A Summary About My Work On CFG Matching

Chen Chen

September 6, 2013

* This is not a paper

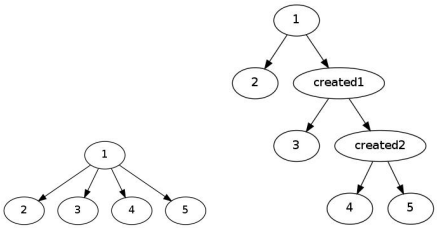
Abstract

We noticed some good properties in CFG. For example, any CFG is or can be converted into a linear structure, besides, any node in CFG can be regarded as no more than two children(in fact, the case where a node has more than 2 children can be converted to several nodes with no more than 2 children). Therefore, we can abstract the CFG into some simplified graph by different levels, and comparing different CFGs in different abstraction levels is a very effective method to compare the similarity between them. We develop a algorithm for that and then prove its ability to be applied into any cases. We obey the rules not destroying the actual control flow throughout the whole algorithm, and it turns out working perfectly.

1 Introduction

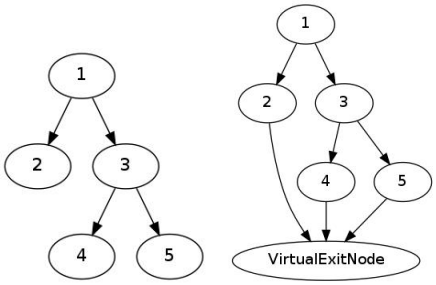
CFG matching is of great use in malware detecting nowadays. And traditional mathods are mainly about subgraphs. They are easy to understand and can work well under some circumstances. However, it would be hard for them when it comes to big graphs and they may work very slowly.

In this paper, we proposed a novel method using abstraction conception. We first abstract the graph into a single node with a tree in it. And then we compare different CFGs by comparing the trees from the root level by level.



(a) Before operation (b) After operation

Figure 1: Pre-operation method 1



(a) Before operation (b) After operation

Figure 2: Pre-operation method 2

2 Algorithm

A. PRE-OPERATION:

As mentioned in ABSTRACT, there are two good and useful properties in CFG:

i) Most nodes in CFG have no more than two children;
ii) Most CFGs have only one enter-point and one exit-node

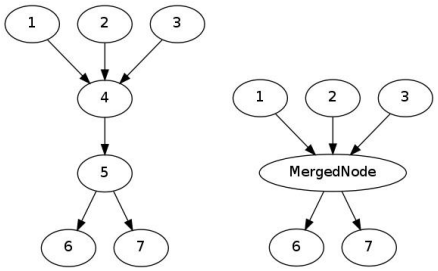
However, we have to admit that there exist some cases where the two properties are not satisfied. Thus, to ensure that following algorithm can be applied to any case, we have to do some pre-operations:

pre-operation 1 To those nodes who have more than 2 children, we will convert them into several nodes, all of which having two children.

pre-operation 2 It's true that each function has only one enter-node, but may not only one exit-node. However, since any function, no matter how many exit-nodes it has, will return to the very function who has called it, we can set a virtual exit-node for each function. In this way, we can deal with all the functions as if they have only one exit-point.

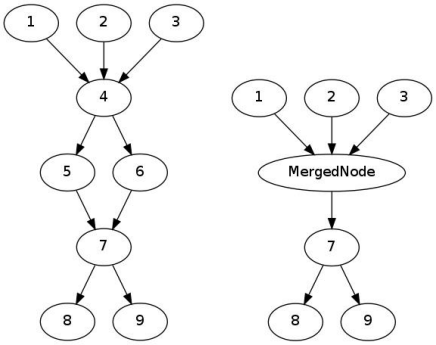
B. ABSTRACTION METHOD:

There are three kinds of operation methods as follow:



(a) Before abstraction (b) After abstraction

Figure 3: Abstraction method 1(1)



(a) Before abstraction (b) After abstraction

Figure 4: Abstraction method 1(2)

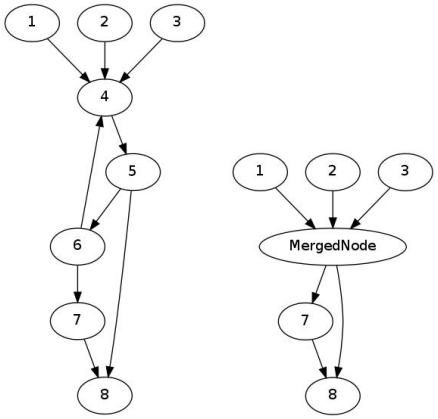
abstraction method 1 (1)Two subsequent nodes A(father) and B(child) can be abstracted into a single node if the the father node A has only one child(B) and the child B has only one father(A); (2) and four nodes which have made up a "diamond" can be simplified into two node, as the picture has showed below:

abstraction method 2 All the nodes inside a loop can be simplified to a single node, and all the fathers and children are inherited by the created node. Since loops make the nodes inside them share the same control flow, this kind of abstraction does not conflict with the principle to reserve actual control flow.(About how to detect loops: we use a modified DFS to detect the loops in directed graph and it turns out working very well. However, since it's not the most crucial part of this algorithm, we will not illustrate it here.)

abstraction method 3 A node that has n other nodes pointing to can be replicated into n nodes which share the same offspring while have a single father. It's obvious that this way will not change the control flow.

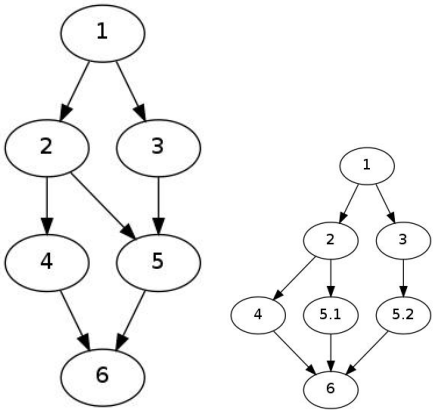
Here is the algorithm about the order of pre-operations and abstraction methods:

begin
using pre-operation method 1 to modify CFG
using pre-operation method 2 to modify CFG



(a) Before abstraction (b) After abstraction

Figure 5: Abstraction method 2



(a) Before abstraction (b) After abstraction

Figure 6: Abstraction method 3

Do While CFG has been changed
  Do While CFG has been changed
    using abstraction method 1 to abstract CFG;
  EndWhile
  using abstraction method 2 to abstract CFG;
  using pre-operation method 1 to modify CFG;
EndWhile
Do While CFG has been changed
  Do While CFG has been changed
    using abstraction method 1 to abstract CFG;
  EndWhile
  using abstraction method 3 to abstract CFG;
EndWhile
End

C. COMPARING METHOD:

To record the process of abstraction, we keep a tree in each node of the graph. Let's call the nodes in the tree as *tnodes* in order to distinguish them with the nodes in the graph. There are also several properties in such tree and its tnodes:

Property 1. Each tnode keeps two variable: NUM(int) and IFLOOP(bool). NUM records the number of leaves in this tnode's offspring, and IFLOOP shows whether there are loops in its offspring. Those two variables are calculated when the tnode is created.

Property 2. If two nodes A and B are abstracted into one node C (there is already a tree in A with a root *rootA*, and also a tree in B with *rootB*), then as the first step to create a tree, we create a tnode(*rootC*) in C as C's root, which has two children *rootA* and *rootB*. It should be noted that *rootA* and *rootB* should be kept in order as children(i.e. the bigger NUM is, the former the position in the ChildList), which is to avoid the obfuscation about the order in future comparison.And all the nodes as the offspring of *rootA* and *rootB* are copied into the new tree in C. NUM in *rootC* is calculated by adding the NUM in *rootA* with that in *rootB*. And if either of the IFLOOPs in *rootA* or *rootB* is 1, the new IFLOOP in *rootC* is 1. Similar actions happen when three nodes are abstracted(namely, a diamond is being abstracted).
textitProperty 3.

Property 3. If more than three nodes are merged into a single node(namely, a loop is being abstracted), we do similar thing with *Property 2* except that the variable IFLOOP in the root of created node is set to 1.

After abstraction following the steps above, we can get a single most-abstracted node with a tree inside. Here comes to how we shall compare two different CFGs with such two nodes.

* This has not been decided clearly yet, I think some parameter should be decided only after doing machine learning in order to ensure the best accuracy.

* Of course, concerning the robustness, the comparing algorithm can be modified.

begin
If the difference between the two NUMs in roots of the two CFGs is bigger than a set value **then**
record that the two CFGs are not similar;
exit;
EndIf
level := 1;
While level less than a set value **Do**
If the number of the tnodes at current level is not the same **then**

record that the two CFGs are not similar;
exit;
Else
If IFLOOP in the tnodes of current level is not same **then**
record that the two CFGs are not similar;
exit;
Else
If the cosine similarity of the two vectors composed by the NUMs in the tnodes of current level is less than a set value **then**
record that the two CFGs are not similar;
exit;
EndIf
EndIf
EndIf
level := level + 1
EndWhile
record that they are similar
End

3 DEMONSTRATION:

* I have not listed the detailed demonstration, here is only a brief description about the reasonable line of thinking.

step 1 We needn't to consider the existence of loops since we merge all the nodes in a loop as a single node. It's sure that every loop can be dealt with in that way. We only need to prove that our algorithm can be applied to CFGs with no loops inside.

step 2 First we demonstrate that a graph in which all the nodes (except the enter-point and the virtual exit-point) has only one father and no more than 2 children can be simplified.

step 3 Then we demonstrate that all the graphs without loop can be converted to the graph described in 2.

4 Conclusions

Personally I think our final work should be composed of three parts: *Algorithm*, *Demonstration* and *Testing*. And more work is needed after I go back to test this method.