Static Analysis of programs for checking malicious behavior

B.Tech. Project Report

Submitted in partial fulfilment of requirements for the degree of
Bachelor of Technology

by

Sanket Kanjalkar
Roll No : 120050011

under the guidance of
Prof. R.K. Shyamasunder

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

November, 2015

Contents

Abstract

Metamorphic viruses transform their code as they propagate, thus evading detection by static signature-based virus scanners, while keeping their functionality. To achieve this, metamorphic viruses use several code transformations, including register renaming, code permutation, code expansion, code shrinking and garbage code insertion. A metamorphic virus is one that can transform based on the ability to translate, edit and rewrite its own code. Antivirus scanners have a difficult time detecting this type of virus because it can change its internal structure, rewriting and reprogramming itself each time it infects a computing system.[2]

In this project, a method based on static analysis is described which will detect metamorphic viruses . First, we extract the control flow graph in terms of basic blocks of the program. Then, assuming we have a database of known virus semantic signatures, we check whether in any possible execution of the program could be malicious. The advantage of this approach is that it will detect and predict variants of Metamorphic viruses with only few signatures while the currently existing techniques and tools for detecting Metamorphic viruses require (almost) one signature per each variant of the Metamorphic virus.

Once we have a semantic signature there are 2 ways of operating on this signature. One is to check whether the there is an exact match with the known bad malware databases. On the other hand, we can cluster the known bad malware data to get a suitable clustering for a set of malware. We, then try out various clustering metrics to check the accuracy of our clustering. We then compare various approaches with the given dataset and finally compare the clustering approach with deterministic dataset.

Chapter 1

Introduction

Since metamorphic virus change their structure and undergo various transformations, it is difficult to detect them by maintaining signatures for every variation of the virus. This creates a need for static signatures. Assuming a database of known semantic signatures, the aim of this project was to automatically check in possible *direct* executions of C programs whether in any execution it could be malicious[1]. For this, two approaches were tried and implemented.

The first approach involved extracting the library call graph from abstract syntax tree and manually creating the control flow graph only maintaining the important nodes. This approach was previously implemented was limited in scope because the problem of generating *Control Flow Graph*(CFG) from *Abstract Syntax Tree*(AST) is equivalent to writing a compiler for C. Thus, the approach was prone to errors and was not exhaustively covering all the C grammar. The results of this approach were tried Sachin Basil John and showed errors in complicated control flow of program.

The approach in this project however, uses basic gcc directive to generate a *CFG* which then can be used to check for behaviour. This removes the grammar limitations of above approach and is guaranteed to be correct. This approach basically cuts down all the steps for getting CFG from AST. There are, however, some new challenges which are discussed in Chapter 3. After getting CFG, the remaining approach is same as first approach were we determine whether the program is malicious based on execution paths. The algorithm implemented matches exactly from library call to library call thereby requiring a large amount of signatures, a machine learning approach with clustering could be employed to solve this problem.

There were 2 different distance metrics tried out to cluster graph data. Various clustering approaches and techniques are discussed in Chapter 4.

Chapter 2 contains the relevant background required for understanding this report, Chapter 3 deals with a deterministic way of identifying a malware. Whereas in chapter 4, we focus

more on clustering approaches to determine based on clusters/hueristics whether the given program is a malware. Finally, Chapter 5 contains Conclusion and Summary of this report.

1.1 Objective

Write a definition of C program in terms of system calls and check *statically* whether the program C program is harmful even in case of metamorphic viruses.

1.2 Procedure

Following is the high level procedure for statically detecting malware. Detailed explanation of the steps is shown in Chapter 3.

1. Extract the Control Flow Graph of function in C.
2. Interlink the Control Flow Graph of each function to produce the overall control flow graph
3. Analyze all the possible paths that might be invoked on executing the program.
4. Check whether the sequences of library calls (given in the database of harmful program) match to any such sequence in the generated library call graph.
5. If any sequence matches, the program is a malware and it should not be executed.
6. Else, the program is safe to execute.

1.3 Procedure for Clustering

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters).

1.3.1 Data points

The data points in our cluster correspond to signature (compressed graph). The essence of clustering algorithms lies in distance function between 2 data points. As the data points here are graphs, defining a distance function is a challenge, (as discussed in Chapter 4), is a challenge in itself.

1.4 Motivation

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis). In this case the analysis is performed on the source code. Static source code analyzers attempt to find code sequences that, when executed, could result in malicious activities Source code analyzers are effective at locating a significant class of flaws that are not detected by compilers during standard builds and often go undetected during runtime testing as well. The analyzer determines potential execution paths through code, including paths into and across subroutine calls.

Static analysis would detect the malware over all inputs of user space inputs and would could detect a malware which is dormant during runtime analysis but could be malicious in future. Static analysis is done without executing the code, therefore it prevents any damage to system.

Clearly, statically analyzing a program which is very long and running through a malware data-base would take up a long time as it is proportional to the number of signatures in database. This motivates a faster non-deterministic approach via Machine learning.

1.5 Project Scope

Only direct function paths are checked through this approach. There may be multiple indirect ways to call a function (eg. via function pointers) which are impossible to determine via statically because we don't know beforehand which function is going to get called. There may also be accesses to memory via pointers which we may not be able to determine statically.

As the approach in this project details only library calls, program memory is not checked only the possible execution is checked.

For the purpose of this project, all input programs are considered to be direct flows. Direct flows are flows which transfer control directly from program statements without the use of pointers.

Chapter 2

Background

2.1 Basic blocks

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The control flows through basic blocks, thus captures all possible flows in program.

2.2 Control Flow Graph

The control graph of the following simple programming statements is shown below.

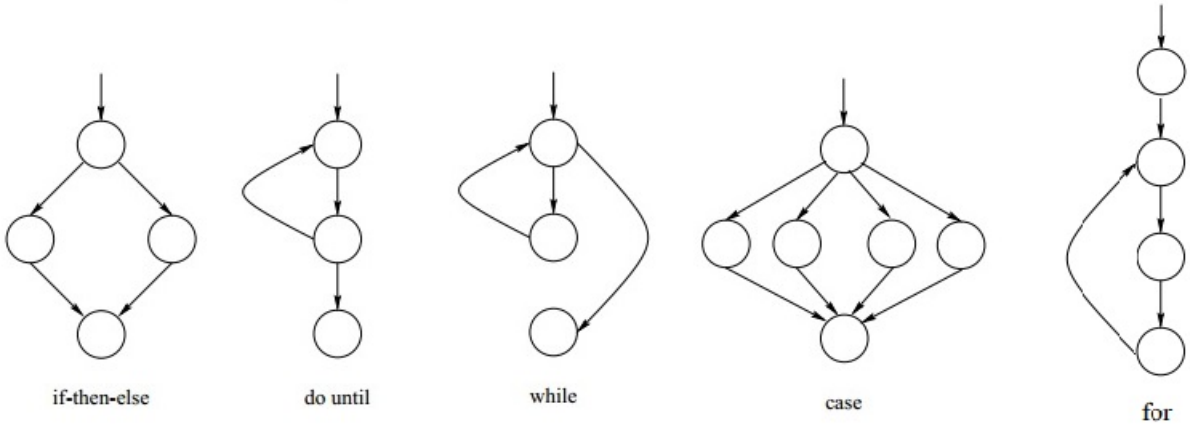
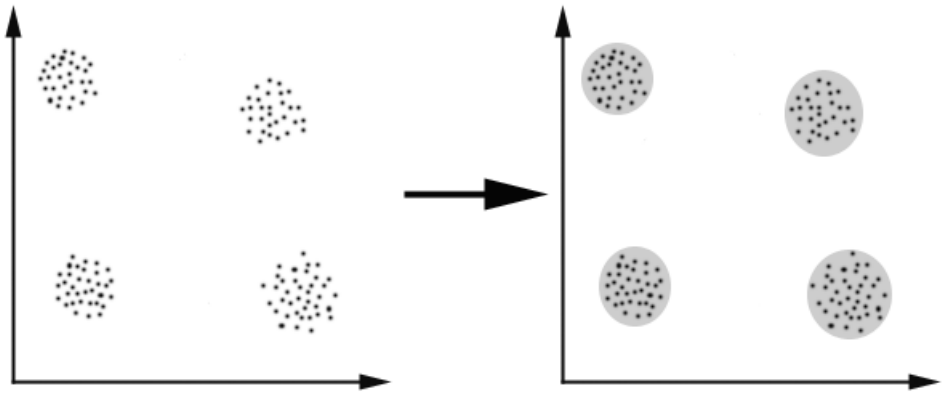


Figure 1: Flow graph representation.

2.3 clustering

Clustering deals with finding a structure in a collection of unlabeled data. A loose definition of clustering could be "the process of organizing objects into groups whose members are similar in some way". A cluster is therefore a collection of objects which are "similar" between them and are "dissimilar" to the objects belonging to other clusters. We can show this with a simple graphical example:



Clustering itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms that differ significantly in their notion of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances among the cluster members, dense areas of the data space, intervals or particular statistical distributions.

Chapter 3

Deterministic Approach

3.1 Approach 1: Getting system call graph from Abstract syntax tree

The credit for designing and starting off with this approach goes to Sachin Basil John. In this project attempted to improve to his basic work.

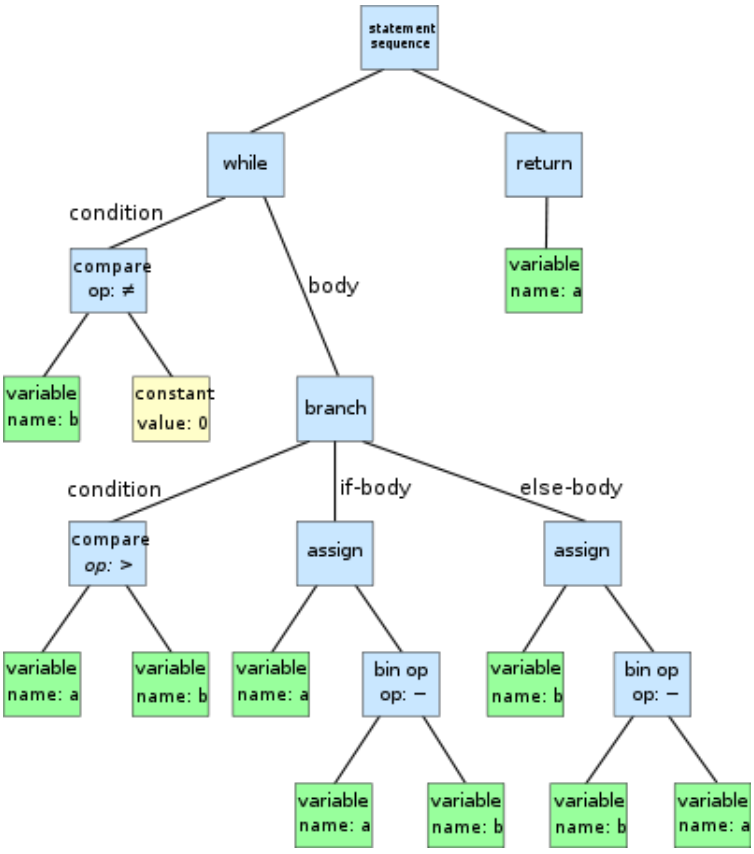
3.1.1 Method

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of the source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. Abstract syntax trees are often built by a parser during the source code translation and compiling process. An AST is usually the result of the syntax analysis phase of a compiler.

Parsing is the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. A parser is a software component that takes input data in some computer programming language (i.e. the source code) and builds a data structure – often some kind of parse tree, abstract syntax tree or hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parser is preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions.

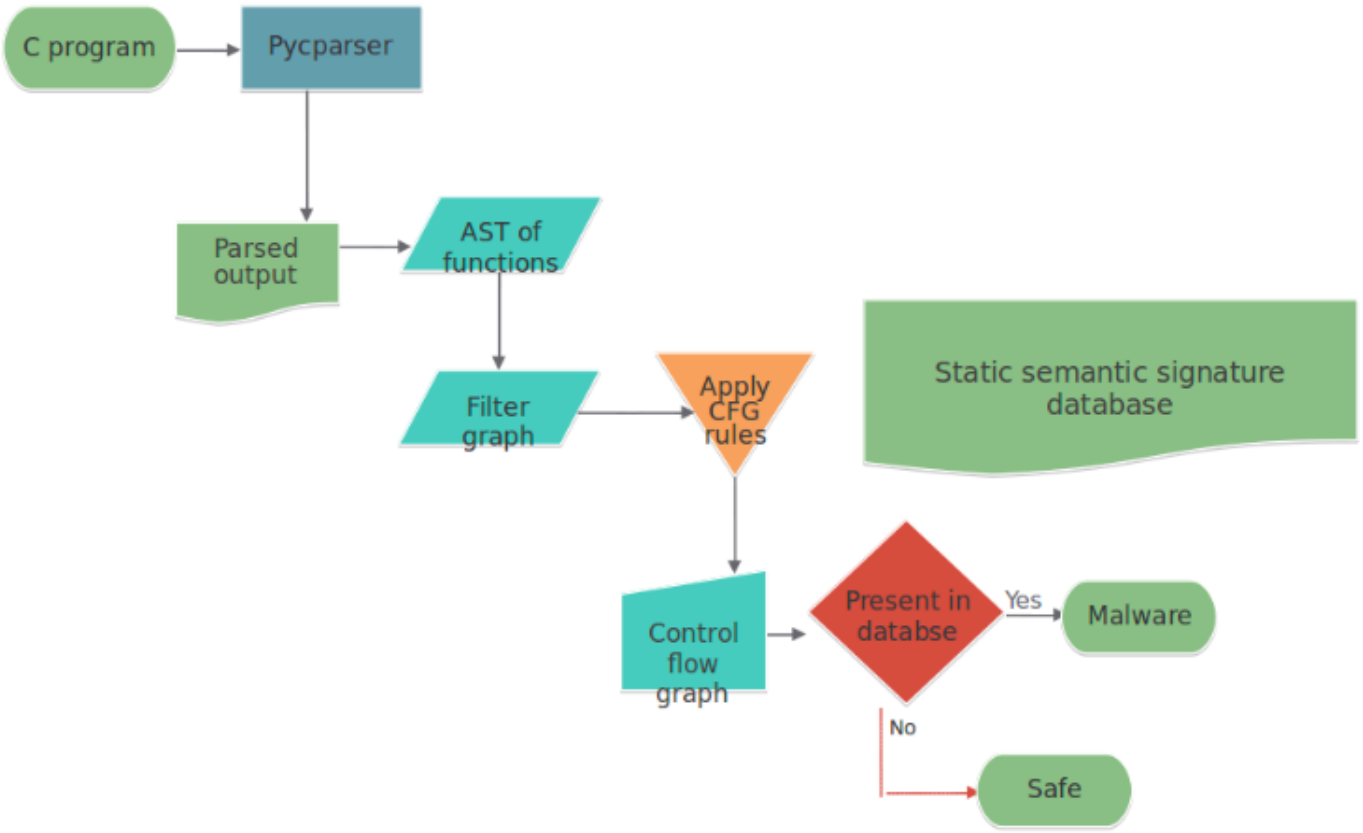
```
while (b != 0)
if (a > b)
      a  = a - b;
   else
```

```
    b = b - a;
  return a
```



pyCparser is a C parser and AST generator written in Python which takes a C program as input and generates its corresponding abstract syntax tree. Thus, we have used pyCparser as a tool to generate ASTs of the programs written in C language.

The flow diagram shown below captures this method.



3.1.2 Drawbacks

This program implemented took C program as input and actually created the compressed graph of important library functions along by writing a compiler like program on AST generated by pycparser[3]. Since, the gcc grammar rules cannot be captured in a thousand lines of code, there were bound to be errors in parsing in more complicated cases. Although, this approach worked fine for small cases, it failed in large cases and was difficult to debug. Moreover, the legacy code by Sachin created a parser for parsed output from pycparser for AST which added the number of errors.

3.2 Second Approach

The new method tried out in this project is to use gcc basic directive to split the code in basic blocks and construct the control flow graph out of it. Note that this captures all direct flows of program and shows the egdes in forms of goto. This does not still fully construct the CFG because edges need to constructed explicitly for functions call and special syscalls such as exit() and abort().

```
#include <unistd.h>
int main()
{
   int a,b;
   a=5;
   while(a--)
   {
      fork();
      b=a+1;
   }
}
```

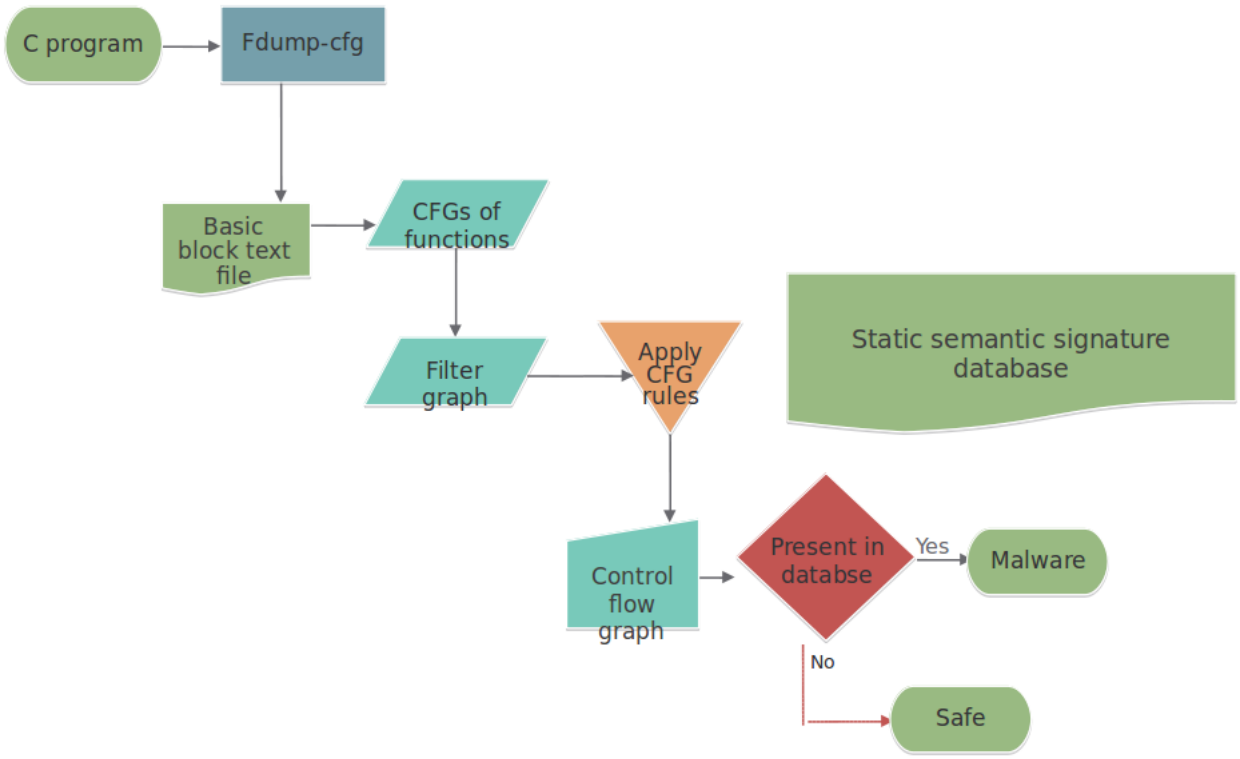
The code shown below is translated to the basic block diagram which is then parsed to produce CFG.

```
int main() ()
{
  int b;
  int a;
  int D.3997;
  int a.1;
  bool retval.0;
  <bb 2>:
  a = 5;
  goto <bb 4>;
  <bb 3>:
  fork ();
  b = a + 1;
  <bb 4>:
  a.1 = a;
  a = a.1 + -1;
  retval.0 = a.1 != 0;
  if (retval.0 != 0)
```

```
    goto <bb 3>;
  else
    goto <bb 5>;
<bb 5>:
  return D.3997;
}
```

Following are the steps involved in Approach 2.

1. Extract the Control Flow Graph of function in C in a .cfg file
2. Parse the .cfg file to get only important functions.
3. Interlink the Control Flow Graph of each function to produce the overall control flow graph
4. Analyze all the possible paths that might be invoked on executing the program.
5. Check whether the sequences of library calls (given in the database of harmful program) match to any such sequence in the generated library call graph.
6. If any sequence matches, the program is a malware and it should not be executed.
7. Else, the program is safe to execute.



The algorithm for generating matching library calls to database sequences is shown below. Imp function in the below algorithm refers to user defined functions which contain security relevant calls, while imp calls are calls which are security relevant.

3.2.1 Special Case about recursive calls

Algorithm 1 without an additional stack cannot be used to represent recursive calls.

To understand this a high level, let us model the entire control graph as DFA. According to DFA, all transitions are forward and in the case of recursive functions we may need to memorize the return address in stack for returning to appropriate values as returning to random value would result in wrong answer. Thus, we need to remember the count(depth) of recursion in node and this is not possible to model in graph(DFA). The proof of this goes by inequality of CFG and DFA[4]. Since, remembering counts requires atleast CFG and we are only using DFA, it is not possible to model recursive functions without explicitly storing the stack of function calls.

Algorithm 1 String matching by DFS

```
1: procedure TRAVERSE(COND_ARR IND CURR_NODE CURR_F)
2:     curr_node.pos ← curr_node.pos[] + ind
3:     Hashset val ← ind
4:     if ind == cond_arr.size() then return true        ▷ Check if all string has matched
5:     for s in curr_node.calls do:
6:         for x in val do:
7:             if s is imp function then traverse(condàrr,ind,curr_node,s)
8:                 val[] = return_value_set
9:             else if s is imp_call and s == cond_arr[x] then
10:                remove x from val, add x+1 to val
11:            else if s is imp_call and s is not cond_arr[x] then
12:                return false
                                  ▷ We have reached the branching part of basic block here
13:    for x in val do:
14:        for ( donode in curr_node.nbrs)
15:            if node is not return node then
16:                traverse(Cond_arr,x,node,f)
17:            elsef.return_value_set.add(x)
```

3.3 Limitations

Both the methods share the basic limitations of false positives. For example, the program listed above executes the while loop exactly 5 times, which may not be possible to know in compile time. This approach would term the above program malware for sequence fork() repeated 6 times but it practice it could only the function 5 times.

Also, there may be possibility that a else part of if-else statement is never executed but our assumes this as possibility. Such could a case where condition part of simple if statement is always true but not possible for compiler to verify in compile time or it could be result of a complicated control flow resulting to this statement.

Chapter 4

Clustering data points

As described before, Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters).

4.0.1 Data points

The data points in our cluster correspond to signature (compressed graph). The essence of clustering algorithms lies in distance function between 2 data points. As the data points here are graphs, defining a distance function is a challenge. Traditional approaches deal with clustering data with a set number of features, then feature engineering to get the correct number of features. And finally, applying some clustering algorithm to produce clusters. Following distance metrics were tried out:

1. First n grams method
2. Distance from diameter method

Standard distance methods such as euclidean distance do not produce nice results as they do not capture the links of graph correctly. Methods such as finding a random common path also do not work correctly. Therefore, the above methods were used.

4.1 First n grams

We first have a set of programs which are known to be malwares. We then traverse their complete CFG to produce a set of possible $n-grams$ and then finally cluster them. The distance functions could be simple distance functions like euclidean distance, hamming distance etc.

Chapter 5

Conclusion and Summary

5.1 Future Work

1. Special case coding for control flow altering calls such as exit, abort
2. To include memory region checking as a part of static semantic signatures.

5.2 Summary

Virus writers and anti-virus researchers generally agree that metamorphism is the way to generate undetectable viruses. Several virus writers have released virus creation kits and claimed that they possess the ability to automatically produce morphed virus variants that look substantially different from one another. Metamorphism is moving to other malware, such as Trojans and Spyware, which are typically distributed from infected or malicious websites. Some of these malwares are changed each time a user visits the infected website upon clicking on a link of a phishing e-mail. This strategy can be described as metamorphism, since the code of the malware itself is changed, often without any encryption. Thus, Metamorphic viruses become extremely difficult to detect. But the method described by us represents C source programs at the system call level. The number of ways to achieve a particular functionality using different sequence of library calls each time is very few.

The method used for implementing this uses basic gcc directives which are proved to be correct. Thus, this covers all grammar and maintains correctness which was not sure in previous cases. Finally, we used clustering algorithms to cluster data points of

Bibliography

[1] N.V.Narendra Kumar R.K.Shyamasundar George Sebastian Saurav Yashaswee. Algorithmic Detection of Malware via Semantic Signatures

[2] Stephen Wolthusen, Evgenios Konstantinou. Metamorphic Virus : Analysis and Detection. Technical Report, RHUL-MA-2008-02, 2008.

[3] pyCparser Too. https://github.com/eliben/pycparser

[4] Proving DFA to CFG Construction Justin Kruskal