Spring 2015

Static Analysis of Malicious Java Applets

Nikitha Ganesh
San Jose State University

Static Analysis of Malicious Java Applets

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Nikitha Ganesh

May 2015

The Designated Project Committee Approves the Project Titled

Static Analysis of Malicious Java Applets

by

Nikitha Ganesh

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Dr. Mark Stamp	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science
Fabio Di Troia	Università del Sannio

ABSTRACT

Static Analysis of Malicious Java Applets

by Nikitha Ganesh

In this research, we consider the problem of detecting malicious Java applets, based on static analysis. In general, dynamic analysis is more informative, but static analysis is more efficient, and hence more practical. Consequently, static analysis is preferred, provided we can obtain results comparable to those obtained using dynamic analysis. We conducted experiments with the machine learning technique, Hidden Markov Model (HMM). We show that in some cases a static technique can detect malicious Java applets with greater accuracy than previously published research that relied on dynamic analysis.

ACKNOWLEDGMENTS

I would like to thank my project advisor Dr. Mark Stamp for his continuous guidance and for also believing in me. Without his guidance, mentoring and support, this project would not have been completed. Also I would want to thank him for his patience throughout the process.

I would also like to thank the committee members Mr. Fabio Di Troia and Dr. Sami Khuri for their valuable inputs and to monitor the progress of the project closely.

Furthermore, I would like to thank my parents for being there with me in each and every step through out my masters program. Also I would like to thank my friends for their encouragement and support throughout the completion of this project.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Malware, which is a term used in short for malicious software, is used to steal personal data and credentials of an individual. It is also used to manipulate the data and online banking transactions. Manipulating the data can create a lot of trouble, ranging from inappropriate message displaying to the deletion of files and drives. It is also capable of launching denial of service attacks. To infect a system, malware developers can use social engineering techniques [1] in tricking the user into executing the program, or even without user interaction [18] they can exploit security vulnerabilities to acquire access to a system. Gameover Zeus and Cryptolocker are popular malwares that are dangerous and mainly targets the data in finance [20]. As well, we should also pay attention and be careful with a lot of other types and variants of credentials stealing Trojans [20].

In the internet environment, Zeus family is one among the most advanced credential-stealing Trojans [20]. The Zeus/Zbot Trojan which belongs to Zeus family are known by many other names like PRG and Infostealer. This family of virus has the record of infecting around 3.6 million systems in the United States. In 2009, the statistics says that the Zeus family had infected more than 70,000 bank accounts including the businesses or organizations such as NASA and the Bank of America.

In simple terms software vulnerabilities can be explained as a security flaw or like a glitch found in a software. It can lead to many security concerns. More often we hear about vulnerabilities with respect to client software, for example, email applications and web browsers. These client software can be exploited by malicious

programs. An attacker could exploit these in a number of ways depending on the behavior of the applications, for example by using a specially crafted email attachment or by persuading the user to visit a website which performs malicious activities. Web browsers are the usual targets. Other well known targets are Adobe Acrobat applications, Quick Time player, Macro-media flash and Java Run-time Environment.

Web browsers are particularly of interest when it comes to software vulnerability. This is because it allows the exploiters to interact with the victim even before actually infecting the browser. For example, an attacker can target the applications like Real Player, Quick Time, or even the victim's Antivirus program, that the browser uses to properly render a website. Also, an attacker can submit a malicious JavaScript request or Java Applet to the browser. This is used to determine which plugins are installed and thus which exploit could be used against a particular target. Once the application is compromised, a creative attacker can get access to the user's browsing history, or to the information which is currently present in user's screen or clipboard. That information can include the user's credentials like the username and password or even the credit card details.

The Java applets, like the typical Java applications consists of a groups of compiled Java classes. They are usually bundled in a jar archive. Java applets are typically embedded inside HTML pages, explicitly specifying the main class as well as additional parameters that should be provided to the applet as attributes. The important part here is the host must be protected from unrestricted access. So the Java applets are subjected to a security manager. The first thing the malicious applets does is, it tries to disable this security manager, thus allowing them to access the restricted resources in the system.

Our goal here is to perform static analysis of Java applets using Hidden Markov

Model. The area under the ROC curve (AUC) will serve as our measure of success. We will compare our results to those obtained using a dynamic analysis techniques similar to that in [12] on the same test set.

The paper is organized as follows. Chapter 2 gives an overview of malware and its detection techniques, as well an overview of applets and its essentials. Chapter 3 gives the details of Hidden Markov Model. Chapter 4 explains code obfuscation and its techniques. Chapter 5 gives an overview of static and dyamic analysis and also the discussion of relevant work to which we are compering our results. Chapter 6 explains the relevant work in the project and the experimental results. Furthermore, Chapter 7 gives the conclusion and the future work related to this project.

CHAPTER 2

Background

This chapter consists of introduction to malware, the malware detection techniques, followed by overview of applets, its essentials and how to execute it.

2.1 Malware

Malware is a set of instructions designed or programmed by the attackers, which when run on the computer system, makes the system to behave in an abnormal way. It makes the system to behave like how the attacker wants it to behave. Usually they are intended to break software security of a system. The malicious activities of these malware range from a simple prank to creating harm to a computer. Once these malware infects, if we do not want data loss, then these malwares should be identified and removed using antivirus software.

Some typical categories of malware are viruses, worms, trojan horse or trojan. We can also find other categorization of viruses like rabbit, spyware, trapdoor or backdoor. The explanation and differences of each of these categories of malware is given in [28].

The first computer virus was created in 1986. It was called the Brain virus. It was developed to infect the boot sector of the storage media. Virus creation and detection methodologies have evolved to a greater extent ever since Brain virus was created. Malware can also be a source of generating money for a malware developer [22]. Malware developers develop malwares which try to evade detection from popular detection techniques. Once a virus evades an anti-virus system and manages to infect

quite a large number of computers, the techniques of virus detection are usually updated so that further infection is detected and hence avoided.

2.2 Malware detection techniques

It is not possible to build a perfect virus/malware detector. We can build the detectors to detect signatures which can be considered as fingerprint like. We can build the detectors to detect changes and anomalies, and also which can detect by performing static analysis or similarity analysis. Likewise, we can use different characteristics of malware in a variety of detection techniques. The following sub-sections will give more details of general approaches that are usually used to detect the malware.

2.2.1 Signature based detection

It is the most common and well-known method that antivirus software uses to identify a program or an application as malware. When files in a system are scanned for viruses, it checks the files against virus signatures present in the dictionary. The dictionary contains the signatures of previously known viruses. A signature [28] is a set of bits or a string of bits that are found in a file, which might also include wildcards. A hash value could also serve as a signature.

This method is very effective in detecting previously known malwares and for those malwares where a common signature can be easily extracted. Also this method has minimum overhead on users and administrators, since the only work to be done here is to maintain the file in which signatures are stored by keeping them up to date.

A fundamental problem with this method of detection is that we can only detect the known signatures, i.e., we can detect only previously known malware. Even a slight variant of a known virus might be missed.

2.2.2 Change based detection

In this method if we detect that there is a change somewhere on a system, then it may imply that the system has been infected by the virus. Usually the changes are determined with the use of the hash functions. It works as follows. Compute hashes of all files present on the system and securely store those hash values in a file. After regular intervals we can compute the hashes of those files again and compare the new hash values with those stored in the file before. If any file has changed in the order of one or more bits we can notice that the hash value computed for that will not match the previously computed one for that particular file.

Fundamental advantage in this method is that we can detect previously unknown malware. However, a major disadvantage is that there are more chances of false positives since files on a system keep changing very often. This can be considered as an overhead on the part of users and administrators.

2.2.3 Anomaly based detection

This technique is used in determining any unusual activity caused by malwares. The challenge with respect to this detection technique is to find out what is supposed to be normal and what is not, and importantly, being able to distinguish between the two. A major difficulty which arises here is, the definition of what is fixed as normal for now might change in the future, and the system must have the capacity to adapt to the changing definition.

The main advantage of this detection method is that we can possibly detect previously unknown malware. There is a work around for this detection technique for a patient attacker to evade the detection. This can be done by being able to make an anomaly appear to be normal. This detection technique alone is not enough to

detect the virus. It is usually combined with signature based detection technique.

2.2.4 Statistical based detection

The most common approach like signature based detection technique cannot be used in the case of malwares which are metamorphic in nature. This is because, these metamorphic malwares evade signature detection by morphing their code. These metamorphic malwares can vary from each other to a vast extent. Yet some of the statistics of the metamorphic malwares remains unchanged to support its actual functionality.

2.2.4.1 Hidden Morkov Model based detection

Hidden Markov Models(HMMs) [13] can be defined as statistical Markov models in which the system being modeled is assumed to be a Markov process with unobserved(hidden) state. HMMs are commonly used for statistical pattern analysis. They are especially known for their applications like speech recognition [23], malware detection and biological sequence analysis [16]. They are also used to detect software piracy [15] and also in protein modeling [16].

2.2.5 Similarity based detection

This detection technique is used in detecting metamorphic malware. It works as follows. It first finds the characteristics that are common to members of a particular metamorphic family. Once we know what the common characteristics are, we can use them to detect metamorphic virus which belonging to the same family.

2.3 Overview of applets

There are two different forms of Java programs. They are Java applications and Java applets [4]. The Java applications run on their own. The Java applets run inside a web page. It is a small application, written in Java programming language and is sent to users in the bytecode form.

When we invoke a Java applet, it gets executed in a Java Virtual Machine(JVM), in a process which is separate from the web browser. When it gets executed, the executable code of the applet gets downloaded to your local system. The web page along with the applet, arrives from the server. It is then executed on the local system [32] by the browser. This supports the web page to have dynamic features. Applets can showcase dynamic graphics on the web page as well. It also lets the user to interact with it. When user interacts with applets, it makes a connection back to the server to query the database accordingly.

The web browser provides the needed infrastructure to support the applet and to help its execution. Applet is platform independent. This is because, the environment provided by the browser acts like an insulation shield for the applets from the underlying operating system. This feature allows the applet to execute on any platform in which the browser is Java-enabled. Dealing with applets is simple because their code is compact. Also, standard libraries are provided on client platform such as libraries to access graphical user interface and network access. So only the code which is unique to applet needs to be transferred across the Internet. Simple example of an applet is as shown in Figure 1.

There are three main parts in the program shown in Figure 1. Import statements, class declaration and custom method (paint method). They are discussed in the

```
import java.awt.*;
import javax.swing.*;
public class JavaApplet extends JApplet {
    public void paint(Graphics g) {
      super.paint(g);
        g.drawString("The Java Applet", 30,20);
    }
}
```

Figure 1: Simple example of an applet

subsequent subsections.

2.3.1 Import statements

The first two line in the Figure 1 contains import statements. The importance of these statements are, they allow reuse of the code which already exists and hence we need not reinvent the wheel again. There are a lot of standard classes provided by Java distribution. They are called as Java application programming interface. For instance, if we want to create an applet and want to display the text on the screen, then there is an existing code to draw things on the screen. We can use that class by referencing it.

The 'Graphic' class is the example here. When the compiler looks for Graphics.class file, it needs to know where it is. So, we need to explicitly tell the compiler as to where it should look for, by using import statement. Here we are specifying that it is in java.awk package.

JApplet class can be found in javax.swing package. So the second import statement indicates the compiler that, it should look for classes in that package to get the code related to JApplet. This has the code to display the output within a web page.

2.3.2 Class declaration

Every classes that we create will need a class declaration. In the example above the name of the class is JavaApplet.

Because we are programming an applet, it is necessary that we have to first specify that our class is an applet. For that, our class should extend the JApplet class. By doing this we inherit the applet functionality without rewriting the entire code to make an applet. It is important that the applet has public access specifier to ensure that we can run the program directly since applets do not have main function.

2.3.3 Custom method

The core of the program lies at the paint method in the above example. Here, we are drawing the text "The Java Apple" inside our applet. The Graphics class helps accomplish this task.

If we have to use the Graphics class, first we need to declare a variable reference for it. In the example above we use the variable g. Now we can make a call to the methods on this object as g.<methodName>, where <methodName> refers to the name of the method in that class. In our example we have used the method drawString. This method called with Graphics object prints the text. When this method is being called, we are supposed to pass some paramethers. The parameters of drawString in our example are as follows: First, "The Java Apple", it is the text that needs to be printed. Second, 30, it is the x coordinate of where we want the text to be drawn. Third, 20, it is the corresponding y coordinate to draw the text. The super keyword in the example above refers to the parent class. It forces the applet to first draw itself before executing the drawing code.

2.3.4 Essential of applets

The essential requirement for an applet to run is HTML file [4]. It is required for displaying the applets. Applets are supposed to be embedded inside the web pages. This part of the code needs to be in a different file from that of the .java file which has the actual source code for the applets. Figure 2 shows the code to get our applet to display.

```
<html>
   <body>
      <applet code="JavaApplet.class" width=400 height=500>
      </applet>
   </body>
</html>
```

Figure 2: HTML code to display the applet

2.4 Executing the applet

To execute the applet, open any text editor and copy the applet described in section 2.3. Save the file as JavaApplet.java. The name of the file should be same as class name. Compile this java program. The java compiler javac is being used. Use the the command: javac JavaApplet.java. When the source code is run through the compiler without any errors, we get a bytecode file JavaApplet.class. Java applet execution method is different from that of the method used to run a Java application program. There are two ways to run a Java applet. One way is by using Java compatible web browser. The other one is by using Applet Viewer which will be included with the JDK.

2.4.1 Using Java compatible web browser

Open any of the text editors, for example, notepad. Type the html code described in section 2.3.4. Here the width and height correspond to your output applet window size. Save the file as html file in the same location as where we have saved JavaApplet.java file. If any other location is preferred, then we have to give the absolute path as the value for 'code' attribute which is used with the tag 'applet'. After the html file has been saved, open it with any Java compatible web browser. We will be able to see the applet in that browser.

2.4.2 Using applet viewer

Usually, JDK contains an Applet Viewer for the purpose of viewing applets. For executing the applet using this method, just add the applet tag containing statement in Figure 2 just before the class in the java program as shown in Figure 3.

```
import java.awt.*;
import javax.swing.*;
/* <applet code="JavaApplet.class" width=400 height=500> */
public class JavaApplet extends JApplet {
    public void paint(Graphics g) {
      super.paint(g);
        g.drawString("The Java Applet", 30,20);
    }
}
```

Figure 3: Inserting applet tag in Java code

After adding the applet tag line in Figure 3, save and compile it. After the compilation is successful, type "appletviewer JavaApplet.java" at the command prompt to run the applet.

CHAPTER 3

Malware Detection Using Hidden Markov Models

Markov model is usually used to model a system which keeps changing its states randomly. It follows Markov property which assumes that the future states of the system depends only on the current state and not on any other sequence of events which has occurred before. In this model, the states are visible. On the other hand, the states of HMM are not directly observable. The HMM models a state machine where each state is assigned the probability of observing a set of observation symbols. Also the transition from one state to another have fixed probabilities. The HMM is considered to be a machine learning model where we can train it, so that it represent a particular set of data, by using some observation sequences of that data, so that it will be able to identify or match an observation sequence in future for a new set of data tested against a trained HMM to find the probability of occurrence of such a sequence. If the probability obtained is more, it implies that the observation sequence is almost similar to the training sequences. We use the notation in Table 1 to describe an HMM [27].

We can define an HMM with the help of the matrices A, B and π as follows: $\lambda = (A, B, \pi)$. Here the matrices A, B, and π are row-stochastic. That means, each of the element in the matrix is a probability and sum of all the elements in each row of the matrix will sum to 1. This implies that each row is a probability distribution.

Figure 4 illustrates the generic form of HMM.

Here the state of HMM at any point of time t is represented by X_t. Also the observation at any point of time t is represented by $\mathcal{O}_t$. In the figure, the part above

Table 1: Hidden Markov Model notation

notation	explanation
T	length of the observation sequence
N	number of states in the model
M	number of observation symbols
Q	distinct states of the Markov process, $q_0, q_1, \ldots, q_{N-1}$
V	possible observations, assumed to be $0, 1, \ldots, M-1$
A	state transition probabilities
B	observation probability matrix
π	initial state distribution
$\mathcal{O}$	observation sequence, $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}$

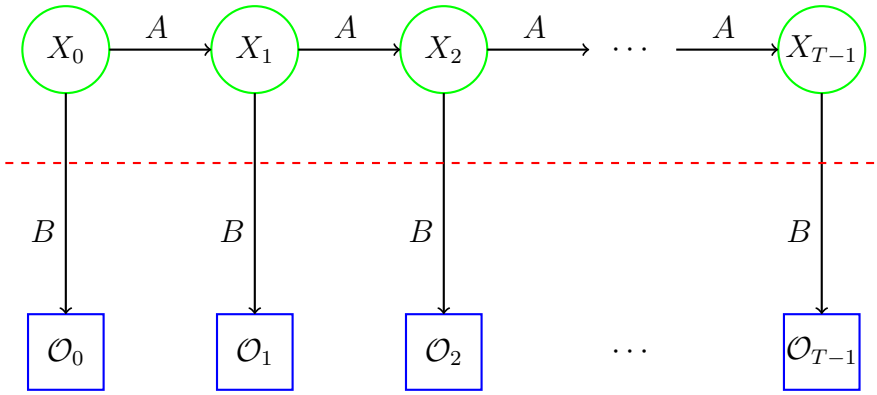

Figure 4: Illustration of HMM

the dashed line represents the hidden part. Here the A matrix drives the (hidden) Markov process, where as the B matrix relates the observations to the hidden states.

3.1 Problems solved using HMMs

First Problem For a model $\lambda = (A, B, \pi)$, given an observation sequence $\mathcal{O}$, we have to find $P(\mathcal{O}|\lambda)$. Here, given the model, we want to find out the likelihood of the occurrence of the observed sequence $\mathcal{O}$. That is, the observation sequence given is scored against the trained model to see if it fits the given model [27].

Second Problem Given a model $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, we need to find out an optimal state sequence for the Markov process. By doing this, we will be actually uncovering the hidden part of the HMM [27].

Third Problem Given an observation sequence $\mathcal{O}$, the dimensions N and M, we need to find a model λ in such a way that it maximizes the probability of occurrence of $\mathcal{O}$. That is, we need to build the model in order to best fit an observation sequence [27].

In the analysis of malicious Java applets we use the algorithm implementation for the first problem discussed above. Here, we train the HMM to represent the sequencet of bytecodes, which is obtained from the malicious Java applets. Then the resulting model is used to score other malware or benign samples against this model.

The solutions for the above three problems discussed are obtained using the Forward algorithm, the Backward algorithm and the Baum-Welch re-estimation algorithm.

The forward algorithm [27], also called as α-pass is used to find the probability of occurrence of the sequence given the model, referred as $P(\mathcal{O}|\lambda)$. It works as follows: For $t = 0, 1, \ldots, N-1$ and $i = 0, 1, \ldots, N-1$, define

$$\alpha_t(i) = P(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_t, x_t = q_i|\lambda)$$

Then the probability of the partial observation sequence up to time t is given by, $\alpha_t(i)$, where the underlying Markov process is in state q_i at time t.

Here, $\alpha_t(i)$ can be computed recursively as follows [27]:

1. Let $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$ for $i = 0, 1, \ldots, N-1$

2. For $t = 1, 2, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, compute

$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ij} b_i(\mathcal{O}_t)$$

3. Clearly we know that, $P(\mathcal{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$.

The backward algorithm [27], also called as β-pass is used to determine a most likely optimal state sequence. It works similar to α-pass, except for the part that it starts at the end and it works backwards to the beginning. This algorithm is as follows. For, $t = 0, 1, \ldots, N-1$ and $i = 0, 1, \ldots, N-1$, define

$$\beta_t(i) = P(\mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \ldots, \mathcal{O}_{T-1} | x_t = q_i, \lambda)$$

Then $\beta_t(i)$ can be computed as follows [27]:

1. Let $\beta_{T-1}(i) = 1$, for $i = 0, 1, \ldots, N-1$.

2. For $t = T-2, T-3, \ldots, 0$ and $i = 0, 1, \ldots, N-1$ compute

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$$

For $t = 0, 1, \ldots, T-2$ and $i = 0, 1, \ldots, N-1$, define

$$\gamma_t(i) = P(x_t = q_i | \mathcal{O}, \lambda)$$

The relative probability up to time t is given by

$$\gamma_t(i) = \alpha_t(i) \beta_t(i) / P(\mathcal{O}|\lambda)$$

From the above definition of $\gamma_t(i)$, we can say that, the most likely state any time t is the state for which $\gamma_t(i)$ is maximum.

Baum-Welch algorithm helps in continuously re-estimating the parameters A, B and π. Here we want to adjust the model parameters so that it best fits the observations. The size of the matrices are constant. That is, the number of states N and number of unique observation symbols M are fixed. However, we can change other elements like A, B and π such that it follows row stochastic condition. The algorithm is as follows:

1. Initialize the model elements A, B and π with values which might be a reasonable guess or any random values. For instance,

$$\pi \approx 1/N, A_{ij} \approx 1/N, B_{ij} \approx 1/M$$

2. Now compute the values of the elements $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$ as seen before. Also compute $\gamma_t(i,j)$ which is referred to as a di-gamma. It can be defined in terms of all these elements as follows:

$$\gamma_t(i,j) = \alpha_t(i)a_{ij}b_j(\mathcal{O}_{t+1})\beta_{t+1}(j)/P(\mathcal{O}|\lambda)$$

   The relation between $\gamma_t(i)$ and $\gamma_t(i,j)$ is:

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i,j)$$

3. Verify that model $\lambda = (A, B, \pi)$ can be re-estimated as follow:

   For $i = 0, 1, \ldots, N-1$ let,

$$\pi_i = \lambda_0(i)$$

   For $i = 0, 1, \ldots, N-1$ and $j = 0, 1, \ldots, N-1$, compute

$$a_{ij} = \sum_{t=0}^{T-2} \lambda_t(i,j) / \sum_{t=0}^{T-2} \lambda_t(i)$$

For $j = 0, 1, \ldots, N-1$ and $k = 0, 1, \ldots, M-1$, compute

$$b_j(k) = \sum_{t=0,1,\ldots,T-1,\mathcal{O}_t=k} \gamma_t(j) / \sum_{t=0}^{T-2} \gamma_t(j)$$

4. if value of $P(\mathcal{O}|\lambda)$ increases, then go to step 3 and repeat the process.

3.2 Malware detection using HMMs

First the model is trained for a particular sequence. In this project it is trained for a sequence of bytecodes extracted from malicious Java applets. By training the HMM we can differentiate between malware and benign data-sets. Any new data set is tested against this trained model. The scores of these benign and malicious data-sets are plotted on a graph and we can see that there is a range of values for which the benign and malicious data-sets do not overlap, this is called the threshold. By using this, malicious data-sets can be separated from those of benign data-sets.

The benign file scores and malicious files scores are used to plot Area Under Curve (AUC) [5]. This AUC will server as our measure of success.

CHAPTER 4

Code Obfuscation

In the internet today we can find many exploit kits which tries to exploit the Java Runtime Environment. Java by itself is the main target when considering the exploits of the web. To exploit it, embed a malicious Java applet inside the web page which when retrieved form the server by the browser, gets downloaded and executed. These malicious Java applets will often be obfuscated to a higher degree so that it evades the signature based detection technique which is used as a technique to detect viruses in most of the antivirus software.

According to the paper [12], if we want to detect the malware which possess obfuscation of its source code, the approach that gives significant results is obtained when performed the dynamic analysis of those samples.

According to the research and experiments done in this project, static analysis using Hidden Markov Model performs better compared to the results of dynamic analysis which is done using HoneyAgent despite various obfuscation techniques used.

In the field of software, obfuscation [21] means deliberately making the code in such a way that, it would be difficult to understand by the humans. We have seen that there are a lot of software which are distributed in executable form. These software can be reverse engineered to find out the vulnerabilities in the application when utilized wisely can cause security breaches. By performing reverse engineering, we can obtain the assembly code from the machine code by the process of disassembling. Various decompilation steps may follow this process to obtain high-level source code from the assembly code.

If we have to complicate this reverse engineering process with respect to Java applets, then the bytecodes of Java applets can be obfuscated [6] using various tools [17]. Malicious applets use obfuscation to evade detection by signature based technique (different kinds of virus detection methods are discussed in next chapter).

Also the problem of reverse engineering can be solved by encrypting the software and decrypting it when its executed [3] or we can also use specialized hardware [19] for this purpose. This approach is effective, but comes with an overhead with respect to performance. Another approach would be to perform code obfuscation.

4.1 Goal of code obfuscation

The goal of code obfuscation is to avoid attacks due to reverse engineering. This can by achieved by makes reconstruction of high-level abstracts difficult and complicated through obfuscation. Section 4.3 gives common code obfuscation techniques used by malwares.

4.2 Difference between compilation and reverse engineering

Compilation is the process of converting the source code into machine code. It involves many steps where each of these steps produces lower-level representations.

Reverse engineering is the process of obtaining high level code similar to source code from machine code. We can broadly divide this process into two steps. Fist one is disassembly. Here we can get assembly code from the machine code of the executable. The second one is decompilation. Here high-level constructs of the software is obtained from the assembly code.

4.3 Code obfuscation techniques in viruses

Code obfuscation is a technique to produce different versions of the same virus so that it can break the signature of that particular virus and hence can evade the signature detection technique used. Hence virus developers commonly performs code obfuscation techniques to avoid detection.

There might be cases where this has worked the other way round. That is, where the implementation of obfuscation techniques helps in detecting the virus. This is possible if there is excess of obfuscation involved. The following subsections explains the techniques used to achieve code obfuscation.

4.3.1 Instruction reordering

In this method, as the name says, code obfuscation is caused by reordering the instructions in random way. This might lead to the concern of control flow. It is taken care by using labels and inserting jump instructions as and where it is needed, so that previous control flow is maintained. Thus obfuscation is achieved here without actually changing the program flow. Since reordering is involved here, it can lead to breaking of the signature. Suppose if there is a lot of reordering, then we can see more umber of jump instructions. The malware detector may consider this as abnormal behavior and detect is as virus.

4.3.2 Subroutine reordering

This is similar to the instruction reordering described in the above section. Here, instead of instructions being reordered, the whole subroutine will be reordered and the control flow of source code would be maintained as before, again by the use of labels and jump statements.

4.3.3 Interchangeable instructions

Here, some of the instructions are being replaced by its equivalent instructions. Here equivalent instruction implies the instructions with similar functionality. This method produces little of metamorphism as the opcode sequence would be different because of the substitution. This method will manage to evade signature detection technique.

4.3.4 Dead code insertion or garbage instructions

The dead code or the garbage instructions will have no effect on the functionality of the program under execution. This can be done by simply inserting dead code or garbage code between the lines of the actual code which contribute for the functionality of the program. Here too much of dead code or garbage code insertion is done is also a problem. It should be done within in the limits of threshold value otherwise, it will be detected by the malware detectors.

4.3.5 Swapping of registers

Here the registers are substituted with registers which are equivalent. Here again the idea is to attempt to obtain different opcode sequence and hence evade the signature based detection technique.

Dynamic analysis of Java applets using HoneyAgent, bypasses all the usual obfuscation techniques. Also we can see that all of these obfuscation techniques described above are currently being used in the wild. As a result of this, it indicates that static analysis might not be reliable in detect malicious activity in recent Java applets. Moreover, obfuscations can be used in benign samples as well, the presence of code obfuscation alone might not be able to conclude that an applet is malicious.

CHAPTER 5

Static and Dynamic Analysis

In this chapter we discuss what is static analysis and dynamic analysis [9]. What are the advantages and disadvantages of each of it [9] and the reason for choosing static analysis for this project.

5.1 Overview of static and dynamic analysis

Static and dynamic analyses emerged from diverse groups and advanced along in parallel however in separate tracks. They have been seen as different areas with researchers expertised in one area or the other.

Static analysis is analyzing and evaluating the application(its code) without actually running it. It inspects the source code and reasons out any behavior that might emerge at run-time. A standard example of static analyses is the compiler optimization [10]. Static analysis works by creating a model which represents the state of the program, then deciding how the program responds to these states. Since there are a numerous way of executing a program, the model is used to keep record of multiple different possible states.

Dynamic analysis is analyzing and evaluating the application at run-time which performs by executing a program and observing the executions. For instance, testing and profiling are dynamic analyses [10]. In dynamic analysis there is less overhead because there is no approximation needed to be done. Here the analysis are used to find the actual, exact run-time behavior of the program.

5.1.1 Primary advantage of dynamic analysis

It uncovers very minute defects, the cause of which would be too complex to be caught by static analysis. The main goal of dynamic analysis is to find the errors and to debug them. It has an important role in security assurance.

5.1.2 Primary advantage of static analysis

It analyzes all the paths and values of the variables that could be possible to be taken during execution and not just those paths and values that would be taken when the program is invoked. Thus we can see that practically, static analysis is capable of revealing errors that may not have been found out with dynamic analysis until weeks, months or years. This part of static analysis is particularly significant in security assurance since any attack related to security is done by running the application in unforeseen and untested ways.

Furthermore, dynamic analysis takes lesser time and can be done as fast as program execution. Some static analyses do run comparatively faster, but in general scenario, getting accurate results takes a lot of computation and hence more time, especially when the program we are analyzing is very large.

The disadvantage of dynamic analysis is that from its outcomes we can not generalize the results for the future executions. Also it has some overhead with respect to run-time [31]. With this into considerations we can say that static analysis is more practical and thus more efficient. Hence we have chosen static analysis as the method to detect malicious java applets and compared the results with dynamic analysis in [12].

5.2 Dynamic analysis of Java applets

According to previous research [12] HoneyAgent has been introduced which performs dynamic analysis of Java applets by overcoming the common obfuscation techniques used. This allows for faster detection of malicious behavior.

In recent approaches [26], to reliably find the exploits caused by obfuscation, the run-time behavior of the applications are preferred to static analysis. There are a lot of tools that are used to inspect the web pages with the help of JavaScript, to examine the PDF documents [26] or Flash programs [11] which are inserted in a web page.

5.2.1 Features of HoneyAgent

The features of HoneyAgent [12] are as follows. It is intended to investigate the run-time behavior of Java applets. It does it by observing how Java applets interact with the default Java Run-time Environment when it is executed. As a result of this, it can detect any operations like file downloads, changes in the file system or even process creation. By intercepting the function calls made when the program is in execution, honey agent safeguards its host system by preventing any changes due to malicious activity. Yet it remains invisible during the process of analyzing the applets. HoneyAgent is capable of identifying malicious Java applets independently without the help of human hands just by applying little of heuristics on the run-time behavior that is observed.

5.2.2 Detection performance of HoneyAgent

Experiments are performed using a total of three sets of Java applets of which two of them are malicious and another set is non-malicious. When the dynamic analysis

was done using HoneyAgent, it was able to detect 96% of malicious samples correctly as malicious, without any false positives for the benign Java applets.

5.2.3 Preventing detection

Malicious applets, to evade the signature based technique of detection, uses code obfuscation to modify the binary footprint of the file. The ways of obfuscation is previously discussed in Chapter 4. These techniques can be used at bytecode level to prevent bytecode from being decompiled [7]. Thus we cannot apply the static analysis method which is present currently based on decompilation [25]. According to previous research [12] it states that code obfuscation is the main reason because of which static analysis does not give better and efficient results.

Also there might be function calls that are encrypted, which will get decrypted at run-time. This makes it very hard to detect the malicious activity of an applet using static analysis techniques.

5.2.4 Requirements to detect malicious behavior at runtime

At run-time the code in examination interacts with the run-time environment. This is the major challenge in dynamic analysis of Java applets. Therefore, to simulate and observe the malicious behavior it is necessary for us to provide an environment which is similar to the one an applet would need on a system which is vulnerable. As well it is necessary to hide the code which is used for analysis purpose as far as possible.

On the contrary, providing an environment is not necessary in static analysis since the code will not be executed, instead the code would just be statically analyzed.

5.3 Functionality of HoneyAgent

HoneyAgent has several functionalities [12]. The subsequent subsections gives a brief description of them all.

5.3.1 Bytecode instrumentation

One of the features of Java's common security component is verification of the bytecodes. The purpose of this is, it processes the bytecodes even before it is fed into JVM. If there is any loophole in the verification part of the bytecode, then can in some cases cause invalid bytecodes to be fed into JVM. This provides an opportunity for malicious Java applets to perform malicious activity.

Now with newer Java versions which have improved version of bytecode verifier are capable of detecting invalid bytecode or its sequence thus avoid the execution of that part of the code which is flagged invalid in the JVM. Because of this feature, HoneyAgent does not get to analyze the complete behavior of the respective applet. But HoneyAgent overcomes this by instrumenting the bytecodes(replaces the invalid or malicious part of bytecodes with valid ones) of the Java applets even before it is checked by the Java's security verifier.

5.3.2 Determining dynamic run-time interception

HoneyAgent does run-time analysis by intercepting some of the system calls made to the API methods. This is done by observing the applet especially the stack frame content at the time of execution. This leads the dynamic run-time analysis to explain the behavior of the applets which were examined without the overhead of run-time which comes into picture due to breaking of the method calls.

5.3.3 Detecting malicious behavior

As we have seen before, HoneyAgent does dynamic run-time analysis of Java applets and finds out from their activity or behavior if they are malicious or benign. This leads to the need for defining what kind of behavior is actually considered as malicious. Usually the applet data-sets are run inside a sandbox, thus restricting its interaction with the host. Malicious applets here tries to interact with the host. There are a number of interactions that these applets can try to perform. Thus we can say that an applet is malicious if it is trying to do some activities which are prohibited.

Following is the list of prohibited activities [29] which when performed is considered as malicious. If the data-set tries to access resources of the clients, for instance, file system. If the sample tries to download or get some contents form third-party server. If it tries to load the native library files. If it tries to modify or disable the security manager of the system. It also should not try to create a class loader or read some of the properties which are considered restricted.

5.4 Drawback of HoneyAgent

There are security concerns when dynamically analyzing applets using HoneyAgent [12]. There are limitations when compared to other dynamic analysis performed [12]. The subsequent subsections gives an outline on all these drawbacks.

5.4.1 Security

When we are dynamically analyzing applets with the help of HoneyAgent they will be interpreted by a JVM which is deployed on the actual host. Because of this, there is a possibility that this JVM is vulnerable to the malicious activities performed by the applets in execution.

5.4.2 Limitation

Major drawback of HoneyAgent compared with other approaches of detecting malicious Java applets is that, only the current path of execution is being observed here. There might be other existing functionality which might lead to malicious behavior that is not being used by applet at current execution which might go unobserved during the run-time analysis.

5.4.3 Evasion

During execution of applets, they can try to see if they are analyzed by HoneyAgent. If they detect the present of it, the applets might try to suppress its malicious activity. The common place where the applets looks for the agent is the current stack trace.

5.4.4 Specific data-set

In the previous research using HoneyAgent only unsigned applets are considered for analysis purpose. This is because the applets which are signed with a valid certificates executes without any concern. It will not have any kind of restrictions. In this case applets execution behavior is similar to that of a typical Java application. Hence these applets which are signed are allowed to perform any activity which we considered prohibited in the previous section. Because of this feature, it gets difficult to distinguish between malicious and non-malicious applet using HoneyAgent. Thus they have considered only unsigned applets.

5.5 Data-set

One set of data-set used in dynamic analysis of Java applets using HoneyAgent are downloaded from Virustotal website. The other set is taken from Contagio malware dump. For the purpose of static analysis, same set of data is being used.

5.6 Detection rate

HoneyAgent was able to successfully determine malicious activity in 94% of all samples which were taken from the Contagio data-set and 97% in the Virustotal data-set. This was done without generating any false positives for the applets which are benign. On an average, the detection rate of malicious Java applets comes up to 96% .

CHAPTER 6

Experimental Results

Java applets consists of compiled Java classes which are usually bundled in a jar archive similar to any usual Java application. We carried out static analysis on these jar archives of malicious Java applets with Hidden Markov Model using the same data sets as used in dynamic analysis using HoneyAgent [12], discussed in Chapter 5.

Using Hidden Markov Model discussed in Chapter 3, we train the model for the particular sequence. Here the sequence would be the sequence of bytecodes appearing in the class files present in the jar archive corresponding to the malicious Java applets. That is, the applets we work on are in the form of .jar files from which we extract the contents which are a set of .class files. From these .class files we extract the bytecodes and use it to train the model.

6.1 Extracting the contents of jar file

The command which is used in extracting the components of JAR file is

jar xf <jar-file>

The options and arguments in this command line are as follows. *x*, it is an option which is used to *extract* the jar files. *f*, it is an option which says that the name of the jar file whose contents are to be extracted has to be given in the command line. *jar-file*, is an argument that indicates the *filename* of the jar file. We can also give a path to the filename as a value for this argument.

In the above command, order in which the options x and f are used will not matter. But we have to make sure that they are not joined together, that is there is a

space between them. The Jar tool works by making copies of the required files when you are extracting the files. It then writes the copied files to the current directory. Here the original JAR file is not modified.

6.2 The bytecodes

We know that Java source code is compiled into bytecodes, which are present in the class file. Bytecodes are the key reason for Java's platform independence. From the section 6.1 we know how to get the class files from the jar archives which correspond to Java applets.

Now, We can use *javap* tool which acts as a disassembler for one or more Java class files to view the bytecodes. The output of this command depends on the options that are used with it.

We use the following command with the option 'c'.

javap -c <class-file-name>.class

The options and arguments in this command line are as follows: +extitc, this option prints out the instructions for each of the methods in the class. These instructions consists of Java bytecodes. Basically this option is used for printing the disassembled code. *class-file-name*, this argument is name of the class file and it must have the extension of .class. Figure 5 is the screen-shot of how the output looks when you run the above command. The output of javap is always printed to stdout.

First capture the output of *javap -c* into a text file. From the above figure we know that along with bytecodes, we do get other information as well. Using simple python script with the help of regular expression we can extract just the bytecodes and copy it to a file. These files containing bytecodes are used in training the HMM,

```
public DocFooter();
  Code:
     0: aload_0
     1: invokespecial #1                  // Method java/applet/Applet."<init>":()V
     4: return

public void init();
  Code:
     0: aload_0
     1: sipush        500
     4: bipush        100
     6: invokevirtual #2                  // Method resize:(II)V
     9: aload_0
    10: aload_0
    11: ldc           #3                  // String LAST_UPDATED
    13: invokevirtual #4                  // Method getParameter:(Ljava/lang/String;)Ljava/lang/String;
    16: putfield      #5                  // Field date:Ljava/lang/String;
    19: aload_0
    20: aload_0
    21: ldc           #6                  // String EMAIL
    23: invokevirtual #4                  // Method getParameter:(Ljava/lang/String;)Ljava/lang/String;
    26: putfield      #7                  // Field email:Ljava/lang/String;
    29: return
```

Figure 5: Output from javap -c JavaApp.class

i.e., for the static analysis of the applets.

So now, for each class file we have a file which contains only its corresponding bytecodes (no other information except bytecodes).

6.3 Results using Contagio data-set

With this experiment, the AUC obtained from static analysis of the data-set from Contagio malware dump is 0.95156 . It is as shown in the Figure 6. In comparison, dynamic analysis using HoneyAgent [12] was able to successfully detect malicious behavior in 94% of all samples from the Contagio data set without generating any false positives for the benign applets. This implies that in this scenario, static analysis is efficient and gives better result than dynamic analysis.

6.4 Results using Virustotal data-set

The AUC obtained from static analysis of the data-set from VirusTotal website is 0.77 .It is as shown in the Figure 7. In comparision, dynamic analysis using HoneyA-

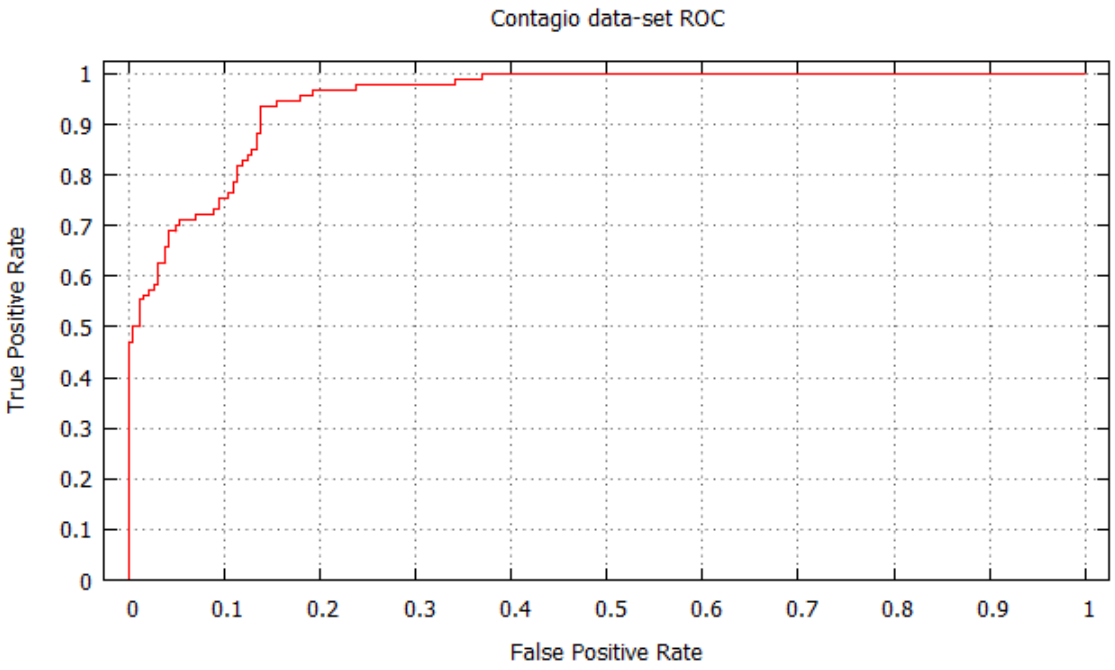

Figure 6: AUC for data-set from Cantagio website

gent [12] was able to successfully detect malicious behavior in 97% of the Virustotal data set without generating any false positives for the benign applets. Assumption for the difference here in static analysis is, a lot of dead code being injected or more percentage of code obfuscation in the data-set downloaded from Virustotal, because of which adjacency of bytecodes will differ a lot.

So the total result of 94% detection rate with respect to sample from Contagio dump and 97% detection rate with respect to sample from Virustotal website, the results come to total detection rate of 96% in average in dynamic analysis.

We even did analysis outside the data-set specified here. We downloaded extra sample from VirusTotal website with the search criteria matching the previous search as used in downloading VirusTotal data-set. We searched for "type:jar p:+5 and not type:android" and downloaded top 100 samples. This search implies that we are

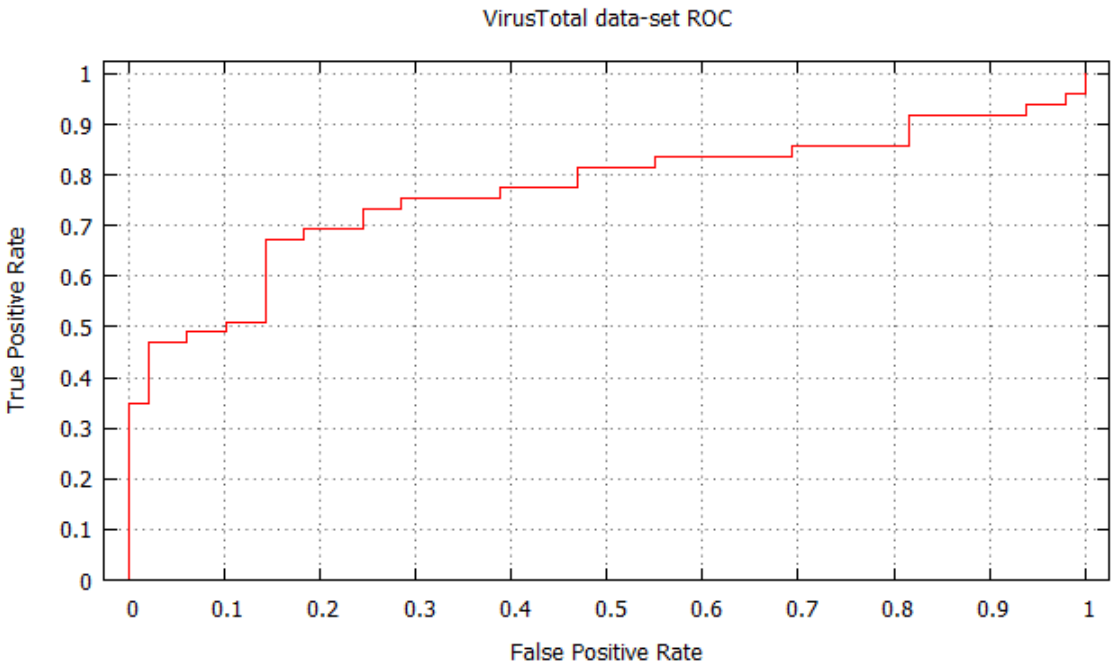

Figure 7: AUC for data-set from Virustotal website

searching for malicious Java applets which are detected by atleast five anti-virus and those applications which are not android. When we did the static analysis on this extra data-set, the AUC obtained is 0.905.

6.5 Deleting uncommon bytecodes

For the purpose of getting better results we tried training the HMM by manipulated data-set. That is, we deleted uncommon bytecodes whose frequency of occurrence is less than or equal to different range. Table 2 gives the outcome of this action performed on VirusTotal data-set and its graph is given in Figure 8. Table 3 and Table 4 gives the results of this action performed on Contagio data-set and extra data sample downloaded from VirusTotal respectively. Also Figure 9 and Figure 10 gives the graphs of the data given in Table 3 and Table 4 respectively.

Table 2: AUC after deleting uncommon bytecodes from VirusTotal data-set

Number range	% removed	AUC
0	0%	0.772
6	0.55%	0.795
12	1.07%	0.806
13	1.18%	0.808
14	1.25%	0.804
15	1.33%	0.8
16	1.14%	0.796
20	1.72%	0.78
30	2.56%	0.758
40	3.38%	0.725
80	5.95%	0.664
100	6.86%	0.643
400	18.42%	0.453

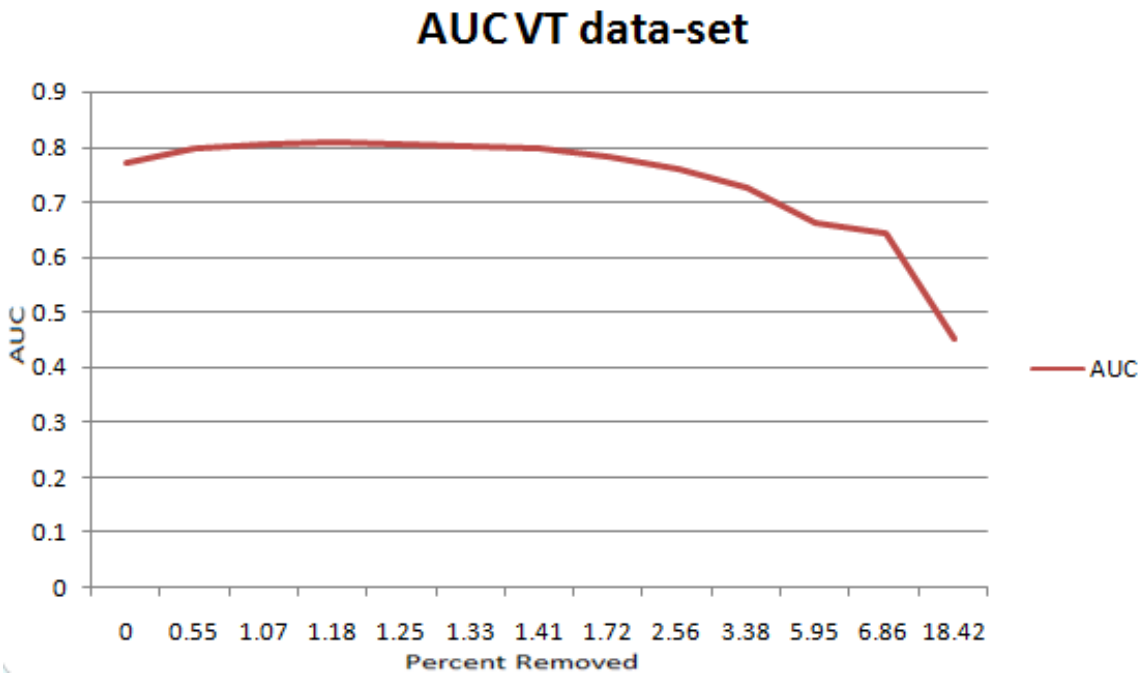

Figure 8: AUC after deleting uncomman bytecodes from VirusTotal sample

From this experiment we can observe that area under ROC curve (AUC) increased to 0.808 after deleting bytecodes whose frequency of occurrence is less than

Table 3: AUC after deleting uncommon bytecodes from Contagio data-set

Number range	% removed	AUC
0	0%	0.952
1	0.014%	0.951
2	0.06%	0.932
6	0.24%	0.926
12	0.5%	0.916
20	0.94%	0.914
50	2.26%	0.851
100	4.04%	0.789

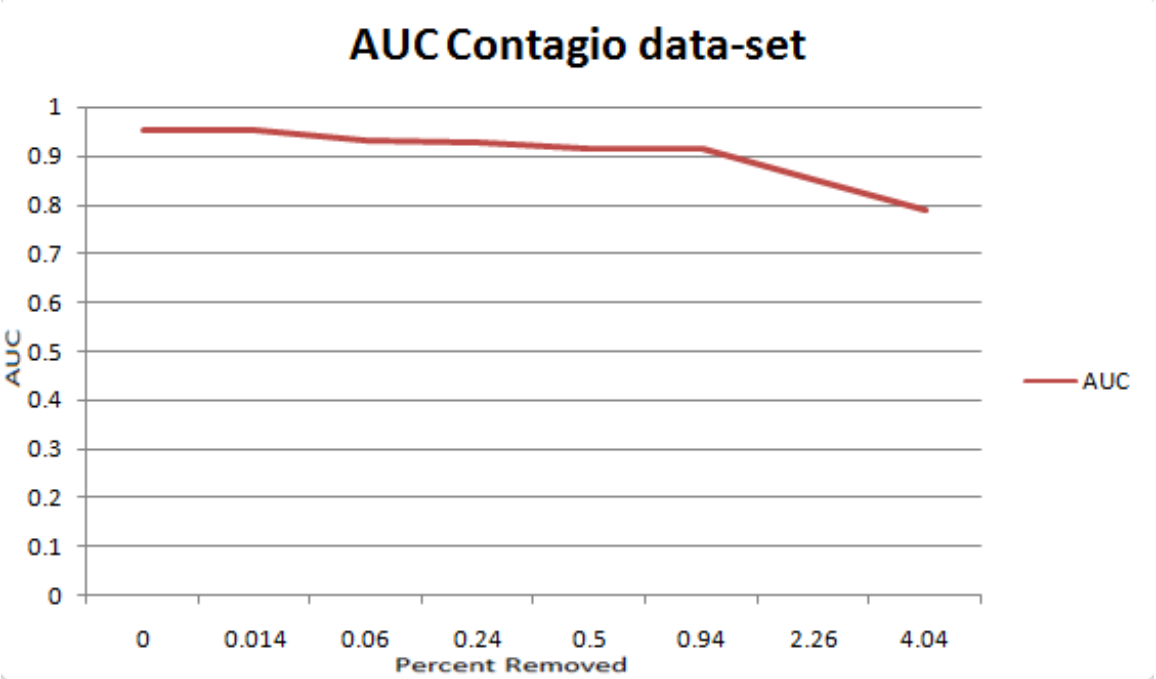

Figure 9: AUC after deleting uncomman bytecodes from Contagio data-set

or equal to 13 in VirusTotal sample. This is the maximum that the AUC reaches, after which there is decline in its value. With respect to data-set from Contagio, when uncommon byecodes are deleted, there is a constant decline in the value of AUC which was initially 0.952. Where as with the extra data-set, AUC value increases to a very small extent to 0.91 when a very small fraction of bytecodes(0.02%) were removed.

Table 4: AUC after deleting uncommon bytecodes from extra sample downloaded from VirusTotal

Number range	% removed	AUC
0	0%	0.905
1	0.02%	0.91
2	0.08%	0.902
3	0.1%	0.901
6	0.4%	0.848
12	0.84%	0.805
20	1.56%	0.779
50	3.95%	0.697
100	6.87%	0.567
400	25.2%	0.38

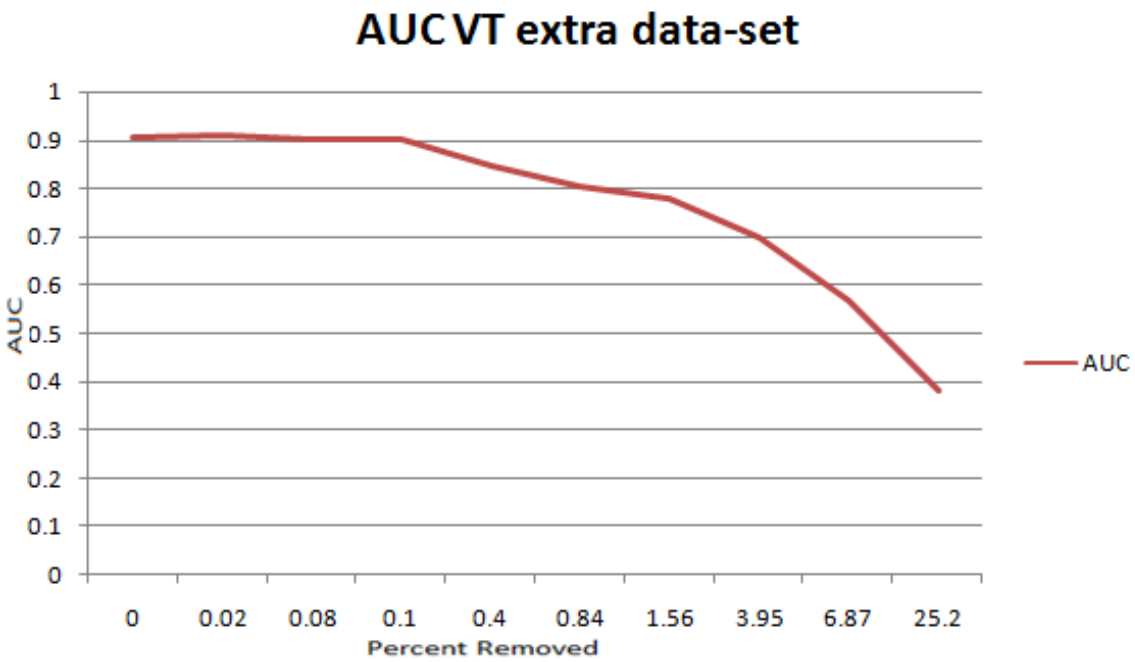

Figure 10: AUC after deleting uncomman bytecodes from the extra sample downloaded from VirusTotal

After which we can see that there is again decrease in its value.

This shows that if we delete the uncommon bytecodes that are used in training

the model, the accuracy of detecting the malicious applet increases to some extent until certain point. We can say that, this will help increase the detection rate of malware.

Now when we compare the detection rate of malicious applets obtained from dynamic analysis of Contagio dump which is 94% and the one with static analysis, the AUC is 0.952. We can say that detection rate of static analysis is 95.2% and hence with respect to this data-set static analysis performs better than dynamic analysis. Where as with VirusTotal data-set, dynamic analysis result is 97% and static analysis result after deleting uncommon bytecodes, the maximum AUC obtained is 0.808 and hence we can say that, detection rate of static analysis her is 80.8% . In the static analysis done with extra data-set, the maximum AUC obtained after deleting uncommon bytecodes is 0.91 and thus we can say that here the detection rate is 91% . Figure 11 gives the graph of this description.

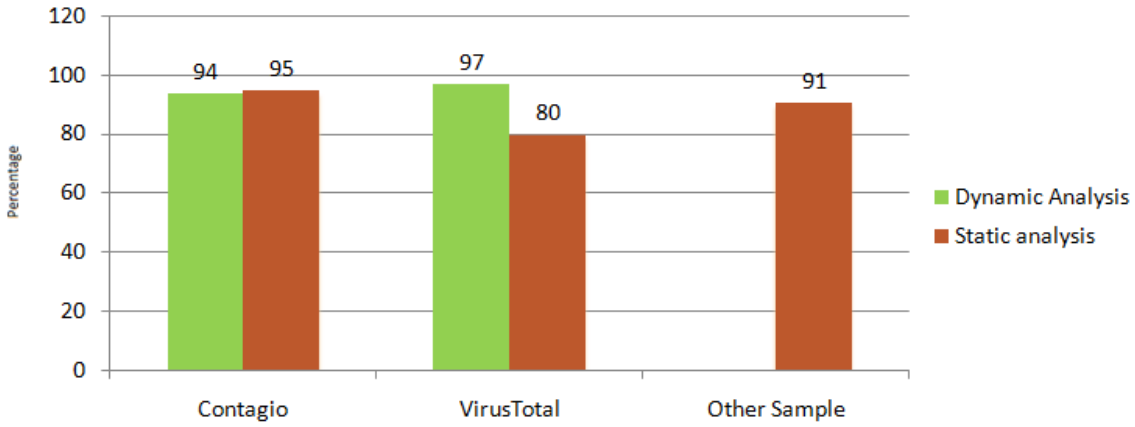

Figure 11: Comparison of static analysis results after deletion of uncommon bytecodes with dynamic analysis results

CHAPTER 7

Conclusion and Future Work

In this work we presented experiments based on static analysis of malicious Java applets using Hidden Markov Model and compared results to the dynamic analysis when working with the same data-set as that used by dynamic analysis using HoneyAgent. We also performed static analysis on extra data sample other than that used in dynamic analysis using HoneyAgent. We also deleted uncommon bytecodes before training the model and observed that we get better results when certain percentage of the bytecodes are removed, compared to training the model with all the bytecodes.

Through the experiments conducted here, we can say that static analysis of malicious Java applets in some cases give better results than dynamic analysis. Since static analysis is better and efficient, and also gives better results, this method should be preferred to dynamic analysis with respect to analysis of Java applets. In certain cases static analysis might not perform better if the sample that we are using to train the model has more of dead-code injection or a high degree of code-obfuscation due to which bytecode adjacency might differ.

These experiments are conducted with limited set of Java applet samples. Availability of malicious Java applets data-set in the internet is very difficult. If we get more set of samples and perform these experiments extensively on all those data-sets and if we get good results in all of these we can say that static analysis is preferred to dynamic analysis most of the times. So future work would be to collect more data samples and perform the static analysis. Also in this project we have considered static analysis only using Hidden Markov Model. We can also use other static analysis

methods for performing the experiments. If they do not give good results, then their scores can be combined to analyze the detection rate.

LIST OF REFERENCES

[1] S. Abraham and I. Chengalur-Smith, An overview of social engineering malware: Trends, tactics and implications, *Technology in Society*, vol. 32, no. 3, pp. 183-196, 2010.

[2] C. Annachhatre, T. H. Austin and M. Stamp, Hidden Markov models for malware classification, to appear in *Journal of Computer Virology and Hacking Techniques*,
`http://link.springer.com/article/10.1007/s11416-014-0215-x`

[3] D. Aucsmith, Tamper resistant software: an implementation, *Information Hiding: First International Workshop*, 1174(1):317–333

[4] E. S. Boese, *An Introduction to Programming with Java Applets*, third edition, Jones 'I&' Bartlett, 2009

[5] A. P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms,
Pattern Recognition, 30(7):1145–1159, 1997

[6] J. -T. Chan and W. Yang, Advanced obfuscation techniques for Java bytecode, *Journal of Systems and Software*, 71(1):1–10, 2004

[7] V. Diaz, Anti-decompiling techniques in malicious Java Applets,
`https://securelist.com/analysis/37162/anti-decompiling-techniques-in-malicious-java-applets/`

[8] E. Daoud and I. Jebril, Computer virus strategies and detection methods, *International Journal of Open Problems in Computer Science and Mathematics*, 1(2):29–36, 2008

[9] Dynamic Analysis vs. Static Analysis, by Intel `https://software.intel.com/sites/products/documentation/doclib/iss/2013/inspector/lin/ug_docs/index.htm#GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm`

[10] M. D.Ernst, Static and dynamic analysis: synergy and duality, 2003, `https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf`

[11] S. Ford, M. Cova, C. Kruegel and G. Vigna, Analyzing and detecting malicious flash advertisements, *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, Honolulu, HI, 2009

[12] J. Gassen and J. P. Chapman, HoneyAgent: Detecting malicious Java applets by using dynamic analysis, *Proceeding of the 9th International Conference on Malicious and Unwanted Software* (MALCON 2014), Fajardo, Puerto Rico, October 28-30, 2014

[13] Hidden Markov model, `http://en.wikipedia.org/wiki/Hidden_Markov_model`

[14] T. Katsuki, Crisis: The advanced malware, 2012, `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/crisis_the_advanced_malware.pdf`

[15] S. Kazi and M. Stamp, Hidden Markov models for software piracy detection, *Information Security Journal: A Global Perspective*, 22(3):140–149, 2013

[16] A. Krogh, An Introduction to hidden Markov models for biological sequences, *Computational Methods in Molecular Biology*, 46–63, 1998

[17] E. Lafortune, ProGuard, `http://proguard.sourceforge.net/`

[18] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, Anatomy of drive-by download attack, *Proceedings of the Eleventh Australasian Information Security Conference-Volume 138.* Australian Computer Society, Inc., 2013, pp. 49-58

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, Architectural support for copy and tamper resistant software, *Proceeding of 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, pages 168-177, New York, NY, November 2000

[20] A. Neagu, The top 10 Most Dangerous Malware That Can Empty Your Bank Account, `https://heimdalsecurity.com/blog/top-financial-malware/`

[21] Obfuscation(software), `http://en.wikipedia.org/wiki/Obfuscation_(software)`

[22] OECD, Malicious Software (Malware): A Security Threat to the Internet Economy. `http://www.oecd.org/dataoecd/53/34/40724457.pdf`

[23] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, 77(2):257–286, 1989

[24] H. Rana and M. Stamp, Hunting for pirated software using metamorphic analysis, *Information Security Journal: A Global Perspective*, 23(3):68–85, 2014

[25] J. Schlumberger, C. Kruegel and G. Vigna, Jarhead analysis and detection of malicious Java applets, *Proceeding of the 28th Annual Computer Security Applications Conference*, New York, 2012

[26] F. Schmitt, J. Gassen and E. Gerhards-Padilla, Pdf scrutinizer: Detecting javascript-based attacks in pdf documents, *Privacy, Security and Trust(PST), 2012 Tenth Annual International Conference on*, Paris, 2012

[27] M. Stamp, A revealing introduction to hidden Markov models, 2012, `http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf`

[28] M. Stamp, *Information Security: Principles and Practice*, second edition, John Wiley'I&' Sons, Inc. , 2011

[29] What Applets Can and Cannot Do, by Oracle `http://docs.oracle.com/javase/tutorial/deployment/applet/security.html`

[30] W. Wong and M. Stamp, Hunting of metamorphic engines, *Journal in Computer Virology*, 2(3):211–229, 2006

[31] S. H. Yong, S. Horwitz, Using Static Analysis to Reduce Dynamic Analysis Overhead, *Formal Methods in System Design*, 27(3):313–334, 2005

[32] R. M. Yorston, Presenting Archaeological Information with Java Applets, *Proceeding of the 25th Anniversary Conference*, University of Birmingham, April 1997