Matching Control Flow of Program Versions

Nagarajan, V., Gupta, R., Zhang, X., Madou, M., & De Sutter, B. (2007). Matching Control Flow of Program Versions. In Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. (pp. 84-93). IEEE COMPUTER SOC. 10.1109/ICSM.2007.4362621

Matching Control Flow of Program Versions

Vijay Nagarajan, Rajiv Gupta
University of California, Riverside

Xiangyu Zhang
Purdue University

Matias Madou, Bjorn De Sutter
Koen De Bosschere
Ghent University

Abstract

In many application areas, including piracy detection, software debugging and maintenance, situations arise in which there is a need for comparing two versions of a program that dynamically behave the same even though they statically appear to be different. Recently dynamic matching [18] was proposed by us which uses execution histories to automatically produce mappings between instructions in the two program versions. The mappings then can be used to understand the correspondence between the two versions by a user involved in software piracy detection or a comparison checker involved in debugging of optimized code. However, if a program's control flow is substantially altered, which usually occurs in obfuscation or even manual transformations, mappings at instruction level are not sufficient to enable a good understanding of the correspondence. In this paper, we present a comprehensive dynamic matching algorithm with the focus on call graph and control flow matching. Our technique works in the presence of aggressive control flow transformations (both interprocedural such as function inlining/outlining and intraprocedural such as control flow flattening) and produces mappings of interprocedural and intraprocedural control flow in addition to mapping between instructions. We evaluated our dynamic matching algorithms by attempting to match original program with versions that were subjected to popular obfuscation and control flow altering transformations. Our experimental results show that the control flow mappings produced are highly accurate and complete, for the programs considered.

1 Introduction

In many application areas, situations arise in which there is a need for comparing two versions of a program that dynamically behave the same even though they statically appear to be different. Let us consider applications such as software piracy detection and debugging of optimized code, in each of which one program version is created by transforming the other version. In the first application, code obfuscation transformations may have been performed to hide piracy [3, 15]. In the second application, transformations are applied to generate an optimized version from the unoptimized version. We proposed the idea of *dynamic matching* [18] recently, which automatically produces a mapping between subsets of statements from two program versions that are executed for the same given input. However, our matching technique [18] assumes that the function to function correspondence is known from the symbolic debugging information, which may not be true in general, especially for commercial programs and pirated programs. Furthermore, it only produces mappings on instruction level which greatly diminishes its effectiveness in matching two program versions whose control flows differ significantly.

Obfuscation [4] is such a technique that substantially alters a program's control flow in order to thwart the understanding of the program. It can be used by software pirates or malicious programmers to hide the real identify of a program [4]. Matching control flow can greatly facilitate the understanding of obfuscated programs. Moreover, in obfuscation transformations, a significant amount of bogus instructions are intentionally injected in order to further confuse a program reader. These bogus instructions incur a lot of false matches for the matching technique in our prior work[18]. In this case, control flow matching technique proposed in this paper can deliver more accurate mappings.

In this paper, we present a comprehensive dynamic matching algorithm that works in the presence of aggressive control flow altering transformations. We evaluated our dynamic matching algorithms by attempting to match the original program with versions that were subjected to three existing obfuscation transformations including *Control flow flattening* [15], *Static Disassembly Thwarting* [10] and insertion of *Binary Opaque Predicates* [4] and other control flow altering transformations including function inlining. Our experimental results show that the control flow mappings produced are highly accurate and complete.

1.1 Overview of Dynamic Matching

We illustrate the complexities and goals of dynamic matching using the example in Fig. 1. This example considers a simple program with three functions: Main, B and C. As we can see in the original version, the functions B and C are called by Main; additionally B is again called within

```
Main(){
 ...
 call B()
 ...
 call C()
 ...
}

function B{
 i1: a=b+c;
     if(a>2) goto i3;
 i2: b=b+1;
 i3: a=a+1;
     if(b<9) goto i1;
     return;
}

function C{
 i4: m=n;
     call B()
 i5: m=m+1
     return;
}
```

```
function C'{
 i4': m=n;
 x1: x=0;
 l1: switch(x)
 case 0:
   i1': a=b+c;
   x2 : x=1;
        goto l1;
 case 1:
   i2': b=b+1
   x3 : x=2;
        goto l1;
 case 2:
   i3': a=a+1
   x4 : x = (b<9) ? 0:3;
        goto l1;
 case 3:
   i5': m=m+1
        return;
}
```

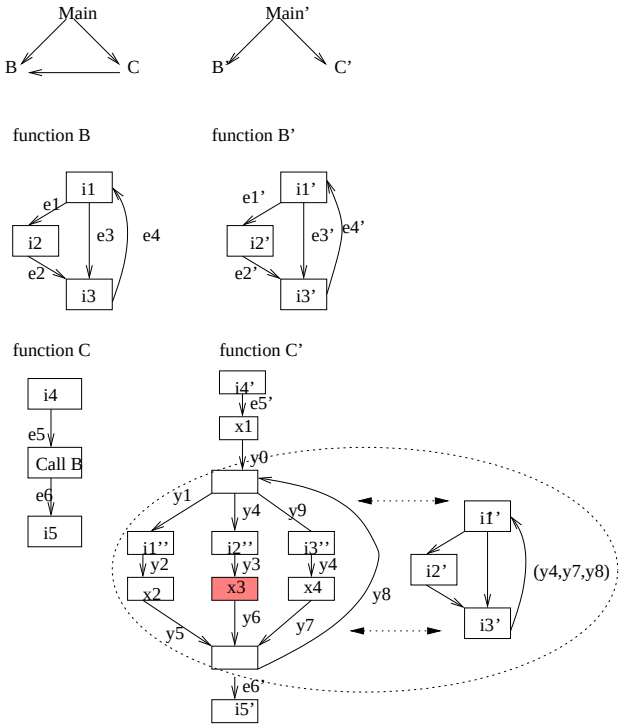

Figure 1. Before and After Versions of Code, Call Graphs and Control Flow Graphs.

Table 1. Interprocedural control flow Matching

Modified Version	Original Version
Main'	Main
B'	B
C'	C, B

Table 2. Instruction matching

Modified Version	Original Version
i4'	i4
i1', x3	i1
i2'	i2
i3'	i3
x1,x2,x4	-
i5'	i5

Table 3. Intraprocedural control flow matching

Modified Version	Original Version
e5'	e5
(y1,y2,y5,y8,y4)	e1
(y3,y6,y8,y9)	e2
(y4,y7,y8)	e4
(y2,y5,y8,y9)	e3
e6'	e6

function C. In the transformed version, while Main and B remain unchanged, C is transformed by inlining the call to function B. Further, the inlined code is *flattened* using the obfuscation transformation developed by Wang et al.[15]. If we look at the two versions we can see that the two programs appear to be highly dissimilar. Moreover if we have the mappings between instructions only, it is hard to see that the two are the same. This is because the control flow structure of the programs differ greatly and the transformed program contains many instructions that have no match in the original program.

Dynamic matching is a three step process consisting of interprocedural control flow matching (or call graph matching), instruction matching, and intraprocedural control flow matching. Each of the above steps uses dynamic information collected from program runs and produces a mapping. Let us illustrate the mappings produced using the example in Fig. 1. In the first step, the call graphs of the two versions are matched to produce a mapping as seen in Table. 1. While the mappings between Main' and B' with their respective counterparts are straightforward, the fact that C' is mapped with both C and B indicates that B has been inlined within C.

In the second step, instruction matching is performed within the corresponding functions using the dynamic execution histories of the instructions. This step is basically the same as in prior work [18] and is not the main contribution of this work. Let us assume we have the matches for the instructions in function C' as shown in Table 2. Observe that several instructions (x1, x2, x4) in the transformed program have no corresponding matches in the original version. Also note that the instruction i1 in the program has two matches i1' (correct match) along with a false match with x3. (which should have remained unmatched). This scenario is quite common as reported in [18]; we also later confirm in our evaluation that the instruction matching step though devoid of false negatives includes a non-trivial amount of false positives. One of

the advantages of control flow matching is to prune out these false matches. In the corresponding example we are able to prune this out by reasoning that there is no direct control flow edge between i2 and i1 but there is an edge between i2' and x3, which means i1 cannot match x3. In the third step, intraprocedrual control flow matching is performed. We use the fact that C' contains inlined code of B and the fact that the instructions x1 and x2 are not matched with any instructions in the original version to produce the mappings given in Table 3. Finally it should be noted that while the dynamic matching algorithm proposed in prior work would only have produced the mappings in Table 2, the dynamic matching algorithms in this paper will, in addition, generate interprocedural and intraprocedural control flow mappings as seen in Table 1 and Table 3.

2 Comprehensive Dynamic Matching

In this section, we present a dynamic matching algorithm that produces comprehensive mappings between two program versions including interprocedural and intraprocedural control flow mappings, in addition to instruction mappings. Dynamic matching involves the following three steps: Callgraph matching, instruction matching within the matched functions and intraprocedural control flow matching.

Dynamic matching hinges on our prior work, *Whole Execution Trace* (WET) [17] , a representation that stores the comprehensive execution history of a program in a highly compact form. The following is the set of dynamic information from the WET which we utilize in our matching:

- Dynamic Call graph DCG. This is the subset of the *static call graph* that is actually *exercised* during an execution.
- Dynamic Control flow graph DCFG. This is the subset of the *static control flow graph*. An edge or a node in the static control flow graph is also a part of DCFG if the edge or node is *executed at least once*.
- Dependences exercised. For each instruction the set of all instructions upon which it is dependent is captured. Thus we form the dynamic data dependence graph (DDDG).
- Values produced. A stream of results produced by an instruction during the execution is abstracted into a set of distinct values along with its frequency.

Each of the matching steps takes two directed graphs as inputs. In the callgraph matching step, we attempt to match the DCGs of the two versions; in the instruction matching step, the DDDGs of the two versions are matched and in the intraprocedural control flow matching step, the DCFGs of the two versions are matched. The goal of the matching algorithms is to produce *matches* between these graphs. A match is essentially a mapping between the nodes (edges) of the directed graphs. While the callgraph and instruction matching algorithms match functions and instructions, which are nodes, the intraprocedural matching algorithm matches the edges of the DCFG. To enable the matching of the above graphs, we annotate the nodes (edges) of the graphs with dynamic information collected. These labels are known as *signatures*. We arrive at the final match relation based on two behavioral dimensions: the compatibility of labels or signatures and the structural isomorphism of the directed graphs. Thus, essential to each of the above algorithms are the two prior steps that first, define the signatures of the node(edges) and second, define the exact condition in which the two signatures of the nodes(edges) are said to be compatible.

2.1 Callgraph Matching

The aim of the *interprocedural control flow* or *call graph* matching step is to produce mappings between functions of the two versions. The mappings are of the following form: for each function or a set of functions in the transformed version we identify a corresponding function or set of functions in the original version of the program. There are two reasons for performing interprocedural control flow matching. First, it indicates what interprocedural transformations were employed in creating the transformed version from the original version. Second, it breaks down the overall matching problem into subproblems such that instruction matching and intraprocedural matching can be performed only between corresponding functions.

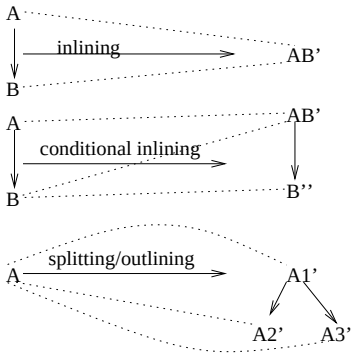

Figure 2. Interprocedural transformations

We observe that there are basically two interprocedural transformations that can change the structure of the call graph: function inlining and function splitting (outlining). In the *inlining* transformation, a call to a function is replaced by the code of the entire function. Note that inlining can be conditional as in the second example above. In function *splitting* or *outlining*, one function is split into multiple functions. One of the obfuscation transformations [10] inserts branch functions into program code and can be viewed as an instance of function outlining/splitting.

2.1.1 Function Signatures/Labels

To compute the matches, we define the *signature* of a function as the set of all *(distinct value, frequency)* pairs produced by the instructions in that function. Alternatively, the signature consists of the set of values, each value repeating as many times as its frequency. Hence the *multi-set* of the values produced in a function is considered as its signature. By multi-set we mean that the set contains repetitions, but the order of the occurrence does not matter. We believe that this would serve as a good signature since the repetition information of values produced (frequencies) serve as a good reflector of the inlining and outlining transformations.

Now let us develop the conditions under which two signatures are considered *compatible*. Note that this condition should be such that it avoids false negatives. In other words, whenever two functions match, their signatures must match. One obvious compatibility metric would be set equality, but clearly this is too strict a metric when obfuscation or optimization transformations are applied as these transformations may change the set of values produced by a function. We then looked at the subset operator using which, two signatures are said to match, if either of them is the subset of other. Thus the *sMatch* operator which decides whether two functions A and B are signature-compatible is defined by:

$$sMatch(A, B) \equiv B \subseteq A \vee A \subseteq B$$

Although this practically ensures that there would not be any false negatives, intuitively, one would expect a lot of false positives; a lot of functions that should not eventually match may be considered signature compatible. This is not really a problem because we also use the structure of call graph in addition to signature compatibility.

2.1.2 Algorithm

We now present our call graph matching algorithm. Given the dynamic call graphs of the two program versions, this algorithm produces mappings between functions of the two call graphs. Assume that functions A and B are the roots of the call graphs of the two versions. Since they are roots they are potential matches. If the signatures of A and B are compatible according to the definition of $sMatch$ given in the preceding section, we add (A, B) to the set of all matched functions. But since our algorithm accounts for inlining (and outlining), we should allow for other potential matches with A and B. Here we make the observation that if there is indeed a function F that matches with A, its signatures should match with the values of function A, that have not yet been matched with B. Thus, whenever we match the signatures of two functions, we update their signatures to compute *residual signatures* to be used in further matching as follows: $sign(A) = sign(A) - sign(B)$ and $sign(B) = sign(B) - sign(A)$. Since the functions A and B matched , each of the children of A (referred to as S_A)

```
Match(A,B) {
    MatchSet ← MatchSet ∪ (A, B);
    sign(A) = sign(A) − sign(B);
    sign(B) = sign(B) − sign(A);
    S_A ← children(A);
    S_B ← children(B);
    for each (X ∈ S_A, Y ∈ S_B) :
                sMatch(X, Y) ∧ NotMatched(X, Y)
        Match(X, Y);
    end for
    for each X ∈ S_A : sMatch(X, B) ∧ NotMatched(X, B)
        Match(X, B);
    end for
    for each Y ∈ S_B : sMatch(A, Y) ∧ NotMatched(A, Y)
        Match(A, Y);
    end for
    for each (X ∈ S_A, Y ∈ S_B) :
                NotMatched(X, −) ∧ NotMatched(Y, −)
        (P, Q) = Find(X, Y);
        Match(P, Q);
    end for
}

Find(A,B) {
    if sMatch(A, B) return (A, B);
    for each M ∈ (A ∪ children(A)) , N ∈ (B ∪ children(B))
        if (M, N) ≠ (A, B) Find(M, N);
    end for
    return false; }

Main() {
    (A, B) = Find(ROOT, ROOT′);
    Match(A, B);
}
```

Figure 3. Call Graph Matching Algorithm

potentially can match those of S_B. As shown in step 1 of Fig. 4, for every function $F \in S_A$, we then use the *signature compatibility* information (the *sMatch* relation) to determine the matching set $Match(F) \subset S_B$.

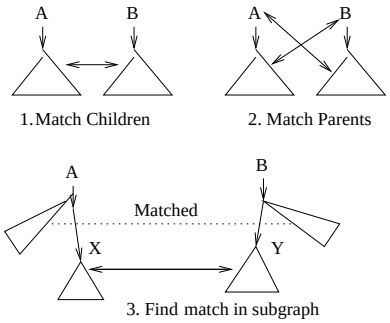

Figure 4. Callgraph matching

After matching the children, we now try to account for function inlining and function splitting. For example A and its callee may have been inlined in B. As far as inlining is concerned, S_B is could potentially match A and similarly B could potentially match with S_A as far as outlining is concerned. Those functions in S_B that are additionally signature compatible with A are thus added to the match set of A. Similarly functions in S_A that are signature compatible with B are added to the match set of B. This is shown in step 2 of Fig 4. Having done all the above matches, now assume that we are left with functions $X \in S_A$ and $Y \in S_B$

which have not yet been matched. Now we attempt to find a match in the subgraphs G_X and G_Y rooted by X and Y respectively. We traverse the two subgraphs until we find $(P \in G_X, Q \in G_Y)$ such that P and Q match. The main idea of performing this step is to account actual functions enclosed in "wrapper" functions for the purpose of obfuscation. For example in Fig. 5, we observe that the root R of the original function does not match with its counterpart X since X is just a wrapper function which in turn calls the actual matching function. Here it is important to note that there may be functions present in both the versions that do not match. Extra functions may have been introduced in the obfuscated version for the purpose of confusion. Similarly functions in the original version may have been optimized away in the optimized version. If this is the case, we will not find any matches within the subgraphs of X and Y.

We summarize the algorithm in Fig.3. The algorithm accepts as input the roots of the two call graphs that need to be matched. It returns $MatchSet$, the set of function matches. We first start by attempting to match the roots of the call-graphs, as structurally they are potential matches. The call to function $Find$ accounts for the wrapper functions scenario that we previously discussed. It traverses the subgraphs rooted at the two nodes it receives as parameters (A and B), and returns a pair of functions that are signature compatible. If the two functions A and B are themselves signature compatible it returns them. The function $Match(A, B)$, basically performs the operations when it is known that A and B are both structure and signature compatible. First, it recomputes the signatures of the functions to account for the match as discussed earlier. Then it proceeds to try and match the children of A and B among themselves. Finally it deals with unmatched children by calling the $Find$ function. In this algorithm, the function $NotMatched(X, Y)$ returns true if the entry (X, Y) is not found in the $MatchSet$. Furthermore, $NotMatched(X, -)$ returns true if X does not belong to any pair in the $MatchSet$.

An Example We apply the above algorithm to the example shown in Fig. 5 and show how the algorithm identifies the correct matches. First notice that the roots of the two call graphs do not match. X and Y in the second call graph are dummy functions, whose only purpose is to call the actual root $R_1A'_1$. So when $Find(R, X)$ is called in the beginning, it returns $(R, R_1A'_1)$ since this is the first match encountered in the subgraph (in this case the whole graph) with R and X as roots. Consequently, $Match(R, R_1A'_1)$ is called and is hence added into the $MatchSet$.

Now the children are matched resulting in A and A'_2 being matched. Then we proceed to match the children with the parents. Accordingly, A is matched with $R_1A'_1$ and R'_2 is matched with R. Let us now consider the call to $Match(R, R'_2)$: R has A as its child but $R2'$ does not

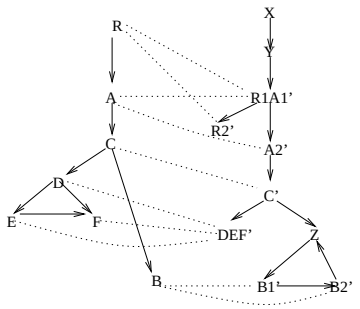

Figure 5. Example matching

have any child. Further, A is already mapped to R'_2's parent. Hence there is nothing to do in this call. Similarly there is nothing to be done in the call $Match(A, R_1A'_1)$. Let us consider the call $Match(A, A'_2)$. Since C and C' directly match, a call to $Match(C, C')$ results. Here it is interesting to observe that statically matching the structure of the subgraph with C and C' as roots will result in a set of wrong matches (where D will be matched to Z and B with DEF'). On the other hand, using our dynamic matching algorithm, D (and then E and F eventually) will be mapped to DEF'. But B and Z cannot be matched. Subsequently, $Find(B, Z)$ is called which returns (B, B'_1). Consequently B and B'_2 are matched. Finally note that the call $Match(B, B'_2)$ terminates even though there is a cycle in the graph. It terminates because all the nodes have been matched. The final $MatchSet$ produced is: $\{(R, R_1A'_1), (R, R'_2), (A, R_1A'_1), (A, A'_2), (C, C'), (D, DEF'), (E, DEF'), (F, DEF'), (B, B'_1), (B, B'_2)\}$.

2.2 Dynamic Instruction Matching

We use the instruction matching algorithm proposed in our previous work [18]. Each instruction is assigned a signature based on the values produced by the instruction. Initially a conservative matching set consisting of pairs of instructions is produced based on the signature compatibility. Next structural isomorphism is taken into account driven by the dynamic dependence graph information. Spurious matches are removed giving the final match set. It is possible that some instructions do not have any matches. Although, the previous work concerned matching of optimized and unoptimized versions, we found that the same approach worked well even under the presence of obfuscation transformations. As reported in the previous work, the final match set, though devoid of missing matches, contains a small number of *false matches*. The algorithm was designed to avoid missing matches while producing false matches because when the mapping is presented to the user, it is easier for the user to detect a spurious match than it is to identify a missing match.

2.3 Intraprocedural Matching

The goal of the intraprocedural control flow matching is to generate mappings between edges of the two program ver-

sions. This step relies on solutions produced by the preceding steps, namely call graph matching and instruction matching. Intraprocedural control flow matching is applied to corresponding functions identified during call graph matching. The instruction matching information (i.e., mapping between instructions and a set of unmapped instructions) is also used during control flow matching. Given the above information, next we describe an algorithm to match the control flow of a pair of corresponding functions. We present the algorithm in two steps. In the first step we present an intraprocedural control flow matching algorithm that assumes that there are no false positives produced in the instruction matching phase. In the second step we deal with the issue of false positives.

2.3.1 Algorithm

```
Match(EdgeList, source_1, next) {
    case next_0 :
        for each edge{next_0, i}
            Match(EdgeList ∪ {next_0, i}, source_1, i);
        end for
    case next_1 :
        if ∃ edge{source'_1, next'_1}
            MatchSet = MatchSet∪
              (EdgeList, {source'_1, next'_1});
        else
            if ∃ path : source'_1, i*_0, next'_1
              for each path : source'_1, i*_0, next'_1
                MatchSet = MatchSet∪
                (EdgeList, {source'_1, i*_0, next'_1});
              end for
            else
              MatchSet = MatchSet ∪ (EdgeList, φ);
        endif
    end case
}
Main() {
    for each edge{a_1, b}
        Match({a_1, b}, a_1, b);
    end for
}
```

Figure 6. Intraprocedural Matching Algorithm

Let us consider two functions F and F' from the original version and the transformed versions respectively. Let us consider the situation in which the original version was optimized aggressively after which obfuscation transformations were applied, finally yielding F'. Because of the application of optimization and then obfuscation transformations, several instructions (and edges) present in F' may not be present in F (due to the effects of obfuscation) and similarly several instructions (and edges) present in F may be absent in F' (due to the effects of optimization). The goal of this algorithm is to find a correspondence between the edges in the DCFGs of the two versions. In the following algorithm a node denoted with subscript 0 (x_0) indicates that the instruction (i.e., x) does not have a match. If a node is specified with a subscript 1, it means that the instruction x_1 has a matching instruction in the other version. Also x'_1 refers to the matching instruction in the transformed function that corresponds to instruction x_1 in the original version. Note that the above information are obtained from the previous instruction matching step.The algorithm works by parsing edges in the original version one by one till a path is formed such that the initial and final nodes have corresponding matches in the transformed version, while the nodes in between(if any) are unmatched. In other words, it parses edges of the form $source_1 i^*_0 dest_1$. Then it attempts to find a corresponding sequence of edges from F' matching the above. For intraprocedural matching we define the signature of an edge (or a sequence of edges) to be the source and destination nodes. The signature compatible edges from the transformed version are those edges whose source and destination nodes have matches in the original version. In other words the signature compatible edges from the transformed version are the set of paths from $source'_1$ to $dest'_1$. But not all such paths are structurally compatible with the original. Only the paths from $source'_1$ to $dest'_1$ that pass through unmatched nodes are structurally compatible. Thus the algorithm generates all possible paths from $source'_1$ to $dest'_1$ that in turn pass through unmatched nodes. This is precisely the required sequence of edges matching the original.

In the function *main*, all edges of the form (a_1, b) are considered for finding a match. The function *match* does the actual work of first parsing the input edges into the required form ($source_1 i^*_0 dest_1$) and searching for matches in the transformed version. To enable this we have the source node, the destination and a list of edges (including the source edge) that has been encountered till now, as the parameters for the *match* function. If the destination (the *next* parameter in the *match* function) turns out to be an unmatched node, then we continue parsing edges till the destination node has a corresponding match. Having parsed the original edges into the required form, we now proceed to find the corresponding matching edges in the transformed version. If there indeed is an edge between the corresponding source ($source'_1$) and destination ($dest'_1$) nodes (in the transformed version), then that edge is added to the match set. Otherwise, all paths from the source to the destination are generated. Out of this, a subset of paths that only pass through unmatched nodes are added to the match set.

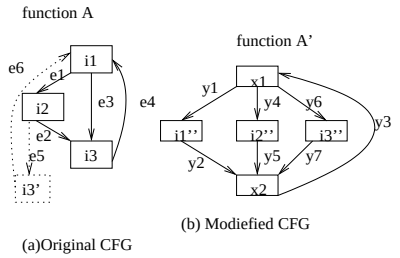

Figure 7. Example matching

An Example. Intraprocedural control flow matching is explained using the example in Fig. 7. In this example, the original version itself contains an obfuscation transformation: an advanced form of opaque predicate transformation in which node $i3$ is duplicated into i'_3 and edges e_5 and e_6 are added. An 'opaque' predicate is added at i_2 which some times leads to the execution of i'_3 and sometimes i_3. The software pirate, who manages to uncover this opaque predicate transformation, himself/herself then adds Wang's control flow flattening in his modified version. Hence the modified version does not contain the opaque predicate but additionally the flattened version of the original. Starting with the node x_1 applying the algorithm, we obtain matches for the sequence of edges in the modified to the original. For example, consider the sequence $(y_1y_2y_3y_4)$. This sequence is basically an edge between two matched nodes (i''_1, i''_2). Since the two instructions correspond to i_1 and i_2 of the original, we check to see if there is an edge between these instruction, resulting in matching $(y_1y_2y_3y_4)$ to e_1. Now let us consider the sequence $(y_6y_7y_3y_1)$. These edges connects i''_3 and i''_1. As i''_3 of the modified version matches both i_3 and i'_3 of the original, the sequence of edges thus match with both e_6 and e_4. Finally let us consider the sequence $(y_4y_5y_3y_1)$. These edges take us from i'_2 to i'_1 which in turn map to i_2 and i_1 respectively. Since neither a direct edge from i_2 to i_1 is present, nor is there a sequence of edges from $i2$ to $i1$ through instructions that have no matches in the transformed version, the sequence $(y_4y_5y_3y_1)$ is not matched. In a similar fashion all other edges are matched.

2.3.2 Dealing with false matches

The above intraprocedural control flow matching algorithm does not account for false matches produced in the instruction matching step. The presence of false matches affects

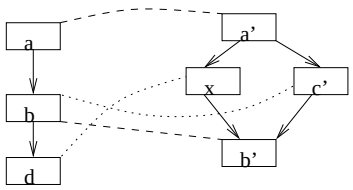

Figure 8. False Matches

the above algorithm in two ways. First, the effects of false instruction matches are carried over to this stage and produce false edge matches. For example in Fig. 8 instruction $b \in V(P)$ is (correctly) matched with b', but is also (falsely) matched with c'. Because of this, the edge $(a, b) \in E(P)$ will also be matched with the edge (a', c') in addition to the correct match, (a', x, b'). Second, and more importantly, this may even cause some false negatives in the intraprocedural control flow matching. In Fig. 8, the instruction $x \in V(T)$ is supposed to be unmatched. This would have enabled the correct matching of the $(a', x, b') \in T$ with $(a, b) \in E(P)$. But if x is falsely matched with d (say), then the correct edge match will not be formed eventually resulting in a false negative.

We use prioritization of the matching instructions to solve the problem. During the instruction matching phase, whenever a node in the original program matches multiple instructions in the transformed program, we prioritize the instructions using the structure of the graphs. Suppose $a \in V(P)$ is the original instruction that was matched with the set $C = c_1, c_2...c_n$ of instructions in the transformed version, we prioritize each c_i by computing the *confidence* measure for each one of them. In the following definition of confidence, the expression ***reachable(a,b)*** evaluates to true if b is reachable from a and the expression ***C(a,b)*** evaluates to true if a and b are signature compatible.

Definition. (Confidence of the match of $a \in V(P)$ with $c_i \in V(T)$)

$$\frac{|v_1 : \{reachable(a, v_1) \wedge (\exists v_2 : reachable(c_i, v_2) \wedge C(v_1, v_2))\}|}{|v_1 : \{reachable(a, v_1)\}|}$$

where $v_1 \in V(P)$ and $v_2 \in V(T)$

Let us first consider the denominator. It refers to the cardinality of the set of nodes reachable from the node a in the original CFG. The numerator refers to the cardinality of the set of nodes reachable from a, each of which additionally should be signature-compatible with some node v_2 (of the transformed CFG) which in turn should be reachable from c_i in the transformed CFG. Intuitively, *confidence* of c_i is the measure of the total reachable nodes from a having matching counterparts in the transformed CFG that are reachable from c_i. Having obtained the confidence values for each of c_i we then proceed to sort them and assign each of c_i a number based on its position in the sorted order. Using these priority values, we can order the matching edges in the intraprocedural matching algorithm. Now let us deal with the second problem. Consider a node $x \in V(T)$. Suppose it is matched with nodes $a_1, a_2...a_n$ ($a_i \in V(P)$) with priorities $p_1, p_2...p_n$ respectively, we compute a confidence metric for x which is the confidence that $x \in T$ is actually unmatched. Also note that the lowest of all the priority values among p_i determines the the best match of x. Intuitively, higher this value, higher the confidence that this node is actually unmatched. Hence we use the least priority value as the measure of unmatched confidence.

$$Unmatched.confidence(x) = Min(p_1, p_2...p_n)$$

We now describe the use of priority values in the intraprocedural control flow matching algorithm. Assume we need to find the edge(s) in the transformed version corresponding to the edge $(a, b) \in E(P)$. Earlier in the description of intraprocedural-procedural control flow matching algorithm,

we assumed that there were unique a' and b' in the transformed version corresponding to the original. But now there could be potentially a set of nodes A' and B' corresponding to the original. First we choose $a' \in A'$ and $b' \in B'$ in the order of their priorities in matching the original a and b. This takes care of the first problem. But there could be be several paths from a' to b'. We prioritize the paths from a' to b' as follows. For satisfying the structural compatibility the path from a' to b' should pass through unmatched nodes. We use the *unmatched.confidence* values of the intervening nodes as a measure of the path's confidence. For each path from a' to b' we calculate the confidence of the path as:

$$Confidence of Path(a' x_i b') = \frac{\Sigma Unmatched.confidence(x_i)}{|x_i|}$$

Intuitively the confidence of a path is greater, if all the intermediate nodes in the path are more likely to be unmatched nodes. Using the confidence values for each path, the paths can be prioritized.

3 Experimental Evaluation

To evaluate our matching algorithms, we implemented some well-known obfuscation transformations in addition to function inlining and other optimizations into DIABLO, a binary rewriting tool [5]. All the programs were statically linked as diablo works on statically linked programs. We then modified the tool *lackey* of *Valgrind* [19] to collect the dynamic execution histories consisting of values and dependencies exercised. We zeroed out all the values that were addresses, as the actual values themselves are likely to be different across executions. The interprocedural transformations we applied were *function inlining* and *static disassembly thwarting* [10], an obfuscation transformation that can be thought as a variant of outlining. Before applying the intraprocedural obfuscation transformations we applied *dead code elimination*, as obfuscation of code that is not executed can be easily filtered and is not very useful. Finally we applied the *opaque predicates transformation* [4] that have a property that is known at obfuscation time, but which is hard to discover afterward. The effect of this transformation is that the control flow of a program can be cluttered with unrealizable paths.

We evaluate the effectiveness of our matching algorithms in terms of accuracy and speed. Recall that the first two transformations, Function Inlining and Static Disassembly Thwarting, mainly change the interprocedural control flow, while the last two obfuscate the intraprocedural control flow of the programs. Thus we experiment with three versions of programs. First, the original unobfuscated version *(u)*. Second, a version in which the first two transformations are applied to mainly alter the interprocedural control flow *(inter)*. Finally the third version in which the last two transformations are *also* applied in addition to the two used in the previous version *(intra+inter)*.

3.1 Effects of applied transformations

We evaluated our algorithms using eight C-programs of the SPECint 2000 suite. For collecting the traces, we ran the three versions (u,inter,inter+intra) with *test* inputs. The effect of the control-flow altering transformations on the programs used in this evaluation are summarized in Table 4. For each version of the benchmark we give the following information: the static number of functions and instructions, the dynamic number of instructions and functions calls and the dynamic number of branch statements. The large variation in the values across different versions, shows that the transformations alter the programs aggressively.

Table 4. Effects of the Transformations.

Bench.	Functions		Instructions		Br.(10^6)
	Static	Dyn.(10^6)	Static	Dyn.(10^6)	
bzip	86	17.6	5105	3760	510
	69	19.5	5766	4300	510
	69	157	11625	12100	1350
gzip	94	8.21	3660	975	160
	86	8.53	3997	1150	160
	86	78.8	8368	4350	441
mcf	107	.460	4623	127	27.0
	89	3.89	4854	165	27.0
	89	23.8	7383	603	60.1
crafty	169	1.73	22588	148	13.6
	151	2.83	25355	168	13.6
	151	11.3	51872	367	39.8
twolf	220	2.21	18513	223	28.2
	180	7.54	20641	280	28.2
	180	38.6	49132	816	85.0
parser	244	34.6	17659	1891	367
	201	63.1	20390	2205	367
	201	315	36813	7456	992
gap	282	10.3	16483	466	71
	240	19.2	18801	561	71
	240	94.2	31123	1522	251
perl	424	7.6	25574	42.9	6.67
	368	12.8	29283	48.1	6.67
	368	21.6	51765	167	25.0

3.2 Interprocedural Matching

In this experiment, we match the original callgraph with the one in which function inlining and static disassembly thwarting is applied. Accuracy of the matches are measured using the metrics of false matches and missed matches. It is important to note that since we had access to the obfuscation code and the code for performing inlining, we knew the exact matches. This served as an oracle for evaluating the matching algorithms. The results are summarized in Table 5. As we see, we do not miss any correct matches in all the benchmarks and we do not obtain false matches in six out of eight of the benchmarks. The false matches arose when functions from the original version were said to be inlined although they were not. The false matches arose because the signatures of some of the functions were too small. All functions in the original version were matched, and all but one function in the transformed version were matched. This corresponds to the *branch function* added by the obfuscation transformation. We also measured the time taken for performing interprocedural matching on a Pentium 4 3GHz machine with

1 GB RAM. As we see the time taken for performing the match is very small. A maximum of 16 seconds were needed by `bzip`.

Table 5. Interprocedural Matching

Bench.	False Matches	Missed Matches	One-Many Matches	Time (secs)
bzip	0	0	18	16
gzip	0	0	9	11
mcf	0	0	8	2
crafty	0	0	16	10
twolf	7	0	48	9
parser	0	0	44	12
gap	0	0	43	9
perl	5	0	57	1

Table 6. Instruction Matching

Bench.	Actual Matches(%)		Generated Matches(%)		Time
	Orig.	Obfus.	Orig.	Obfus.	secs
bzip	99	56	99	72	251
gzip	100	44	100	63	157
mcf	100	63	100	76	120
crafty	99	43	99	58	186
twolf	99	37	99	51	183
gap	99	55	99	70	212
perl	98	50	98	66	85
parser	100	47	100	68	282
Mean(%)	99	49	99	65	184

3.3 Instruction Matching

Using the function mappings produced, we then proceed to match the instructions inside the each matched functions. The first two columns of Table. 6 represent the percentage of original and obfuscated instructions that are actually matched. The actual instruction matches are obtained beforehand by making the obfuscation tool emit the matching instructions. We find that most of the original instructions (98-100%) are matched. The corresponding value for the matches generated by our algorithm is the same. In other words, there are no missed matches. This is not surprising, since the matching algorithm first forms a conservative match set based on signature compatibility and then proceeds to iteratively prune the set using structural information. We observe that only around 49% of the obfuscated instructions are actually matched. This is because the rest of them correspond to the additional code added by the obfuscation transformations. Next we see that the corresponding value generated by our algorithm is 65% (i.e. higher that 49%). This is because the algorithm is not able to prune all the false matches due to some instruction signatures being too short and the dynamic dependence graph structure being too simple. We also measured the time taken to match the instructions and it amounts to 3 minutes on the average which is very reasonable.

3.4 Intraprocedural Matching

In this experiment, we evaluate the effectiveness of intraprocedural matching. As we discussed earlier, the output of this stage is the mapping between the original edges and the obfuscated sequence of edges. This algorithm thus emits a set of mappings of edges between the two versions. But unlike the previous evaluation, we do not have the actual mappings between the control flow edges even though we have the correspondence between instructions and functions. Therefore the actual correspondences (the oracle data) between the control flow edges can only be identified by manual inspection. Hence we performed the evaluation of this algorithm by manually inspecting the control flow graphs. Since it was too tedious to inspect the control flow matches within each and every function, we used the following criteria to select one function from each benchmark for this evaluation. In particular, we selected a sufficiently large function for which the number of distinct executed instructions in the two versions differed the most.

Table 7. Intraprocedural Matching

Bench.	Actual Matches	Without Pr.		With Pr.	
		Missed	False	Missed	False
bzip	278	86	21	12	17
gzip	150	34	5	6	8
mcf	245	61	7	7	8
crafty	189	43	12	5	8
twolf	557	147	29	21	27
gap	238	94	14	13	18
perl	168	52	24	4	10
parser	221	95	16	19	25
Mean(%)	100	29.9	6.2	4.2	5.9

It should be noted that although the mappings are *inspected manually* for evaluation purposes, they are *generated automatically* using our matching algorithm. We would also like to note that this procedure mirrors the actual application of matching algorithms to piracy detection. During piracy detection a manual inspector is needed to view the control flow matches generated by our algorithm and use this evidence to decide whether the transformed version is indeed a pirated one.

As we see in Table. 7, the first column refers to the actual number of matches that exists between the two versions found by manual inspection. The next two columns correspond to the application of the interprocedural algorithm without using prioritization (of Section 2.5.2) and in the presence of false instruction matches. Next we applied prioritization producing the results in the last two columns. In this process, we consider only the match with the highest priority, i.e we only choose the path with the highest *Unmatched.Confidence* as the legal match and reject the rest. We find that this simple heuristic is highly effective and is able to significantly reduce the missed matches to around 4% and the false matches to around 6%.

4 Related Work

Static differencing algorithms. An existing class of algorithms that compare two program versions are *static differencing* algorithms [1, 6, 9, 20, 8]. These algorithms perform differencing at different levels: while [1, 9] find differences

by comparing control flow graphs, [6] compares input/output dependences of procedures and [20, 8] work by comparing source code. These algorithms work with source or intermediate code representations of the program versions. In contrast our matching algorithms work at binary level and they match instructions that dynamically behave the same even though they statically appear to be different.

Differencing dynamic histories. Research has been carried out on *differencing dynamic histories* of program executions. The benefits of such algorithms for software maintenance have been recognized. In [12] Reps et al. made use of path profiles to recognize Y2K bugs in programs. Wilde [16] has developed a system that enables a programmer to visualize the changes in the dynamic behavior of a program. However, in these works dynamic histories of different executions, corresponding to two different inputs, of a single version of a program are compared. In contrast, our work considers matching of dynamic histories of two program versions on the same input.

Comparison checking [7] is a technique that determines whether erroneous behavior of the optimized version is being caused by a bug in the original unoptimized version that was unmasked by optimizing transformations or whether it is due to a bug that was introduced due to an error in the optimizer. Matching of executed instruction histories can be used to identify whether the bug is in fact in the original version or in the optimization transformation [18]. While the focus of our work has been to use matching for comparison checking and software piracy detection, variants of matching have also been proposed for carrying out impact analysis to assist in regression testing [11]. There has been work on deobfuscation [13] of the control-flow flattening transformation using a combination of static and dynamic analysis. While the goal of the above work is to transform an obfuscated binary into the original, the goal of our work is to match the executions of the two versions. Also our techniques are general and not targeted towards any particular obfuscation transformation.

5 Conclusions

In this paper, we presented a comprehensive dynamic matching algorithm that works in the presence of aggressive control flow transformations including interprocedural transformations like function inlining/outlining and and intraprocedural transformations like control flow flattening. Our results show that our algorithms are able to match the dynamic execution histories of the original and transformed versions with a high degree of accuracy and speed, for the benchmarks considered.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments. This work is supported by grants from Microsoft and NSF grants CNS-0719791, CNS-0708199, CNS-0614707 and CCF-0541382.

References

[1] T. Apiwattanapong, A. Orso, M.J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," *IEEE International Conf. on Automated Software Engineering*, pages 2-13, 2004.

[2] S. Chow, Y. Gu, H. Johnson, V. Zakharov, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", *G. Davida and Y. Frankel, editors, Information Security*, ISC 2001, volume 2200 of Lectures Notes in Computer Science (LNCS).

[3] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," *IEEE International Conference on Computer Languages*, pages 28-38, Chicago, IL, 1998.

[4] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," *Techreport, The University of Auckland, New Zealand*.1997.

[5] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere, "Link-Time Optimization of ARM Binaries", *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), 2004.

[6] D. Jackson and D.A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," *IEEE Conference on Software Maintenance*, pages 243-252, Nov. 1994.

[7] C. Jaramillo, R. Gupta, and M.L. Soffa, "Comparison Checking: An Approach to Avoid Debugging of Optimized Code," *7th European Software Engineering Conference and ACM SIGSOFT 7th Symposium on Foundations of Software Engineering*, LNCS 1687, Springer Verlag, pages 268-284, Toulouse, France, September 1999.

[8] R. Komondoor and S. Horwitz, "Semantics-Preserving Procedure Extraction," *27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 155-169, 2000.

[9] J. Laski and W. Szermer, "Identification of Program Modifications and its Applications to Software Maintenance," *IEEE Conference on Software Maintenance*, pages 282-290, Nov. 1992.

[10] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", *Proceedings 10th. ACM Conference on Computer and Communications Security* (CCS 03), Oct 2003.

[11] M.K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: Automatically Detecting Variations Across Program Versions," *IEEE/ACM International Conference on Automated Software Engineering*, ASE 2006.

[12] T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *6th European Software Engineering Conference and ACM SIGSOFT 5th Symposium on Foundations of Software Engineering*, pages 432-449, 1997.

[13] S. Udupa, S. Debray, and M. Madou, "Deobfuscation: Reverse Engineering Obfuscated Code", *12th Working Conference on Reverse Engineering*, (WCRE), Novemeber 2005.

[14] Z. Wang, K. Pierce, and S. McFarling, "BMAT - A Binary Matching Tool for Stale Profile Propagation," *The Journal of Instruction Level Parallelism*, 2, May 2000.

[15] C. Wang, J. Davidson, J. hill, and J. Knight, "Protection of Software-based Survivability Mechanisms," *International Conference of Dependable Systems and Networks*, pages 193-202, Goteborg, Sweden, July 2001.

[16] N. Wilde, "Faster Reuse and Maintenance Using Software Reconnaissance," Technical Report SERC-TR-75F, SERC, Univ. of Florida, CIS Department, Gainesville, FL, July 1994.

[17] X. Zhang and R. Gupta, "Whole Execution Traces," *IEEE/ACM 37th International Symposium on Microarchitecture*, Portland, Oregan, December 2004.

[18] X. Zhang and R. Gupta, "Matching Execution Histories of Program Versions", *10th European Software Engineering Conference* and *13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 197-206, 2005.

[19] *http://valgrind.org/*

[20] *MOSS: http://theory.stanford.edu/ aiken/moss/*