國立臺灣大學管理學院資訊管理系

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

以三值樹狀自動機為基礎之惡意程式分析

Malware Analysis with 3-Valued Deterministic Finite Tree Automata

王奕翔

Yi-Hsiang Wang

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 100 年 11 月

November, 2011

以三值樹狀自動機為基礎之惡意程式分析

Malware Analysis with 3-Valued Deterministic Finite Tree Automata

本論文係提交國立臺灣大學

資訊管理研究所作為完成碩士學位

所需條件之一部分

研究生：王奕翔 撰

中華民國一百年十一月

謝辭

二年多的研究生涯，七百多個日子，如今終於要劃下了句點。這段日子裡，許多不同的經驗打開了我的視野也寬廣了我的見聞。除了感謝老天，我還有許多值得我感謝的人事物。

首先要感謝的當然是我的指導教授蔡益坤老師。從老師的身上，我學到了持續累積的重要，也見識到老師對於學問的堅持與追求。此外，還要感謝實驗室的所有人。明憲學長的強大，我想就不用多言了。儘管已經結婚了，還能同時顧及研究以及家庭，小弟自愧不如。晉碩學長與怡文學姐，感謝你們幫助我了解 GOAL 實作的演算法，祝你們白頭偕老。智斌學長，開發網站時我學到了很多東西，是個有趣的經驗，希望日後你能夠順利的創業。睿元學長，你總是熱心的關心學弟妹，感謝你提供的考古題。昇峰學長，當初報告時很常麻煩你，祝你之後工作順利。任峰，雖然你先畢業了，不過還是要感謝你的幫忙還有你提供的沙發。辰旻，身為台大資管旻，就算你不常來實驗室也是可以諒解的，祝你之後都能開開心心的。啟祥以及靖婕，修課分組時辛苦你們了，雖然你們現在感覺很辛苦，不過你們一定會準時畢業的。瑞舜與暐獻，可惜後來就沒再跟你們玩桌遊了，祝你們也順利畢業。

另外，要感謝我的家人，來台北讀書後已很少回台中，感謝你們一直關心我。此外，也要感謝孟璘，這二年多裡你幫助我應付許多困難，給了我很大的支持。

最後，要感謝這二年來所有碰到的不如意，這些經驗讓我能夠成長，面對壓力，面對現實。這二年來所學到的東西，我一定會學以致用，也祝福大家都能夠開心順利。

王奕翔　謹識
于台灣大學資訊管理研究所
民國一百年十一月

論文摘要

學生：王奕翔　　2011 年 11 月

指導教授：蔡益坤

以三值樹狀自動機為基礎之惡意程式分析

網路上存在許多不同的資安威脅，其中最惡名昭彰的就是惡意程式。惡意程式指的是那些帶有惡意企圖並且有惡意行為的程式。典型的惡意程式包含了病毒、蠕蟲、木馬與間諜軟體。惡意程式偵測軟體可降低我們被惡意程式攻擊的風險。不同的偵測軟體有其各別的偵測方法，但在現今的偵測軟體中最基本也最普遍被應用的方法是靜態特徵碼比對。然而這種偵測方法已經普遍的被認為無法對抗現今更為進階的惡意程式。進階的惡意程式使用程式混淆的手法改變程式本身的結構，也因此能夠非常輕易的避過偵查軟體。幸運的是，程式本身的語意在經過混淆後通常仍會保持一致，因此一個可行的對策就是設計一個以程式語意為基礎的偵測方法。

在這篇論文裡，我們提出一個以程式語意為基礎的惡意程式偵測方法。觀察近年來被提出的各種辦法，我們發覺字串仍然被廣泛的使用為一種特徵碼的形式。我們可以將字串擴充為樹，樹不但更為一般化而且帶有更多的語意資訊。因此，我們使用樹做為我們偵測軟體的特徵碼形式。我們的偵測軟體需要一組惡意程式跟一組正常程式做為輸入。這些程式的語意會以系統呼叫的資料相依圖表示。接著，我們將這些相依圖解析為樹。使用文法推論的方法，我們最後可以得到一個三值的樹狀自動機。一個三值的樹狀自動機有三個互斥的最終狀態：接受、拒絕、與未知。如果我們使用三值樹狀自動機做為我們的惡意程式偵測軟體，根據輸入的程式他會有三種可能的輸出值。如果輸入的程式是一個惡意程式，偵測軟體會輸出是。如果輸入的程式是一個正常程式，偵測軟體會輸出否。其餘的狀況下他會輸出未知。根據我們的實驗結果，我們的偵測軟體有著相當低的誤報。然而，相對的代價是許多程式經過軟體檢查後的結果是未知。

關鍵字：惡意程式分析、惡意程式偵測軟體、文法推論、三值自動機、程式語意、系統呼叫

THESIS ABSTRACT

Graduate Institute of Information Management
National Taiwan University

Student: Wang, Yi-Hsiang Month/Year: November, 2011
Advisor: Tsay, Yih-Kuen

Malware Analysis with 3-Valued Deterministic Finite Tree Automata

There exist many security threats on the Internet, and the most notorious is malware. Malware (malicious software) refers to programs that have malicious intention and perform some harmful actions. Typical malware includes viruses, worms, trojan horses, and spyware. The first line of defense to deter malware is malware detector. Each malware detector has its own analysis method. The most basic and prevalent methods used in commercial malware detectors are based on syntactic signature matching. It is widely recognized that this detection mechanism cannot cope with advanced malware. Advanced malware uses program obfuscation to alter program structures and therefore can evade the detection easily. However, the semantics of a malware instance is usually preserved after obfuscation. So, it is feasible to develop a malware detector that is based on program semantics.

In this thesis, we propose a semantics-based approach to malware analysis. Observing recently proposed methods for malware detection, we notice that string-based signatures are still used widely. It is natural to extend from string to tree, which is more general and can carry more semantics. Therefore, we use trees as signatures. Our malware detector requires a set of malware instances and a set of benign programs. The semantics of each input program is extracted and represented as a system call dependence graph. The graph is then transformed into a tree. With the set of trees generated from malware and benign programs, we use the method of grammatical inference to learn a 3-valued deterministic finite tree automaton (3DFT). A 3DFT has three different final states: accept, reject, and unknown. If we take this 3DFT as the malware detector, it outputs three different values. If an input program is a malware instance, the detector outputs true. If an input program is a benign program, the detector outputs false. Otherwise, it outputs unknown. According to our experiments, our detector exhibits very low false positives. However, there is a tradeoff that many programs are identified as unknown.

Keywords: Malware Analysis, Malware Detector, Grammatical Inference, 3-Valued Automata, Program Semantics, System Call

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Nowadays, many different services are provided on the Internet. As more trusts have been put on the Internet, more attentions have been paid to Internet security. There exist many security threats on the Internet, and the most notorious is malware. Generally speaking, **malware** (malicious software) refers to programs that have malicious intention and perform some harmful actions. Typical malware includes viruses, worms, trojan horses, bots, and spyware.

As underground economy flourishes, malware has become a profitable tool. Malware can be used to launch zero-day attacks, inject bots, send fishing web-sites, and crash systems. What worse, malware is used widely to steal private information. As observed by Symantec in 2010 [21], an underground economy advertised $0.07 to $100 for each stolen credit card number. Besides, they also observed that 10,000 bots are promoting with $15. Bots are widely used for spam or distributed denial of service attacks (DDoS).

It is reasonable that the amount of malware samples have increased every year. As showed in Figure 1.1, the number of new malware samples that AV-TEST collected [1] is increased with amazing speed. In 2009, the amount of new malware samples is about 12 millions, however, in 2010, it is increased to 17 millions.

We mostly rely on malware detector to cope with malware. A malware detector ac-

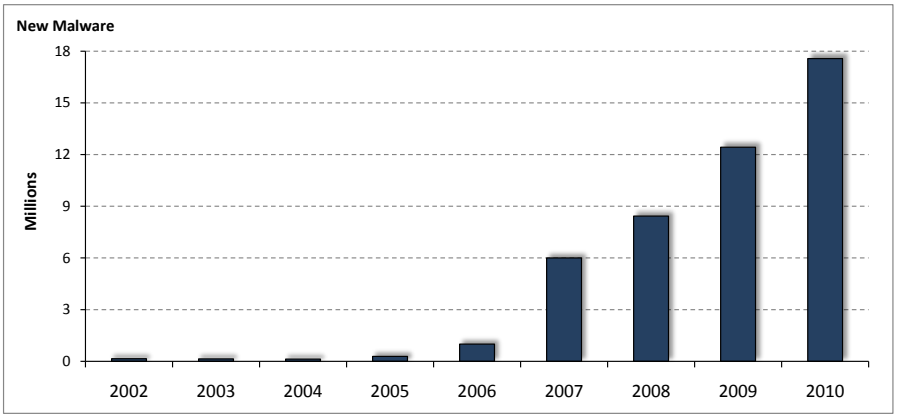


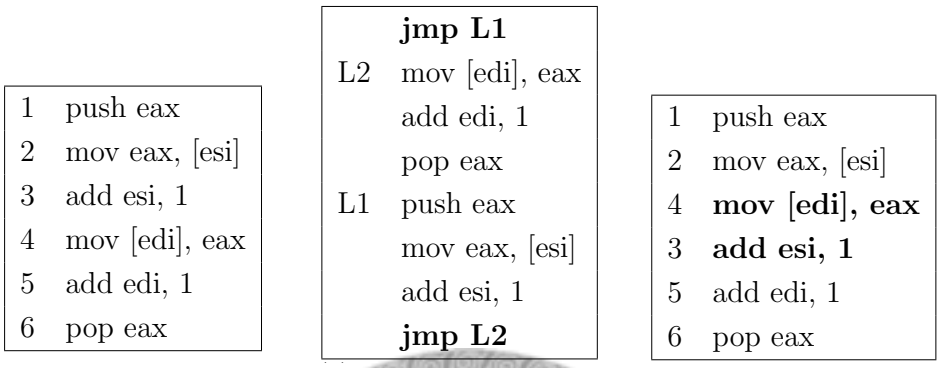
Figure 1.1: The amount of new malware samples collected by AV-TEST

cepts a suspicious program as input, and determine whether it is a malware instance or a benign program. There are many detection approaches, but the most widely used is *syntactic signature matching*.

A *signature* refers to a pattern that is only present in a particular malware instance or malware family. If a program matches a pattern (signature), that program is regarded as a malware instance. *Strings* are the most commonly used format for signatures. For example, a sequence of machine instructions, a sequence of magic numbers, and an regular expressions can be used as signatures.

Syntactic signature matching has the advantages that detection is efficient and is easy to implement. However, syntactic signature can't cope with advanced malware which uses **program obfuscation**, **encryption**, and **packing**. Program obfuscation changes the syntactic structure of a program while preserving its original semantics. Encryption changes the original entry point (OEP) of a program. Packing changes the format of a program using lots of different skills. These techniques were initially designed to prevent reverse engineering. A corporation use these skills to prevent their rivals imitate their products. Unfortunately, malware writers also use these techniques to protect malware. The most common program obfuscation techniques includes garbage codes insertion, equivalent instruction substitution, and instruction reordering.

For example, Figure 1.2(a) is the original program. Figure 1.2(b) inserts jmp instructions and Figure 1.2(c) reorders instruction 3 and instruction 4. However, the semantics for these three programs are identical.



(a) Original program (b) Instruction reordering 1 (c) Instruction reordering 2

Figure 1.2: Program obfuscation

Therefore, rather than using syntactic signatures, it is more effective to use **semantics-based** signatures. Malware writers usually change the syntactic structures to create new malware variants. However, these variants usually preserve the original semantics. If a signature is semantics-based, it should be able to detect several different malware variants as long as they have same program semantics.

1.2 Motivation and Objective

An ideal malware detector can identify malicious programs correctly. However, it is a **co-evolution** between the malware writers and the malware detectors. Every time a new malware sample is designed, malware detectors are improved to detect this malware sample. Whenever this malware sample can be detected, more advanced malware will be proposed to evade detection. A practical malware detector inevitably has false positives and false negatives.

We would like to design a malware detector that identifies programs as malware, benign program, or unknown. Only programs are identified as unknown need a double-check. For the other two cases, programs are recognized as malware instances or benign programs with great confidence. We design a malware detector like so because it is always need a double check in reality. In a highly security-sensitive environment, more than one malware detectors are used to cope with malware. The results from the former detectors may need to re-precess in the latter detectors. If we can explicitly separate programs into three groups: malware, benign, and unknown; only programs identified as unknown need to re-process. Therefore, it is more efficient.

As showed in Figure 1.3, the outermost rectangle represents all possible programs. Each circle is a benign program and each triangle is a malware instance. For the circles in the left rectangle, they are correctly identified as benign programs. For the triangles in the right rectangle, they are correctly identified as malware instances. The remain circles and triangles in the middle rectangle are identified as unknown and need a double check.

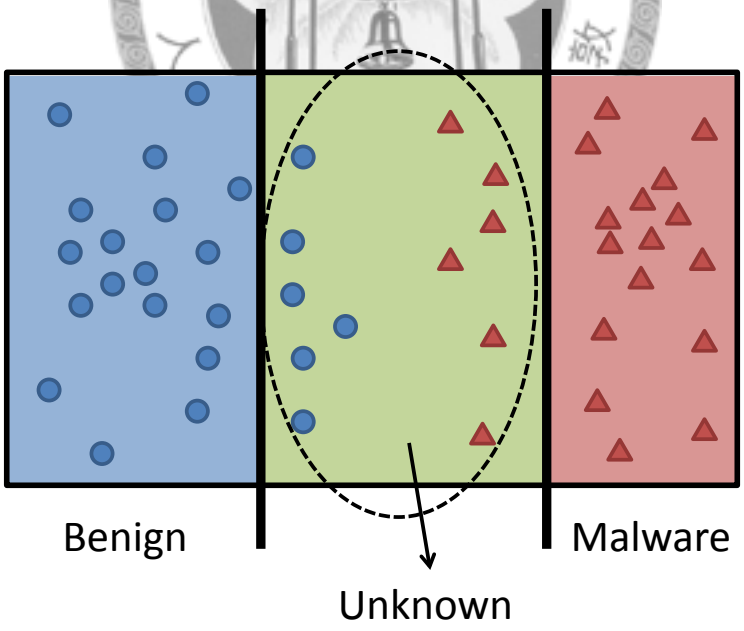


Figure 1.3: A malware detector identifies programs as malware, benign, or unknown

Besides, we would like to use **trees** as semantics-based signatures. As mentioned before, strings has been used for syntactic signature matching widely. We can still select strings as semantics-based signatures, but strings as signatures are too specific and easy to evade. Strings can represent sequential relation and the basic dependence relations. However, trees can represent more subtle dependence relations, such as shared dependence.

Some proposed detection approaches require a priori knowledge about malicious behaviors. They specify a set of behaviors as malicious or design benign policies and regard programs as malware if any behaviors violate the policies. It is inefficient and error-prone. We would like to learn the malicious behaviors using **grammatical inference**. Given a set of malware samples and benign programs, we use grammatical inference to learn signatures. Compared with training from the given knowledge, our signatures can reflect more implicit knowledge.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

- In Chapter 2, we introduce several related literatures.
- In Chapter 3, we give some preliminaries about this thesis, includes finite ordered trees and finite tree automata.
- In Chapter 4, we describe our detection approach.
- In Chapter 5, we describe our implementation and show the results of our experiments.
- In Chapter 6, we summarize our contributions and indicate some possible research direction in the future.

Chapter 2

Related Work

In this chapter, we describe some related works about our research. At first, we give the introduction about grammatical inference, and we specifically focused on tree automata inference. Later, related works about malware analysis with executed system calls will be introduced. And we will also review two related works that combine malware analysis with tree automata.

2.1 Grammatical Inference

Grammatical inference is concerned with learning language representations from given information, which can be text, examples and counter-examples, or anything that can provide us insight about the elements of the target language[13]. The learner sometimes is called an inference machine, or a learning algorithm. The research on grammatical inference cross a number of related fields, includes artificial intelligence, machine learning, formal language theory, pattern recognition, computational linguistics, and speech recognition. This field has a variety of learning models, they all have different and discriminative environment setting, from the target of learning, the language representation, the available information or the information presentation. However, there are three major established formal models:

- Gold's identification in the limit,
- Angluin's active learning model,
- Valiant's probably approximately correct(PAC) model.

As that Angluin's learning model has a tight link with other work, we will give a brief introduction. The introduction about other two learning models can be referenced in [12],[20], and [13].

In Angluin's active learning model (also called query learning meodel), there is a teacher which can answer specific kind of queries about the unknown grammar, and the learner learns the target language by asking teacher this queries. Angluin has described several types of queries in [5]. In [4], Angluin proposed the L^* learning algorithm that can learn a regular language from minimally adequate teacher. A minimally adequate teacher (MAT) is assumed to answer correctly two types of queries from the learner about the target language.

- Membership query: given a string t, returns "yes" if t is the member of the target language, returns "no" otherwise.
- Equivalence query: given a hypothesis grammar, if the hypothesis is equivalent to the target, returns "yes", returns "no" otherwise. In the case of returning "no", the counter-example will also be returned. The counter-example is the symmetric difference of the input grammar and the target grammar.

The learner use the membership queries to infer grammar structure. Every time the learner inferred a structure, it asks teacher equivalence queries to check equivalence. If there is a counter-example returned, modifies the inferred structure with the help of membership queries and asks equivalence queries again. The process is repeatedly continuing until the inferred structure is somehow identical to the target. The details about L^* learning algorithm is omitted here, as will be describe in latter section. The interactions between the learn and teacher is depicted in Figure 2.1.

2.1.1 Tree Automata Inference

As mentioned in section 1.2, we would like to infer a tree automata from the example. Here, we give an brief introduction about the related works. The main focus of research in grammatical inference has been placed on learning regular grammars or deterministic

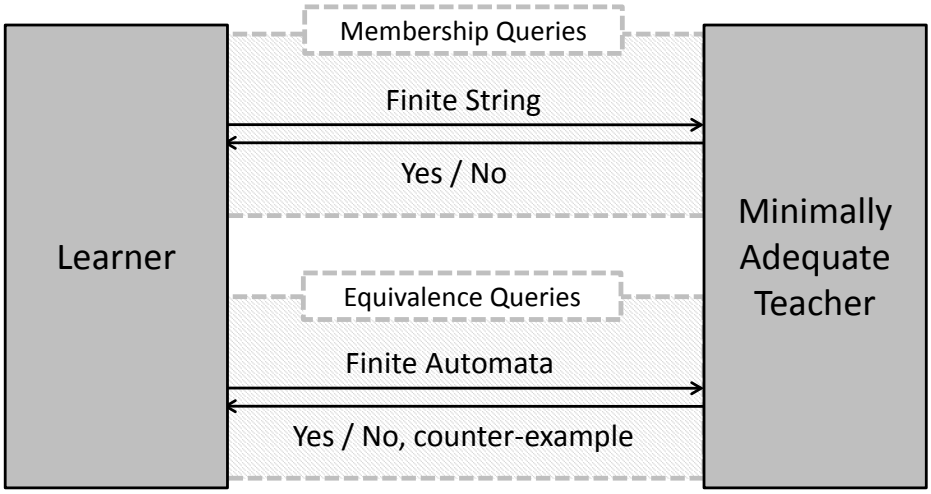


Figure 2.1: The interactions between learner and teacher in Angluin's L^* algorithm

finite automata (DFA). The reason for so is that this problem seems simple enough as it is less general than context-free grammar. For tree automata, it can be seen as the direct extension of finite automata that the input is finite trees instead of strings. Therefore, it is straightforward to extend finite automata learning algorithm inference to tree automata inference.

Brayer and Fu [9] proposed an tree automata inference algorithm that extend the original k-tail inference method. Similar to [9], Fukuda and Kamata [16] use the k-follower to infer the tree automaton. Both these algorithms infer the tree automaton from a given sample set. García and Oncina [17] extend the RPNI (Regular Positive and Negative Inference) algorithm to tree automata inference, obviously, the learning is completed with given positive examples and negative examples. The idea is that they build the subtree automaton at first (recognize each subtree of positive sample), and merge the automaton state while not accept the negative sample. The algorithm works in polynomial time with the size of the input data.

Drewes [14] extend Angluin's L^* learning algorithm to tree automaton inference. Just

like Angluin's algorithm, they retain an observation table during the learning process and use this table to construct the automaton. To extend observation table, they ask teacher membership queries and equivalence queries. In [14], they proposed two different tree automata learning algorithm, L_*^{tfta} and L_{fta}^*. The discriminative difference between this two algorithms is that what L_*^{tfta} learned is a total finite tree automata, however, the automata L_{fta}^* learned is a partial finite tree automata, so has less transitions and states. The idea behind the partial finite tree automata is that they removed the dead state from the automaton. The term dead state means that the corresponding equivalence class of states can not be any subtree of the target language. More detail about this learning algorithm will be discussed in chapter 4.

Another algorithm that also extends Angluin's active learning model is [7]. In [7], Besombes and Marion proposed the learning algorithm that learn the automata from positive examples and membership queries. Without the help of equivalence queries, the learner is initially provided with the representative samples for compensate. Informally, an representative sample is the set of languages such that each transition of the target automaton will be used to process the samples. By checking the consistent of the observation table, they can get the new differentiating context.

2.2 Malware Analysis with Executed System Calls

As mentioned in section 1.1, the new generation malware detectors focused on the behavior of the program rather than the syntactic structure. There exists many kinds of behavior models, one of the prevalence models is based on the executed system calls of program. We know that in order to access the system resources, the user-level program has to invoke system call. By tracking the executed system calls of malware, we can realize the purpose of the program and the resources being request. However, if only using the sequence of system calls as malware signature, it is easy to evade by system call reordering skills. Some more informative representation that based on system calls have been proposed, we give a brief introduction as follows.

2.2.1 Effective and Efficient Malware Detection at the End Host

In [18], Kolbitsch et al. proposed the malware detection approach that is efficient to implement at the end host. To model the behavior of the program, they rely on the executed system calls. As mentioned above, it is unwise to represent the program behavior as system call sequences. Instead, they use the dependence graph of system calls, also called *behavior graph* in the paper.

The *behavior graph* $G = (V, E, F, \delta)$, where

- V is the set of vertices, each represents a system call $s \in \Sigma$
- E is the set of edges, $E \subseteq V \times V$
- F is the set of functions $\bigcup f : x_1, x_2, ..., x_n \rightarrow y$, where each x_i is an output argument of system call, and y is the input argument of system call
- δ, which assigns a function f_i to each system call input argument a_i

For example, we list partial behavior graph for malware Netsky in Figure 2.2.

To detect the malware at the end host, the suspicious program must be dynamically executed. The invoked system calls will be used to match the graph vertices from the

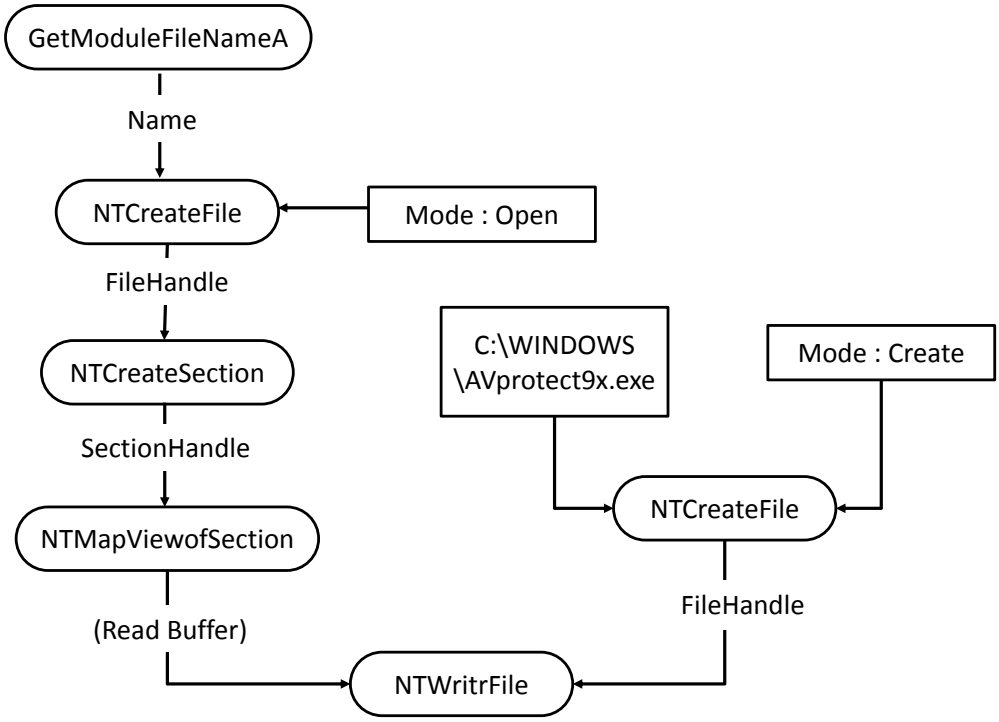


Figure 2.2: Partial behavior graph for malware Netsky, redraw from [18]

collected malware behavior graphs. However, the dependence between the graph vertices needs to preserved. And it generates significant overhead to dynamically taint analysis at the end host. Therefore, they use the program slicing skills. For the data flows between x and y in the malware, the instructions that are responsible for reading the input and transforming it into output is extracted. This program slice can be used to derive the symbolic expression that represents the semantics of the slice. Using this symbolic expression, when the system call x has been invoked at the end host, the expected output can be pre-computed. Later, when the system call y has been invoked, checks whether the value of the arguments is identical to the expected output. If the value is equal, the dependence data flow has been detected.

To test the detection effectiveness and false positives, they set up the experiments with six common malware family. By randomly selecting 100 samples for each malware family and extract the behavior graph with 50 samples, the detection rate of their detector is about 93% for the remain test samples. Despite the experiment is only focused on the six

known malware family, the detection rate is still impressive. For false positives, they test five common benign applications and report no false positives. Although this result is promising, but the exact amounts of test samples is not clear from the experiments, and we cannot comment how well their detector is in goodware misclassified. But as their method shows, the dependence graph of system calls has truly a tight relation with the program semantic.

2.2.2 A Layered Architecture for Detecting Malicious Behaviors

Martignoni et al. [19] proposed a layered architecture for detecting malicious behaviors. However, the focus of this method is on botnet detection, and it is reasonable to extend the method to malware detection. To capture the behavior of the malicious program, the basis of their method is the executed system calls. But the system calls are used in a smarter way. For each malicious behavior, they use a layered representation. Each behavior is composed of events, and events is generated from observed system calls. For example, Figure 2.3 shows an example of the hierarchy of events used to specify the high-level behavior: **downloading and executing a program**.

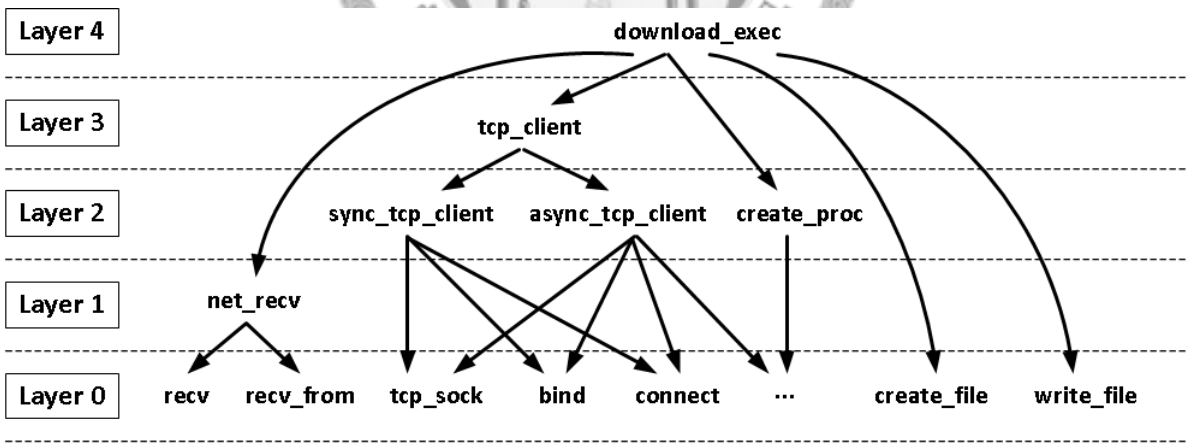


Figure 2.3: Layered behavior specification, redraw from [19]

High-level behaviors are decomposed into multiple layers. Events are represented as bold strings, the edges related the events indicate the dependence relation. The lowest layer, Layer 0, is the invoked system calls. The events in Layer 1 aggregate Layer 0 events

that have a common side effect. In Figure 2.3, event **net_recv** is generated whenever any of the Layer 0 events **recv** or **recvfrom** occur. Events at Layer 2 and upper layer identify correlated sequences of lower-layer events that have some aggregate, composite effect. In Figure 2.3, event **sync_tcp_client** identifies a synchronous TCP socket has been created, bound, and connected upon. As we can see, the events at upper layers have a more rich semantics. By decomposing the high-level behavior into multiple layers, the behavior specifications are configurable, less error-prone and easy to update.

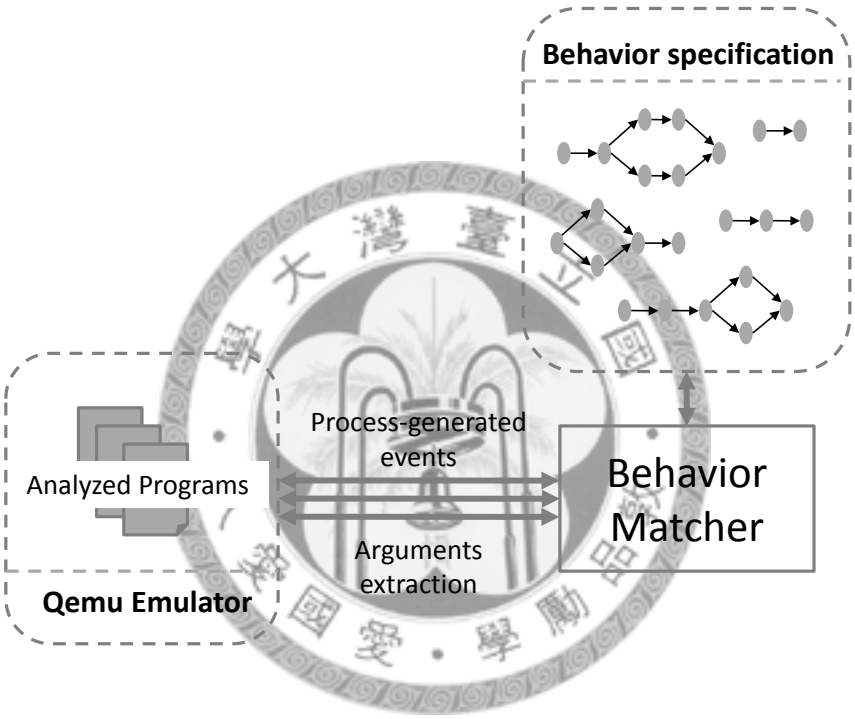


Figure 2.4: The architecture of the detector, redraw from [19]

The overall architecture of the detector is presented in Figure 2.4, it includes *analysis environment* and a set of *behavior specifications* and a *behavior matcher*. It works as follows:

1. The suspicious program is dynamically executed in the emulator, and each invoked system call will send to *behavior matcher*.

2. Every time the matcher receives an invoked system call, it attempts to match this with the node of the behavior specification.

For each behavior specification, it is composed as the layered architecture in Figure 2.3. It is a graph structure, each node can be the system call or an event. There is a single output node in each graph. If overall nodes except the output node of the graph have been matched, then the output event will be generated. This event can also used to compose other behavior specification.

3. If the high-level behavior specification has been matched, then the malicious behavior has been observed.

In their implementation, the sets of behavior specification are manually extracted with domain knowledge and analysis of tens of gigabytes of executions traces. Although it is inefficient to construct the behavior specifications, to modify or update the specification is simple. And the extracted behavior specifications also provide the detector a good behavioral signature.

2.3 Malware Analysis with Tree Automata

Below, we review the works that implement malware detection with the help of tree automata. In addition to give a brief introduction of the methods, we also comment their advantages and drawbacks.

2.3.1 Architecture of a Morphological Malware Detector

In [8], Bonfante et al. proposed a malware detector that makes use of tree automata. To capture program semantics, the detector relies on the extracted control flow graph (CFG). The input program is transformed to the abstracted language first, and the corresponding CFG is extracted. The vertices of the extracted CFG includes **inst**, sequential instructions; **jmp**, uncondition jumps; **jcc**, conditional jumps; **call**, function calls; and **end**, function returns or undefined instructions. Besides, in order to deal with the classical obfuscation skills, they also design an rewriting engine that reduces the extracted CFG. After this steps, the reduced CFG of a malware can be regarded as a signature. To build the signature database, collect the set of CFG of malware and transform its into tree representation. Using the finite set of trees, a minimal tree automata which recognizes these trees can be build. And this automata can be used to detect malware infection.

The overall architecture of the detector is depicted in Figure 2.5. For the malware samples, extract their control flow graph and reduced with rewriting engine. The reduced graph is then transformed into tree. The set of trees is then used to construct the tree automata, and after minimizing the tree automata, it will be used as the malware detector. For malware detection, the suspicious program is proceed with graph extraction, graph rewriting, and transform from graph to tree. The final output tree is used for malware detection. If the output tree of the program is accepted by the tree automaton, this program is identified as malware, otherwise, regard this program as benign program.

The detector Bonfante et al. proposed can be constructed automatically. Besides, the properties of tree automata give their tool an efficient performance to detect malware.

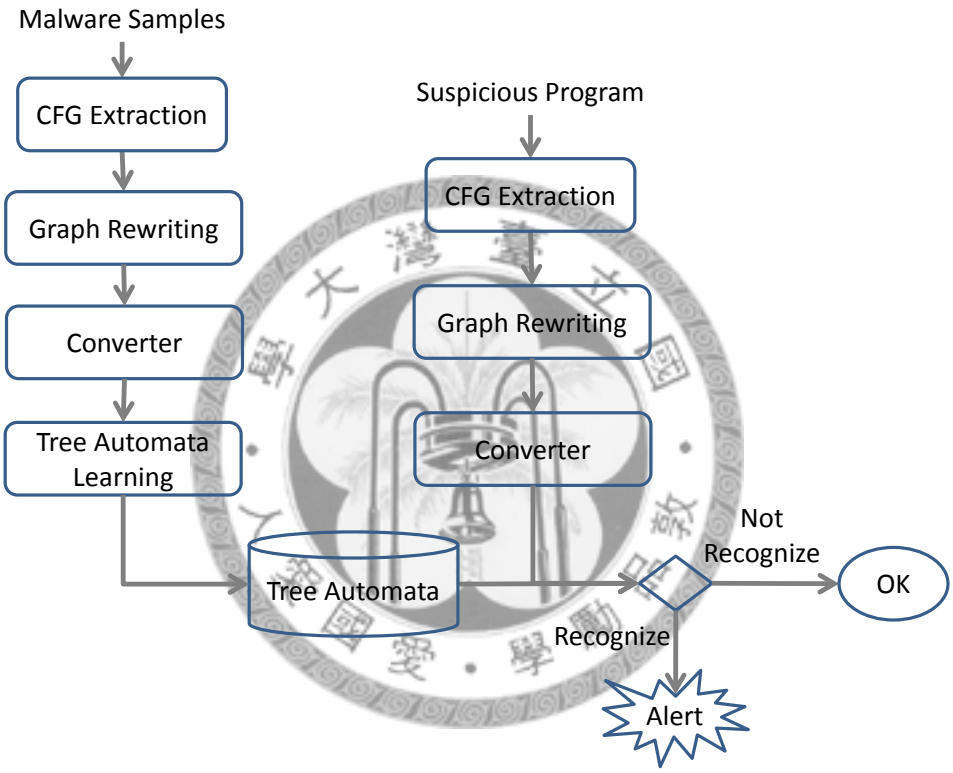


Figure 2.5: The architecture of a morphological malware detector, redraw from [8]

However, that detection can only handles the basic malware such has same CFG as the malware signature. To detect a more general malware infection, the detection have to based on CFG subgraph isomorphism, and it increases the complexity of detection. When there is a demand to add new malware signature, the work can be done by computing the union of automata, and can be computed in linear time. However, the method of automata construction do not mentioned in [8]. So we cannot realized the capability of their tree automata and the effects of automata union. And the experiments they make only mentioned the false positives but neglect the false negatives. Although the false positives(0.09% when lower bound of CFG size is 15) of their detector is really amazing, it's still not clear how effective and efficient of their malware detector. But the conclusion can be made from their experiments that it is worthwhile to combine tree automata with malware analysis.

2.3.2 Malware Analysis with Tree Automata Inference

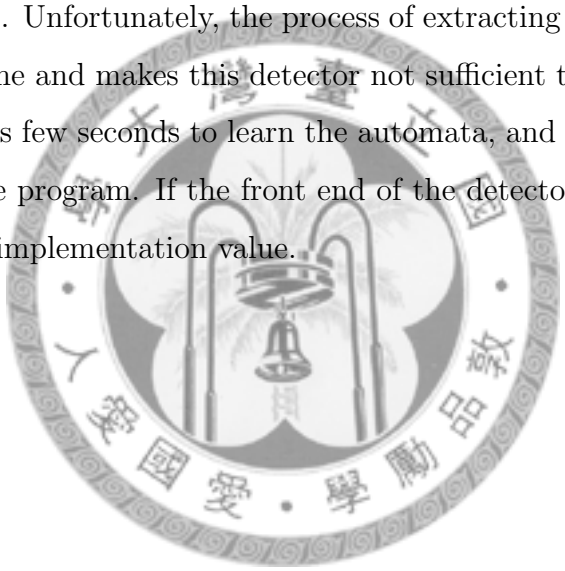
Similar to [8], Babic et al. [6] proposed the malware analysis method with tree automata inference. However, the major contribution of their work is the automata inference algorithm.

For each malware, they use dynamic taint analysis to construct the data flow dependence graph of system calls. As mention in section 2.2, the executed system call reflects the accessed system resources of malware. And it has been used in the community for a while. Besides, by focusing on the data flow dependence, the graph can resist some common obfuscation skills. After build the graphs, in order to avoid the exponential blowup when expanding into trees, they learn the tree automata directly from the graph.

In [6], they only use the set of positive samples (malware) to learn the automata. Instead of learning the automata from regular tree languages, they learned from the subclass of regular tree languages, the k-testable tree languages. These languages are defined as a finite set of k-level-deep tree patterns. And this kinds of regular languages can be identified from positive samples only. The value k is a tunable factor, it influences the

level of generalization. The smaller the k factor, the more abstract the inferred automaton. Therefore, by adjusting factor k, the detector can get balance between false positives and false negatives. There already exists some proposed methods for k-testable tree automata inference. But the algorithm Babic et al. designed has better performance, with complexity $O(kN)$, where N is the size of the graphs.

As their experiments showed, when $k = 4$, their detector has 20% false negatives and 5% false positives. And increasing the value of k above 4 does not make a significant improvement of detection rates. Therefore, they determined that $k = 4$ is the optimal abstraction level. Besides, they also test the classification ability of their detector. The experiment results can only support that the tool has some kind of classification ability but the noise still exists. Unfortunately, the process of extracting system call dependence graph spends lots of time and makes this detector not sufficient to implement at ad hoc. However, it merely costs few seconds to learn the automata, and takes less than the timing jitter to analyze the program. If the front end of the detector can be improved, this detector will has more implementation value.



Chapter 3

Preliminaries

In this chapter, we give a formal definition about finite ordered trees and finite tree automata.

3.1 Finite Ordered Trees

First, we define the term *ranked alphabet*. A *ranked alphabet* Σ is a finite set of symbols together with a function *rank*: $\Sigma \to \mathbb{N} \cup 0$. For each symbol $a \in \Sigma$, $rank(a)$ is the rank (arity) of a. We denote by Σ_n, $n \geq 0$, the set of all symbols a with $rank(a) = n$. A *finite ordered tree* (abbreviated as tree in the following) t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions:

- $dom(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$,
- for each $p \in dom(t)$, if $t(p) \in \Sigma_n$, then $\{\ i \mid pi \in dom(t)\ \} = \{1, ..., n\}$

We call each sequence $p \in dom(t)$ a *node* of t, and use ϵ to denote an empty sequence. A *root* node of the tree t is the node which $dom(t) = \epsilon$. A *frontier* node of the tree t is the node p such that $\forall j \in \mathbb{N}$, $pj \notin dom(t)$. We denote by $T(\Sigma)$ the set of all such trees over the ranked alphabet Σ.

For example, in Figure 3.1, we depict a tree t in the left side and its domain in the right side. For the tree t, $\Sigma_0 = \{a, b\}$, $\Sigma_2 = \{f, g\}$, and $\Sigma_3 = \{h\}$. A root node of the tree t is the node ϵ. And the frontier nodes of the tree t is the set of nodes $11, 12, 21, 221, 222, 231, 232$.

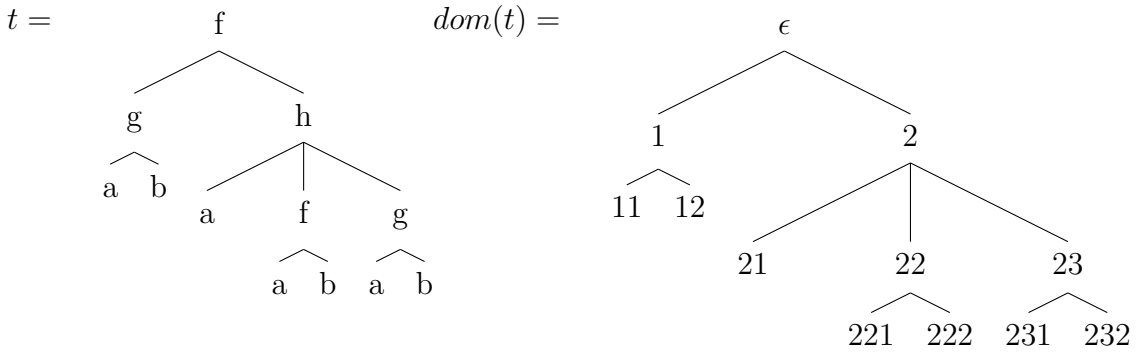


Figure 3.1: A finite ordered tree t and its domain.

3.2 Finite Tree Automata

A finite tree automaton (FTA) is a tuple $\mathcal{A} = (\Sigma, Q, \Delta, F)$ where:

- Q is a finite set of states,
- $F \subseteq Q$ is a finite set of final states,
- Σ is a ranked alphabet,
- Δ is a set of transition rules with following formats:
  $f(q_1, q_2, ..., q_n) \to q$, where $q_1, ..., q_n, q \in Q$, $f \in \Sigma_n$

A finite tree automaton is *deterministic* if there do not exists two transition rules with the same left-hand side, which is denoted by **DFT**. Given a tree t, we use the transition rules of FTA to traverse the tree t from the bottom to top. We can notice that a FTA does not have an initial state, but the transition rule is as the form $f() \to q$ when $f \in \Sigma_0$, and this kind of rules can be regarded as the initial rules.

We give a more formal definition, a run π of a finite tree automaton $\mathcal{A}$ over a tree t is a mapping from $dom(t)$ to Q. It starts from the leaves rules, $a(\ ) \to q$, where $a \in \Sigma_0$. A run π assigns every node $p \in dom(t)$ a state with following rules: if $f(q_1, ..., q_n) \to q \in \Delta$ and $t(p) = f$, $\pi(pi) = q_i$ for each $i \in \{1, ..., n\}$, then $\pi(p) = q$. We say a run π of $\mathcal{A}$ on tree t is successful if $\pi(\epsilon) \cap F \neq \emptyset$. A tree t is **accepted** by $\mathcal{A}$ if there is a successful run of $\mathcal{A}$ on t. A tree language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of trees accepted by $\mathcal{A}$. Two

FTAs are said to be **equivalent** if they recognize the same tree language.

Let us look at an example. Let $\Sigma = \{0, 1, not, and, or\}$, with $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{not\}$, and $\Sigma_2 = \{and, or\}$. Consider a DFT $\mathcal{A} = (\Sigma, Q, \Delta, F)$ where $Q = \{q_0, q_1\}$, $F = \{q_1\}$, and Δ contains the following rules.

$$\begin{array}{rclrcl}
0 & \to & q_0 & 1 & \to & q_1 \\
not(q_0) & \to & q_1 & not(q_1) & \to & q_0 \\
and(q_0, q_0) & \to & q_0 & and(q_0, q_1) & \to & q_0 \\
and(q_1, q_0) & \to & q_0 & and(q_1, q_1) & \to & q_1 \\
or(q_0, q_0) & \to & q_0 & or(q_0, q_1) & \to & q_1 \\
or(q_1, q_0) & \to & q_1 & or(q_1, q_1) & \to & q_1
\end{array}$$

For a given tree t in Figure 3.2 , the run π of $\mathcal{A}$ over tree t is also presented in Figure 3.2. Because this automaton is a deterministic finite tree automaton, it only has one possible run on t . Since $\pi(\epsilon) \cap F \neq \emptyset$, π is not a successful run. Therefore, this tree t is not accepted by $\mathcal{A}$.

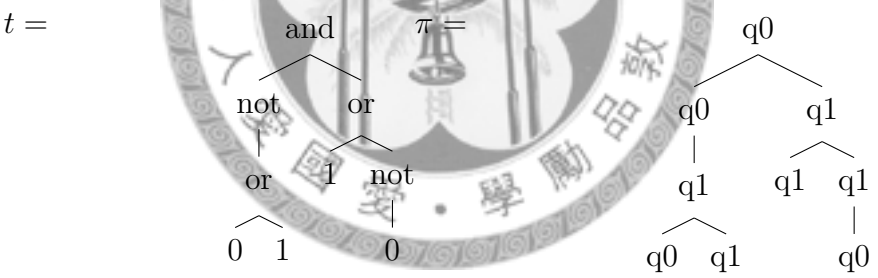


Figure 3.2: A finite ordered tree t and a run π of $\mathcal{A}$ over tree t.

Chapter 4

Approach

In this chapter, we present our malware detection approach. As mentioned in Section 1.2, our detector uses semantics-based signatures. We assume that program semantics is captured by **system call data-flow dependence graphs**. Because we want to use trees as signatures, we unfold dependence graphs to trees. We use a **finite tree automaton** to represent correlated signatures and take this automaton as a malware detector.

Our tree automaton is a 3-valued tree automaton which has three disjoint final states: *accept*, *reject*, and *unknown*. If we take a 3-valued tree automaton as a malware detector and give it a suspicious program as an input, there are three possible outcomes:

- An accept implies that the input program is a malware instance.
- A reject implies that the input program is a benign program.
- An unknown means no conclusive answer.

Therefore, our detector works as in Figure 4.1. Every circles and triangles is an input program. For the circles inside the left circle, they are rejected by our tree automaton. For the triangles inside the right circle, they are accepted by our tree automaton. For the remaining circles and triangles, they are unknown for our tree automaton.

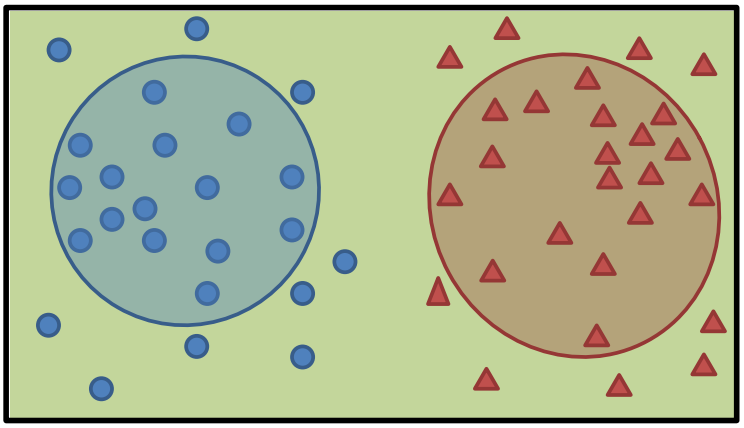


Figure 4.1: Our malware detector

4.1 Architecture

Our detector is trained with positive examples and negative examples. The positive examples are the set of malware instances and the negative examples are the set of benign programs. We use a system call dependence graph to represent each input program's semantics. A parser will then parse the graph to a tree. Using the tree automata learning algorithm, we can learn a 3-valued finite tree automaton. We take this automaton as our malware detector.

To test a suspicious program, follow the similar steps and use the 3-valued tree automaton to test the generated tree. The overall architecture is depicted in Figure 4.2. The dash line separates the architecture into detector construction and suspicious program testing.

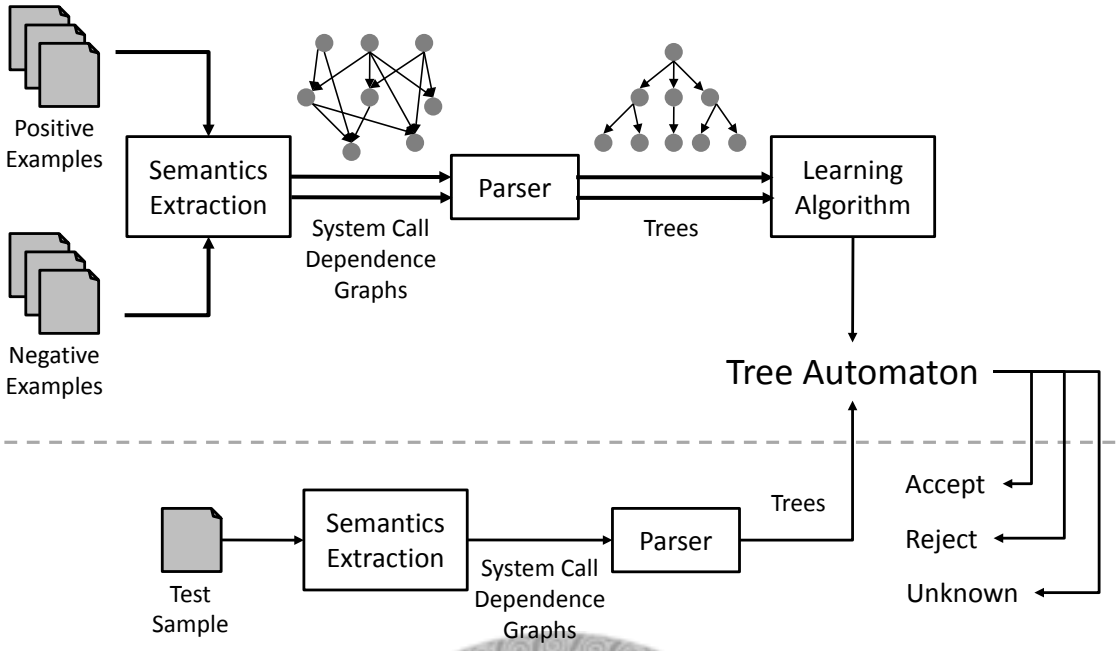


Figure 4.2: Architecture

4.2 3-Valued Deterministic Finite Tree Automata

We have mentioned that our tree automaton is a 3-valued tree automaton. Chen et al. [10] have proposed the notion of finite state automata with three disjoint final states. We extend their definition to finite tree automata. A **3-valued deterministic finite tree automaton** (3DFT) is a tuple $\mathcal{A} = (\Sigma, Q, \Delta, Accept, Reject, Unknown)$ where:

- Σ is a ranked alphabet,
- Q is a finite set of states,
- $Accept \subseteq Q$ is a finite set of states, disjoint with $Reject$ and $Unknown$,
- $Reject \subseteq Q$ is a finite set of states, disjoint with $Accept$ and $Unknown$,
- $Unknown = Q - (Accept \bigcup Reject)$,
- Δ is a set of transition rules of the following form:
  $f(q_1, q_2, ..., q_n) \to q$, where $q_1, ..., q_n, q \in Q$, $f \in \Sigma_n$

The definition of a run over 3DFT is similar to that in finite tree automata. We say that a tree t is accepted by a 3DFT $\mathcal{A}$ if there is a run π such that $\pi(\epsilon) \cap Accept \neq \emptyset$. A tree t is rejected by a 3DFT $\mathcal{A}$ if there is a run π such that $\pi(\epsilon) \cap Reject \neq \emptyset$. If a tree t is neither accepted by $\mathcal{A}$ nor rejected by $\mathcal{A}$, we say t is unknown for a 3DFT $\mathcal{A}$.

4.3 Semantics Extraction

As we know, programs have to invoke system calls to acquire system resources. We can discover the intension of a malware instance by observing executed system calls. Besides, the dependence relation between system calls is also important. It is easy to reorder system calls that have no dependence relations. However, reordering system calls that have dependence relations always change the program semantics. Therefore, if we only record the executed system calls but not the dependence relations, this kind of signature is easy to evade.

We represent a system call data-flow dependence graph as a **behavior graph**. A *behavior graph* is a directed acyclic graph $G = (V, E)$:

- V is a sets of vertices, where each vertex is a system call $s \in S$.
- A directed edge $< v_1, v_2 >$ from $v_1 \in V$ to $v_2 \in V$ represents a dependence relation that the input arguments of v_2 are somehow dependent on the outputs of v_1.

For example, Figure 4.3 is a behavior graph example for malware Allaple.b which is redrawn from [18]. The directed edge from NtOpenFile to NtCreateSection means that an input argument of NtCreateSection depends on the output of NtOpenFile. In this example, the dependence relation between NtOpenFile and NtCreateSection is the shared file handle.

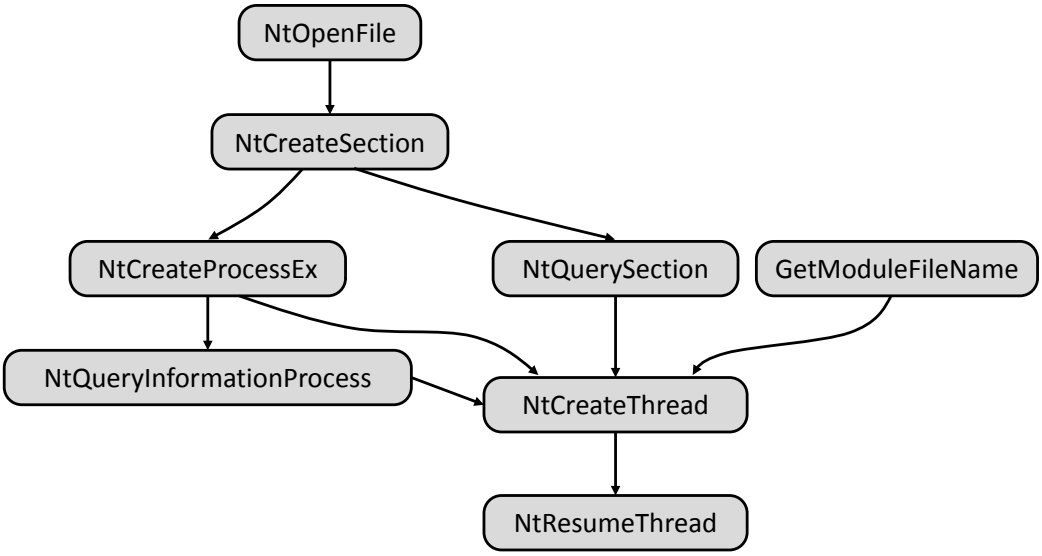


Figure 4.3: A behavior graph

4.4 Graph Parser

For a given behavior graph G, we extract all dependence relations and output a tree t^G. For each directed edge $< v_1, v_2 >$, we create a tree $t_{<v_1,v_2>}$ with root v_1 which has a child v_2. All of the trees generated with the above formalism from a behavior graph G form a set of trees T^G. For each $v_i(v_j) \in T^G$, merge with $v_i(v_k) \in T^G$, where $v_j \neq v_k$ and form a new tree $v_i(v_j, v_k)$.

We got two different trees that are generated from a single behavior graph. For the first type of tree, we call it a **one-leveled dependence tree**. We create a single tree t^G from a behavior graph G with an artificial root *Root*. Each tree in T^G is the child of *Root*. Because the maximal height of T^G is 2, a one-leveled dependence tree has height at most 3. It is called a one-level dependence tree because it captures the direct dependence from the graph, which is a one-level depth. We show a one-leveled dependence tree in Figure 4.4 which is parsed from the behavior graph in Figure 4.3.

For the second type of tree, we call it a **two-leveled dependence tree**. We expand the trees in T^G with one level deeper depth. For each $v_i(v_j) \in T^G$, merge with

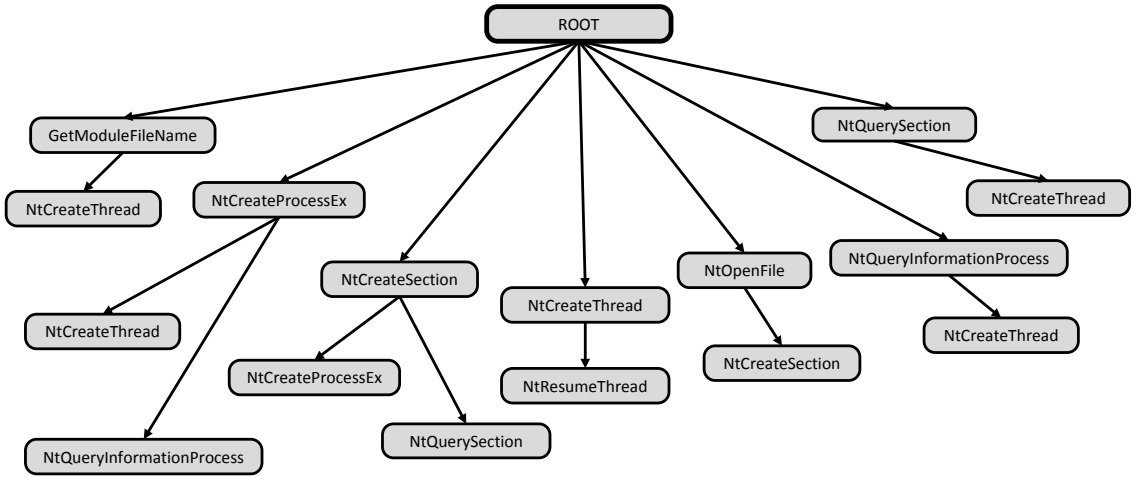


Figure 4.4: An one-leveled dependence tree

$v_j(v_k) \in T^G$ and form a new tree $v_i(v_j(v_k))$. We create a single tree t^G from a behavior graph G with an artificial root *Root*. Each tree in T^G is the child of *Root*. Because the maximal height of T^G is 3, a two-leveled dependence tree has height at most 4.
We show a two-leveled dependence tree in Figure 4.5 which is parsed from the behavior graph in Figure 4.3.

Observe that each child of the root of t^G represents a dependence relation, and there does not exist an ordered relation between the children. To tackle the problem that an identical dependence relation in a behavior graph may generate different trees, we give each tree an ordered relation. Each vertex in a behavior graph is a system call, which is a string. We can sort a set of trees with the following rules:

1. Compare two different trees from the root node to the frontier nodes.

2. Nodes are sorted according to their corresponding system call's dictionary orders.

Trees in Figure 4.4 and 4.5 are sorted according to the above rules.

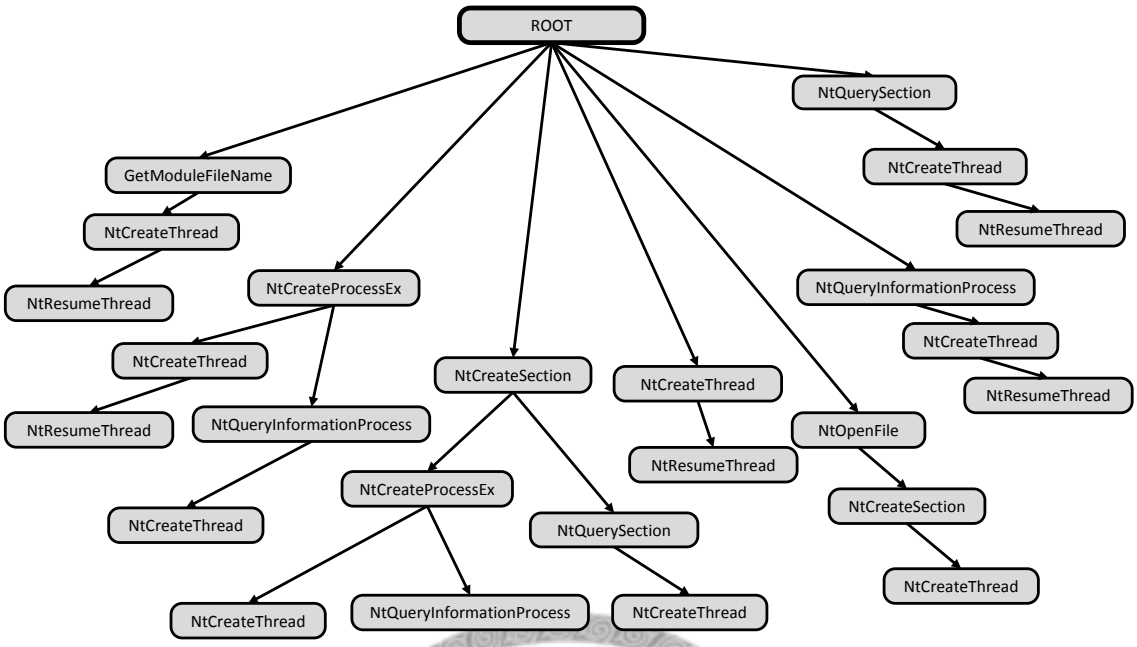


Figure 4.5: A two-leveled dependence tree

4.5 Automata Learning Algorithm

We have mentioned in Section 2.1.1 that Drewes proposed two different tree automata learning algorithms. Because L^*_{fta} is the main algorithm of our learning algorithm, we will take a deeper view.

4.5.1 Learning Algorithm of Drewes

For a set of trees T, let $\Sigma(T)$ denote the set of all trees of the form $f(t_1, ..., t_k)$, where $f \in \Sigma_k$, and $t_1, ...t_k \in T$. Recall that $T(\Sigma)$ denotes the set of all trees over the ranked alphabet Σ. Let $\square \notin \Sigma$ be a special symbol with rank 0, and let $C(\Sigma)$ be the set of all trees in $T(\Sigma \cup \{\square\})$ with exactly one occurrence of $\square$, which we call *contexts* over Σ. The *concatenation* $c \cdot t$ with $c \in C(\Sigma)$ and $t \in T(\Sigma) \cup C(\Sigma)$ is the tree obtained from c by replacing $\square$ with t.

Similar to Angluin's L^* algorithm, they use an observation table Ω to construct a tree

automaton. The observation table Ω can be separated into two parts: the upper table Ω_U and the lower table Ω_L. The rows of Ω_U are indexed by the trees in S, $S \subseteq T(\Sigma)$. The rows of Ω_L are indexed by the trees in T, $T \subseteq \Sigma(S)$. The columns of Ω are indexed by contexts from a finite set $C \subseteq C(\Sigma)$. We use $Mem_L : T(\Sigma) \to \mathbb{B}$ to represent the membership relation of the tree t in the tree language L. If $t \in L$, $Mem_L(t) = True$, otherwise, $Mem_L(t) = False$. The cell in the place of row t and column c in the observation table is filled with $Mem_L(c \cdot t)$, which represents the membership relation of the tree $c \cdot t$ in the tree language L. We use $\langle t \rangle$ to denote the row of t in Ω, and extend to the finite tree set T such that $\langle T \rangle = \{\langle t \rangle \mid t \in T\}$.

The idea behind Angluin's L^* algorithm is to construct an automaton by exploiting the Myhill-Nerode congruence of the target language. The Myhill-Nerode congruence $\equiv_L$ on $T(\Sigma)$ is defined as follows: $t \equiv_L t'$ iff for all $c \in C(\Sigma)$, $Mem_L(c \cdot t) = Mem_L(c \cdot t')$. We say tree t and t' are equivalent with respect to C iff for all $c \in C$, $Mem_L(c \cdot t) = Mem_L(c \cdot t')$.

To construct the finite tree automaton $\mathcal{A}_\Omega$ from Ω, two properties have to hold:

1. Ω is *closed*, that is $\langle t \rangle \in \langle S \rangle$, for every $t \in T$.
2. Ω is *consistent*. Let $\Sigma_\Box(S) = C(\Sigma) \cap \Sigma(S \cup \{\Box\})$, the observation table Ω is consistent if $\langle c \cdot s \rangle = \langle c \cdot s' \rangle$, for all $c \in \Sigma_\Box(S)$ and all $s, s' \in S$ with $\langle s \rangle = \langle s' \rangle$. If $\langle c \cdot s \rangle \neq \langle c \cdot s' \rangle$, then s and s' are not equivalent with respect to $\Sigma_\Box(S)$. And there exists an separating context c that witnesses this inequivalence.

After assuring that Ω is both closed and consistent, we can construct $A_\Omega = (\Sigma, Q, \Delta, F)$ as follows:

- The set of states Q is $\langle S \rangle$
- $\langle s \rangle \in F$ if $s \in L$
- For every tree $t = f(s_1, ..., s_k) \in \Sigma(S)$, the corresponding transition rule is $f(\langle s_1 \rangle, ..., \langle s_k \rangle) \to \langle t \rangle$.

Now, let us describe the L^*_{fta} learning algorithm. As an extension of Angluin's L^*, the learning algorithm asks teacher membership queries and equivalence queries. At first, the

observation table is started with $S = \emptyset$ and $C = \{\Box\}$. Then, it constructs an automaton from the observation table and performs an equivalence query. If there is a returned counter-example, update the observation table. Repeat this process until no counter-example is returned. The pseudo code of the algorithm is presented in Algorithm 4.1.

Algorithm 4.1 L^*_{fta}

Input: $\Omega = \{S, T, C\}$, Teacher.
Output: $\mathcal{A}_\Omega$.

```
1: loop
2:     construct A_Ω;
3:     t := EquivalenceQuery(A_Ω);
4:     if t = yes then
5:         return A_Ω;
6:     else
7:         update(Ω,t);
8:     end if
9: end loop
```

When updating the observation table (Algorithm 4.2), decompose the returned counter-example t from the bottom to top and get a subtree t' that is not in S, where $t = c \cdot t'$ for $c \in C$. If t' is also not in T, add t' to Ω_L and assure that Ω is closed. Otherwise, find the equivalence tree t_e in Ω_U and replace t' with t_e to get a new tree $t_{new} = c \cdot t_e$. If $Mem_L(t) = Mem_L(t_{new})$, decompose t_{new} with above process again. Else, $Mem_L(t) \neq Mem_L(t_{new})$, and we find a separating context c. Add the context c to observation table and assure that the table is closed. The algorithm of updating the observation table is in Algorithm 4.2. The function **close** in Algorithm 4.2 checks whether $\langle t \rangle \in \langle S \rangle$, for every $t \in T$. If there is a tree $t \in T$ which $\langle t \rangle \notin \langle S \rangle$, move t from T to S.

This algorithm has several interesting properties. For every tree $t \in T$, there is exactly one tree $s \in S$ such that $\langle s \rangle = \langle t \rangle$. In other words, there is no redundant information being record. Therefore, there is no need to check table consistent. Besides, it can be assured that the amount of contexts is no more than the states in Ω_U. The L^*_{fta} algorithm outputs a finite tree automaton $A = (\Sigma, Q, \Delta, F)$ with $O(r \cdot |Q| \cdot |\Delta| \cdot (|Q| + m))$,

Algorithm 4.2 update

Input: $\Omega = \{S, T, C\}$, Counter-example t.
Output: Ω.

```
1: loop
2:     decompose t into t = c · t', where t' ∈ Σ(S) \ S;
3:     if t' ∈ T then
4:         let s be the unique tree in S with ⟨s⟩ = ⟨t'⟩;
5:         if membershipQuery(c · s) = membershipQuery(t) then
6:             t := c · s;
7:         else
8:             C := C ∪ {c};
9:             return close(Ω);
10:        end if
11:    else
12:        T := T ∪ {t'};
13:        return close(Ω);
14:    end if
15: end loop
```

where m is the maximum size of counter-examples returned from the teacher, and r is the maximum rank of symbols in Σ. The algorithm requires $|Q| + |\Delta| + 1$ equivalence queries, and $m + |Q| \cdot (|\Delta| + 1)$ membership queries. As mentioned in [14] , the major disadvantage of L^*_{fta} is the number of equivalence queries.

4.5.2 Tree Automata Learning Algorithm

Now, we show how we can learn a 3DFT by adapting Drewes's L^*_{fta} algorithm. What we have are a set of positive examples (malware) and a set of negative examples (benign programs). In order to use Drewes's learning algorithm, we need a teacher to answer membership queries and equivalence queries. Therefore, we simulate the teacher with the given positive and negative examples. The positive examples and negative examples are trees rather than behavior graphs.

For *membership queries*, we check whether a given tree t belongs to positive examples or negative examples. The term *belong* indicates that we check whether a tree is in the set of positive examples or negative examples. If a tree t is in the positive exam-

ples, returns true. If a tree t is in the negative examples, returns false. If a tree t is in both the positive examples and negative examples or in neither of them, return unknown.

For the last case of membership queries, there is a possibility that the positive examples and negative examples have the common members. That is the case where the malware sample is conscious of being analyzed and is pretending as a benign program. Then, the generated behavior graph will be identical to the benign programs.

For *equivalence queries*, we check that whether the samples in the positive examples are accepted and the samples in the negative examples are rejected. For the case that a tree t is both in the positive examples and negative examples, tree t is identified as unknown. If there is a sample that violates this rule, it will be returned as a counter-example. We proceed from the positive samples to negative samples.

Chapter 5

Implementation and Experiments

5.1 Implementation

We implemented a prototype TALA (Tree Automata Learning Algorithm) based on the 3DFT learning algorithm and we take it as our malware detector. TALA is written in C++, and it currently provides following functionalities.

- 3DFT learning algorithm.
- Parsing from a graph to a tree.
- Tree-language membership testing.

Inside TALA, we adapt the library libSFTA [3]. libSFTA is a symbolically encoded finite tree automata library and supports basic automata operations. The term *symbolically encoded* means that they use a multi-terminal binary decision diagrams(MTBDD) to represent transition functions of tree automata. The alphabets of transition functions are encoded into boolean variables. More implementation details can be referred in [3].

5.2 Experiments

To test the capability of our malware detector, we have performed several different experiments. Our detector requires a set of malware instances and a set of benign programs. For input data, we use the data that are publicly available on Babic's website [2]. They are provided as the system call data-flow dependence graph, which are generated by using the tool designed by Daniel Reynaud and the tracing library libwst. The provided

graphs are generated from 2632 malware instances and 35 benign programs. We directly use their data as our behavior graph in the experiments.

The set of positive examples (malware samples) are pre-classified into 48 different malware families. As mentioned in [6], the methods they used for classification are based on the work of Christodorescu et al. [11] and Fredrikson et al. [15]. We list the complete 48 malware families and the number of samples included in each malware family in Table 5.1.

Table 5.1: 48 malware families and the amount of contained samples

Malware Family	Samples	Malware Family	Samples
ABU,Banload	16	Hupigon,AWQ	219
Agent,Agent	42	IRCBot,Sdbot	66
Agent,Small	15	LdPinch,LDPinch	16
Allaple,RAHack	201	Lmir,LegMir	23
Ardamax,Ardamax	25	Mydoom,Mydoom	15
Bactera,VB	28	Nilage,Lineage	24
Banbra,Banker	52	OnlineGames,Delf	11
Bancos,Banker	46	OnLineGames,LegMir	76
Banker,Banker	317	OnLineGames,Mmorpg	19
Banker,Delf	20	OnLineGames,OnlineGames	23
Banload,Banker	138	Parite,Pate	71
BDH,Small	5	Plemood,Pupil	32
BGM,Delf	17	PolyCrypt,Swizzor	43
Bifrose,CEP	35	Prorat,AVW	40
Bobax,Bobic	15	Rbot,Sdbot	302
DKI,PoisonIvy	15	SdBot,Sdbot	75
DNSChanger,DNSChanger	22	Small,Downloader	29
Downloader,Agent	13	Stration,Warezov	19
Downloader,Delf	22	Swizzor,Obfuscated	27
Downloader,VB	17	Viking,HLLP	32
Gaobot,Agobot	20	Virut,Virut	115
Gobot,Gbot	58	VS,INService	17
Horst,CMQ	48	Zhelatin,ASH	53
Hupigon,ARR	33	Zlob,Puper	64

The set of negative examples (benign programs) includes 35 different samples of frequently used applications. The complete lists are list in Table 5.2.

Table 5.2: The list of benign programs

Adobe_Reader	Apple_Software_Update	Autoruns
Battle_for_Wesnoth	Chrome	Chrome_Setup
Copy_to_system_folder	Firefox	Freecell
Freeciv	Freeciv_server	GIMP
Google_Earth	Hello_world	Internet_Explorer
iTunes	Minesweeper	MSN_Messenger
Netcat_port_listen	Netcat_port_scan	NetHack
Notepad	OpenOffice_Writer	Outlook_Express
ping	Self_extracting_archive	Skype
Solitaire	System_information	Task_Manager
Tux_Racer	uTorrent	VLC
Windows_Media_Player	WordPad	

We have tested the detection ability and classification ability of our detector. The procedure of testing detection ability is depicted as follows.

1. The samples in each malware family are separated into two disjoint sets, one for learning and one for testing.
2. The benign programs are also separated into learning sets and testing sets.
3. We parse each sample into a tree.
4. We use the learning samples to learn a 3DFT for each malware family.
5. We use the testing samples to test detection ability.

As mentioned in section 4.4, we extracted the dependence from the behavior graph and build a tree for a graph. In our experiments, we have parsed two types of trees from a graph. The first type of tree has height at most 3, and is defined in section 4.4 as *one-leveled dependence tree*. The second type of tree has height at most 4, and is defined in

section 4.4 as *two-leveled dependence tree*. For each type of dependence tree, we executed 4 experiments with different combinations of learning samples and testing samples. The ratio of the number of samples in the learning set and testing set are presented in Table 5.4.

Table 5.3: Experiments for testing detection ability

	Positive Examples		Negative Examples	
	Learning	Testing	Learning	Testing
Experiments 1	80%	20%	100%	0%
Experiments 2	80%	20%	80%	20%
Experiments 3	50%	50%	100%	0%
Experiments 4	50%	50%	50%	50%

The procedure of testing detection ability is depicted as follows.

1. We use the automata in malware family Banker,Banker that are generated from the above experiments.
2. Taking the samples in other malware family as testing samples.
3. Test samples in other malware families.

Table 5.4: Experiments for testing classification ability

	Positive Examples		Negative Examples	
	Learning	Testing	Learning	Testing
Experiments 5	80%	20%	100%	0%
Experiments 6	50%	50%	100%	0%

5.2.1 Experimental Results

We describe the results that are experimented with one-leveled dependence trees at first.

For Experiment 1, all test samples are identified as *accept* or *unknown*, no one is identified as *reject*. In Figure 5.1, we showed the accepting rates for testing samples in each malware family. Since some malware family have test samples less than 10, it is meaningless to discuss their accepting rates, therefore, the first 10 malware families which have most samples are labeled with different format in Figure 5.1. We can see that although the accepting rates are not good enough, but they are on average more than 50%.

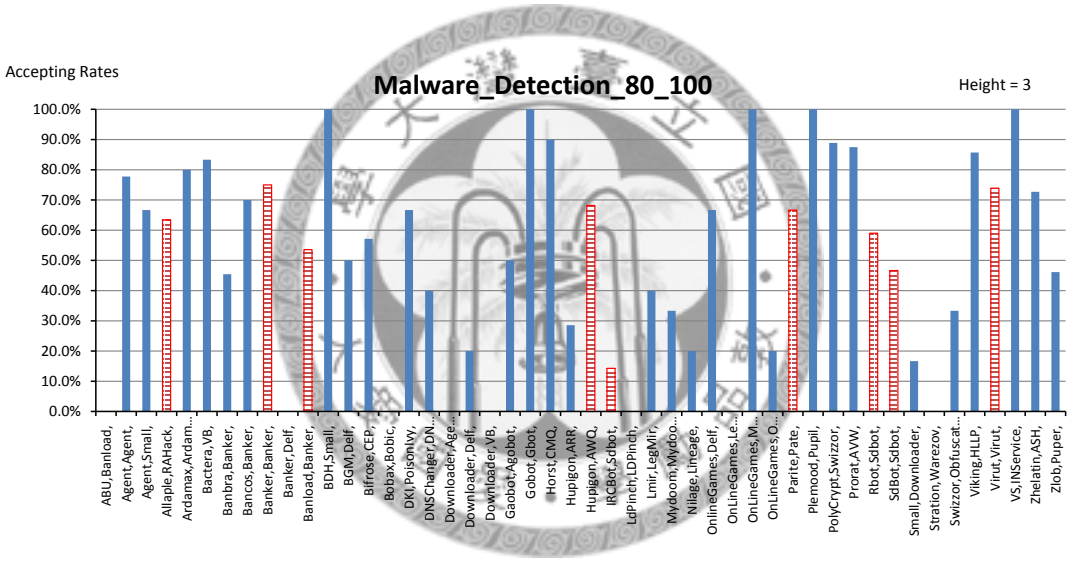


Figure 5.1: Experiment 1 results with one-leveled dependence tree

For Experiment 2, we take 20% of positive examples and 20% of negative examples as the test samples. The accepting rates for positive test samples are identical to Experiment 1. And all test samples in the negative set are identified as *unknown*.

Notice that the samples in the negative test set are all identified as unknown, the possible reason is that the samples in the negative set have great difference with each other, and they cannot be identified by the detector. Besides, the accepting rates in the

positive test sample are identical to Experiment 1, which reflects that the accepting rates are more relied on the input positive examples rather than negative examples.

In Experiment 3, 50% of positive examples are used as test samples. The results are showed in Figure 5.2. We can observe that the accepting rates are similar but less than Experiment 1. It is quiet straightforward since learning with less examples will let the detector identified less behaviors and identified more programs as unknown.

For Experiment 4, 50% of positive examples and 50% of negative examples are used as the test samples. The accepting rates for positive test samples are identical to Experiment 3. And the accepting rates for all negative test samples are identified as unknown.

As we can see, in this experiments, all test samples in negative set are identified as unknown. It supports our previous inference that the samples in the negative sets are quiet different with each other, and cannot be identified by the detector.

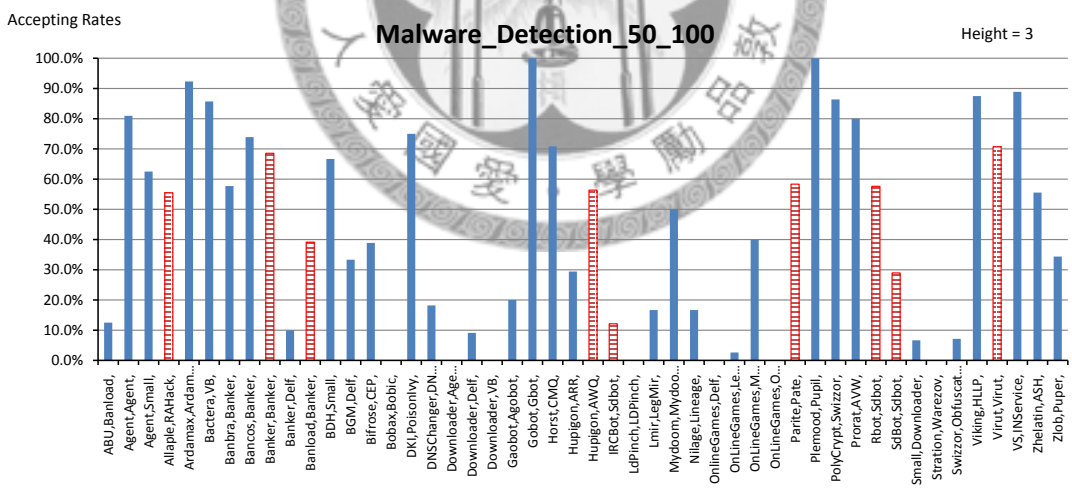


Figure 5.2: Experiment 3 results with one-leveled dependence tree

We look at the results that are experimented with two-leveled dependence trees now. Actually, the results we get are very similar to previous experiments. In Figure 5.3, we showed the results for Experiment 1. In Figure 5.4, we showed the results for Experiment

3. We can see that the accepting rates are increased compared with previous experiments. The accepting rates in the malware families which have the most samples are increased a little bit. For the remain families, their improvement in accepting rates are more obvious. All testing samples in the positive set are identified as *accept* or *unknown* and all testing samples in the negative set are identified as *unknown*, just identical to previous experiments. By these experiments, we can say that if we learn a 3DFT with a more accurate dependence tree (two-leveled dependence tree), the detection rates will increase, especially for the family that has less samples.

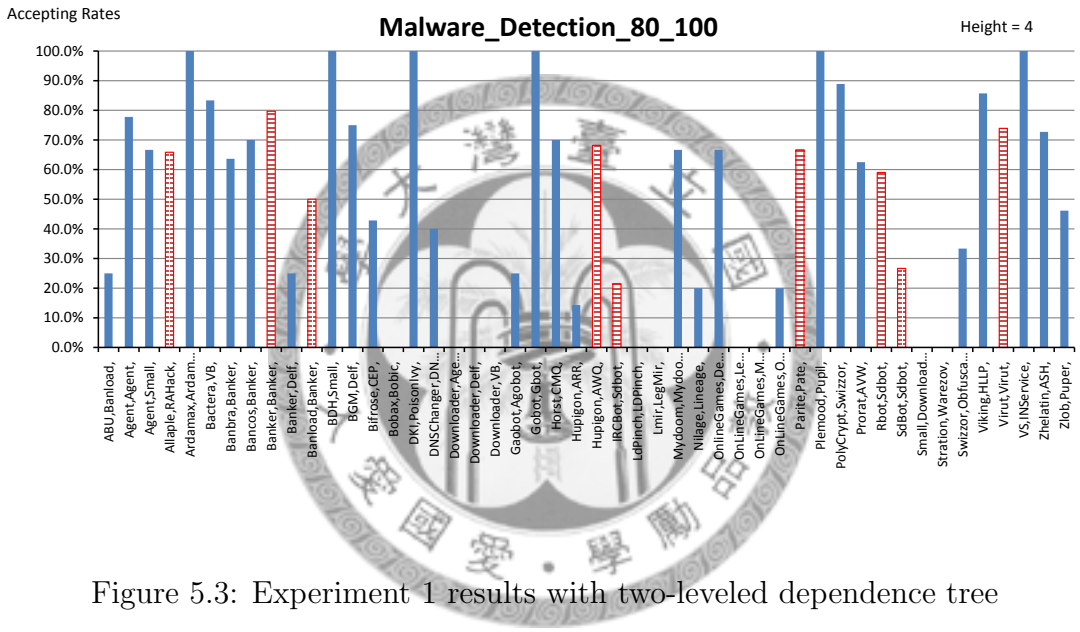


Figure 5.3: Experiment 1 results with two-leveled dependence tree

So far, we have tested 4 different combinations of learning samples and testing samples. Besides detection rates, the time required for learning and the size of the generated automaton are also the important factors to reflect the effectiveness of the detector. We recorded the size of the automata which generated from Experiment 1 to Experiment 4 and also recorded the time expense for learning each automaton.

The total states of each automaton are showed in Figure 5.5 and the total transitions of each automaton are showed in 5.6. For the time expense, they are depicted in Figure 5.7. We can see that the automaton generated from Experiment 1 has the largest size, and

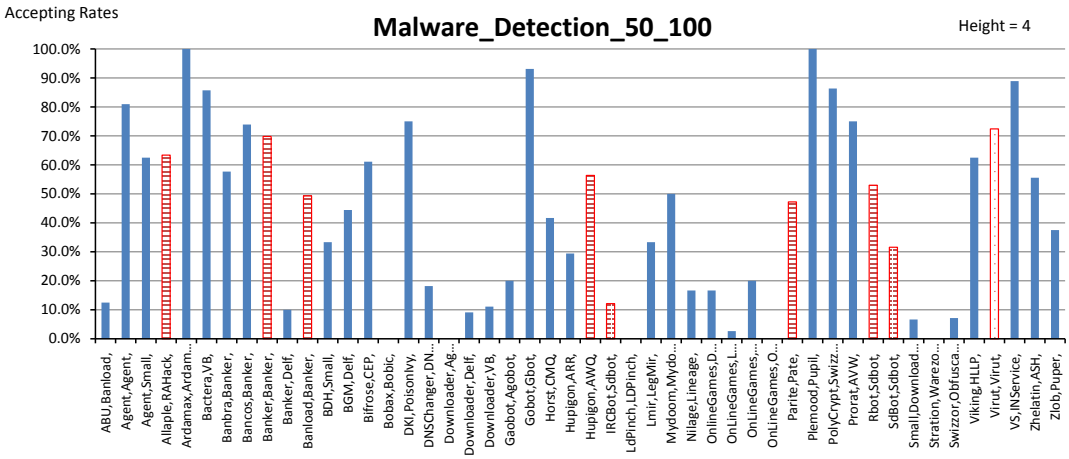


Figure 5.4: Experiment 3 results with two-leveled dependence tree

the automaton generated from Experiment 4 has the smallest size. However, although the automaton generated from Experiment 3 is learned with less positive examples compared with the automaton generated from Experiment 2, the size of the automaton is not less than the automaton generated from Experiment 2. We guess that is because the samples in the negative examples are more complicated and discriminative, therefore, the detector needs to learn more rules to process this differences.

For classification, we use the automaton generated from Experiment 1 and Experiment 3 to test malware samples in other malware families. The automaton is learned for single malware family, and the samples in other malware families are used to test samples. In Figure 5.8, it is the automaton learned from Banker,Banker family and is generated from Experiment 1. Figure 5.9 is the automaton learned from Banker,Banker family and is generated from Experiment 3. As we can see, the malware families which have similar name with Banker,Banker have higher accepting rates compared with others.

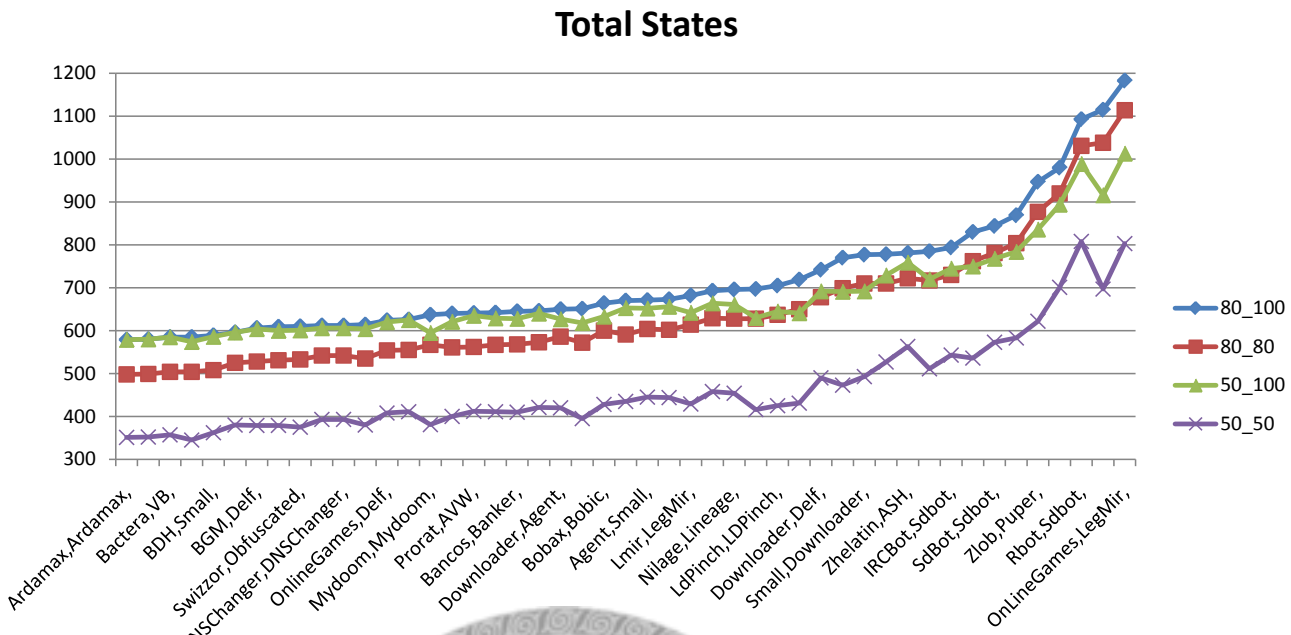


Figure 5.5: Total states of generated automaton

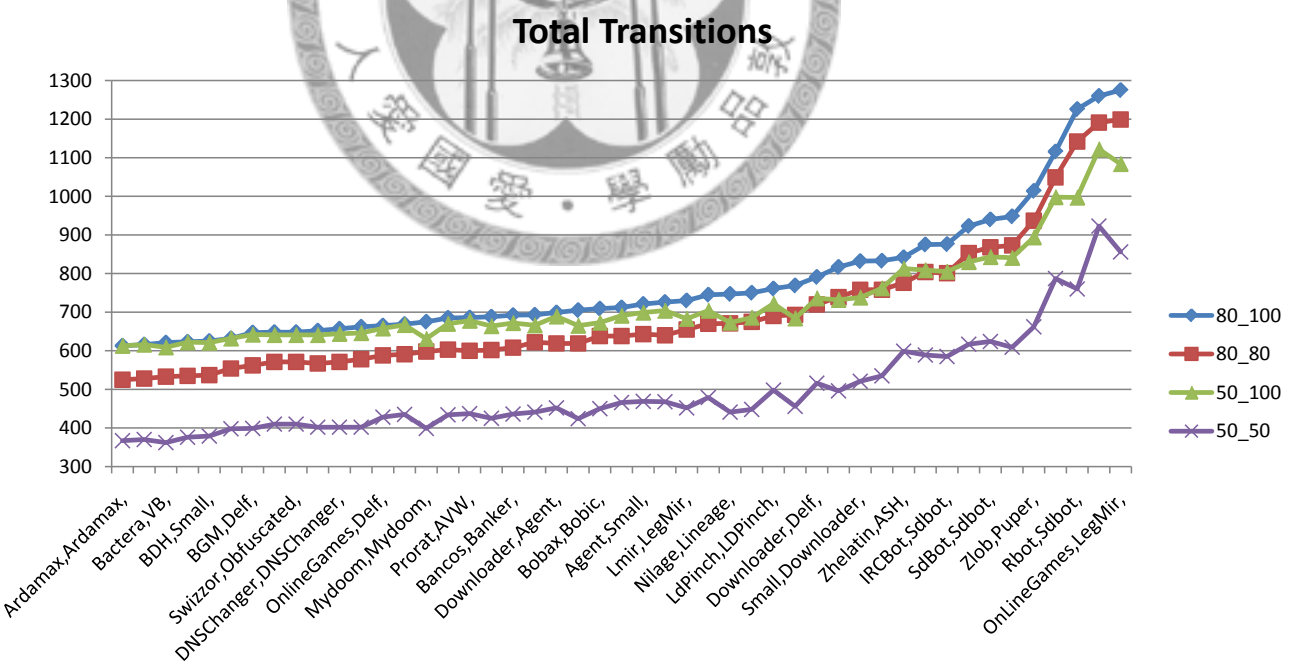


Figure 5.6: Total transitions of generated automaton

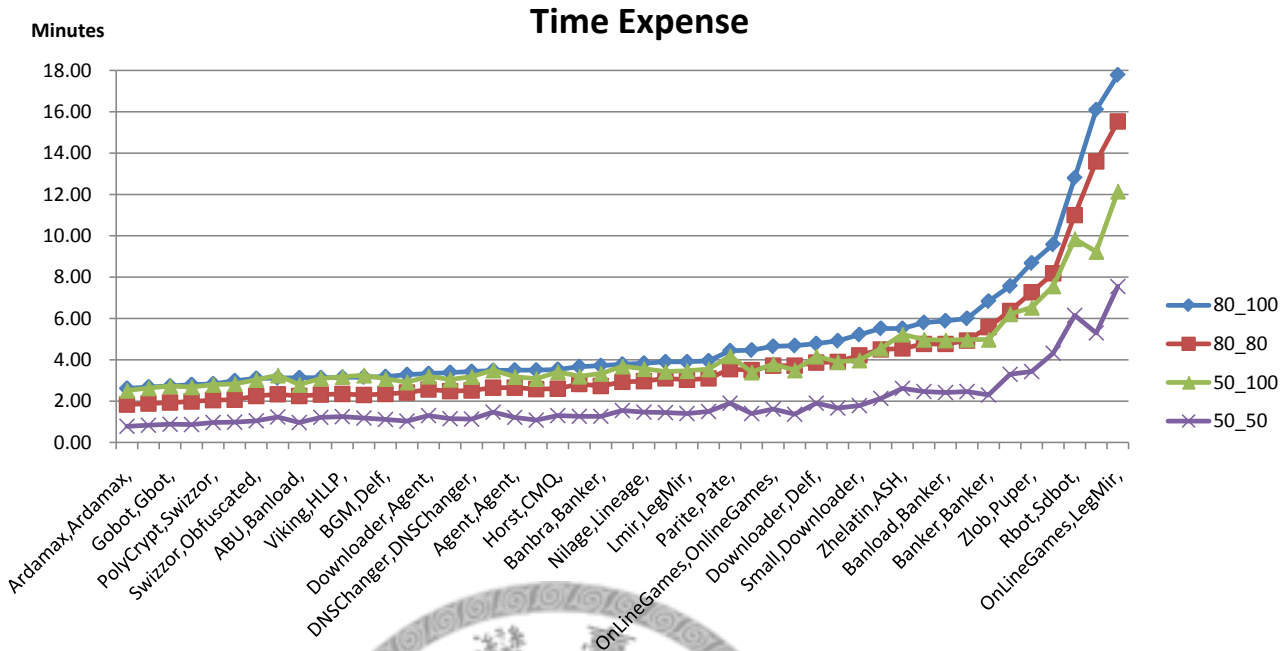


Figure 5.7: The time expense for learning automaton

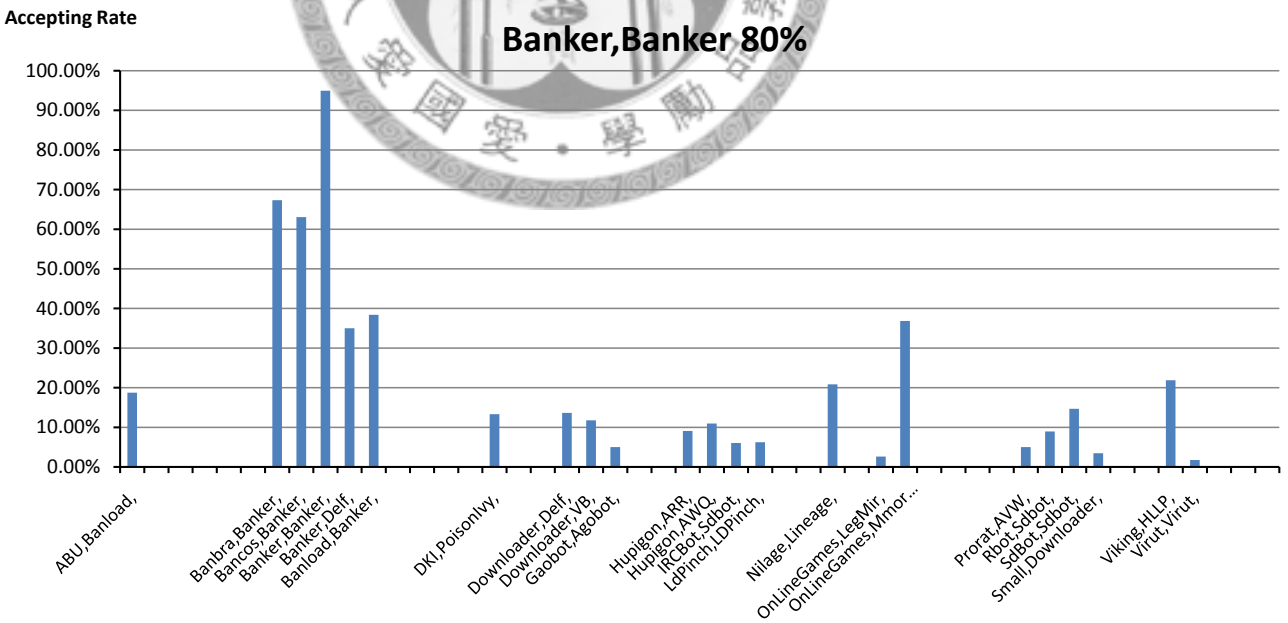


Figure 5.8: Classification results for automaton generated from Experiment 1

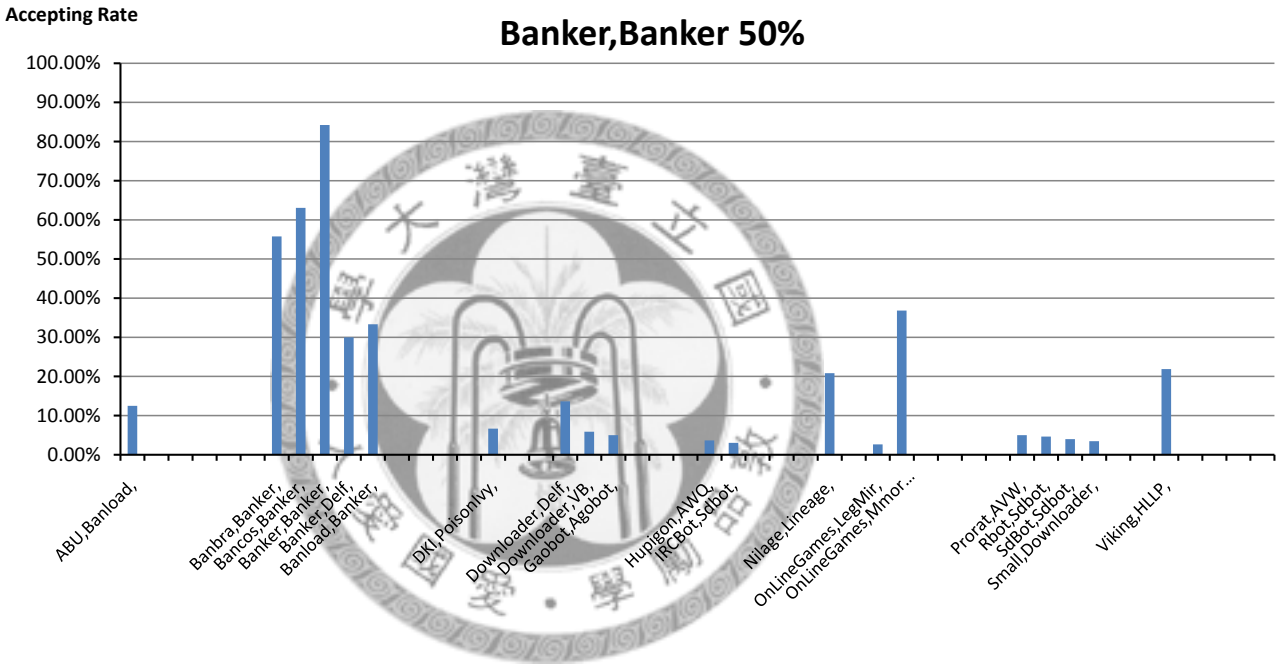


Figure 5.9: Classification results for automaton generated from Experiment 3

Chapter 6

Conclusion

In this thesis, we proposed a malware detection approach that is behavior-based and uses a tree representation as the signature. Our detector requires a set of malware samples and a set of benign programs. The semantics of each input program is extracted and represented as a system call dependence graph. The graph is then parsed into a tree. We take the set of trees generated from malware and benign programs as inputs to our tree automata learning algorithm. For automata learning, we adapted the methods proposed by Drewes [14], which was adapted from Angluin's L^* algorithm [4]. When the learning algorithm is terminates, it outputs a 3-valued deterministic finite tree automaton (3DFT). A 3DFT contains three disjoint sets of final states: accept, reject, and unknown. Therefore, using the 3DFT as the malware detector, it outputs corresponding three different values: true, false, and unknown. If the input program is a malware instance, the detector outputs true. If the input program is a benign program, the detector outputs false. Otherwise, it outputs unknown. According to our experiments, our detector exhibits very low false positives. However, there is a tradeoff that many programs are identified as unknown.

6.1 Contributions

We summarize our contributions as follows:

- We introduced 3DFT for malware analysis.
  Babic et al. [6] and Bonfante et al. [8] have proposed to use finite tree automata in malware analysis. We generalized finite tree automata to 3-valued deterministic

finite tree automata. A 3DFT has three different final states: accept, reject, and unknown. We take a 3DFT as the malware detector, it outputs 3 different values. If an input program is a malware instance, it outputs true. If an input program is a benign program, it outputs false. Otherwise, it outputs unknown.

- We proposed a learning algorithm for 3DFT from the sets of positive examples and negative examples.

  We adapted the learning algorithm of Drewes [14]. Our algorithm requires a set of positive examples and negative examples and outputs a 3DFT. We sequentially process the samples from the set of positive examples to negative examples. Every sample in the set of positive examples is accepted by the 3DFT and every sample in the set of negative examples is rejected by the 3DFT.

- We implemented a prototype malware detector.

  We implemented the tool TALA for our 3DFT learning algorithm and we take its outputs as malware detectors. The prototype currently supports the following functionalities: 3DFT learning algorithm, parsing from a graph to a tree, and tree-language membership testing.

6.2 Future Work

There are several issues that can be considered as future works, and we list them as follows.

- Improve the accepting rates for our malware detector.

  Currently, the detection rates of our detector are on average 50%. The time required to learn a 3DFT is ranged from 1 minute to 18 minutes. It is inefficient for a practical use. There are several possible solutions to consider. First, unfold the graph with a more appropriate tree representation. We unfold the data-flow dependence graph with 2 different depths. It is worthwhile to try another representation. Second, improve the tree automata learning algorithm we used. We adapted the algorithm proposed by Drewes [14] to learn a 3DFT. However, in some cases we have to double check every example, and it increases the run time.

- Find the minimal tree automaton that is consistent with a 3DFT.

  We use a 3DFT as our malware detector, which outputs three different values. However, it is valuable to learn a minimal tree automaton that is consistent with a 3DFT. A tree automaton is consistent with a 3DFT if it accepts the trees that are accepted by a 3DFT, and rejects the trees that are rejected by a 3DFT. The states that are identified by a 3DFT as unknown can be either accepted or rejected by the consistent tree automaton.

Bibliography

[1] AV-TEST Statistics. http://www.av-test.org/en/statistics/malware/.

[2] Babic's website. http://www.domagoj-babic.com/index.php/ResearchProjects/MalwareAnalysis.

[3] libSFTA. http://www.fit.vutbr.cz/research/groups/verifit/tools/libSFTA/.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[6] Domagoj Babic, Daniel Reynaud, and Dawn Song. Malware analysis with tree automata inference. In *CAV*, pages 116–131, 2011.

[7] Jérôme Besombes and Jean-Yves Marion. Learning tree languages from positive examples and membership queries. *Theor. Comput. Sci.*, 382(3):183–197, 2007.

[8] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[9] John M. Brayer and King-Sun Fu. A note on the k-tail method of tree grammar inference. *Systems, Man and Cybernetics, IEEE Transactions on*, 7(4):293 –300, april 1977.

[10] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning Minimal Separating DFA's for Compositional Verification. In *TACAS*, pages 31–45, 2009.

[11] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA*, pages 34–44, 2004.

[12] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[13] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[14] Frank Drewes. MAT learners for recognizable tree languages and tree series. *Acta Cybern.*, 19(2):249–274, 2009.

[15] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[16] H. Fukuda and K. Kamata. Inference of tree automata from sample set of trees. *International Journal of Parallel Programming*, 13:177–196, 1984. 10.1007/BF00979871.

[17] Pedro García and Jose Oncina. Inference of Recognizable Tree Sets. Technical Report DSIC-II/47/93, Universidad de Alicante, 1993.

[18] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 351–366, 2009.

[19] Lorenzo Martignoni, Elizabeth Stinson, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 78–97, 2008.

[20] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185(1):15–45, 1997.

[21] Symantec. Symantec Internet Security Threat Report, Trends for 2010. http://www.symantec.com/business/threatreport/, 2010.