ABSTRACT

This project implements and evaluates a bounded feature space behavior modeling (BOFM) framework for scalable malware detection. BOFM models the interactions between software (which can be malware or benign) and security-critical OS resources in a scalable manner. A malware, performs various actions on one or more OS resource instances. In the proposed BOFM, for each OS resource type, the set of actions performed by malware on an individual OS critical resource instances are monitored, and based on these collected features, BOFM algorithm can decide whether it is a malware or a benign application. The approach which is proposed is scalable, which is achieved by placing an upper bound to the number of the feature extracted.

The document mainly covers about the background, previous research, motivation from the previous research, architecture of the proposed solution, functionality of the application which is designed to detect the malwares in online applets and JAR files, different test cases of the project and finally conclusion and future work.
# TABLE OF CONTENTS

Abstract ........................................................................................................................................ ii

Table of Contents .................................................................................................................. iv

List of Tables ......................................................................................................................... vi

List of Figures ........................................................................................................................ vii

1. Background and Rationale ......................................................................................... 1

   1.1 Attacker Tools .......................................................................................................... 2

      1.1.1 Backdoor ......................................................................................................... 3

         1.1.1.1 Zombies .................................................................................................. 3

         1.1.1.2 Remote Administration Tools ............................................................. 3

      1.1.2 Rootkits ......................................................................................................... 4

   1.1.3 Web Browser Plugins ..................................................................................... 4

   1.1.4 Non Malware Threats .................................................................................... 4

2. Previous Research and Motivation ............................................................................. 6

   2.1 Previous Research ............................................................................................... 6

   2.2 Motivation .......................................................................................................... 9

3. Architecture ..................................................................................................................... 11

   3.1 Bounded Feature Space Behavior Modelling ................................................... 11

   3.2 Upper Bound ...................................................................................................... 14

      3.2.1 Hypothesis .................................................................................................. 15

   3.3 Flow Diagram for the Proposed System ........................................................... 15

   3.4 Input and Output Flow Diagram ....................................................................... 17

      3.4.1 Flow Diagram if the input is a JAR file ....................................................... 17
3.4.2 Flow Diagram if the input a URL of Applet

3.5 Implementing BOFM in JAVA Based Threat Detector

3.6 Use Case Diagrams

3.7 Project Development Phases

3.8 Environment

3.8.1 Java, HTML Parse: Jsoup

3.8.2 ASM – Java API

4. Functionality of the Application

4.1 Interface of the Threat Detector Application

4.1.1 Java Based Threat Detector

4.1.2 Different Scanning Options

4.1.3 If the selected options is ‘Computer’

4.1.4 Scanning an Online Applet

4.1.5 Copying the URL of the Applet

4.1.6 Entering the URL in the PopUp window

4.1.7 If the selected option is ‘Scan JAR file’

4.1.8 JAR file selected from hard drive

4.2 Malicious activities of JAR file and Online Applet

5. Testing and Evaluation

5.1 Test Cases

6. Conclusion and Future work

Bibliography and References
LIST OF TABLES

Table 3.1: OS RESOURCES AND CORRESPONDING ACTIONS ..................................13
LIST OF FIGURES

Figure 2.1: Different Users affected by malware in the year 2014 ..........................8
Figure 2.2: Countries with highest number of malware attacks .........................9
Figure 3.1: Flow Chart for the proposed system ..................................................16
Figure 3.2: Flow Diagram if the input is a local JAR file .....................................17
Figure 3.3: Flow Diagram if the input is a URL of online applet .........................18
Figure 3.4: Use Case Diagram 1 ...........................................................................20
Figure 3.5: Use Case Diagram 2 ...........................................................................20
Figure 4.1: Simple Java Applet ...........................................................................24
Figure 4.2: Java Based Threat Detector Home Window .....................................25
Figure 4.3: Different Scanning Options .................................................................26
Figure 4.4: If the selected option is ‘Computer’ ....................................................27
Figure 4.5: PopUp window with message ............................................................27
Figure 4.6: If the selected option is ‘Check Online Applet’ .................................28
Figure 4.7: Pop Up displayed ...............................................................................28
Figure 4.8: Copying the URL of the Applet ..........................................................29
Figure 4.9: Copied URI entered into Pop Up window .........................................29
Figure 4.10: If the selected option is ‘Scan JAR file’ .............................................30
Figure 4.11: JAR file selected from hard drive ....................................................31
Figure 5.1: List of JAR and Class files generated ...............................................34
Figure 5.2: Downloading a JAR file from the web .............................................35
Figure 5.3: Scan the JAR file from the local machine ...........................................36
Figure 5.4: Result obtained after scanning the JAR file ........................................38
Figure 5.5: Downloading a JAR file from the website ........................................39
Figure 5.6: Scanning the JAR file from the local machine.................................40
Figure 5.7: Result obtained after scanning the non malicious JAR file...............41
Figure 5.8: Field to enter the URL of the online applet.................................42
Figure 5.9 Copy the URL of the online applet...............................................42
Figure 5.10: The URL of the online applet is entered .....................................43
Figure 5.11: Result obtained after scanning the online applet .........................43
Figure 5.12: Copy the URL of an online Applet ............................................45
Figure 5.13: URL of the online Applet is entered ........................................45
Figure 5.14: Results obtained after scanning the online Applet .......................46
1. BACKGROUND AND RATIONALE

Exponential development of malware (noxious programming) is a real risk in the software industry. Given the alarming advancement of malware, a considerable measure of investigation has focused on proposing diverse malware detection methods to direct this issue. The malware detection strategies can be divided into two general classifications: static and dynamic malware detection (static is also called signature based and dynamic is also called behavior based malware detection). Signature based malware detection has more advantages than behavior based malware detection, because signature based detection technique inspects the static content of the malicious code and hence, has the capacity to accomplish full code coverage. By using signature based malware detection technique, it is even conceivable to discover malicious applications even before executing. The major limitation and restriction of signature based malware recognition is that it can be effortlessly avoided by obfuscation techniques [6]. Even though this technique has limitation, it is one of the predominant malware detection techniques till date. [8]

In order to overcome the limitations of signature based malware detection technique, many security analysts have proposed several behavior based malware detection techniques [7, 8, 9]. The proposed techniques, mainly concentrate on the semantics of the malicious applications. Specifically, behavior based malware detection also inspect the run-time behavior of the malicious binary and examine the system calls which are to be invoked during execution. Some of the behavior based malware detection technique includes bags of system calls [3] and sequence of system calls like the n=gram
model [7, 3]. However, scalability (Size of extracting malware feature from the execution trace) of the existed behavior based malware detection techniques is highly tricky, which is one of the main drawbacks of the existing malware detection models are, it requires a high computational complexity and consumption of memory is high. For example, to extract the specification of the malware using an algorithm, it would take around 12 to 48 hours, which is practically not acceptable.

The proposed malware detection technique is BOFM; this technique captures the malicious behavior of the malware, like the interaction between the malware and security-discriminating OS resources in a versatile way (scalable manner) and it can recoil or spring back from the basic obfuscation techniques [6].

By using BOFM which is a behavior based malware detection technique, the extracted feature space is of fixed measurement, which is achieved by placing an upper bound [10], it does not increase with the number of malware examples (samples) under examination [10] and is three orders of size smaller than the best reported systems for malware behavior modelling. Accordingly, calculation time and memory utilization for extracting malware characteristics and malware detection are greatly reduced.

1.1 Attacker Tools [14]:

Attacker tool is nothing but a part of the malware infection. The attacker tools which are of the type malware are used by the attackers to attain an unauthorized access to the system resources and confidential data. The attacker tools can be delivered to a system by the malware or it can be transferred to a system or host, after the attack occurs. For example, if a system is infected by a worm, then the system which is infected by the
worm is directed to perform some malicious action like downloading malicious software from the web, and installing those malicious software on your system. Some popular attacker tools are discussed below.

1.1.1 Backdoor:

The general term which is used instead of backdoor is a malicious program, which will listen for certain command using TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Most of the backdoor malicious programs will have a client and a server. The intruder can remotely perform some actions on the computer which is infected, and the remote intruder will have some degree of control of the infected system. Some other important capabilities of backdoor are.

Zombies:

A zombie is a backdoor malicious application which is installed on a system to attack other systems. Using the zombie, an intruder can give commands remotely to many of the agents at a time, so the agents which received that remote command will perform a coordinated attack against a system [17].

Remote Administration Tools:

These are used to get access to the system completely by the remote intruder. Some Remote Administration Tools will attain complete access to the system, like attaining control over devices of the system like webcams, speakers and monitors every thing that appears on the screen of the system [17].
1.1.2 Rootkits:

Rootkits are used to tamper the standard system functionalities. On windows operating system, rootkits have the ability to modify files that are in the memory and it can modify the built-in OS system calls. If many changes are made to the rootkit it can preserve or hide the existence of rootkits and it can even hide the changes made to the system calls. Mainly rootkits are used to install different attacker tools [17].

1.1.3 Web Browser Plug-ins:

Web browser plugins are nothing but a software that will add some additional functionalities to the Web browser. Typically, they are used to add toolbars and to support different video, audio and graphic formats. Attackers will create a malicious plugin that acts as a spyware, when this spyware in installed on your browser, the plugin can monitor all the pages and websites which the user visits [17].

1.1.4 Non Malware Threats:

The different forms of non-malware threats are phishing and Virus Hoaxes. Both phishing and virus hoaxes are used by the attacker to fetch some sensitive data from the user, by downloading some unwanted files and executing them on the victim's system which appears to be benign but they are actually malicious [17].

**Phishing**

In the phishing attack the common technique which is used to download malicious software to the victim's system is by banner advertising or pop up window. If the user clicks the popup window, it will let the malicious software downloaded to the user system. And this will allow accessing the personal data, passwords of all the websites which the user is interacting [17].
Virus Hoaxes

Virus hoaxes are nothing but false alarms. Even though hoaxes may not cause any adverse effects, some type of hoaxes performs malicious actions like, directing the user to “alert some OS settings or delete the file” which could lead to “Security and Operational problems” [17].
2. Previous Research and Motivation

2.1 Previous Research

The study by Canali [3] on behavior based malware detection is most related to the current work, after doing a lot of study and analyzing different behavioral malware detection techniques, they designed this malware detection techniques consisting of three dimensions.

(1) Granularity of the model elements – system calls (with and without parameters) at various levels.

(2) Basic model elements relationship.

(3) Cardinality of each element.

The experimental result shows that it achieved 99% detection rate and 0.4% false positive rate [3]. Even though this has a high detection rate, the main issue about this technique is the scalability of this approach which is precarious and this technique will develop large feature space and it is more problematic. Another important practical limitation of this approach is that memory consumption is high [3].

The other behavior based malware detection technique proposed by Lanzi [7] which is based on a system-centric malware model. In this model benign sample and the OS resources interaction is monitored and it is modelled as a system-centric manner.

Other malware detection method which is proposed by Kolbitsch et al. [11] is a detection process at the end host and this detection process is efficient and effective [18]. In this model, malware detection is done using graph matching at the end host. The main drawback of the proposed approach is, it works only with a limited number of samples.
In the same way other approach which is proposed by Fredrikson et al. [12] is a malware detection process using dependency graph mining. This approach has a high detection rate, but the main drawback is that “computationally very intense”.

“A layered malware detection model using VMM” [13] is other malware detection modelling technique; in this technique malware is detected using virtual machine based anti-malware system. The main drawback of this approach is on the top of virtual machine monitor there are two other virtual machines, one is the guest OS (user) and the other is the host OS monitor, the user cannot directly monitor the guest and not even configure the mechanism which is used to detect, in the guest OS.

The key observations which are made from the above mentioned detection approaches are [3, 7 and 8]:

1. The general malware detection technique is coined as four dimensions.
   - Size of feature space
   - Computational complexity
   - Overhead
   - Detection accuracy

2. Dependency and behavioral malware model are complex and they generally require high computation time.

3. The malware detection models are simple, but they require huge feature space. So to overcome this problem they require a different mechanism which is described in the proposed solution.
In the above mentioned malware detection techniques, there are several drawbacks associated with these techniques like the feature space is more, the approach is computationally very intense, the proposed solution for malware detection is not scalable. So based on these factors the proposed solution is “Malware detection using a bounded feature space behavioral modelling” [10] this approach is computationally not very intense, scalable and the feature space is also limited.

The statistical reports for different types of user who are affected in 2014 are:

![Figure 2.1 Different Users affected by malware in the year 2014 [15]](image-url)
Figure 2.2 represents the top 15 counties around the world which had the highest malware attacks in the year 2013 to 2014.

![Figure 2.2 Countries with the highest number of malware attacks][16]

### 2.2 Motivation

A lot of studies prove that, the effect of malware on the financial sector is becoming worse day by day. So it is very important to know the different types of malwares, the adverse effects which are performed by the malware and the different detection techniques. From the figure 2.1 it is obvious that the malware attacks are more in the financial field than compared to the general users and figure 2.2 portraits the countries with highest malware attacks in 2014.

Malware is a malicious code which performs some unwanted and unnecessary action to the host (system) and this may lead to some adverse effects like, loosing confidential
data, interrupting the default OS related system calls and this may lead to damage of the
host machine, and there will be a lot of cost which is associated with replacing the
damaged system with a new one or, by repairing it.
3. ARCHITECTURE

3.1 Bounded Feature Space Behavior Modelling (BOFM):

A malware performs different activities on one or more OS resources. The proposed system BOFM monitors the interaction between the malware and OS critical resources. For every OS resource type “Activities which are performed by malware on an individual resource will constitute a feature of the malware” [10].

Different Operating System Resource Types which BOFM will keep track are [10]:

- File system: Programs which run on the OS are composed of individual files which are nothing but an opened file or I/O resource.

- Registry: Application and some system components store data and can retrieve that data from the registry which acts as a repository for the applications.

- Process and Thread: Set of threads needs a virtual address space and the control information. So Process will provide the control information and virtual address space for the thread which are under execution.

- Network: Network related operations are performed.

- Synchronization: By using multiple threads or processes, shared resources can be protected.

- Section: Section is nothing but a shared memory, where a process can share its address spaces with other processes by using sections.
The key properties of BOFM:

- Property 1: If the same set of actions are being performed many times on similar type of OS resources, then the set of actions are considered as a single BOFM feature [10].

- Property 2: In this malware detection technique, while constructing the feature space the sequence in which these malware actions are performed are not considered [10].

- Property 3: If the actions which are performed on two different OS resource type are identical, then the identical action set is considered as a single feature [10].

The main goal of malware is to perform malicious actions on the system critical OS resources. Action refers to the set of related system calls (which are high level operation) through which main objective is achieved. The main advantage of using actions over system calls is that different operating systems have different versions and each version has a different name for the system call, but will have the same functionality [2]. So by analyzing the system calls directly, results in processing large amounts of data. In-order to prevent that, actions are used instead of system calls.

For example, the two system calls OpenFile and ReadFile can be mapped to a single action Open file using a mapping algorithm [5].
A thorough list of activities that a malware can perform on every OS asset sort is given in Table 3.1.

**TABLE 3.1: OS RESOURCES AND CORRESPONDING ACTIONS [10]**

<table>
<thead>
<tr>
<th>OS Resource Types</th>
<th># of Actions</th>
<th>List of Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>File system</td>
<td>14</td>
<td>CreateDirectory, QueryDirectory, CreateFile, SetFileInformation, UnLockFile, LockFile, OpenFile, WriteFile, QueryFileAttributes, QueryFileVolume, DeleteFile, ReadFile DeviceControl, QueryFileInformation</td>
</tr>
<tr>
<td>Registry</td>
<td>7</td>
<td>CreateKey, DeleteKey, DeleteValue, SetValue, OpenKey, NotifyChangeKey QueryValue</td>
</tr>
<tr>
<td>Synchronization</td>
<td>6</td>
<td>CreateMutex, OpenSemaphore, CreateSemaphore, OpenMutex, ReleaseMutex, ReleaseSemaphore</td>
</tr>
<tr>
<td>Network</td>
<td>1</td>
<td>NetworkConnection</td>
</tr>
<tr>
<td>Section</td>
<td>4</td>
<td>OpenSection, CreateSection, QuerySection, MapViewOfSection</td>
</tr>
</tbody>
</table>
3.2 Upper Bound

Now let us consider OS resources are in, the number of possible actions that can be performed on each OS resource type \( j \) \((1 \leq j \leq n)\) is always fixed and predefined. Table 3.1 describes the possible number of actions.

The total number of malware actions that can be performed on a resource instance \( j \) is \( k_j \) and is a constant.

The maximum number of action sets (features) for a resource type \( j \) is \( m_j \)(constant value) and the value of \( m_j \) can be computed as.

\[
m_j = c_1^{k_j} + c_2^{k_j} + c_3^{k_j} + \ldots + c_{k_j}^{k_j}
\]

Where 
\[
c_h^{k_j} = \frac{k_j!}{h!(k_j-h)!}, (1 \leq h \leq k_j)
\]

\( k_j \) is the total number of malware actions performed on an OS resource \( j \).

\( m_j \) is the maximum number of malware actions performed on an OS resource \( j \).

The total number of features possible, which are extracted from all the resource type after applying the BOFM is always a constant (N) which is nothing but the sum of the maximum number of malware actions performed on each OS critical resource.

\[
(N) = \sum_{j=1}^{n} m_j
\]

This equation shows that the number of malware features completely depends on the number of actions performed on each OS critical resource type. In the previous detection techniques the feature space grows in proportion to the number of malware samples [18] but in the approach which is described above, the malware features extracted are bounded by the upper bound of value N. Here N is the total number of features that the JAVA Based threat detector can track the interactions, which the JAR file of Online applet does.
So at most it can find $N$ interactions, between the file which is being scanned and the OS critical resources. An upper bound is indirectly used in the Java Based Threat detector (which is developed) by making the designed application to scan only a fixed number of interactions which most of the malware tries to interact with.

### 3.2.1 Hypothesis

Generally malware performs a combination or set of actions on the OS critical resources to achieve a malicious objective and the actions which are performed by benign application are significantly different. [10]

### 3.3 Flow Diagram for the Proposed System

Figure 3.1 will represent the execution of the given input file and determines whether it performs any malicious actions or not.

The proposed system is implemented completely in JAVA without using any tools and Figure 3.1 shows the clear description of the different options available and the flow of the system if a particular option is selected.
Figure 3.1 Flow Chart for the proposed system
3.4 Input and Output Flow Diagram

The different inputs for the Java Threat Detector Application are 1) Online Applet URL, 2) JAR file which are present on the local machine and the third command which can be given to application is, to scan for JAR file in the local machine.

3.4.1 Flow diagram if the input is a JAR file present on the local machine

Figure 3.2 represents the input flow, if the input to the application is a JAR file from the local machine.

![Flow Diagram](image)

Figure 3.2 Flow Diagram if the input is a local JAR file

Malware is a malicious software which does some malicious interaction. The Java Based Threat detector application which is developed without using any tool, prints out the total interaction made by the JAR file, the user should analyze the data, if the user
finds any unwanted interaction which have some adverse effect, then that particular JAR file is considered as malicious.

3.4.2 Flow diagram if the input is a URL of online applet

Figure 3.3 represents the input flow, if the input to the application is a URL for the online applet.

Malware is a malicious software which does some malicious interaction. The Java Based Threat detector application which is developed without using any tool, prints out
the total interaction made by the Online Applet, the user should analyze the data, if the user finds any unwanted interaction which have some adverse effect, then that particular Online Applet is considered as malicious.

3.5 Implementing BOFM in JAVA Based Threat Detector:

Figure 3.2 and 3.3 describes that, the extracted classes are compared to predefined threat classes, here the Bounded Feature Space Behavior Modelling is used, in a way that at most it can compare the extracted class with only a fixed number of predefined threat classes. The upper bound is applied in an indirect way, on the number of features its going to compare. If it finds any match, then that particular interaction is logged and finally prints all the log messages in the text area.

3.6 Use Case Diagrams

Figure 3.4 is the use case diagram for the actions which are performed by the user. When the user wants to scan for a JAR file on the local machine. The use case diagram will also represent results that are generated by the Java Threat Detector.
Figure 3.4 Use Case Diagram 1

Figure 3.5 is the use case diagram for the actions which are performed by the user. When the user wants to scan an online applet for malicious activities and the results that are generated by the Java Threat Detector.

Figure 3.5 Use Case Diagram 2
3.7 Project Development Phases

The proposed system is implemented completely by using JAVA programming language in Eclipse IDE. The below mentioned steps describes about the process in which the application will work.

1. First scan the class file and JAR file
2. Then take the input class file and represent its internal format. Similarly, if it is a JAR file then create an internal JAR representation.
3. The JAR file contains the data in a byte code format so using a special API is JAVA called .ASM the byte code is converted into source code format by keeping it in a node after the class name is determined and then load the byte code which is present in the class file.
4. If the input is an Online Applet, using Jsoup API the HTML link of the online applet is parsed and from the parsed HTML data the applet tag is extracted.
5. Using that applet tag the location of the source code is found and then extract all the class files present.
6. Extract all the class files and keep it in a node and compare the class files extracted with predefined threat class.
7. If there is any match log that particular interaction and the related interaction message.
8. All the logged messages are printed in the text area.
9. Finally a print message is displayed in the log containing the output, Now the user should whether it is malicious or a genuine file by analyzing the interactions made.

The complete project is developed using JAVA without using any tools. In order to completely implement the project some special API’s like JSOUP and ASM API’s are used. Section 3.7 will describe about the different special API’s used in the project development.

3.8 Environment

3.8.1 JAVA, HTML Parse: JSOUP

JSOUP is nothing but a JAVA library which is used to work with the HTML files. JSOUP library is the efficient and convenient option to manipulate and extract data from HTML using JQUERY method [19].

The different functionalities of the JSOUP API are:

1. Parse the HTML using URL
2. Extract the data
3. Manipulate the data

3.8.2 ASM- JAVA API

ASM is a simple JAVA API, “Byte code manipulator” and “Analysis Framework” which is used to read the byte code, manipulate the JAVA class binary code and it can even generate a JAVA binary class file by “generating new classes altogether” [20].
The above mentioned API’s, which are described in section 3.7.1 is used to extract the JAVA code from the online applet and the API which is described in 3.7.2 is used to read the byte code of it. Then followed by the detection mechanism which is from ‘JAVA Threat Detector’ detects the threat associated with the applet [20].
4. FUNCTIONALITY OF THE APPLICATION

The important functionality of this application is a “Java Based Threat Detector“, so the application can track the malicious actions performed by the malicious Java applications. The different sources through which the application can look through, to detect a malicious Java application are.

1. Check Online Applets
2. Scan a particular Jar file
3. Scan the Computer

Java Applet is nothing but a Java program, which will be transferred to a system and then it is executed on the browser. Many users in one way or the other will come across these applets, without having proper knowledge about the consequences; users will allow these Java programs to run on their machine. Figure 4.1 is a simple Java applet.

![Java Applet](image)

Figure 4.1: Simple Java Applet [17]
4.1 Interface of the Java Threat Detector Application

The main functionality of the application is malware detection which is achieved in the form of an application, which scans for malicious activities.

4.1.1 Java Base Threat Detector

Figure 4.2 is the developed ‘Java Based Threat Detector’ home window, which contains the fields File, Edit and Scan.

![Figure 4.2  Java Based Threat Detector Home Window](image)

4.1.2 Different Scanning Options

The developed application can detect threats from 3 different sources. The first scanning option is ‘Check online Applets’, this option checks the malware which is related to online applets. The second scanning option is ‘Scan JAR files’, This application will scan a particular jar file and will display the threats with the jar file. The third scanning option is ‘Computer’ it will scan all the jar files in the machine.
4.1.3 If the Selected option is ‘Computer’

If the selected scanning option is a computer (shown in figure 4.4), then the application will generate a pop up message saying “Scanning your computer for archives. This could take a while”. Figure 4.5 shows the pop up message window.
Figure 4.4 If the selected option is ‘Computer’

Figure 4.5 PopUp window with message

A list of Jar file present in our system will be displayed
4.1.4 Scanning an Online Applet

A particular online applet can be scanned for malicious activities using the URL of it. By choosing the ‘Check online Applets’ (shown in Figure 4.6), from the scan field in the application. It will generate a pop up window (dialogue box) requesting to enter the URL (Shown in figure 4.7).

![Figure 4.6 If the selected option is ‘Check online Applets’](image)

![Figure 4.7 Pop Up displayed](image)
4.1.5 Copying the URL of the Applet

The URL of the particular online applet is copied from its website [21].

![Figure 4.8 Copying the URL of the Applet](image)

4.1.6 Entering the URL in the PopUp window

The URL which is copied is then given as input to the JAVA Threat Detector.

![Figure 4.9 Copied URL - entered into pop up window](image)
4.1.7 If the selected is ‘Scan Jar file’

A particular JAR file on the local machine can be scanned and checks for any malicious activities. By selecting the ‘Scan JAR files’ option from the scan field in the application (shown in figure 4.13), it redirects to select the destination of the JAR file in the machine.

![JAVA BASED THREAT DETECTOR](image)

Figure 4.10 If the selected option is ‘Scan JAR file’

4.1.8 JAR file selected from the hard drive

The JAR file is selected from a location on the hard drive and fed as input to the JAVA Threat Detector.
4.2 Malicious activities of JAR file and Online Applet:

Online Applet is a JAVA code which is used to play online games on the web, generally an Online Applet should communicate with only one website which acts as a source repository (for the JAVA code) for the applet to run, if an Online Applet tries to interact with more websites or if it tries to communicate with other application on the local machine where it is being scanned, then this particular action is considered malicious and based on these factor a decision is made by the user whether it is a malicious applet or a genuine applet.
Malicious activities performed by JAR file. General malicious action performed by a JAR file.

1. Deleting an existing file
2. Writing data into an existing file.
3. Moving the location of the file.
4. Interacting with malicious website.
5. Make the system content available to remote person.
6. Creating servers on your computer.

If a JAR file performs any of these actions then the user can decide whether it is a malicious JAR file or not.
5. TESTING AND EVALUATION

The project contains a Java threat detector which performs the following test cases:

1. Try to detect the total number of programs in which the system tries to connect to the internet and find their malicious activities.

2. Try to detect existing applications present in the system and find the vulnerabilities present in the system that are caused by “complexity, familiarity, connectivity, password management flaws, fundamental operating system design flaws, Internet website browsing, software bugs, unchecked user input”, Buffer overflows, SQL injection, code injection, mail injection, default permit.

3. Try to log the connection details between malware and source repository. The two different version control systems are centralized and distributed, based on the control system the entire arrangement of data in the repository is kept on a single server or broadcasted to every client framework.

4. Using the virus dictionary, capture the details which happen during the scanning, to detect the malicious behavior like “unpacking of malcode”, “modifying the host files”. By observing such kind of actions will helps us to detect the existence of malware which are not seen before, on the system which is protected.

5. Obtain the scan details about the malicious activities and save them in the text file for future references.

6. Capture standalone programs which run on websites such as applets which also form malicious bot in the system. The memory usage and computational time for capturing applets is very low.
5.1 Test Cases

Test Case 1: Scanning a malicious JAR file

Step 1: Scanning all the JAR files in the local machine

Figure 5.1 shows the list of JAR files and class files present in our local machine, so that the user can scan the required JAR file in that particular location using the JAVA Threat Detector.

![JAVA BASED THREAT DETECTOR](image)

**Figure 5.1 List of JAR and Class file generated**

C:\ProgramFiles\Java\jdk1.7.0_51\lib\missioncontrol\plugins\com.jrockit.mc.ui.ja_5.2.0.157284.jar

C:\ProgramFiles\Java\jdk1.7.0_51\lib\missioncontrol\plugins\com.jrockit.mc.ui.zh_CN_5.2.0.157284.jar

C:\ProgramFiles\Java\jdk1.7.0_51\lib\missioncontrol\plugins\com.jrockit.mc.ui_5.2.0.157284.jar
Step 2: Scanning a malicious JAR file.

Downloading a jar file from a website and place the file on the local machine (shown in Figure 5.2). The JAR file downloaded is given as input to the JAVA Threat Detector.
According to the description given below the link, the JAR file is an improvement to the minecraft server and downloading this JAR file will move the craftbukkit JAR file.

**Step 3: Scanning the JAR file**

![Figure 5.3 Scan the JAR file from the Local machine](image)

Figure 5.3 shows that the JAR file which is downloaded is given as input to the “JAVA Threat Detector”.
Step 4: Result obtained after scanning the JAR file

Java Based Threat Detector

File Edit Scan

Scanning: C:\Users\muralijagdev\Desktop\craftbukkit-1-5-1-R0-1.jar
Time: Wed Apr 15 08:06:56 CDT 2015

Creates one or more data output streams so that it can write data to:
Characters are being written to this stream.
Words are being written to this stream.

Tampers with the data of files!
One or more files get cleared, and new data gets written onto them!

Is encrypting and decrypting data, could be runnable Java code!
Interaction: AES/CBC/PKCS5Padding
Interaction: AES/CBC/PKCS5Padding

Creates one or more input streams so that it can read data from websites, files, and servers!
Interaction: 2147483647
One or more input streams are reading data from either files, websites, or servers!

Tampers with the data of files!
One or more files get cleared, and new data gets written onto them!

Interacts with one or more servers/computers! (tcp)
Data gets read from the one or more servers in which the application(s) interact with! (tcp)
Data gets written to one or more servers in which the application(s) interact with! (tcp)

Interacts with one or more files!
Interaction:
Interaction: META-INF
Interaction: .
Interaction: level.dat
Interaction: level.dat
Interaction: crash-reports
Interaction: .
Interaction: banned-players.txt
Interaction: banned-ips.txt
Interaction: region
By observing the above mentioned results(Figure 5.4) it is obvious that, the particular file which has been scanned is a malicious file because it make interaction with many other websites, the JAR file which is scanned will tamper with the data from files and one or more files get cleared, and new data gets written onto them. Based on these factors we can conclude that the JAR file is a malicious one.

The user can decide it as a malicious file, not from the number of interactions made by the JAR file but from the malicious interaction (as mentioned above) which it tried to do with the OS critical resources.
Test Case 2: Scanning a Non malicious JAR file:

Step 1: Downloading a jar file from the website and it contains the description of the JAR files.

Figure 5.5 Downloading a JAR file from the website

Figure 5.5 shows the JAR file which has been downloaded from the website has a description below it, the description clearly specifies that “JavaMod is a multi-platform audio player with a focus on the playback of old Amiga files” which indirectly specifies that the JAR file will try to make some interaction with websites.
Step 2: Scanning the JAR file

Figure 5.6 shows that the JAR file is selected from the local machine.

![Figure 5.6 Scanning the JAR file from the local machine](image)

Step 3: Result obtained after scanning the JAR file

![JAVA BASED THREAT DETECTOR](image)
The result obtained after scanning this particular JAR file is shown in figure 5.7, which shows that the JAR file tries to make some interactions with the websites which is already specified and it does not perform any malicious actions which are not required, so this jar file doesn’t perform any malicious activities and hence it is not a malware.
Test Case 3: Scanning an Online Applet

Step 1: Field to enter the URL of the online Applet

Figure 5.8 shows the application field where the URL of the online applet should be entered and Figure 5.10 shows that the URL is entered.

Figure 5.8 Field to enter the URL of the Online Applet

Step 2: URL of the online applet entered

Figure 5.9 Copy the URL of the online applet
Figure 5.10 The URL of the online applet is entered

Figure 5.10 shows, the URL address of the online applet is entered into the given field and figure 5.9 shows that the URL of the online applet is copied from the browser.

Step3: Result obtained after scanning the online Applet

Figure 5.11 Result obtained after scanning the online applet

Total result: Figure 5.11 shows the result obtained

Looking for archives on: http://www.jpowed.com/free_java_game/alienwar
Figure 5.10 shows the result obtained after scanning the URL of the online applet. This shows that the online applet does only one interaction with one website, which is the source repository for the applet and it doesn’t perform any other malicious actions. So this online applet is not a malware.

**Test Cases 4: Scanning a Malicious online Applet**

1. Try to connect to the URL of the Applet which was loaded.
2. Parse the URL of the applet with Jsoup
3. Locate the Applet tag inside that URL of the HTML document and then extract the Jar file.
4. Then extract the Classes and methods from the JAR file.
5. Map the methods with our threat classes and internal threat methods which are determined.
6. And depending on the match result the interactions are determined and finally printed on the console.
Step 1: URL of the online Applet.

Figure 5.12 shows that the URL of an online Applet is copied.

Figure 5.12 Copy the URL of an online Applet

Step 2: URL of the online Applet is entered

In step 2 the URL of the online applet which is copied in figure 5.12 is entered as shown in Figure 5.13.

Figure 5.13 URL of the online Applet is entered
Step 3: Result Obtained after Scanning the Online Applet

Figure 5.14 Results obtained after scanning the online Applet

Figure 5.14 shows the result obtained after scanning the online Applet. “The domain from where the applet executable has been downloaded is the only domain to which the usual (unsigned) applet is allowed to communicate. This domain can be different from the domain where the surrounding HTML document is hosted”. But the applet creates one or more connection to read or write data. So the Applet is malicious.
The last statement in figure 5.13 “One or more stream created for a website so that the scanned application can read data” shows that it’s a malicious applet because Online Applet will no scan any applications on the local machine.
7. CONCLUSION AND FUTURE WORK

This system mainly discusses about the malware detection solution which will unite a new malware behavior modeling technique (BOFM) [10], to recognize malware from amiable programs. The main objective is, to complement traditional anti-virus software on end host machines in an effective and scalable manner. Since the behavior of malware is characterized as an interaction between the operating system resources and the malware in a scalable manner, and the interaction involves high level actions, so the detection model is not simple. Even with the growth of the number of malware samples the feature space generated by BOFM remains the same.

The proposed system is tested for malware in both jar files and online applets in the testing section. The testing procedure is performed with a specific goal to confirm whether all the features are working in the application.

The future scope of this project, is to add some extra functionality like, the project is implemented only for JAVA jar files, but this detection technique should also work for any type of file format. And the user interface can also be enhanced and make it much more appealing. The Java Threat Detector is not completely automated in making decision about the malware, the scope for enhancement is to make the system completely automatic and an approach should be designed to calculate the efficiency of this malware detection technique.


50
Class Diagrams:

Class diagram if the option selected in Online Applet:

Class diagram if the option selected is JAR file:
 private void scan()
{
    JSyntaxAnalyzerAnalyzerScan(archive.getName());
    JSyntaxAnalyzerAnalyzerScan("time": new Date(System.currentTimeMillis()) + "\n");
    Map<ThreadClass, List<ThreadMethod>> threatMap = getThreatMap();
    Iterator localIterator2;
    for (Iterator localIterator1 = this.archive.iterator(); localIterator1.hasNext(); localIterator2.hasNext())
    {
        ClassNode classNode = (ClassNode)localIterator1.next();
        localIterator2 = classNode.methods.iterator();
        continue;
    }
    printScanResults(threatMap);
}

 private Map<ThreadClass, List<ThreadMethod>> getThreatMap()
{
    Map<ThreadClass, List<ThreadMethod>> threatMap = new HashMap<>();
    for (Class<? extends ThreadClass> threatClass : this.threatClasses)
    {
        try
        {
            ThreadClass threatClassInstance = (ThreadClass)threatClass.newInstance();
            threatMap.put(threatClassInstance, threatClassInstance
            .getThreadMethodInstances());
        }
        catch (InstantiationException e)
        {
            e.printStackTrace();
        }
    }
    catch (IllegalAccessException e)
    {
        e.printStackTrace();
    }
    return threatMap;
}

 private void scanMethodInsnNode(MethodInsnNode methodInsnNode, Map<ThreadClass, List<ThreadMethod>> threatMap)
{
    String methodInsnNodeOwner = "l" + methodInsnNode.owner + ";
    for (Map.Entry<ThreadClass, List<ThreadMethod>> threatEntry : threatMap.entrySet())
    {
        ThreadClass threatClassInstance = (ThreadClass)threatEntry.getKey();
        if (methodInsnNodeOwner.equals(threatClassInstance.getName()))
        {
            threatClassInstance.setUsed();
            addInteraction(threatClassInstance, methodInsnNode);
            for (ThreadMethod threatMethodInstance : threatEntry.getValue())
            {
                if (methodInsnNode.name.equals(threatMethodInstance.getName()))
                {
                    threatMethodInstance.setUsed();
                    addInteraction(threatMethodInstance, methodInsnNode);
                }
            }
        }
    }
    for (Map.Entry<String, Boolean> entry : pluginThreads.entrySet())
    {
        if (methodInsnNode.owner.equals(entry.getKey()))
        {
            entry.setValue(Boolean.valueOf(true));
        }
        if (((methodInsnNode.name + methodInsnNode.desc).equals(entry.getKey()))
        {
            entry.setValue(Boolean.valueOf(true));
        }
    }
}

 private void addInteraction(ThunkObject threatObject, AbstractInsnNode abstractInsnNode)
{
    AbstractInsnNode previousNode = abstractInsnNode.getPrevious();
    if ((previousNode instanceof VarInsnNode))
    {
        int var = ((VarInsnNode)previousNode).var;
        AbstractInsnNode nextNode = previousNode.getPrevious();
        }
while (nextNode != null)
{
    if ((nextNode instanceof VarInsnNode))
    {
        VarInsnNode varInsnNode = (VarInsnNode)nextNode;
        if ((varInsnNode.getOpcode() > 53) &&
            (varInsnNode.getOpcode() < 87) &&
            (varInsnNode.var == var))
        {
            previousNode = varInsnNode.getPrevious();
            break;
        }
    }
    nextNode = nextNode.getPrevious();
}

if (! (previousNode instanceof LdcInsnNode))
{
    LdcInsnNode ldcInsnNode = (LdcInsnNode)previousNode;
    threatObject.addInteraction("interaction: " + ldcInsnNode.cst);
}

private void printScanResults(Map<ThreatClass, List<ThreatMethod>> threatMap)
{
    boolean regularThreatsFound = false;
    ThreatClass threatClassInstance;
    for (Map.Entry<ThreatClass, List<ThreatMethod>> threatEntry : threatMap.entrySet())
    {
        threatClassInstance = (ThreatClass)threatEntry.getKey();
        if (!threatClassInstance.isUserUsed())
        {
            regularThreatsFound = true;
            JScanner.log(threatClassInstance.getReply());
            printInteraction(threatClassInstance);
            for (ThreatMethod threatMethodInstance : threatEntry.getValue())
            {
                if (!threatMethodInstance.isUserUsed())
                {
                    JScanner.log(threatMethodInstance.getReply());
                    printInteraction(threatMethodInstance);
                }
            }
            JScanner.log("\n");
        }
    }
    if (!regularThreatsFound) {
        JScanner.Log("No regular threats found in this archive!\n");
    }
    boolean pluginThreatsFound = false;
    for (Object pluginEntry : pluginThreats.entrySet())
    {
        if (!((Boolean)((Map.Entry)pluginEntry).getValue()).booleanValue())
        {
            pluginThreatsFound = true;
            JScanner.log((String)((Map.Entry)pluginEntry).getKey() + " is used.");
        }
    }
    if (!pluginThreatsFound) {
        JScanner.Log("No plugin threats found in this archive!\n");
    }
}

private void printInteraction(ThreatObject threatObject)
{
    for (String interaction : threatObject.getInteractions())
    {
        JScanner.Log(interaction);
    }
}
WebPage.java file:

```java
public List<Archive> getArchives()
{
    List<Archive> archives = new ArrayList();
    Iterator localIterator1;
    for (Iterator localIterator1 = this.getDocument.getElementsByTagName("applet").iterator();
        localIterator1.hasNext();
        localIterator1 = this.getDocument.getElementsByTagName("applet").iterator())
    {
        Element appletElement = (Element)localIterator1.next();
        String archiveAttributeName = "archive";
        int commaCount = 0;
        for (char character : archiveAttributeName.toCharArray()){
            if (character == ',') {
                commaCount++;
            }
        }
        List<String> archiveAttributeValueList = new ArrayList();
        for (int i = 0; i <= commaCount; i++) {
            archiveAttributeValueList.add(archiveAttributeName.split(",\"))[i].trim());
        }
        java.lang.String code = appletElement.getAttribute("code");
        String codebase = appletElement.getAttribute("codebase");
        if (!codebase.isEmpty())
        {
            for (int i = 0; i < archiveAttributeValueList.size(); i++) {
                archiveAttributeValueList.set(i, resolvePath(codebase,
                                                      (String)archiveAttributeValueList.get(i)));
            }
            code = resolvePath(codebase, code);
        }
        File classFile = getClassFile(resolvePath(this.getDocument.baseUri(),
                                                   code));
        try {
            if (classFile != null)
            {
                archives.add(new ClassArchive(classFile));
                classFile.delete();
            }
        } catch (Exception e) {
            e.printStackTrace();
            localIterator2 = archiveAttributeValueList.iterator();
            continue;
        }
        return archives;
    }
}

private String resolvePath(String parentDirectory, String childDirectory)
{
    try {
        return new URI(parent).resolve(childDirectory).toString();
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
    return null;
}

private File getClassFile(String url)
{
    try {
        File classFile = new File("temp.class");
        URL u = new URL(url);
        InputStream inputStream = u.openStream();
        FileOutputStream outputStream = new FileOutputStream(classFile);
        byte[] bytes = new byte[1024];
        int length;
        while ((length = inputStream.read(bytes)) != -1) {
```
MenuItemApplet.java

```java
package com.jsparser.gui.component.impl;

import com.jsparser.JParser;

public class MenuItemApplet
    extends ComponentMenuItem
{
    private static final long serialVersionUID = 1L;

    public MenuItemApplet()
    {
        super("Check online Applets");
    }

    public void actionPerformed(ActionEvent e)
    {
        String url = JOptionPane.showInputDialog("Enter url below.");
        if (url != null)
        {
            JParser.log("Looking for archives on: "+url);
            List<Archive> archives = new com.jsparser.util.WebPage(url).getArchives();
            if (archives.size() > 0)
            {
                for (Archive archive : archives)
                {
                    new ArchiveScanner(archive);
                }
            }
            else
            {
                JParser.log("Could not find any archives on: "+url);
            }
        }
    }
}
```

56
import com.jscanners.archive.Archive;

public class FileManager
{
    private static final File[] DIRECTORIES = {
        getDirectory("/plugins")
    };

    private static File getDirectory(String directory)
    {
        return new File(System.getProperty("user.home") + "/JScanner/" + directory);
    }

    public static void createDirectories()
    {
        for (File directory : DIRECTORIES) {
            directory.mkdirs();
        }
    }

    public static Archive getSelectedArchive(ComponenMenuSet componentMenuSet)
    {
        JFileChooser jFileChooser = new JFileChooser();
        jFileChooser.setFileFilter(new FileNameExtensionFilter("Archives", new String[] { "class", "jar" }));
        if (jFileChooser.showOpenDialog(componentMenuSet) == 0)
        {
            File selectedFile = jFileChooser.getSelectedFile();
            if (selectedFile.getName().endsWith(".class")
            {
                return new ClassArchive(selectedFile);
            }
        }
        try
        {
            return new JavaArchive(new JarFile(selectedFile));
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return null;
    }

    public static List<Archive> getPlugins()
    {
        List<Archive> plugins = new ArrayList<>;
        for (File file : DIRECTORIES[0].listFiles())
        {
            String name = file.getName();
            if (name.endsWith(".class"))
            {
                plugins.add(new ClassArchive(file));
            } else if (name.endsWith(".jar"))
            {
                try
                {
                    plugins.add(new JavaArchive(new JarFile(file)));
                }
                catch (IOException e)
                {
                    e.printStackTrace();
                }
            }
        }
        return plugins;
    }
}
MenuItemApplication.java

```java
package com.jsceans.gui.component.impl;

import com.jsceans.archive.Archive;

public class MenuItemApplication
    extends ComponentMenuItem
{
    private static final long serialVersionUID = 7501648013168982367L;

    public MenuItemApplication()
    {
        super("Scan JAR files");
    }

    public void actionPerformed(ActionEvent e)
    {
        Archive archive = FileManager.getSelectedArchive(this);
        if (archive != null) {
            new ArchiveScanner(archive);
        }
    }
}
```