Malware Detection Techniques
Ayush Tewari, Ishan Rastogi, Shubanshu Agarwal
IIIT – Hyderabad, India

Abstract—Malware is code designed for a malicious purpose, such as to disrupt computer operation, gather sensitive information, or gain access to private computer systems. A malware detector identifies malware and thus prevents it from adversely affecting a host. Detectors which use pattern matching to identify malware, can easily be evaded by simple code transformations and various obfuscations. To overcome this shortcoming various malware detectors have been proposed which rely on semantic signatures and employ static analysis techniques such as model checking and theorem proving to perform detection. While these techniques are quite successful in identifying malwares, there are certain obfuscations which make the malwares undetectable. In this paper, we try to study the design of a semantic based malware detector which undoes the obfuscations performed by a malware writer. We also study dynamic approaches which are resistant to most obfuscations but are resource intensive. We try to improve on the current techniques and propose a hybrid system which will hopefully improve the quality of malware detector.

Index Terms—Malware Analysis, Static Analysis, Dynamic Analysis, Code Obfuscations (key words)

I. INTRODUCTION
Malicious code (or malware) is defined as software that fulfills the harmful intent of an attacker[1]. Current Malware detection are largely classified in two main categories, static analysis and dynamic analysis. With static analysis, we study a program without actually executing it. As we can analyse parts of a suspect program under specific conditions which normally do not execute, static analysis has a nice prospect. With dynamic analysis, we study a program as it executes. The advantage of dynamic analysis is that it can be accurate and we have access to its behavior. Limitations of dynamic analysis can be seen that it is not possible to predict the behavior of a non-trivial program. It is also not possible to make a suspect program traverse all paths through its code if there are many. As malware detectors use advanced detection techniques, malware writers use better techniques to try to fool the malware detector. Some common practices to evade code detection are polymorphism, metamorphism, reordering etc. For ex. a virus can morph and encode its malicious payload and decrypt it during execution. Also the malware writer can change the code in various ways by reordering the code and using appropriate jump statements, writing a particular statement in non-trivial way, unrolling the loop, distributing the loop into multiple parts etc. These become helpful to evade static detection but does not affect dynamic detection much.
A malware obfuscation technique is a way of constructing a malware that make it more difficult to detect [2]. The following are commonly used obfuscation techniques:

- **Self-Encryption and Self-Decryption** - some malwares can encrypt and decrypt themselves, concealing themselves from direct examination
- **Polymorphism** - polymorphism is a nice way of self encrypting. a polymorphic malware makes several changes to the encryption code, and finally the code has the same functionality but different appearance
- **Metamorphism** - The idea behind metamorphism is to alter the content of the malware itself, rather than hiding the content with encryption
- **Stealth** - A stealth malware uses various techniques to conceal the characteristics of an infection
- **Armoring** - The intent of armoring is to write a malware so that it attempts to prevent malware detectors from analyzing the malware functions through disassembly, traces, and other means.
- **Tunneling** - A malware that employs tunneling inserts itself into a low level of the operating system so that it can intercept low-level operating system calls. By placing itself below the malware detectors, the malware attempts to manipulate the operating system to prevent detection by malware detectors.

Static malware analysis uses some templates and tries to do pattern matching against the suspect program. Dynamic analysis runs the suspect program in controlled environment and collects the statistics for memory access, order of memory access, Instructions, system calls; after collecting these statistics it is compared to the already trained data and then classified as a malware or not using some model of representation. For static analysis the popular tools used are disassemblers, decompilers, source code analyzers whereas for Dynamic analysis the popular tools are debuggers, function call tracers, machine emulators, logic analyzers, and network sniffers.
II. STATE OF THE ART METHODS

A. Dynamic Analysis

Various dynamic analysis methods exist in the literature\cite{4,5}. Most of them have a similar approach to the problem which is to construct a mathematical representation of the program, defining similarity between the programs and using it to learn the malware. The recent state-of-art method we discuss here is the graph based approach\cite{5}. Their method is described below.

1. The first step is to get the instruction trace of the program which is done using Ether analysis system\cite{4}. The analysis program and the input program are run on a virtual machine. In this step, programs which look malicious to the analysis system due to use or access of some memory or restricted variables are already identified. The output of this step is the in-order list of executed instructions.

2. Every program is represented as a graph \( G = (V,E) \) from the instruction trace. \( V \) is the set of vertices and each vertex is an instruction ignoring the values of operands. The set \( V \) is common to all programs and is the set of all unique instructions observed. An edge \((i,j)\) is constructed in the graph when instruction \( i \) and instruction \( j \) are executed successively. The matrix representation of this graph is constructed and every row is normalized so that it represents a markov chain.

3. The metric for comparing any two graphs is defined in terms of kernels in the inner product space. Gaussian Kernel,
\[
K_G(x,x') = \sigma^2 \exp \left( -\frac{1}{2\lambda^2} \sum (x_{ij} - x'_{ij})^2 \right)
\]
Laplacian Kernel,
\[
K_L(x,x') = \sigma^2 \exp \left( -\frac{1}{2\lambda^2} \sum \left( \phi_k(L|x) - \phi_k(L|x') \right)^2 \right)
\]
where \( L(x) \) is the laplacian of adjacency matrices and and \( \phi_k \) are the eigen vectors of \( L(x) \).

4. The intuition behind defining these two kernels is that the first captures differences at local level i.e. Penalizing every change while the second compares the global structure of the graphs using eigen vectors. The actual kernel used is a linear combination of both.

5. A linear SVM is trained using these graphs which identify malware from normal programs. Though the SVM used is linear, non-linear boundaries can be learnt using the “kernel trick”\cite{5}. The kernel defined above is a valid kernel which is an inner product in a higher dimension space. The SVM is learnt using this kernel.

The results reported by the author show it to be better than the static approaches. The major limitation is the runtime of the algorithm. Though most of the work is done offline, the kernel computations have to be done for every program. Also, this representation cannot encode the correct order of execution. It just encodes the probability of an instruction occuring after another. This may result in wrong answers because instructions could be used again which do not affect the outcome of the program such that it can evade such methods.

B. Static Analysis

Static Analysis uses approach to check if a given code sequence is a malware or not by comparing it with some known data. The paper we analyze\cite{6} introduces a novel method which quantifies the semantics of malware and to some extent the functionality of the individual modules of different malwares i.e. malware behaviour. This paper models malware behaviour by considering the modules as "templates" which are basically code written in low level intermediate representation. The goal is to match these templates with the input software. They present techniques to solve the above mentioned Template Matching Problem resonably. We briefly describe their method's characteristics, strengths and limitation as it is crucial to understand our work.

Template matching Problem can be crudely defined as following : "Given a code in Intermediate Representation and an instruction sequence, determine if they behave similarly i.e. both specify the same state transformation". Intuitively, a template is just a code written in some IR(similar to some compiler's but more powerful) and state transformation determines how memory is affected when the instruction is executed from an initial memory state. The formal definition of template and instruction sequence as given by the authors are: A template \( T \), \((I_T, V_T, C_T)\) is a 3-tuple, where \(I_T\) is a sequence of instructions and \( V_T \) and \( C_T \) are the set of variables and symbolic constants that appear in \(I_T\). There are two types of symbolic constants: an n-ary function (denoted as \( F(n) \)) and an n-ary predicate (denoted as \( P(n) \)). For determining the final and initial memory states, we need to execute a code over a machine or at least simulate it logically. A template is provided by an execution context which assigns values to the symbolic constants to make this execution possible. An execution context for a template \( T \), \( E_T = (I_T, V_T, C_T) \), is an assignment of values of appropriate types to the symbolic constants in the set \( C_T \). Formally, an execution context \( EC_T \) for a template \( T \) is a function with domain \( C_T \), such that for all \( c \in C_T \) the type of \( c \) and \( EC_T(c) \) are the same. An execution context for the template shown in Figure 1(a) is shown in Figure 1(c). Given
an execution context $EC_T$ for a template $T$, let $EC_T(T)$ be the template obtained by replacing every constant $c \in C_T$ by $EC_T(c)$.

The Template Matching Problem is stated formally. We say that an instruction sequence $I$ contains the behaviour specified by template $T$ iff:

i) For same initial memory patch we get same final memory patch for both $T$ and $I$.

ii) The events that happen e.g. interrupts etc if any are all same and in same order in both $T$ and $I$.

iii) If program counter at the end of template points to some changed memory location, so must be the case of $I$.

In general, the above first condition can be loosened a bit by checking for only "core" memory locations such that locations which store temporaries can be ignored.

Above mentioned TMP is undecidable in the general case and can be proved easily. Refer to the paper for the proof. Algorithm which solves the TMP is based on two conditions:

i) All nodes in the template must match with some node in the instruction sequence.

ii) Def-Use Path is a sequence of nodes $<n_1, n_2, ..., n_k>$ such that there is at least one variable defined in $n_1$ and used in $n_k$ with the condition that any variable defined in $n_1$ is not defined in $n_i$ for $1 < i < k$. Preservation of def-use paths i.e. if two nodes $n_1$ and $n_2$ in the template are related by a def use path and are mapped to $m_1$ and $m_2$ in the instruction sequence, then $m_1$ and $m_2$ must also be related by a def use path with the same binding variable that is defined and used in the def use path).

The algorithm is sound i.e. if an instruction sequence matches some template $T$, the match is a correct match though it is not complete i.e. it may not match a semantic similar template $T$ to matching instruction sequence.

Some rules while matching a template node and an instruction sequence node are:

i) A variable in the template can be unified with any program expression, except for assignment expressions.

ii) A symbolic constant in the template can only be unified with a program constant.

iii) The predefined function memory : $F(1)$ (traditionally written in array notation memory[. . .]) can only be unified with the program function memory : $F(1)$.

iv) A predefined operator in the template can only be unified with the same operator in the program.

v) An external function call in the template can only be unified with the same external function call in the program.

Their algorithm adheres to above restrictions. To appreciate their strengths and weaknesses it is enough to know the above points and hence we do not state their algorithm.

Illustration 1: Example of a template
III. IMPROVEMENTS AND FINAL SYSTEM

The final system we propose consists of three parts. We describe them below.

A. Static Semantic Analysis

This is the method as described in the previous section. Some limitations of the method are:

1. Strict memory ordering is required in the instruction sequence to match with the template. This comes from the use of def-use path constraints. A def-use path encodes the order of memory updates and the same order has to be followed in the instruction sequence. If this order is different but the results are same, the algorithm fails to give the correct output.

2. If some instruction, I is replaced with similar instructions which together are equivalent to I, it is not possible to match them to the template.

<table>
<thead>
<tr>
<th>Template T</th>
<th>Program P</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; 10; i++) a[i] = 0;</td>
<td>for (i = 0; i &lt; 10; i++) a[i] = 0;</td>
</tr>
<tr>
<td>for (i = 1; i &lt; 10; i++) a[i] = 0;</td>
<td></td>
</tr>
</tbody>
</table>

Illustration 3: Example of limitation

In this case, the same array is being accessed in a different order to do the same task but it does not match with the template.

B. Re-evaluation using Modified program

As the problem of TMP is undecidable, there can never exist a solution which can solve the general case of matching. We try to improve the algorithm reasonably to give better results for most of the cases and overcome these limitations.

We modify the semantic model such as to return partial matches along with the complete match i.e. it returns the percentage of match between every template and the program.

Illustration 2: Flowchart of Malware Analysis

If the semantic model says that a program is malware, as it is a sound algorithm, we know that it is a malware and end our algorithm. In case the semantic model is not able to determine the output, we check the amount of partial matches. If the partial match with any template is more than a specific threshold, we re-run the semantic model after modifying the code for all those templates for which the match was greater than the threshold.

We do the following modifications to the program

1. Combining Loops

   This is to overcome the first limitation where the order of memory access prevents from matching correct matches. We want to find out if there are more than one loops in the program which may be equivalent to 1 loop in the template. For this, we try to combine compatible loops and trying to match again. We first identify all the loops in the program using the control flow graph.

   If n > 1, n loops can be combined if:
   1. The statements inside all the loops are same.
   2. No statement in between any two loops are related by a def-use path with any statement of the individual loops. If this happens, then the loops cannot be combined because it may affect the values of the variable in the def-use path before the second loop.
   3. Intersection of the set of values of counter variables of any two loops is empty. Suppose $S_i$ is the set of values over which the counter variable iterates in Loop $L_i$ and there are $n$ loops, then

   $$S_1 \cup S_2 \cup \cdots \cup S_n = S_T$$

   where $S_T$ is the set over the loop in the template.

   $$S_i \cap S_j = \emptyset \quad \forall 1 \leq i \neq j \leq n$$

   For a specific template, we identify the loops which have not been matched yet. For each such loop, we find out the set of loops of the program it could have matched with i.e. where the internal statements can be unified. We try to combine the loops which are compatible and run the matching algorithm again. Ideally, we should try all possible combinations but as it would be exponential, we put a limit on maximum number of loops which can be combined as a constant.

   We use a similar approach for the second limitation by trying to combine statements and simulation. In this case, if the match with some template is greater than a threshold, we try to combine individual statements of the program which are
compatible. For \( n > 1 \), \( n \) expressions can be combined and matched to a template expression if:

1. The set of variables used in the RHS of the expression are the same and correspond to the same assignment (LHS).
2. For any statements between these expressions in the program, there should be no def-use path between them and the statements to be combined.

For any expression in the template which has not been matched yet, we identify the set of statements of the program which can be combined and matched to this statement and then try all possible combinations restricted to a constraint. This is done using evaluating these expressions for some random inputs for the RHS and comparing the values of LHS.

For example, Template T: \( a = a \times 8 \)

Program P: \( a = a^2 \)

\[
\begin{align*}
  a &= a^2 \\
  a &= a^2
\end{align*}
\]

Once we know that we can combine the 3 expressions of P, we give random values to a at the start and evaluate both of them and compare the value of a after execution. If the values are same, we declare it as a matched node.

After this step, if any template is matched, we declare the program as a malware. If not, we again find the amount of partial matches for the templates. If the maximum partial match is above a threshold, we run the third step and if it is less, we consider the program as safe.

C. Dynamic Analysis

In this step, we have a program as an input which is a probable malware. As dynamic analysis is secure against most of the obsfuscations, after this step, we consider the output of dynamic analysis as the final result. As dynamic analysis is resource and time intensive, we run it for a small subset of programs which pass after step A and step B. For dynamic analysis, we use the graph-based dynamic analysis as discussed with some improvements. We had observed that we cannot encode information about the order of instructions fully in the graph. For ex. \( S_1, S_2, S_3 \) and \( S_1, S_2, S_3 \) represent the same graph where \( S \) represents instruction i . For representing this information, we replace this graph with a bipartite graph where each vertex is replaced with \( t \) vertices, where \( V \) is the vertex \( v \) at time \( i \mod t \). We can select the value of \( t \) using validation in present datasets.

Dynamic Analysis using Static Information

We can further reduce the time required using the information from the static analysis methods. Instead of executing the whole program for dynamic analysis, we can identify the parts of program which are useful and only execute them. We can identify the useful part of any program with reference to a template T as:

1. The starting point can be identified as the instruction which matches with the first node of the template.

2. For determining the end point, we move upwards from the exit node in the template, with the objective to find a sub-template, which has no statement which transfers the flow to any node outside the sub-template. Once we identify such a sub-template and find its corresponding matching nodes in the program. The last matched node can be taken as the end point of the useful part of the program.

We compare the useful part of the program and the template by executing them using random memory patches as input and comparing their output on multiple such inputs. Using this method also gives a better solution to find out the match in the case where memory is accessed in different order as discussed as a limitation of semantic approach.

IV. Results

The results as reported by semantic analysis \([6]\) are very good and detect malware in about all the cases. Also, false positives are 0 as the algorithm is sound. All false positives happen because of incorrect template which does not uniquely identify the malware.

<table>
<thead>
<tr>
<th>Malware family</th>
<th>Template detection</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decryption loop</td>
<td>Mass-mailer</td>
</tr>
<tr>
<td>Netsky</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Bi(e)agle</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Sober</td>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Illustration 4: Results of detection by semantic-aware static analysis

As our system involves many steps including dynamic analysis, we expect it to work better even in the case where the input program is a malware which cannot be detected from static analysis.

V. Future Work

We have proposed some solutions for work around of the limitations observed in the current approaches. For semantic analysis, there can be better approaches to solve the general case of memory ordering problem. If this is solved, it would also help in solving other limitations. There can be stricter definition of semantics which can capture the meaning of the program better for ex. If two instructions which look completely different and use different operators have the same meaning, they can be combined. In the case of dynamic analysis, there may be better approaches to find the end points for execution in the case where we cannot identify the useful part of the program from our approach. These are interests for future research.

VI. Conclusion

We observe that there are many obsfuscations like packing, polymorphism, opaque constants which are very tough to be detected from a static approach and there is a necessity for a
hybrid system\textsuperscript{[3]} . We have studied current static and dynamic analysis methods, tried to improve them and provide a hybrid system which has the strength of each one of them. We have improved the static analysis module by trying to match those cases where the program is semantically same but accesses loops or statements in different order. We try to do this maintaining the algorithm to be sound. We have added additional outputs for the probability or percentage of partial match to determine if a program is a probable malware or not. We have tried to reduce the time required for dynamic analysis by identifying a sub part of the program and only executing that part. We have combined these models in a way such that the strength of each model is added up in the final system i.e. it is fast because it mainly uses static analysis on most of the programs and dynamic analysis on lesser number of them, and is more reliable as we run dynamic analysis for programs about which we are not sure.

REFERENCES


