Detecting Virus Mutations Via Dynamic Matching

Min Feng Rajiv Gupta
CSE Dept., University of California, Riverside
{mfeng,gupta}@cs.ucr.edu

Abstract

To defeat current commercial antivirus software, the virus developers are employing obfuscation techniques to create mutating viruses. The current antivirus software cannot handle the obfuscated viruses well since its detection methods that are based upon static signatures are not resilient to even slight variations in the code that forms the virus. In this paper, we propose a new type of virus signature, called dynamic signature, and an algorithm for matching dynamic signatures. Our dynamic signature is created based on the runtime behavior of a virus. Therefore, an obfuscated virus can also be detected using a dynamic signature as long as it dynamically behaves like the original virus. We also discuss issues related to deploying our virus detection approach. Our experiments based upon several known mutating viruses show that our method is effective in identifying obfuscated viruses.

1. Introduction

A computer virus is a computer program which can propagate itself without the user's consent. Usually, it has malicious intent, e.g. to damage a computer system or acquire information without owners' permission. An antivirus software is used to identify and remove computer viruses.

Existing commercial antivirus software use static signatures to identify viruses. Each signature is a static pattern of program code which is present in every infection of the virus. A file is declared as infected if it contains a sequence of instructions that is matched by a static signature. For each virus variant, at least one signature is needed so that the virus can be detected. The procedure of creating a signature involves human effort and is time-consuming. With the rapid growth of viruses, commercial antivirus software seems to be in difficulty. Symantec Internet Security Threat Report states "the release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications."

Besides the explosion of virus quantity, the virus developers also update their viruses frequently to evade detection from antivirus software. In each update, they only add or change a small portion of code. Usually, the updates have two purposes. The first purpose is to mutate the code in order to invalidate the static signatures used by the antivirus software. Therefore, the antivirus software developers have to develop new signatures for detecting different variants. The second purpose is to add new anti-detector features. For example, *Bagle*, which is a widespread virus family, was once updated to include a new backdoor which can disable the security system in the local machine.

To further protect their viruses from antivirus software, the virus developers apply obfuscation techniques to the virus programs [16]. The obfuscation techniques they use include instruction substitution, instruction permutation, insertion of superfluous instructions, variable renaming, and control-flow alteration. A *self-mutating virus* is a virus which obfuscates itself each time it infects a new file. It is hard for current antivirus software to detect a *self-mutating virus* since its code keeps changing. Theoretically, two self-mutating virus variants may seem different in static code but dynamically behave the same. Therefore, it is difficult to develop a single static signature which can be used to identify all variants of a *self-mutating virus*.

This paper aims to design a virus detection approach which is resilient to obfuscation. In this paper, we propose a novel type of virus signature, called *dynamic signature*, which is created based on the runtime behavior of the viruses. Each dynamic signature is a backward slice of a library function call which is unique to the corresponding virus' runtime trace. No matter how the viruses mutate their static code, as long as they dynamically behave in a similar fashion, their dynamic signatures do not change. We also present an algorithm for matching dynamic virus signatures and extracting a good dynamic signature from a set of known virus variants. The matching algorithm in this paper is effective in matching virus signatures – matching algorithm presented in our prior work was not found to be effective for this purpose [26]. To make our dynamic virus detection approach practical, we come up with a solution for deploying our detection method and develop a few techniques for reducing the overhead. Specifically, in this paper we make the following contributions:

1. *Dynamic signature*. We propose a new type of virus signature, dynamic signature, which is much more resilient to obfuscation than static signature. The criteria for extracting a dynamic signature is formally defined.

2. *Signature matching algorithm*. We present a method for searching a signature in the runtime trace of a target program. This algorithm can handle a set of obfuscation transformations, including variable renaming, instruction permutation, insertion of superfluous instructions, control flow alteration, and limited set of instruction substitutions.

3. *Evaluation.* Our evaluation based upon known viruses shows that our virus detection method is very effective in identifying different variants belonging to the same virus family including self-mutating viruses. Moreover, in our experiments we did not observe any false positives.

The remainder of the paper is organized as follows. In section 2 we develop the notion of *dynamic virus signatures* and develop a *dynamic matching* algorithm to identify these signatures in execution traces of programs. We also discuss practical issues of safety and cost in deploying a dynamic virus detection technique. In section 3 we discuss our implementation and present the results of our experimental evaluation. In section 4 we discuss related work and we conclude in section 5.

2. Dynamic Virus Detection

In this section we motivate the need for dynamic detection of viruses and then develop the formal definition of *dynamic signature* for virus detection. We also provide an algorithm for searching for a dynamic signature through the runtime trace of a program run. To give an intuitive explanation of the underlying ideas, we use the sample program shown in Fig.1.

```
1   lea edi, ptr [ebp+0x4025]   // edi = mem[ebp+...]
2   mov ecx, 0x3ec5             // ecx = 0x3ec5
3   mov edx, 0xef4013a0         // edx = 0xef4013a0
    loop:
4   mov al, byte ptr ds[edi]    // al = mem[ds+edi]
5   sub al, dl                  // al = al - dl
6   sub al, dh                  // al = al - dh
7   xor al, cl                  // al = al ^ cl
8   rol edx, cl                 // rotate edx by cl bits
9   mov byte ptr ds[edi], al    // mem[ds+edi] = al
10  inc edi                     // edi = edi + 1
11  dec ecx                     // ecx = ecx - 1
12  jnz loop                    // jump
13  push edi                    // push args into stack
14  call 0x7c92a950             // call a lib func
```

Figure 1. Sample code extracted from *Bagle*.

This example is extracted from the assembly code of *Bagle*, which is a widespread email-based virus. For ease of presentation and understanding we have performed some simplifications on the original code. The key part of the sample code is a loop formed by instructions 4 through 12. The instructions preceding the loop (i.e., instructions 1 through 3) initialize the loop counter (`ecx`), starting address (`edi`), and another variable (`edx`). During each iteration, the loop fetches a value from the data segment, performs a calculation based upon that value, and then finally puts the new computed value back into the data segment. Following the loop, the program calls a library function that uses the newly computed values – the use of these values triggers the actions of the virus. Many different kinds of obfuscation transformations [16] can be applied to this piece of code to affect mutations of the *Bagle* virus. We will consider such mutations to illustrate the functioning of our dynamic signature-based virus detection technique.

2.1. Static Signatures

Before presenting our dynamic technique, in this section, we briefly discuss how static signatures are used for virus detection and how the detection can be thwarted when the original virus is mutated via obfuscation transformations. Static signature based virus detection is the most common method that is employed by traditional antivirus software for identifying viruses. When antivirus software scans a file for viruses, it compares the contents of the file to a dictionary of known virus signatures. If a virus signature is found in the file, then the file is treated as infected and the antivirus software can take actions that remove the virus.

A static virus signature consists of a sequence of machine language instructions which are both unique to the virus and are present in every one of the virus' infections. Every time a new virus is encountered, it is analyzed to identify its signature which then inserted in the dictionary of static signatures used by the antivirus program so that future encounters with this virus can be detected and dealt with. For example, we could use the sequence of instructions shown in Fig.1 as the virus signature to identify whether a file is infected by *Bagle*. Suppose the above segment of code is unique to *Bagle*, this is a very efficient way to identify *Bagle* infections.

However, as the antivirus software evolves, the virus authors also develop ways to hide their viruses. One way is that the authors can regularly update the virus codes, e.g. mutating independent statements or changing the data segment in the virus, in order to hide the viruses from the antivirus software. Another more sophisticated way is to create a virus which can avoid detection by mutating itself each time it infects a new program. Each new mutation essentially performs the same task as its parent. These type of virus is called a *self-mutating virus*. Because both methods involve code mutations, they are effective in thwarting detection by traditional static signature based antivirus software. For example, in the sample program shown in Fig.1, we could change the relative order of instructions 2 & 3, 7 & 8, and 10 & 11 without altering the semantics of the code since instructions in each pair are independent of each other. After mutation, the antivirus software cannot use the

original static signature to identify the new infection any more. In order to detect a virus and its variants, the traditional antivirus detectors have to use shorter signatures. For example, in the sample program, instruction 4-6 could be used as a signature to identify both the original and mutated viruses since they are dependent to each other and thus hard to mutate. But as signatures get smaller, the likelihood of *false identification* increases. Therefore, it is easy to see that the traditional virus signature does not work well when a virus mutates repeatedly.

To identify the viruses in the same family, some new detection methods based on control-flow or call graphs were proposed [10]. However, current obfuscation techniques can alter the program's structure by introducing useless conditional and unconditional branch instructions such that at runtime, the order in which the program executes instructions is unchanged. For example, the sample program could become the one shown in Fig.2 after being obfuscated.

```
1   lea edi, ptr [ebp+0x4025]   // edi = mem[ebp+...]
2   mov edx, 0xef4013a0         // edx = 0xef4013a0
3   mov ecx, 0x3ec5             // ecx = 0x3ec5
    loop:
4   mov al, byte ptr ds[edi]    // al = mem[ds+edi]
5   mov ebx, 0                  // ebx = 0
    l1:
6   cmp ebx, 1                  // ebx == 1 ?
7   je l2                       // jump to l2 if true
8   sub al, dl                  // al = al - dl
9   inc ebx                     // ebx = ebx + 1
10  jmp l1                      // jump to l1
    l2:
11  sub al, dh                  // al = al - dh
12  rol edx, cl                 // rotate edx by cl bits
13  xor al, cl                  // al = al ^ cl
14  mov byte ptr ds[edi], al    // mem[ds+edi] = al
15  sub ecx, 1                  // ecx = ecx - 1
16  inc edi                     // edi = edi + 1
17  sub edi, 0                  // garbage code
18  jnz loop                    // jump
19  push edi                    // push args into stack
20  call 0x7c92a950             // call a lib func
```

Figure 2. Mutated version of sample code.

In the mutated code shown in Fig.2, we permute the order of three pairs of instructions (2 & 3, 12 & 13, 15 & 16), substitute "sub ecx, 1" for "dec ecx", insert a superfluous instruction 17, and alter its control flow by adding new instructions (5, 6, 7, 9 & 10). These new instructions will not change the program semantics but prevent the antivirus software from identifying the virus using static signature.

From the above example, we can see that it is easy to change the static structure of a virus while keeping its runtime semantics the same. Therefore, the static information based antivirus software could be easily overwhelmed by the obfuscation techniques. In the following sections, we present our dynamic signature which is created based on a virus' runtime trace and illustrate how to identify an infection by using dynamic signature.

2.2. Dynamic Signatures

In this section, we develop the notion of dynamic signature for viruses and then present an algorithm for detecting dynamic signatures in the next section. The dynamic signature we develop is in form of a *Dynamic Data Dependence Graph* (dDDG). Thus, unlike the aforementioned static signatures, a dynamic signature will capture a virus' runtime behavior. Searching for the dDDG corresponding to a dynamic signature of a virus in the dDDG for an entire program run will be a highly expensive. However, to make the cost acceptable, we identify short fragments of *Dynamic Backward Slices* (DBSs) [28] computed at selected program points. As we will show, dynamic signature detection, which we refer to as *dynamic matching*, is quite robust in the presence of code obfuscation techniques.

The dDDG of a program run is a directed acyclic graph representing data dependences between different execution instances of statements in a program run. More precisely, a dDDG is defined as follows:

Definition 1. The dDDG of a program run, $dDDG(N, E)$, consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ represents the i^{th} execution instance of statement n in the program; and each edge $m_j \rightarrow n_i \in E$ corresponds to a dynamic *data* dependence of the i^{th} execution instance of statement n on the j^{th} execution instance of statement m.

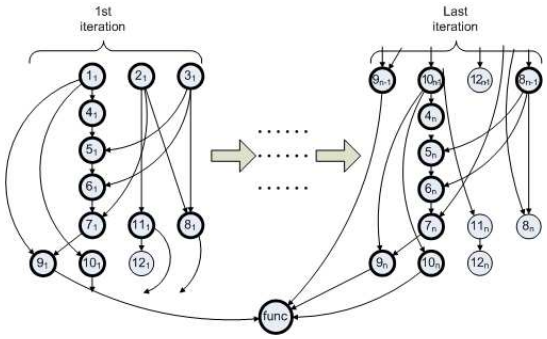


Figure 3. dDDG of the sample program.

Fig.3 shows the dDDG of a program run of the sample code from Fig.1. Beside the first three statements, each statement has an execution instance for each iteration. The last node stands for the whole library function called by statement 14. Within the library function the values stored by statement 9 are read and thus the function node has an edge from each instance of node 9.

Definition 2. Given $dDDG(N, E)$, a Dynamic Backward Slice (DBS) of a statement execution $n_i \in N$ denoted by $DBS(n_i)$ is the subgraph of $dDDG(N, E)$ which includes n_i as well as all other nodes and edges from which n_i is reachable, i.e.

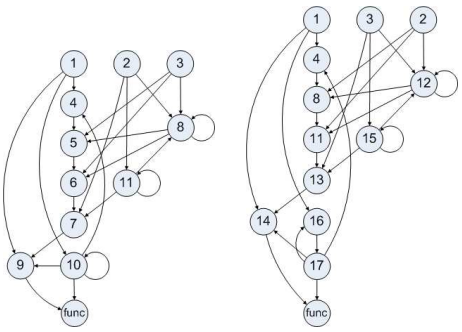
$$DBS(n_i) = (n_i, e | e = m_j \rightarrow n_i \in E) \cup \bigcup_{\forall m_j \rightarrow n_i} DBS(m_j)$$

For any virus, no matter what it aims to do, it finally calls some library function to realize its goal. Most of the statements in the virus are dedicated to preparing data for the library function call. In other words, if a statement's result does not reach any library function call, then the statement is likely to be superfluous code that is intentionally introduced to thwart the detection of the virus by a static signature based antivirus software. Because a virus computes values used in library function calls, it is only appropriate that our dynamic signature based technique look for the presence of dynamic virus signatures within DBSs of library function calls.

An advantage of using DBSs is that the superfluous code introduced to obfuscate the virus will most likely be eliminated during the construction of the dynamic backward slice. The nodes in bold cycles shown in Fig.3 comprise the DBS of the library function called by the last statement in the sample code. Although statement 12 controls the loop, it is not in the DBS because there is no statement having a data dependence on it. Therefore, DBS is immune to control flow mutation since it does not include control dependences.

Our dynamic signature of a virus is a compact representation of DBS of a library function call, which is unique to the virus and present in its every run.

Definition 3. Given $DBS(n_i)$ which is unique to the virus and present in its every run, a dynamic signature $DS(N, E, f)$ is its compact representation where: each node $n \in N$ represents statement n in $DBS(n_i)$ instead of its execution instance, each edge $m \rightarrow n \in E$ means that there is at least one execution instance of statement n in $DBS(n_i)$ which depends on statement m's results, and f represents the library function call statement, based on which $DBS(n_i)$ is created.



(a) (b)

Figure 4. Dynamic signatures of (a) the sample code and (b) the mutated code.

In other words, the original DBS is an execution instance-based data dependence graph while our dynamic signature is a statement-based graph. The conversion process is very simple and fast: we create a node for each statement in DBS and connect node m to n if there exists at least one edge from $\{m_1, m_2, \ldots\}$ to $\{n_1, n_2, \ldots\}$ in DBS. We use a statement-based graph as our virus signature because the original DBS is very large and repetitive, which may cause the matching process to become extremely slow. Fig.4 show the dynamic signatures constructed based on the sample code and the mutated code. They are clearly much smaller than the corresponding DBSs. We also observe that the two dynamic signatures are very similar, which means our dynamic signature is more robust than static signatures in presence of obfuscation techniques.

2.3. Signature Matching

The goal of the signature matching is to automatically establish a correspondence between instructions from our virus signature and DBSs of a program run. Given a dynamic signature $DS(N, E, f)$ and a program run, the matching process contains three phrases:

1. Decomposing the program run. Before matching, we first generate the DBSs for library calls from the dDDG of the program run. All the DBSs are in the compact form discussed earlier.

2. Comparing library function call. We compare the function call in the dynamic signature with that in each DBS. If they are different, the signature and the DBS are considered to perform different tasks. In order to overcome the API substitution technique, we treat two library function calls as compatible if they perform the same task. We discard all the DBSs where the function call does not match the call in the dynamic signature.

3. Matching the signature with each DBS. Finally, we match the dynamic signature with each remaining DBS and calculate their similarity. If there exists at least one DBS whose similarity to the dynamic virus signature exceeds a preset threshold, then the corresponding program is treated as an infection.

We had developed a dynamic matching algorithm in our prior work [26]. There are two main aspects to this matching algorithm: instruction matching uses local value histories of instructions to overestimates of sets of instruction that potentially match each other; and structure matching that refines these overestimates iteratively to produce an accurate match. A high degree of match between the two dDDGs is treated as a true match. However, this algorithm is ineffective in virus detection according to our experiments. Both instruction matching and structure matching had weaknesses which led to matches between virus and its variants being missed. Next we identify these weaknesses and describe the ideas used to deal with them.

Our prior algorithm used the local histories of two instructions to match them to each other. However, in the case of virus detection, matching local histories does not work well. The reason is that the machine on which the dynamic virus signature is created may differ in its environment from the machine on which the end user is using the antivirus software. Such environment differences can lead

Match(DS, DBS)
input:
Dynamic signature $DS = G(N_a, E_a)$,
and dynamic backward slice $DBS = G(N_b, E_b)$.
output:
Similarity between DS and DBS.
begin
Step 1: Initialization of match sets.
$N'_a \leftarrow N_a$
for each node $n \in N_a$ **do**
$Match(n) \leftarrow \{m | m \in N_b \& op(n) = op(m)\}$
if $Match(n) = \emptyset$
Remove n from N'_a and connect $iAnc(n)$ to $iDes(n)$
end for
Step 2: Iterative refinement of Match sets.
while not stable **do**
- Forward pass:
for each node $n \in N'_a$ **do**
for each node $\bar{n} \in Match(n)$ **do**
if $\exists a \in iAnc(n), Match(a) \cap Anc(\bar{n}) = \emptyset$
$Match(n) \leftarrow Match(n) - \bar{n}$
end for
if $Match(n) = \emptyset$
Remove n from N'_a and connect $iAnc(n)$ to $iDes(n)$
end for
- Backward pass:
for each node $n \in N'_a$ **do**
for each node $\bar{n} \in Match(n)$ **do**
if $\exists a \in iDes(n), Match(a) \cap Dec(\bar{n}) = \emptyset$
$Match(n) \leftarrow Match(n) - \bar{n}$
end for
if $Match(n) = \emptyset$
Remove n from N'_a and connect $iAnc(n)$ to $iDes(n)$
end for
end while
Step 3: Calculation of similarity.
$$sim(DS, DBS) \leftarrow \frac{|N'_a| + |\bigcup_{n \in N'_a} Match(n)|}{|N_a| + |N_b|}$$
end

Figure 5. Algorithm for matching a signature with a compact DBS.

to differences in the local histories produced on these machines. For example, consider the *email blaster worm*. In this virus the local histories of some instructions differ for different machines because these histories depend upon the email address lists stored on the machines and these email lists may differ from one another.

To handle the above problem, in our new matching algorithm, we match the two instructions by comparing their opcodes. If they have the same opcode, we consider them to be matched; otherwise, they cannot be matched. The shortcoming of matching instructions by opcode is that this approach cannot deal with the instruction substitution techniques. For example, if the instruction "inc eax" is modified as "add eax, 1", they cannot be matched. Therefore, in order to handle the instruction substitution problem, our signature matching algorithm looks for partial matches of a signature to a DBS when the whole signature cannot be matched. We also believe this problem can be handled by setting up rules describing which kinds of opcodes can be matched with each other.

The structure matching proposed in our new algorithm is also superior. We found that our prior algorithm failed when the two dDDGs being matched contain nodes which had no corresponding matches. In our new algorithm we use the following approach to overcome this problem. When we find nodes in the two graphs that have no potential corresponding matches, we transform the graphs by eliminating these nodes. The elimination is carried out by connecting the immediate ancestors of an eliminated node directly to the immediate descendants of the node. We found that these transformations enabled us to effectively carry out the matching of the remaining nodes.

Next we present our dynamic matching algorithm that uses the above form of instruction and structure matching in detail. We also illustrate the algorithm using an example.

Fig.5 shows our algorithm for matching a signature with a compact DBS, where $iAnc(n)$ ($iDes(n)$) denotes the immediate ancestor (descendant) set of node n, and $Anc(n)$ ($Des(n)$) stands for the ancestor (descendant) set of node n (including n). The algorithm computes the $Match$ set for each node n in N_a to be the subset of nodes in N_b that match n. After initialization, the algorithm iteratively calculates the forward and backward passes. The forward pass ensures that the two matched instructions use the same operands while the backward pass ensures they are used by the same set of instructions. If the $Match$ set of node $n \in N_a$ becomes empty, we remove it from the N_a and connect n's parents directly to its children. This cuts off the chain effect of $Match(n)$ begin empty and keeps the matching process going. The algorithm terminates when all $Match$ sets stabilize. The final similarity is equal to the number of matched nodes divided by the total number of nodes.

Initial matches	Final matches
Match(1)={$\bar{1}$}	Match(1) = {$\bar{1}$}
Match(2)={$\bar{2}$,$\bar{3}$,$\bar{4}$,$\bar{14}$}	Match(2)={$\bar{2}$}
Match(3)={$\bar{2}$,$\bar{3}$,$\bar{4}$,$\bar{14}$}	Match(3)={$\bar{3}$}
Match(4)={$\bar{2}$,$\bar{3}$,$\bar{4}$,$\bar{14}$}	Match(4)={$\bar{4}$}
Match(5)={$\bar{8}$, $\bar{11}$, $\bar{17}$}	Match(5)={$\bar{8}$, $\bar{11}$}
Match(6)={$\bar{8}$, $\bar{11}$, $\bar{17}$}	Match(6)={$\bar{8}$ $\bar{11}$}
Match(7)={$\bar{13}$}	Match(7)={$\bar{13}$}
Match(8)={$\bar{12}$}	Match(8)={$\bar{12}$}
Match(9)={$\bar{2}$,$\bar{3}$,$\bar{4}$,$\bar{14}$}	Match(9)={$\bar{14}$}
Match(10)={$\bar{6}$}	Match(10)={$\bar{16}$}
Match(11)={}	Match(11)={}

Table 1. Matches between the dynamic signatures shown in Fig.4.

We use the examples shown in Fig.4 to illustrate our matching process. The left column in Table.1 shows the initial matching results after the first step in our algorithm, where i denotes node i in Fig.4(a) and $\bar{j}$ stands for node j in Fig.4(b). Since the match set of node 11 is empty, we remove it from the N'_a. In the initial matches, each node is

matched to those nodes which have the same opcode. Then the forward pass refines the matches according to the dependence relationship. It first passes node 1, 2 & 3 since they do not have any parents. Then it examines node 4 and removes $\{\bar{2}, \bar{3}\}$ from the its match set since node $\bar{2}$ & $\bar{3}$ do not depend on node $\bar{1}$. Similarly the forward pass removes $\{\bar{1}\bar{7}\}$ from Match(5) and Match (6) and $\{\bar{2}, \bar{3}, \bar{4}\}$ from Match(9). Because node 11 is removed and not the immediate ancestor of node 7 any more, the match set of node 7 is not emptied due to the empty match set of node 11. Now we perform the backward pass, where we examine the nodes in the reverse order. The match sets of node 5 - 10 do not change in this pass. Let consider node 4. The result of node 4 is used by node 5 but node $\bar{8}$ & $\bar{1}\bar{1}$, which are in the match set of node 5, do not depend on node $\bar{1}\bar{4}$ at all. Therefore, node $\bar{1}\bar{4}$ does not match node 4 and is removed from the match set of node 4. Similarly we remove $\{\bar{3}, \bar{4},\bar{1}\bar{4}\}$ from Match(2) and $\{\bar{2}, \bar{4},\bar{1}\bar{4}\}$ from Match(3). Note that the match sets at this point represent the final results (see Table.1). Node 5 and 6 have two matched nodes $\bar{8}$ and $\bar{1}\bar{1}$ since they have the same opcode and the dependences. If we could build a set of match rules, like "dec *" = "sub *, 1", then node $\bar{1}\bar{5}$ would be matched to node 11. While in this example we performed the forward pass once and backward pass once, in general we may have to apply them for several times. Using the final matches, we can get the similarity between those two signatures as $sim = (10+10)/(11+12) = 87\%$. If we can set up match rules and recognize the garbage instructions like "sub *, 0", we can further improve the result.

2.4. Automated Signature Extraction

Once a virus has been encountered and a few variants of the virus have already been found, we know that we are dealing with a mutating virus and hence we construct a dynamic signature for the virus. This signature is constructed from the already available set of program runs for the same virus family (i.e., virus and its mutants). A good signature is one that can be found in every available program run involving the virus, but at the same time it is unlikely to be found if the virus is not present. In other words, we want to minimize the likelihood of both false negatives and false positives. Usually, a signature for a new virus is chosen by an antivirus expert. However, the procedure of choosing a virus signature is very time-consuming while the number of new viruses being produced continues to increase. Therefore, we developed a method for automatically extracting good dynamic signature from the runtime traces of a virus. The signature extraction procedure consists of three steps:

1. Generating candidate signatures. In this step, given a set of runtime traces of the same virus family, we create a dDDG for each trace. In each dDDG, we search for all library function calls and generate a candidate signature from each library function call's DBS. The subsequent steps select the best signature from the candidates.

2. Filtering out the signatures which could cause false positives. Intuitively, the shorter the signature, the more likely its is that it will result in false positives. In this step, we remove the candidate signatures the sizes of which are smaller than some established threshold.

3. Choosing a signature which minimizes the false negatives. Suppose we have n runtime traces. We use $U_i = \{DS_1^i, DS_2^i, \ldots\}$ to denote the set of candidate signatures generated from the i-th trace. Each candidate signature is assigned a score, which can be calculated as follows:

$$score(DS) = \sum_{i=1}^{n} \max_{DS_j^i \in U_i} \{sim(DS, DS_j^i)\}$$

The score of DS is equal to the summation of the similarities between DS and its corresponding slice in each trace. It can be seen as the possibility of DS being present in a runtime trace of the virus. We select the candidate signature with highest score as the best signature.

Now we have completed our description of how a dynamic signature is obtained and how this can be matched with dynamic backward slices to detect viruses. In the next section we discuss issues that must be addressed in the deployment of a dynamic virus detector.

2.5. Deploying Dynamic Virus Detection

There are two main challenges in deploying dynamic virus detection. First, dynamic detection requires an infected program to be executed. However, such execution will enable the virus to cause the damage that it is intended to cause. We must solve this problem to make dynamic detection practical. Second, it is not desirable to run a program using an emulator forever. However, to stop using the emulator we must develop a mechanism that decides that the virus scanner has sufficiently checked the program. Next we discuss our solutions for the above two problems.

To address the first problem, we exploit *sandboxing* [14]. A sandbox is a CPU emulator where a file can be executed as if it were running on a real computer. When a virus scan is requested for an executable file, our virus detector loads the file into the sandbox and then executes it in this completely isolated environment. Thus, during the execution, even though the suspect file is indeed infected by a virus, it cannot cause any damage to the actual host computer.

To overcome the second problem, we need a mechanism that determines that the program can be cleared as being sufficiently scanned. Since each library call represents a potential point through which a virus can attack, we track the library calls that have been encountered by the emulator and cleared via dynamic virus signature matching. In fact, once a library call has been cleared, we can turn off dynamic signature matching for that library call. However, when the sandbox encounters a library function call that has not been covered before, we match its backward slice with the dynamic signatures and mark it as encountered and cleared.

Once all library function call sites have been cleared by the emulator, we can declare the program as being free of viruses and stop running it using the sandbox.

It is possible that some library function calls in the target program are not encountered for a very long time. This will cause the program to continue being run using the emulator. To tackle this problem, our sandbox terminates the emulation process if no new library function calls have been encountered for a certain period of time. However, we still need to deal with the situation where one of the uncleared library calls is encountered as the program may be infected by a virus at such a call. We instrument the non-cleared library function calls such that when they are encountered in the actual computer, the execution of the program will be suspended and the program is put in the sandbox for checking the newly encountered library function call. If the program is indeed an infection, we terminate that suspended process and isolate the executable file. If our detector cannot find any virus signature at that function call, we can remove the corresponding instrumentation and continue that suspended process. In this way, we can make sure our detector does not miss any virus hidden in the target program.

Finally, our virus detector is comprised of three components: *sandbox*, *trace collector*, and *signature scanner*. At the start of the simulation, the sandbox begins executing the target program and informs the trace collector of the process to be traced. The trace collector then keeps collecting the trace and when a backward slice of some library function call has been generated, it calls upon the signature scanner to search though the slice for known dynamic virus signatures. Our dynamic detector does not have to be used all the time. When a new virus family emerges with no static signature being able to detect all its variants, the antivirus companies can first create a dynamic signature for it and thus the users can turn on our dynamic virus detector to protect their computers from the new virus family in time. After the antivirus companies create the static signatures for all the variants of the new virus family, our dynamic detector can be deactivated since the static detector is more efficient.

3. Experimental Evaluation

The goal of our experiment was to measure the effectiveness of our dynamic signature based virus detector, which can detect different variants in a virus family by using a single dynamic signature. To do that, we applied our detector to a set of real-world viruses. For each virus family, we extract the dynamic signature from three randomly selected variants, and then use it to identify other variants in the same family. We only selected those signatures containing at least 30 nodes – we did not use short signatures as they are likely to result in false positives. Another goal of these experiments is to measure the runtime overhead of dynamic virus detection.

3.1. Implementation

In our implementation, we used QEMU [4], an open source process emulator, as the *sandbox*. The *trace collector* was realized by modifying the source code of QEMU. We use the paging register to identify the process to be traced. To make our technique efficient, we developed a few methods to reduce the overhead of tracing and matching.

3.1.1 Reducing Tracing Overhead

Trace collection is a time and space-consuming procedure. It accounts for a major part of overhead introduced by our detector. The time taken by the trace collecting procedure is usually several times larger than the normal running time. The trace data of an application may take several gigabytes of space. Therefore, to make our virus detector efficient, we need to reduce the cost of tracing in both time and space.

Time overhead. Due to presence of loops, there is a lot of repetitive contents in a runtime trace. Our approach to reducing time overhead is based upon avoiding the recording of repetitive traces. To determine if a program run is infected, our detector decomposes the whole trace into multiple DBSs and scan those DBSs one by one. If there are two same DBSs in a trace, the extra DBS is useless since it does not give extra information to the detector. Therefore, if we could avoid collecting the part of trace corresponding to the extra DBS, we could reduce the time overhead of trace collecting without degrading our detector's performance.

Definition 4. Given a trace, a *User Trace Block* (UTB) is a segment of trace produced from user code and thus, the trace right before or after a UTB is from library code.

Definition 5. Given a trace, a *Library Trace Block* (LTB) is a segment of trace produced from library code and thus, the trace right before or after the LTB is from user code.

Definition 6. Given a trace, a *Paired Trace Block* (PTB) is a pair of UTB and LTB where the UTB is immediately followed by the LTB.

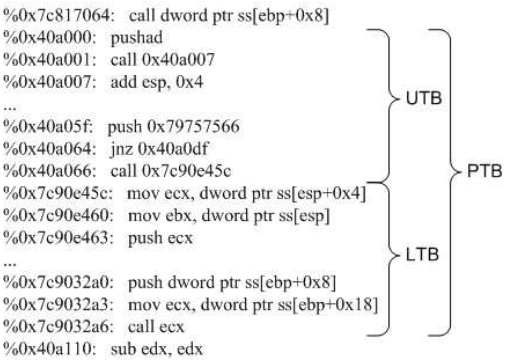


Figure 6. A PTB Example.

Fig.6 shows an example of PTB. Each DBS of a library function call usually corresponds to a PTB. To avoid repetitive PTB, when the collector reaches a new PTB, it checks if the addresses of the first n instructions appear more than m times, where n and m are established parameters. If so,

the collector stalls dumping the trace until the new PTB. This can significantly reduce the tracing overhead especially when the execution time of the process is very long.

Space overhead. The trace of a program usually takes several gigabytes of space. If we save the trace into a file, the disk operation will greatly increase the tracing overhead. In practice, we find that in order to create a DBS for the last library function call during the runtime, we only need to keep the latest 40 megabytes trace, which can be easily kept in the memory.

3.1.2 Reducing Matching Overhead

Using separate core to do matching. As the multicore computers have become available, it is feasible to use a separate core to conduct the matching procedure. While the matching task is performed on one core, the scanned process can still be executed on the other core. Thus, the matching task will not slow down the scanned process.

Early termination of the matching procedure. In our matching algorithm, we keep removing the unmatched nodes away from the node sets until convergence. As we remove more and more unmatched nodes, the similarity decreases correspondingly. To reduce the matching overhead, we could calculate the similarity after each iteration. Once the similarity is below some threshold, we can terminate the matching procedure even though the termination condition is not satisfied, because the intermediate result has shown the signature and DBS cannot be matched.

3.2. Examined Viruses

In our experiments, we apply our dynamic signature based method to seven virus families which in all contain 120 virus variants. These viruses are downloaded from *Offensive Computing* [1] and *VX Heavens* [2] websites.

Table.2 lists all the viruses used in our experiment. All of them are present in infected host files. They cover four different categories of viruses. Except for *Dislex* and *Etap*, all viruses are widespread or were once widespread. Each virus family has many variants – ranging from 11 to 24. To detect these variants, existing commercial antivirus software need at lease one signature for each variant. However, from the view of execution traces, these variants have similar behaviors. Therefore, ideally, we expect our trace matching algorithm to detect all the variants by using only one dynamic signature. *Dislex* and *Etap* are different because they mutate their own codes according to a set of predefined rules when they infect other files. This makes it difficult for the static signature based techniques to detect them.

3.3. Detection Rate

When checking a virus V with a signature DS, we treat V as detected when examining a backward dynamic slice DBS if the similarity of DBS and DS computed by our matching algorithm exceeds 0.5. We observed that the similarity between two different slices is around $10\% - 20\%$ while the similarity between two matchable slices is about $80\% - 90\%$. Therefore, we set the threshold as 0.5 to discriminate the matched slices from the unmatched ones.

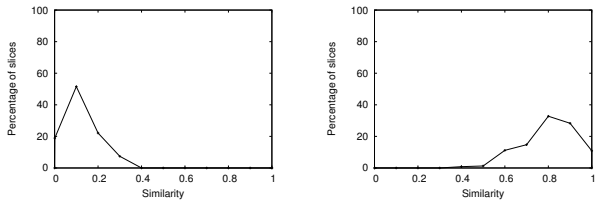
Name	Detection Rate	Avg Sim.	Sig. Size
Agobot	100%	0.89	62
Bagle	100%	0.95	65
Dislex	100%	0.84	48
Etap	93.3%	0.72	44
Kriz	100%	0.82	39
Mytob	100%	0.88	57
Netsky	100%	0.94	63

Table 3. Detection rate. From left to right: virus family name, detection rate, average similarity, and size of the signature.

Table.3 shows the experimental results for all seven virus families. Our dynamic signature based detector identifies all the viruses except one variant in the *Etap* family. Correspondingly, the average similarity between *Etap* and its signature is lowest. The reason is that *Etap* applies instruction substitution technique to mutate its code, which our method cannot handle effectively. For all other viruses, our method works well since the similarities between the viruses and their signatures is very high.

3.4. False Positives

From Table.3, we can see that the size of the signatures we extracted ranges from 39 to 65 nodes, which is large enough to avoid false positive. To verify this, we measured the false positive rates in an experiment. In the trace of a virus, there are a number of different backward slices. Some slices belong to the virus and should be detected by using the corresponding signature while the others should not be detected. We match each slice in the virus traces with the dynamic signatures. The result shows that the false positive rate is 0% for all the slices in the virus traces, that is, there is no non-virus slice which is identified as a virus slice.



non-virus slices virus slice

Figure 7. Similarity distribution.

Fig.7 shows the similarity distribution for both non-virus slices and virus slices. We can see that the similarities of the non-virus slices are all below 40% while most of the virus slices' similarities are around 80%. Since we set the threshold as 0.5, we did not get any false positives even though very few virus slices are identified as non-virus.

3.5. Runtime Overhead

Table.4 shows the overhead of our method on several real virus examples. The data were collected on a Dell Precision T3400 with one Intel quad-core processors at 2.40GHz,

Name	# of Variants	Year	Self-mut?	Category	Description
Agobot	24	2004	N	worm	giving its creator complete access to the infected computer
Bagle	17	2004	N	worm	sending email to addresses it gathers
Dislex	21	2002	Y	virus	–
Etap	15	2003	Y	virus	popping up junk message box
Kriz	11	1998	N	virus	damaging the motherboard and hard disk
Mytob	12	2005	N	worm	gathering personal and financial information
Netsky	20	2004	N	worm	sending email to addresses it gathers

Table 2. Viruses used in the experiment. From left to right: virus family name, number of variants we collected, year virus first appeared, self-mutating or not, category, and description.

4GB of RAM. All the timings in the table are in seconds. From the table, we can see that the tracing overhead of our method is from 2x to 10x while the matching overhead is relatively small, from 1x to 3x. Since the matching procedure can be executed on a separate core in practice, it will not affect the overall performance of our dynamic detector.

Name	# of slices	Exec.	Tracing	Matching
Bagle.at	292	5.13	50.89	13.57
Bagle.au	289	5.24	36.91	10.25
Netsky.t	613	8.54	64.06	11.73
Kriz.3863	378	19.82	43.68	7.23
Dislex	409	6.77	30.02	6.80

Table 4. Overhead. From left to right: Virus name, number of backward slices, execution time, tracing overhead, matching overhead.

The tracing overhead of *Bagle.at* is almost 10x which is the highest. This is because its host program does many different checks on the operating system which produces a large piece of non-repetitive trace. *Kriz.3863* has the lowest tracing overhead (slightly over 2x) because its host program is a notepad. Each time the notepad receives an input, it calls the same library function to show the character on the screen. Therefore, most backward slices in its trace are repetitive, and thus can be neglected. In our experiment, we did not use any trace compression technique [27] in the tracing procedure. Also we did not use any antivirus technique [5] to detect the possible entry-points of virus code in a host program, which can limit the range of code to be traced. We believe the tracing overhead can be further reduced if the aforementioned techniques are employed.

4. Related Work

4.1. Malware Detection

Malware detection techniques can be generally divided into two categories: static analysis and dynamic monitoring.

Static analysis. There are several malware detection work [6, 7, 12] based on matching static semantic patterns. SAFE [6] is a malware detector which uses a malicious code automaton as the semantic signature to match the control flow graph of a program. If a match is found, the program is identified as a malware. SAFE can only handle very simple obfuscations such as garbage insertions. The static analysis technique proposed in [12] employs the symbolic execution to match semantic patterns and is designed to detect kernel-level intrusions (known as rootkits). The malware detector presented in [7] leverages not only symbolic execution but other decision procedures to identify general malicious behaviors. All of these techniques cannot detect the malicious behaviors involving control flow alteration and instruction substitution (Fig.1 and Fig.2). On the contrary, our technique can effectively detect them by eliminating incompatible nodes in the signature matching.

Several other malware detection methods were based on code normalization, which can handle control flow alteration [5] and instruction substitution [23]. However, they both rely on the obfuscation rules and hence can only handle the specific obfuscations that their rules target.

Dynamic monitoring. Generic decryption (GD) [16] is a technique used by several antivirus companies to detect encrypted viruses. It executes the suspect program in a virtual machine and search for virus signatures in the potential virus code region of virtual memory. Petroni and Hicks [20] proposed a method to detect the control-flow change during runtime. Erlingsson and Schneider [9] proposed a binary rewriting method to insert security-purpose inline checks into Java program. Copilot [19] is a kernel integrity monitor designed to detect malicious modifications to an operating system by aggressively examining the host RAM. Laika [8] is a system which identifies the data structures used by a program in its memory image by using Bayesian learning. It can be used to detect polymorphic viruses by comparing their data structures. Yin et al. designed a system, Panorama [25], which detects malwares by capturing certain malicious information access and processing behavior, e.g., password thieves and network sniffers. Rieck et al. [22] proposed a machine learning method which extracts the behavior patterns (e.g. opening a file and locking a mutex) shared in a malware family to detect new instances of the same family.

4.2. Program Differencing

Static differencing. Statically comparing two program versions have been extensively explored. The early work [15] checks the differences in the code using line by line comparison. Several different methods [3, 10, 13] are proposed to assess program equivalence by comparing control flow or call graphs. Jackson and Ladd [11] proposed

a method which measure differences by comparing input/output dependences of procedures. All these methods except [10] work on source or intermediate code representations of the program versions. These methods cannot handle the program versions which statically appear to be different but dynamically behave the same, in which case our method can do well. Recently, Person et al. [18] proposed a differencing algorithm by using symbolic execution, called Differential Symbolic Execution (DSE), which can deal with certain kinds of obfuscations.

Dynamic differencing. Compared to the research work in static differencing, relatively fewer works have been done in dynamic differencing. Reps et al. [21] proposed a method to detect Y2K bugs by using path profiling and Wilde [24] developed a system which visualize the changes in the dynamic behavior of a program. Our prior works [17, 26] presented techniques to match runtime traces of two program versions. Compared to them , the matching algorithm proposed in this paper is more powerful and more suitable for virus detection. First, the prior work compares nodes in dDDGs by comparing their produced values, which does not work in virus detection since the statements of a virus may dynamically produce different values on different machines. Second, in prior work, if the match set of a node is empty (i.e. it cannot match any node in the other program), it may propagate to other nodes so that the match sets of its descendants may all become empty. In this paper, we solved the above two problems. Moreover, both of our prior papers focused on comparing two entire programs while this paper is mainly about finding a small piece of signature in a large program.

5. Conclusions

We gave a new type of virus signature, called *dynamic signature*, which is designed for detecting mutating viruses. Correspondingly, we developed a system of approaches for how to extract and match the dynamic signatures and how to deploy our dynamic signature based virus detector. Experiments demonstrated that our matching algorithm is very effective in identifying different variants in a virus family with a single dynamic signature while producing no false positive. In the future, we could extend our work to detecting other types of malwares, e.g., rootkits. We also plan to reduce the overhead of our detection method using trace compression/optimization techniques in our prior work.

Acknowledgements. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470 to UC Riverside.

References

[1] Offensive Computing. http://www.offensivecomputing.net/.
[2] VX Heavens. http://vx.netlux.org/.
[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. *ASE'04*.
[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC'05*.
[5] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 2007.
[6] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security Symposium 2003*, 2003.
[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *SP'05*, 2005.
[8] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *OSDI 2008*.
[9] U. Erlingsson and F. B. Schneider. IRM enforcement of java stack inspection. 2000.
[10] H. Flake. Structural comparison of executable objects. In *DIMVA'04*.
[11] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM'94*.
[12] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. *ACSAC'04*.
[13] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. *CSM'92*.
[14] S. Miwa, T. Miyachi, M. Eto, M. Yoshizumi, and Y. Shinoda. Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares. In *DETER 2007*.
[15] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1986.
[16] C. Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 1997.
[17] V. Nagarajan, R. Gupta, X. Zhang, M. Madou, and B. D. Sutter. Matching control flow of program versions. *ICSM'07*.
[18] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *SIGSOFT '08/FSE-16*.
[19] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium 2003*.
[20] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS'07*.
[21] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 1997.
[22] K. Rieck, T. Holz, C. Willems, P. Dussel, and P. Laskov. Learning and classification of malware behavior. In *DIMVA'08*.
[23] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. *SCAM'06*.
[24] N. Wilde. Faster reuse and maintenance using software reconnaissance. *Technical Report SERC-TR-75F, SERC, Univ. of Florida, CIS Department*, 1994.
[25] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS'07*.
[26] X. Zhang and R. Gupta. Matching execution histories of program versions. *SIGSOFT Softw. Eng. Notes*, 2005.
[27] X. Zhang and R. Gupta. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.*, 2005.
[28] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE'03*.