.

[126] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A Multifaceted Approach to Understanding the Botnet Phenomenon," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC 2006)*, (New York, NY, USA), pp. 41–52, ACM, 2006.

[127] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your Botnet is My Botnet: Analysis of a Botnet Takeover," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[128] P. Li, M. Salour, and X. Su, "A Survey of Internet Worm Detection and Containment," *IEEE Communications Surveys Tutorials*, vol. 10, no. 1, pp. 20–35, 2008.

[129] C. Smith, A. Matrawy, S. Chow, and B. Abdelaziz, "Computer Worms: Architectures, Evasion Strategies, and Detection Mechanisms," *Journal of Information Assurance and Security*, vol. 4, pp. 69–83, 2009.

[130] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A Taxonomy of Computer Worms," in *Proceedings of the 2003 ACM Workshop on Rapid Malcode(WORM 2003)*, pp. 11–18, ACM Press, 2003.

[131] L. Carettoni, C. Merloni, and S. Zanero, "Studying Bluetooth Malware Propagation: The BlueBag Project," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 17–25, 2007.

[132] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[133] I. K. Rana, *An Introduction to Measure and Integration*. American Mathematical Society, second ed., 2005.

[134] K. Athreya and S. Lahiri, *Measure Theory and Probability Theory*. Springer, New York, 2006.

[135] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *Proceedings of 2006 Annual Conference of the European Institute for Computer Antivirus Research (EICAR 2006)*, April 2006.

[136] D. Mutz, C. Kruegel, W. Robertson, G. Vigna, and R. Kemmerer, "Reverse Engineering of Network Signatures," in *Proceeding of the 4th Annual AusCERT Information Technology Security Conference*, May 2005.

[137] P. Halmos, *Measure Theory*. Springer-Verlag, 1970.

[138] D. Cohn, *Measure Theory*. Birkhauser Advanced Texts, second ed., 2013.

[139] L. Ambrosio, G. D. Prato, and A. Mennucci, *Introduction to Measure Theory and Integration*.

[140] L.-Y. S. Hsieh, "Ergodic Theory of Multidimensional Random Dynamical Systems," Master's thesis, Department of Mathematics and Statistics, University of Victoria, 2008.

[141] R. Gray, *Probability, Random Processes, and Ergodic Properties*. Springer, second ed., 2009.

[142] F. Gebali, *Analysis of Computer and Communication Networks*. Springer, 2008.

[143] C. Shannon, "The Mathematical Theory of Communication," tech. rep., University of Illinois, 1949.

[144] P. Attfield, "United States v Gorshkov Detailed Forensics and Case Study; Expert Witness Perspective," in *Proceedings of the 2005 First International Workshop on Systematic Approaches to Digital Forensic Engineering*, pp. 3–24, November 2005.

[145] R. Duda, P. Hart, and D. Stork, *Pattern Classification.* John Wiley & Sons Inc., second ed., 2001.

[146] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 120–128, IEEE Computer Society, 1996.

[147] H. Shah, J. Undercoffer, and A. Joshi, "Fuzzy Clustering for Intrusion Detection," in *Proceedings of The 12th IEEE International Conference on Fuzzy Systems (FUZZ '03)*, pp. 1274–1278, May 2003.

[148] D. Barbará, C. Domeniconi, and J. Rogers, "Detecting outliers using transduction and statistical testing," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 2006, (New York, NY, USA), pp. 55–64, ACM, 2006.

[149] J. Yao, S. Zhao, and L. Saxton, "A Study on Fuzzy Intrusion Detection," pp. 23–30, March 2005.

[150] D. Denning, "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[151] K. Fukunaga, *Introduction to Statistical Pattern Recognition.* Morgan Kaufmann Publishing, second ed., 1990.

[152] A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K.Yuksel, S. Camtepe, and S. Albayrak, "Static Analysis of Executables for Collaborative Malware Detection on Android," in *Proceedings of the 2009 IEEE International Conference on Communication (ICC 2009)*, June 2009.

[153] J. Bergeron, M. Debbabi, M. Erhioui, and B. Ktari, "Static Analysis of Binary Code to Isolate Malicious Behaviors," in *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises (WET-ICE 1999)*, (Washington, DC, USA), pp. 184–189, IEEE Computer Society, 1999.

[154] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static Detection of Malicious Code in Executable Programs," in *Proceedings of the 1st Symposium on Requirements Engineering for Information Security*, March 2001.

[155] D. Bruschi, L. Martignoni, and M. Monga, "Detecting Self-Mutating Malware Using Control Flow Graph Matching," tech. rep., Universit degli Studi di Milano, 2006.

[156] J. Kolter and M. Maloof, "Learning to Detect Malicious Executables in the Wild," in *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2004)*, (New York, NY, USA), pp. 470–478, ACM, 2004.

[157] M. Siddiqui, M. Wang, and J. Lee, *Wireless Networks, Information Processing and Systems*, ch. Detecting Trojans Using Data Mining Techniques, pp. 400–411. Communications in Computer and Information Science, Springer Berlin Heidelberg, August 2008.

[158] T. Wang, S. Horng, M. Su, C. Wu, P. Wang, and W. Su, "A Surveillance Spyware Detection System Based on Data Mining Methods," in *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, July 2006.

[159] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Control Flow to Detect Malware," in *Inter-Regional Workshop on Rigorous System Development and Analysis*, (Nancy, France), 2007.

[160] G. Vigna, *Malware Detection*, ch. Static Disassembly and Code Analysis, pp. 19–41. Advances in Information Security, Springer US, 2007.

[161] C. Leita, U. Bayer, and E. Kirda, "Exploiting Diverse Observation Perspectives to Get Insights on the Malware Landscape," in *Proceedings of 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, pp. 393 –402, June 2010.

[162] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pp. 29 –44, May 2010.

[163] J. Caballero, N. Johnson, S. McCamant, and D. Song, "Binary Code Extraction and Interface Identification for Security Applications," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, March 2010.

[164] S. Cesare and Y. Xiang, "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost," in *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA 2010)*, (Washington, DC, USA), pp. 721–728, IEEE Computer Society, April 2010.

[165] M. Sharif, V. Yegneswaran, H. Saidi, and P. Porras, "Eureka: A Framework for Enabling Static Analysis on Malware," Tech. Rep. SRI-CSL-08-XX, College of Computing, Georgia Instititue of Technology, April 2008.

[166] I. Sorokin, "Comparing files using structural entropy," *Journal of Computer Virology*, pp. 259–265, November 2011.

[167] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation: Tools for Software Protection," *IEEE Transactions on Software Engineering*, vol. 28, pp. 735–746, August 2002.

[168] I. Popov, S. Debray, and G. Andrews, "Binary Obfuscation Using Signals," in *Proceedings of 16th USENIX Security Symposium (SS 2007)*, (Berkeley, CA, USA), pp. 1–16, USENIX Association, August 2007.

[169] M. Preda, *Code Obfuscation and Malware Detection by Abstract Interpretation*. PhD thesis, Universita degli Studi di Verona, May 2007.

[170] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01)*, (London, UK), pp. 1–18, Springer-Verlag, 2001.

[171] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, December 1992.

[172] J. Borello and L. M, "Code Obfuscation Techniques for Metamorphic Viruses," *Journal of Computer Virology*, vol. 4, pp. 211–220, August 2008.

[173] K. Rieck, T. Holz, C. Willems, P. Dússel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)*, pp. 108–125, July 2008.

[174] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based Spyware Detection," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[175] T. Lee and J. Mody, "Behavioral Classification," in *Proceedings of 2006 Annual Conference of the European Institute for Computer Antivirus Research (EICAR 2006)*, April 2006.

[176] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy (SP 2001)*, (Washington, DC, USA), pp. 156–168, IEEE Computer Society, 2001.

[177] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP 2007)*, (Washington, DC, USA), pp. 231–245, IEEE Computer Society, May 2007.

[178] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, *Botnet Detection*, ch. Automatically Identifying Trigger-based Behavior in Malware, pp. 65–88. Advances in Information Security, Springer US, October 2007.

[179] J. Wilhelm and T. Chiueh, "A Forced Sampled Execution Approach to Kernel Rootkit Identification," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID 2007)*, (Berlin, Heidelberg), pp. 219–235, Springer-Verlag, August 2007.

[180] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, March 2010.

[181] M. Neugschwandtner, P. Comparett, G. Jacob, and C. Kruegel, "FORECAST: Skimming off the Malware Cream," in *Proceedings of the 2011 Annual Computer Security Applications Conference (ACSAC 2011)*, December 2011.

[182] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic Analysis of Malicious Code," *Journal of Computer Virology*, vol. 2, pp. 67–77, August 2006.

[183] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo, "Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses," in *Proceedings of the 5th International Symposium of Recent Advances in Intrusion Detection (RAID 2002)*, (Zurich, Switzerland), pp. 36–53, September 2002.

[184] D. Gao, M. Reiter, and D. Song, "Behavioral Distance Measurement Using Hidden Markov Models," in *Proceedings of the 9th International Symposium of Recent Advances in Intrusion Detection (RAID 2006)*, (Hamburg, Germany), pp. 19–40, September 2006.

[185] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting, "System Call Monitoring Using Authenticated System Calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, pp. 216–229, July 2006.

[186] M. Osborne and A. Rubinstein, *A Course in Game Theory*. MIT Press, 1994.

[187] R. Myerson, *Game Theory: Analysis of Conflict*. Cambridge, Massachusetts, USA: Harvard University Press, 1997.

[188] T. Başar and G. Olsder, *Dynamic Non-Cooperative Game Theory*. San Diego, CA, USA: Academic Press, 1995.

[189] S. Tadelis, *Game Theory: An Introduction*. Princeton University Press, 2013.

[190] P. Walters, *Introduction to Ergodic Theory*. Springer-Verlag, 1982.

[191] R. Goldblatt, *Topoi: The Categorial Analysis of Logic*. North-Holland Publishing Company, fourth ed., 1984.

[192] A. Papoulis and S. Pillai, *Probability, Random Variables and Stochastic Processes*. McGraw Hill Inc., fourth ed., 2002.

[193] A. Danilenko and C. Silva, *Mathematics of Complexity and Dynamical Systems*, ch. Ergodic Theory: Non-singular Transformations, pp. 329–356. Springer US, 2011.

[194] P. Peeples, Jr., *Probability, Random Variables and Random Signal Principles*. McGraw-Hill Inc., fourth ed., 2001.

[195] A. Agah, S. Das, K. Basu, and M. Asadi, "Intrusion Detection in Sensor Networks: A Non-cooperative Game Approach," in *Proceedings of 2004 IEEE International Symposium on Network Computing and Applications (NCA 2004)*, (Cambridge, MA, USA), pp. 343–346, August 2004.

[196] L. Chen and J. Leneutre, "A Game Theoretical Framework on Intrusion Detection in Heterogeneous Networks," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 2, pp. 165–178, 2009.

[197] W. Lu, S. Xu, and X. Yi, "Optimizing active cyber defense," in *Proceedings of the 4th International Conference on Decision and Game Theory for Security (GameSec 2013)*, pp. 206–225, Springer International Publishing, November 2013.

[198] K. Lye and J. Wing, "Game Strategies in Network Security," Tech. Rep. CMU-CS-02-136, School of Computer Science, Carnegie Mellon University, May 2002.

[199] A. Nochenson and C. Heimann, "Simulation and Game-Theoretic Analysis of an Attacker-Defender Game," in *Proceedings of the 3rd International Conference on Decision and Game Theory for Security (GameSec 2012)*, November 2012.

[200] T. Alpcan and S. Buchegger, "Security Games for Vehicular Networks," in *Proceedings of the Allerton Conference on Communication, Control, and Computing*, (Urbana-Champaign, IL, USA), September 2008.

[201] Q. Zhu, C. Fung, R. Boutaba, and T. Baser, "A Game-Theoretical Approach to Incentive Design in Collaborative Intrusion Detection Networks," in *Proceedings of the First ICST International Conference on Game Theory for Networks (GameNets'09)*, (Piscataway, NJ, USA), pp. 384–392, IEEE Press, 2009.

[202] G. Theodorakopoulos, J. Baras, and J. Boudec, "Dynamic Network Security Deployment Under Partial Information," in *Proceedings of the 2008 46th Annual Allerton Conference on Communication, Control, and Computing*, September 2008.

[203] Q. Zhu, H. Tembine, and T. Başar, "Network Security Configurations: A Nonzero-Sum Stochastic Game Approach," in *Proceedings of 2010 American Control Conference (ACC)*, pp. 1059–1064, June 2010.

[204] T. Alpcan and T. Başar, "An Intrusion Detection Game with Limited Observations," in *Proceedings of the 12th International Symposium on Dynamic Games and Applications*, (Sophia Antipolis, France), July 2006.

[205] F. Li, Y. Yang, and J. Wu, "Attack and Flee: Game-theory-based Analysis on Interactions Among Nodes in MANETs," *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, vol. 40, pp. 612–622, June 2010.

[206] A. Patcha and J. Park, "A Game Theoretic Formulation for Intrusion Detection in Mobile Ad Hoc Networks," *International Journal of Network Security*, vol. 2, pp. 131–137, March 2006.

[207] Y. Liu, C. Comaniciu, and H. Man, "A Bayesian Game approach for Intrusion Detection in Wireless Ad Hoc Networks," in *Proceeding from the 2006 workshop on Game theory for communications and networks (GameNets'06)*, (New York, NY, USA), ACM, 2006.

[208] T. Alpcan and T. Başar, "A Game Theoretic Analysis of Intrusion Detection in Access Control Systems," in *Proceedings of the 43rd IEEE Conference on Decision and Control*, (Paradise Island, Bahamas), December 2004.

[209] M. Khouzani, S. Sarkar, and E. Altman, "A Dynamic Game Solution to Malware Attack," in *Proceedings of the 30th IEEE International Conference on Computer Communications (IEEE INFOCOM 2011)*, pp. 2138–2146, April 2011.

[210] Symantec Security Response, "W32.Duqu: The precursor to the next Stuxnet." `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf`, November 2011. Last accessed: 5-July-2014.

[211] S. Russell and P. Norvig, eds., *Artificial Intelligence: A Modern Approach*. Prentice Hall, second ed., 2003.

[212] S. Smith and E. Spafford, "Grand challenges in information security: process and output," *IEEE Security & Privacy*, vol. 2, pp. 69–71, Jan 2004.