Malware Detection based on Dependency Graph using Hybrid Genetic Algorithm

Keehyung Kim
School of Computer Science and Engineering
Seoul National University
599 Gwanak-ro, Gwanak-gu
Seoul, 151-744 Korea
keehyung@snu.ac.kr

Byung-Ro Moon
School of Computer Science and Engineering
Seoul National University
599 Gwanak-ro, Gwanak-gu
Seoul, 151-744 Korea
moon@snu.ac.kr

ABSTRACT

Computer malware is becoming a serious threat to our daily life in the information-based society. Especially, script malwares has become famous recently, since a wide range of programs supported scripting, the fact that makes such malwares spread easily. Because of viral polymorphism, current malware detection technologies cannot catch up the exponential growth of polymorphic malwares. In this paper, we propose a detection mechanism for script malwares, using dependency graph analysis. Every script malware can be represented by a dependency graph and then the detection can be transformed to the problem finding maximum subgraph isomorphism in that polymorphism still maintains the core of logical structures of malwares. We also present efficient heuristic approaches for maximum subgraph isomorphism, which improve detection accuracy and reduce computational cost. The experimental results of their use in a hybrid GA showed superior detection accuracy against state-of-the-art anti-virus softwares.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection—*Invasive software*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms, Experimentation, Security

Keywords

Malware detection, subgraph isomorphism, genetic algorithm, dependency graph

1. INTRODUCTION

Computer Malware is a software or program that damages computer systems or destroys valuable information stored in computers. High use of web browsers and USB devices introduced new types of malwares. Written in script languages such as Visual Basic Script (VBS), javascript, etc., these are known as *script viruses*, distributed in the form of sources, and spreading fast. It is easy to make new variants in high level in that the source code itself is a virus.

Over the time, changing the appearance of malwares, more complex techniques have appeared to avoid detection of anti-virus softwares. One of the most famous techniques is polymorphism. Polymorphic viruses keep the same functionality as the original viruses, while having apparently different structures. It is possible to create a great number of variants based on the one virus by combining several polymorphic techniques.

Furthermore, there exist toolkits that modify the code by junk code insertion and statement reordering, as well as variable renaming, and generate polymorphic viruses automatically [1]. Using those toolkits, anyone can produce polymorphic ones by clicking several buttons in a few minutes. Recent researches show that unknown malwares can be created from the original ones, in the context of evolution, utilizing genetic frameworks [13, 14].

Typical malware detection methods based on signatures therefore have difficulty in detecting polymorphic viruses when they first appear [1, 15] because their signatures are not yet analysed. Although current anti-virus softwares seem sufficient to deal with most of malwares for individual use, they are nevertheless short for fighting against new complex malwares. For these tools, signature-based scanners that search a unique sequence of instructions, are fragile to some variations that can be induced automatically, we need more robust detection algorithms. In this paper, we study how to detect polymorphic malwares, previously unknown, both accurately and effectively.

Polymorphism only changes the appearance of a malware but keeps the contents. Thus, the core part like dependency among important variables and statements of a malware is still valid to be able to perform destructive works as it is supposed to. We also examined polymorphic changes that confuses current anti-virus softwares and tested that they are ineffectual to the dependency graph. The graphs mostly stay similar even when the source code is significantly altered [12]. To figure out whether the target file is a polymorphic variant of a known malware, we model the problem as maximum subgraph isomorphism. Since maximum subgraph isomorphism is NP-hard [6], we propose a genetic algorithm

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

(GA) and heuristics to secure both detection accuracy and time consumption.

Based on the above design, we implemented a dependency graph-based malware detection tool. Experimental results indicate that the proposed method is both effective and efficient. It accurately caught polymorphic malwares and outperformed current anti-virus softwares, and the heuristic approaches reduced time consumption.

The rest of the paper is organized as follows. Section 2 and 3 provides an overview of malware polymorphism and related work. In section 3, we illustrate the detection mechanism and the method we propose. Section 4 describes the malware dataset we collected and generated for the experiment, while section 5 provides experimental results and discussions. Section 6 concludes the paper and gives directions for future work.

2. RELATED WORKS

A number of malware detection methods were proposed to prevent different kinds of viruses, spywares, etc. The simplest and most widely used technology is signature-based one which requires forensic experts to study each malware's behaviour and to update the signature in the database [1]. The signature-based scanner later compares every single file to each malware and tries to find the same sequence of patterns as that of the malware. It is thus hard, if not possible, to detect them in the early stage of their life. Several techniques for virus detection were suggested, to remedy this weakness, including rule learning [17], control flow graph [4], neural networks [18], and data mining [16] approaches.

Since script malwares spread in source-code form, a great degree of freedom in formatting for polymorphism is given to virus makers [1]. Variability in formatting requires the detection system non-case-sensitive approaches different from previous approaches [3]. Unlike other viruses that are propagated in executable format, this leads for forensic experts to a possibility of performing various static analysises. For example, Ko [9] used a flow analysis on macro operations to determine whether it is a malware. The system, based on associated values on variables, extracts the control and data flow from the macro, compares the flow with that of the known suspect, and measures similarity.

A number of polymorphic transformations are also used in the field of code obfuscation. Thus, researches on code plagiarism detection do not only share common traits, but also give meaningful insight into detection on polymorphic script malwares. In an AST-based approach [2, 10], a program is parsed into an abstract syntax tree (AST). Then duplicate subtrees are searched to find similar parts. In a token-based approach [8], a program is first transformed into a sequence of token symbols such as identifiers and keywords. Then, duplicate token subsequences are searched in detection process. Recent research proposed an idea, adapting program dependence graph analysis for core-part detection for software plagiarism [12].

3. MALWARE POLYMORPHISM

Polymorphic viruses confuse virus scanners by changing their appearance and thus make detection hard. There exist many kinds of polymorphism to change appearance of malware codes. Fortunately, most of polymorphic viruses are, however, actually generated by simple rules. The following eight polymorphic techniques are the most well-known which are often used by virus creators and virus mutation or creation engines. Figure 1 shows examples of polymorphic variants. Figures 1(b)-(h) are variants derived from the original, Figure 1(a); they are described in the following.

Format Alteration

It is done by inserting and removing blanks or comments. Format alteration is the simplest and least effective method among others. However, even this simple method sometimes lets malwares avoid detection tools.

Variable Renaming

Identifier names of variables can change consistently without violating program correctness. Variable renaming can confuse human beings, but is almost futile to detection tools [12]. In Figure 1(b), variable identifiers n, p, and i were renamed to a, b, c, respectively.

Statement Reordering

A sequence of some statements can be rearranged while not causing program errors. In Figure 1(c), some statements not dependent on other statements were relocated.

Statement Replacement

Some statements can be replaced with others having the same functionality without damaging the logic. In Figure 1(d), the statement $p = 1$ which assigns value 1 to variable p was replaced by $p = n/5$ exploiting the fact that n equals 5. This is a more complex technique compared to previous three methods in that it changes a statement itself.

Control Replacement

Some control statements perform similar works. For example, a **for** loop and a **while** loop are interchangeable. An example is shown on Figure 1(e). The dependency of statements and variables are still kept.

Junk Code Insertion

Immaterial codes can be inserted to confuse detection, while not disturbing the original logic. As in Figure 1(f), junk codes can be inserted which are inert with respect to the original code; in other words, running junk code does not affect the logic of the original code [1].

Spaghetti Code

Consecutive statements can be scattered and linked together by unconditional jumps such as **goto**, as in Figure 1(g). The execution order remains the same in both pieces of codes.

Subroutine Inlining and Outlining

Code inlining is a technique normally employed to avoid subroutine call overhead, which replaces a subroutine call with the subroutine's code [1]. Outlining is the reverse operation; it need not preserve any logical code grouping, however. Inlining and outlining transformations maintain the original code but deal it in different ways. Figure 1(h) shows an example of outlining.

```
dim n, p, i
n = 5
p = 1
for i = 1 to n do
  p = p * i
end for
```

(a) Original code

```
dim a, b, c
a = 5
b = 1
for c = 1 to a do
  b = b * c
end for
```

(b) Variable renaming

```
dim i, p
p = 1
dim n
n = 5
for i = 1 to n do
  p = i * p
end for
```

(c) Statement reordering

```
dim n, p, i
n = 5
p = n / 5
for i = 1 to n do
  p = p * i
end for
```

(d) Statement replacement

```
dim n, p, i
n = 5
p = 1
i = 1
while i <= n do
  p = p * i
  i = i + 1
end while
```

(e) Control replacement

```
dim n, p, i
n = 5
p = 1
for i = 1 to n do
  if i>0 then
    p = p * i
  end if
end for
```

(f) Junk code insertion

```
dim n, p, i
goto X:
Y:
for i = 1 to n do
  p = p * i
end for
goto Z:
X:
n = 5
p = 1
goto Y:
Z:
```

(g) Spaghetti code

```
dim n, p, i
n = 5
p = 1
for i = 1 to n do
  p = prod(p, i)
end for

function prod(a, b)
return a * b
```

(h) Subroutine outlining

Figure 1: Example of Polymorphism

4. THE PROPOSED SYSTEM

4.1 Overview of the System

The overview of the system is illustrated in Figure 2:

1. The system chooses a known malware P_1's dependency graph G_1 from the virus database and a target file P_2, not classified as either malware or benign yet, to test whether P_2 is a polymorphic variant of P_1.
2. The code of P_2 is parsed and transformed to a code with semantic meaning. As illustrated on Figure 3(a)-(b), each line of the original code is divided into a number of unit statements of semantics.
3. The system extracts a dependency graph G_2 from the semantic code and conducts graph reduction to diminish the size of graph.
4. The system compares the dependency graph G_1 with the target graph G_2 by running heuristics to find the maximum subgraph isomorphism on the two graphs.
5. Based on the results of the Step 4 and a threshold value α, we classify P_2 as a virus and proceed to the next pair, or otherwise goto Step 6.
6. Using a hybrid GA, the system, to draw the final decision, measures the size of maximum subgraph isomorphism.

In the following, we describe each part of the system in detail.

4.2 Variable Dependency Graph

A dependency graph [11] is a directed graph representing the dependencies of objects towards others. Here, we consider a dependency graph based on the relation among the lines of the semantic code. Each vertex represents a line in the semantic code. The dependency between two lines is represented by a directed edge.

Definition 1 (Dependency Edge). *There is a dependency edge from vertex v_1 to vertex v_2 if there is a certain variable X such that X is used on v_2 while the value of X is assigned on v_1.*

Figure 3 provides an example how a dependency graph is generated from the given code.

Simple polymorphisms such as format alteration and variable renaming do not change anything even in the control flow graph. Statement reordering actually changes the order of vertices on the dependency graph, but the structure of that remains the same. More complex techniques such as statement replacement, control replacement, and junk code insertion can add one or more vertices into the dependency graph. However, new vertices do not harm the structure of the previous dependency graph because it should keep the same functionality as before. Spaghetti code alters the control flow of graph and adds vertices for unconditional jump into the dependency graph. Subroutine inlining and outlining is the most complex technique, and changes the dependency graph a great deal. In [12], Liu et al studied that program dependence graph is robust to five disguises in software plagiarism: format alteration, variable renaming, statement reordering, control replacement, and junk code insertion.

4.3 Graph Reduction

The size of a dependency graph may be reduced. Some part of the code where the control flow never reaches can be removed. In addition, vertices that satisfy one of the following four conditions can be eliminated:

- A vertex with only one outgoing edge without any incoming edge. It is mostly the declaration of a variable, which is not critical when only considering the core part of the program.
- A vertex with only one incoming edge without any outgoing edge. It means that the first vertex uses the value of the latter one.
- A vertex with only one incoming and one outgoing edge. It plays a role in conveying a value or data from one vertex to another mainly.

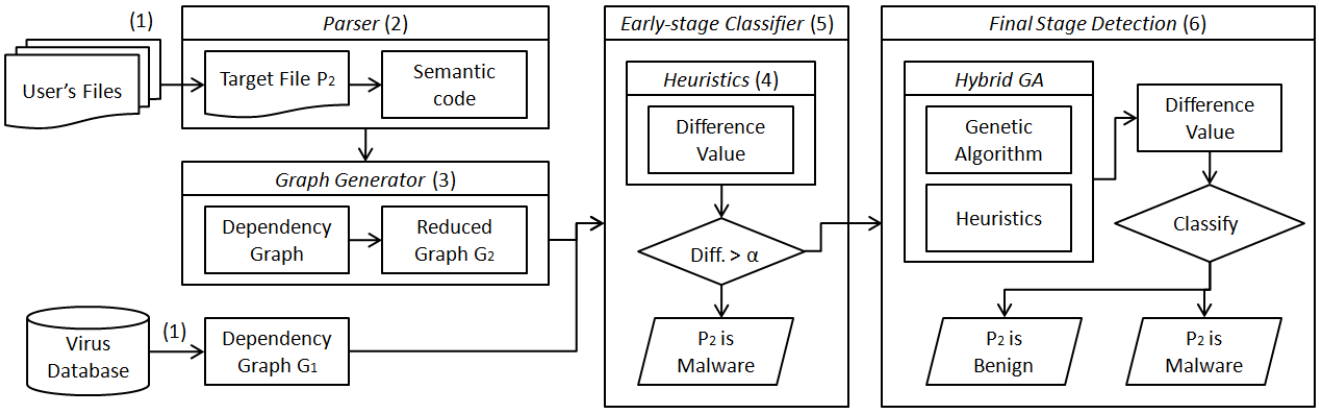


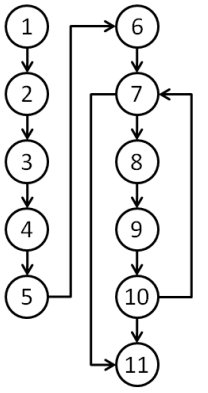
Figure 2: Overview of detection mechanism

```
1: dim n, p, i
2: n = 5
3: p = 1
4: for i = 1 to n do
5:     p = p * i
6: end for
```

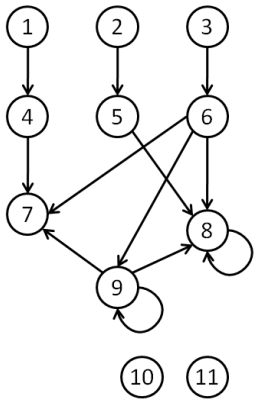
(a) Original code

```
1: dim n
2: dim p
3: dim i
4: n = 5
5: p = 1
6: i = 1
7: if i ≤ n then
8:     p = p * i
9:     i = i + 1
10:    goto 7:
11: end if
```

(b) Semantic code



(c) Control Flow Graph



(d) Dependency Graph

Figure 3: An illustrative example for dependency graph: From the given (a) original code, at first, (b) semantic code is produced. Analysing control flow of the code, the system easily extracts (c) control flow graph. (d) Dependency graph is later constructed based on dependencies of the semantic code and the order of the control flow graph.

- A vertex without any incoming or outgoing edge. It is regarded as a non-necessary redundant part.

In Figure 4, vertices 10 and 11 are unnecessary since the lines 10 and 11 of Figure 3(b) do not have any special semantic meaning. Vertices 1, 2 and 3 have only one outgoing

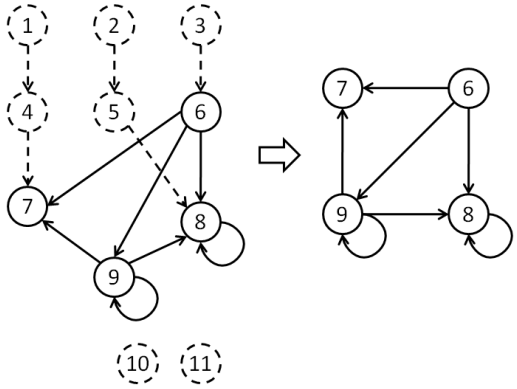


Figure 4: Reduce the size of dependency graph

edge. Vertices 4 and 5 have only one incoming and one outgoing edge. Thus, the original dependency graph with 11 vertices can be reduced to one with 4 vertices.

4.4 Subgraph Isomorphism

We use subgraph-isomorphism testing on the target file and the malware to determine whether the target is a polymorphic variant. Related terminologies are listed below.

DEFINITION 2 (GRAPH ISOMORPHISM). *A bijective function $f : V \rightarrow V'$ is a graph isomorphism from a graph $G = (V, E)$ to a graph $G' = (V', E')$ if*
- $\forall e = (v_1, v_2) \in E, \exists e' = (f(v_1), f(v_2)) \in E'$
- $\forall e' = (v_1', v_2') \in E', \exists e = (f^{-1}(v_1'), f^{-1}(v_2')) \in E$

DEFINITION 3 (SUBGRAPH ISOMORPHISM). *An injective function $f : V \rightarrow V'$ is a subgraph isomorphism from G to G' if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S.*

We finally solve the following optimisation problem.

DEFINITION 4 (MAXIMUM SUBGRAPH ISOMORPHISM). *The problem of maximum subgraph isomorphism is to find $G_S = (V_S, E_S)$ which satisfies subgraph isomorphism to G' if there exists a subgraph $G_S \subset G = (V, E)$ while maximizing $|E_S|/|E|$.*

4.5 Genetic Operators

Since subgraph isomorphism is an NP-hard problem, a GA is appropriate. A GA generates a set of initial solutions and lets them evolve over a number of iterations. When GA meets some condition, the best solution is returned and the algorithm terminates. Our algorithm replaces 20 percent of the population per generation and uses two local optimisation heuristics after crossover and mutation (hybrid or memetic GA). In the following, the algorithm for the proposed system is described.

Representation

Linear encoding is used for each chromosome to represent the arrangement of vertices. Each gene value in a chromosome represents the location of the corresponding vertex in an arrangement (permutation).

Fitness function

We measure the proportion of the number of different edges between two graphs against the number of edges in the smaller graph, to evaluate a chromosome. We call this the *difference*, noted by d, of the graphs. The difference value 0 means that G_1 is a complete subgraph of G_2.

$$I(e, E) = \begin{cases} 0 & \text{if } e \in E \\ 1 & \text{otherwise.} \end{cases} \quad (1)$$

$$d = \frac{\sum_{e \in E_1} I(e, E_2) + \sum_{e \in E_2} I(e, E_1)}{|E_1|} \quad (2)$$

where $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, and $|V_1| \leq |V_2|$. The computation takes $\theta(E)$ time. The smaller the value d, the higher the fitness.

Initialisation

When GA starts, one solution is created in increasing order, and the rest 99 solutions are created at random. That is, the population size is 100. Twenty of them are replaced each generation.

Selection

The roulette-wheel-based proportional selection is used. The probability that the best chromosome is chosen was set to four times higher than the probability that the worst chromosome is chosen.

Crossover and Mutation

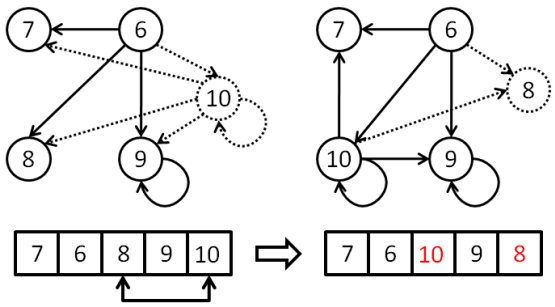
Cycle crossover and PMX [7] are used. We first produce two offspring. Out of them, the better one is chosen as the final offspring. For mutation, we select two genes at random and exchange them. The rate of crossover and mutation is set to 0.9 and 0.2, respectively.

Replacement

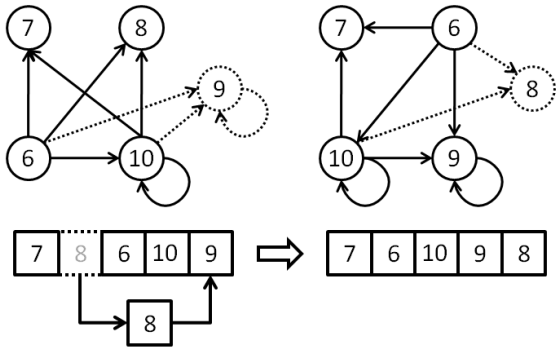
We replace the worst members of the population with the new 20 offspring.

Stopping criterion

The GA stops as soon as we find a complete subgraph isomorphism with d value 0. Otherwise, we set the number of maximum generation to 2,000 and 100 in the standard GA and the hybrid GA, respectively.



(a) Exchange locations of two vertices



(b) Relocate a vertex to another place

Figure 5: Assume a reduced graph on Figure 4, a graph on the left of (a), and that of (b) as G, G_a, and G_b. (a) G is the exact subgraph of G_a if the location of vertex 8 is exchanged with that of vertex 10. (b) G is the exact subgraph of G_b if the location of vertex 8 is moved from the second to the fifth.

Local optimisation

We use two heuristics for local optimisation. Details of those optimisations are described in Section 4.6.

4.6 Heuristics

We devised two heuristics (Figure 5) to improve the quality of arrangements for maximum subgraph isomorphism.

Algorithm 1 Two-vertex exchange heuristic

```
1: for v1 ∈ V do
2:   for v2 ∈ V and v2 ≠ v1 do
3:     Copy and make G' from G
4:     Exchange vertex v1 and v2 on G'
5:     if fitness of G' is less than G then
6:       Copy G' into G
7:     end if
8:   end for
9: end for
```

The first one, shown in Algorithm 1, is similar to 2-Opt [5] in the travelling salesman problem (TSP). On a solution, every possible combination of two vertices are considered for being swapped. The algorithm, then, exchanges the locations of two vertices if the change causes an improvement. There exist $\theta(|V|^2)$ combinations of selecting two vertices.

The other is shown in Algorithm 2. For every vertex, the

Algorithm 2 Vertex relocation heuristic

```
for v1 ∈ V do
  for location do
    Copy and make G' from G
    Remove vertex v1 and insert it into location on G'
    if fitness of G' is less than G then
      Copy G' into G
    end if
  end for
end for
```

Table 1: Malwares used for test

Virus Name	Polymorphic Variants
Neves	Neves.a, Neves.b, Neves.c, Neves.d
Rabbit	Rabbit.a, Rabbit.b
Internal	Internal.a, Internal.b, Internal.c, Internal.f, Internal.g
Small	Small.a, Small.b
Hello	Hello

Table 2: Malwares generated using polymorphic techniques

Variation	Used Polymorphism
Hello.v1	Format alteration
Hello.v2	Statement replacement
Hello.v3	Format alteration, variable renaming
Hello.v4	Format alteration, variable renaming, statement replacement, junk code insertion, spaghetti code

algorithm tries to find the optimal location. A new solution is taken if relocation of the vertex into another location improves the fitness. There exist V vertices to move and $V-1$ possible locations for each vertex.

5. DATASET

We collected fourteen script viruses of five different series of Visual Basic Script (VBS) malwares including polymorphic variants from VX Heaven[1]: Neves, Rabbit, Internal, Small, and Hello. Table 1 shows the categorization of the viruses. Malwares in the same category are regarded as polymorphic variants of the original.

In addition, we also created four polymorphic variants based on Hello virus newly (Table 2). Hello.v1 and Hello.v2 contain only one polymorphic change on Hello virus while Hello.v3 contains two polymorphic changes. Hello.v4 adopts five polymorphic techniques, and thus is considered as a very strong polymorphic virus.

6. RESULTS AND DISCUSSION

A number of experiments were conducted. We provide experimental results of the heuristic approaches, and those of the GA and anti-virus softwares.

6.1 Effect of Graph Reduction

Tables 3 and 4 show the number of nodes and edges on the dependency graph of each virus before (column "Node") and after (column "# Nodes after Reduction") graph reduction. The column "Ratio" represents the rate of "# Nodes after Reduction" to "# Nodes."

For all malwares, the number of nodes and edges in each dependency graph was reduced significantly after graph reduction. On average, the size of a dependency graph was reduced to 36.78% (nodes) and 57.11% (edges) of the original. It means approximately 63% (on nodes) and 43% (on edges) of each dependency graph do not have a meaningful role. The reduced graph, however, still preserves the core characteristics of each malware.

In the case of Rabbit series, there was not any remaining vertex after graph reduction because of simple codes of those

[1]VX Heavens Virus Collection, `http://vx.netlux.org`

Table 3: Graph reduction on Node

Virus	# Nodes	# Nodes after Reduction	Ratio
Hello	53	25	47.17
Hello.v1	53	25	47.17
Hello.v2	53	25	47.17
Hello.v3	53	25	47.17
Hello.v4	50	25	50
Neves.a	60	21	35
Neves.b	72	25	34.72
Neves.c	100	21	21
Neves.d	122	39	31.97
Rabbit.a	13	0	0
Rabbit.b	14	0	0
Internal.a	65	31	47.69
Internal.b	40	18	45
Internal.c	35	13	37.14
Internal.f	39	18	46.15
Internal.g	160	65	40.63
Small.a	14	4	28.57
Small.b	18	10	55.56

Table 4: Graph reduction on Edge

Virus	# Edges	# Edges after Reduction	Ratio
Hello	59	38	64.41
Hello.v1	59	38	64.41
Hello.v2	59	38	64.41
Hello.v3	59	38	64.41
Hello.v4	55	38	69.09
Neves.a	65	32	49.23
Neves.b	81	43	53.09
Neves.c	71	32	45.07
Neves.d	134	73	54.48
Rabbit.a	9	0	0
Rabbit.b	10	0	0
Internal.a	86	68	79.07
Internal.b	47	39	82.98
Internal.c	32	22	68.75
Internal.f	47	39	82.98
Internal.g	163	110	67.48
Small.a	14	5	35.71
Small.b	17	14	82.35

Table 5: Difference value on maximum subgraph isomorphism by Heuristics

Virus	Hello	Neves	Rabbit	Internal	Small
Hello	**0**	81.25	144.44	72.72	60
Hello.v1	**0**	81.25	144.44	72.72	60
Hello.v2	**0**	81.25	144.44	72.72	60
Hello.v3	**36.84**	65.62	88.88	72.72	80
Hello.v4	**36.84**	65.62	88.88	72.72	80
Neves.a	81.25	**0**	111.11	86.36	60
Neves.b	92.1	**34.37**	122.22	86.36	60
Neves.c	81.25	**0**	111.11	86.36	60
Neves.d	97.36	106.25	88.88	**77.27**	80
Rabbit.a	144.44	111.11	**0**	188.88	60
Rabbit.b	180	130	**11.11**	160	60
Internal.a	118.42	121.87	155.55	113.63	**80**
Internal.b	**65.78**	106.25	155.55	113.63	80
Internal.c	72.72	59.37	188.88	**0**	60
Internal.f	**65.78**	87.17	155.55	113.63	80
Internal.g	102.63	81.25	111.11	81.81	**80**
Small.a	60	60	60	80	**0**
Small.b	92.86	**57.14**	100	85.71	80

Table 6: Difference value on maximum subgraph isomorphism by hybrid GA

Virus	Hello	Neves	Rabbit	Internal	Small
Hello	**0**	28.13	22.22	40.91	40
Hello.v1	**0**	28.13	22.22	36.36	40
Hello.v2	**0**	28.13	11.11	36.36	40
Hello.v3	**0**	31.25	22.22	31.82	40
Hello.v4	**0**	34.38	22.22	36.36	40
Neves.a	28.13	**0**	22.22	36.36	40
Neves.b	63.16	**0**	22.22	31.82	40
Neves.c	28.13	**0**	22.22	36.36	40
Neves.d	31.58	**0**	33.33	27.27	40
Rabbit.a	22.22	22.22	**0**	155.56	60
Rabbit.b	20	40	**0**	130	60
Internal.a	55.3	40.6	**22.2**	27.3	40
Internal.b	20	22	77.78	**0**	40
Internal.c	40.91	36.36	155.56	**0**	40
Internal.f	21	22	77.78	**0**	40
Internal.g	44.7	40.6	44.4	50	**40**
Small.a	40	40	60	40	**0**
Small.b	28.57	42.86	57.14	35.71	**0**

malwares. We thus did not adopt graph reduction on Rabbit variants for experiments.

6.2 Experimental Results

Table 5 shows the difference values when only heuristics are applied to detection. Eight variants were detected correctly (with difference value 0) without any mis-categorization. Since heuristics take fairly less time than a GA does, it can save significant time if used for the purpose of filtering at the first stage of the detection system. If we set a more generous threshold, say $\alpha = 40$, then 12 variants out of 18 malwares were detected.

Among totally 18 malwares, 16 variants were well-corrected by the hybrid GA (Table 6). Internal.a and Internal.g have some different contents although they belong to Internal malware series. Thus, they are of more than polymorphism, and detecting those is somewhat beyond the scope of our mission.

Although the heuristics alone are not strong enough, they are useful for screening at the early stage of polymorphism detection in that they can screen out some pairs of graphs before the GA, the time-consuming work, runs.

6.3 Comparison with Anti-Virus Softwares

We tested new polymorphic variants derived from five original malwares. To test that currently existing anti-virus softwares can detect unknown polymorphic malwares, we used 41 softwares available on VirusTotal[2]. Table 7 shows the experimental results. When polymorphic variants of Hello virus were tested, 29%, 12%, 7%, 2%, and 0% of the anti-virus softwares detected the five polymorphic viruses, respectively. The proposed system, on the other hand, detected all of the five.

Twelve of anti-virus softwares detected Hello virus, but the number decreased significantly and remained less for higher degree of polymorphism. Surprisingly, just introducing the simplest polymorphism, format alteration, could confuse more than half of the anti-virus softwares. Furthermore, any of the 41 anti-virus softwares could not detect Hello.v4 variant which contains five polymorphic techniques. It demonstrates how existing anti-virus softwares are weak against polymorphism.

Table 7: Comparison with anti-virus softwares

Virus Variant	Anti-Viruses	Proposed system
Hello	12/41 (29.27%)	Detected
Hello.v1	5/41 (12.2%)	Detected
Hello.v2	3/41 (7.32%)	Detected
Hello.v3	1/41 (2.44%)	Detected
Hello.v4	0/41 (0%)	Detected
Neves.a	32/40 (80%)	Detected
Neves.b	32/40 (80%)	Detected
Neves.c	25/41 (60.98%)	Detected
Neves.d	25/41 (60.98%)	Detected
Rabbit.a	36/41 (87.8%)	Detected
Rabbit.b	33/39 (84.62%)	Detected
Internal.a	14/41 (34.15%)	Undetected
Internal.b	19/41 (46.35%)	Detected
Internal.c	16/40 (40%)	Detected
Internal.f	13/41 (31.71%)	Detected
Internal.g	21/41 (51.22%)	Undetected
Small.a	35/41 (85.37%)	Detected
Small.b	7/41 (17.08%)	Detected

Table 7 also contains the detection rates of other existing polymorphic variants. Although the rates were not as bad as the newly generated variants (Hello.*), one can observe that the existing anti-virus softwares are still poor in detecting polymorphic variants. The hybrid GA successfully detected all of the variants except for two of them that contain more than polymorphic variants. Although, in VX Heaven, five Internal variants are grouped as Internal virus series, some anti-virus softwares also distinguish some of them from others. For example, one anti-virus software recognises both

[2]VirusTotal website, `http://www.virustotal.com`

Internal.b and Internal.f as Internal series while Internal.g as Zeha series. Therefore two undetected Internal variants (Internal.a and Internal.g) are too different from others (Internal.b, Internal.c, and Internal.f) in the concept of polymorphism.

7. CONCLUSIONS

We proposed a malware detection mechanism based on the dependency graph analysis and using GAs. Since a large proportion of malwares are in script format and propagated through USB and web browsers, the dependency analysis of malware codes seems to be fairly useful to detect the unknown polymorphic malwares. We, especially, formulate malware detection as maximum subgraph isomorphism problem and use a hybrid GA; the approach outperformed existing anti-virus softwares in VirusTotal. Especially, graph reduction and the two heuristic approaches are also crucial parts of the system.

A limit of the proposed system is computational cost of GA. Although graph reduction and heuristics save the computational time significantly, the GA still requires fairly more time to process than current anti-virus softwares based on signature-based scanners. The approaches we used on malware detection can be extended to the area of software plagiarism detection, too, and other applications that use maximum subgraph (or graph) isomorphism in some sense. Extensive experiments on the various size and types of graphs are in the set of future topics.

Acknowledgements

This work was supported by the Brain Korea 21 Project and Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2009-0063242). The ICT at Seoul National University provided research facilities for this study.

8. REFERENCES

[1] J. Aycock. *Computer Viruses and Malware*. Springer, 2006.

[2] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pages 368–377, 1998.

[3] V. Bontchev. Macro virus identification problems. *Computers and Security*, 17(1):69–89, 1998.

[4] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *DIMVA '06: Proceedings of the Conference on the Detection of Intrusions and Malwares and Vulnerability Assessment*, pages 129–143, 2006.

[5] T. Bui and B. Moon. A new genetic approach for the traveling salesman problem. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.*, volume 1, pages 7–12, 1994.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[9] C. W. Ko. Method and apparatus for detecting a macro computer virus using static analysis, February 2004. United States Patent #6,697,950 B1.

[10] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. Working Notes of 3rd Workshop on AI and Software Engineering, 1995.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[12] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, 2006.

[13] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq. Evolvable malware. In *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1569–1576, 2009.

[14] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq. Using formal grammar and genetic operators to evolve malware. In *RAID: 12th International Symposium On Recent Advances In Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 374–375, 2009.

[15] S. Pearce. Viral polymorphism. Sans Institute, 2003.

[16] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 38, 2001.

[17] M. Z. Shafiq, S. M. Tabish, and M. Farooq. On the appropriateness of evolutionary rule learning algorithms for malware detection. In *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 2609–2616, 2009.

[18] G. Tesauro, J. Kephart, and G. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, 1996.