Structuring 2-way Branches in Binary Executables

Tao Wei, Jian Mao, Wei Zou, Yu Chen
Institute of Computer Science and Technology
Peking University
{weitao, maojian, zouwei, chenyu}@icst.pku.edu.cn

Abstract

One of the major challenges of control flow analysis in decompilation is to structure 2-way branches into conditionals, loop conditionals and switches. In this paper, we propose a graph-based method to formally describe structures of 2-way branches via the introduction of concepts called "compound branch subgraph" and "cascade branch subgraph". We then present novel structuring algorithms based on such concepts. Compared with previous works, our algorithms are deterministic rather than heuristic, and they do not use complicated data structures such as Interval/DSG. We show that in theory our algorithm is more accurate and efficient than typical current approaches; furthermore, we have applied the algorithm to several real-world binary executables, and experimental results validate such theoretical analysis.

1 Introduction

Decompilation is a key technology in reverse engineering area. Decompilation was initially introduced for porting programs across platforms. It then had been widely used in areas such as software maintenance and compilation verification. Since the 1990s, demand on decompilation from software security analysis community has been growing very quickly due to outbreaks of security vulnerabilities and malicious codes. [1]

When decompiling a program, it is important to correctly recover its underlying structures, such as loops, 2-way branches and n-way branches, from its corresponding binary executable. This paper focuses on how to structure 2-way branches.

2-way branch is a basic element of control flow structures in binary executables. The goal of structuring 2-way branches is to organize scattered 2-way branches into subgraphs, which represent three kinds of high level control structures—**`if`** (conditionals), **`for`**, **`while`** (loop conditionals) and **`switch`**. Structuring 2-way branches into conditionals or loop conditionals was first investigated by C. Cifuentes[2] in 1994; however, structuring 2-way branches into **`switch`**es hasn't been studied yet.

Unlike those straightforward but inaccurate approaches employed in previous efforts, we introduce strict definitions of related control flow structures based on graph theory, in particular, the definitions of ***compound branch subgraph*** and ***cascade branch subgraph***; we also propose novel algorithms to structure 2-way branches into subgraphs according to these definitions mentioned.

Benefiting from these strict definitions, our algorithms are more accurate than those proposed in previous works. Furthermore, our algorithms are deterministic rather than heuristic, and they do not use complicated data structures such as ***Interval/DSG***[5][6]. Hence they are more efficient than current ones.

The main contributions of this paper are:

1) Introduction of strict definitions of related control flow structures based on graph theory, in particular, definitions of ***compound branch subgraph*** and ***cascade branch subgraph***;

2) A novel 2-phase algorithm for structuring 2-way branches into conditionals and loop conditionals based on ***compound branch subgraph***;

3) An innovative algorithm for structuring 2-way branches into **`switch`**es based on ***cascade branch subgraph***;

4) Theoretical and empirical evaluations of various structuring techniques. Both theoretical analysis and experimental results show that our algorithm is more accurate and efficient than typical current approaches.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 provides an overview of preliminaries of structuring 2-way branches. Section 4 introduces the definitions used in this work. Section 5 presents the algorithm to structure compound branch subgraph. Section 6 provides the algorithm to structure cascade branch subgraph. Section 7 shows the theoretic analysis of our scheme. Section 8 reports our experimental results and findings. At last, section 9 concludes our work.

2 Related work

Structuring control flow graphs is an important problem in decompilation. As early as the 1970s Housel, Baker and Lichtblau *et al* proposed many algorithms to structure control flow graphs. However, structuring 2-way branches into conditionals and loop conditionals was introduced much later in 1994 by C. Cifuentes in her famous Ph.D. thesis "Reverse Compilation Techniques"[2], and structuring 2-way branches into cascade branches (**`switch`**) has not been studied yet.

C. Cifuentes proposed a heuristic structuring algorithm for conditionals and loop conditionals in her thesis[2][3]. E. Moretti *et al* proposed an inductive algorithm in 2001[4], and K. Kaspersky presented a manual method in 2004[7].

The heuristic algorithm proposed by C. Cifuentes is most well-known. It repeatedly binds together two 2-way branch nodes based on a matching subgraph showed in Fig.1, until there are no matched branch nodes left. The algorithm works well if **`x`** and **`y`** in Fig.1 are not correlated. However, if **`x`** and **`y`** are correlated, the algorithm might not structure nodes correctly. Fig.2 shows one such example, in which **`x`** is "**`(a && (b==1))`**", **`y`** is "**`(!a && (c==2))`**". Fig.3 is its control flow graph (CFG), where "**`T`**" represents the ***`True`*** branch edge, and "**`F`**" represents the ***`False`*** branch edge. It is easy to see that the graph can not be matched by the heuristic subgraph.

The inductive algorithm proposed by E. Moretti *et al* uses ***Interval/DSG***[5][6] as the boundary of branches, and then it adds predecessors of crossover of branches into branch head nodes. The algorithm handles correlated cases well, but it can not handle loop conditionals or binary executables resulting from compiling optimization. Fig.4 shows one such example: after it is compiled by the command "**`gcc -O2`**", the CFG of the resulting binary executable is shown in Fig.5. Branches (diamond nodes in the figure) belong to one **`if`** branch and two **`switch`** branches respectively, but this algorithm will incorrectly structure all branch nodes into one branch header set.

K. Kaspersky proposed a pure manual method: First, the follow node is determined by hand, and then nodes in branch and head are determined. His proposal provided guidance rather than an algorithm.

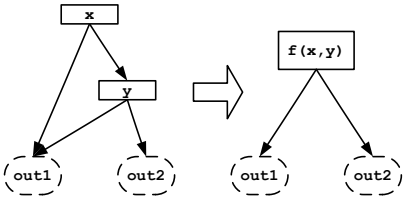

Figure 1. Heuristic subgraph

```
int foo(int a, int b, int c)
{
   if( (a && (b==1)) ||
       (!a && (c==2)) )
      return 1;
   else
      return 0;
}
```

Figure 2. A program containing correlative nodes

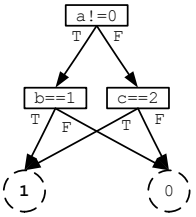

Figure 3. CFG of Fig.2

```
int foo(int a, int b, int *c)
{
      if( a == 1 ) {
            (*c)++;
            switch(b)
            {
            case 1:
                  return 0;
            case 2:
                  return 1;
            case 3:
                  return 2;
            default:
                  return -1;
            }
      }
      else{
            (*c)--;
            switch(b)
            {
            case 0:
                  return 0;
            case 1:
                  return 1;
            case 2:
                  return 2;
            default:
                  return -1;
            }
      }
}
```

Figure 4. A program with adjacent conditionals

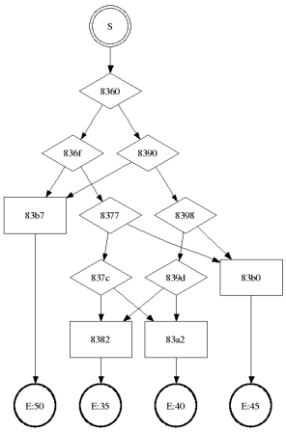

Figure 5. CFG of Fig.4

3 Preliminaries

This section briefly describes some basic conceptions in control flow analysis. The detailed descriptions could be found in [8].

The instructions of a program are organized into basic blocks, where program flow enters a basic block at its first instruction and leaves the basic block at its last instruction.

A control flow graph (CFG) is a connected and directed graph for describing control flow information of a program; it is often represented by a triple (N,E,h), where N is the set of basic blocks of the underlying program, E is the set of directed edges between these basic blocks, and h is the entry of the program.

For a basic block b, $Succ(b)$ is the set of successors of b, and $Pred(b)$ is the set of predecessors of b.

4 Definitions

In this section we first present some basic definitions used in our method, such as ***expanded basic block***, ***expanded CFG***, etc. We then state two key definitions of this paper—***compound branch subgraph*** and ***cascade branch subgraph***. These definitions together formalize the structure of 2-way branches.

Definition 1 ***Predict <=***: For two basic block b and c, if c is the only successor of b, and b is the only predecessor of c, then it is called $b <= c$.

Definition 2 ***Expanded basic block***: In a CFG $G=(N,E,h)$, an ***expanded basic block*** $e=<b_0,b_1,b_2,\ldots,b_n>$ satisfies these constraints:

(1) An expanded basic block is made up by basic blocks, i.e. $\forall i \in [0,n],\ b_i \in N$;

(2) An expanded basic block is continuous, i.e. $\forall i \in [0,n),\ b_i <= b_{i+1}$;

(3) An expanded basic block is maximum continuous, i.e. there is no basic block x in G, which satisfies $x <= b_0$ or $b_n <= x$.

It is obvious that one basic block belongs to one and only one expanded basic block.

Definition 3: Given an expanded basic block $e=<b_0,b_1,b_2,\ldots,b_n>$, let $Succ(e)=\{f \mid \exists b,\ f$ *is the expanded block containing* b, *and* $b \in Succ(b_n)\}$.

Definition 4: Given an expanded basic block $e=<b_0,b_1,b_2,\ldots,b_n>$, let $Pred(e)=\{f \mid \exists b,\ f$ *is the expanded block containing* b, *and* $b \in Pred(b_0)\}$.

The directed edges between expanded basic blocks are defined as the successive relationships of them.

Definition 5 ***Expanded Control Flow Graph***: The expanded control flow graph of a control flow graph $G=(N,E,h)$ is defined as $G'=(N',E',h')$, where N' is the set of expanded basic blocks of G, i.e. $N'=\{e \mid \exists b \in N,\ e$ *is the expanded basic block*

containing b}; ***E′*** is the set of directed edges between blocks in ***N′***, ***E′={<e1,e2>|e1 ∈ N′, e2∈Succ(e1)}***, and ***h′*** is the expanded basic block containing ***h***.

Definition 6 ***Predict <<***: for two expanded blocks e_1, e_2 in an expanded ***CFG G=(N, E, h)***, $e_1 << e_2$ if there is a path from e_1 to e_2.

Definition 7 ***Branch block*****:** a branch block is an expanded basic block which has exactly two successors. For a branch block ***b***, ***|Succ(b)|=2***.

Definition 8 ***Compound branch subgraph:*** Compound branch subgraph ***T=(N′,E′,h′,t,f)*** is a subgraph of an expanded graph ***G=(N,E,h)***, which satisfies:

(1) ***N′*** is the set of branch blocks in ***T***, i.e. ***N′⊂N***, for ∀ ***n∈N′***, ***|Succ(n)|=2***;

(2) ***E′*** is the set of edges in ***T***, i.e. ***E′={<n1,n2>|n1∈N′, n2∈N′-{t,f},<n1,n2>∈E}***;

(3) ***h′*** is the only entry of ***T***, i.e. ***h′∈N′;*** in the graph ***G***, for ∀ ***n ∈ N′-{h′}***, ***|Pred(n)|>0***, ***Pred(n)⊂N′***;

(4) ***t*** and ***f*** is the only two exits of ***T***, i.e. ***t,f∈(N-N′)∪{h′}***, ***t≠f***; in the graph ***G***, for ∀ ***n∈N′***, ***Succ(n) ⊆ N′ ∪ {t,f}***, and ***Pred(t) ∩Pred(f)∩ N′≠ ∅***;

(5) There is no cycle in ***T***.

Because ***T*** is acyclic, the restriction of ***<<*** on nodes in ***N′*** satisfies the transitive relation.

Compound branch subgraph is the topological representation of conditionals and loop conditionals. It can be proved that all compound branch subgraphs can be translated into conditionals or loop conditionals.

Definition 9 ***Cascade branch subgraph:*** Cascade branch subgraph ***S=(N′,E′,h′,C)*** is a subgraph of an expanded CFG ***G=(N,E,h)***, which satisfies:

(1) ***h′*** is the only entry of ***S***, ***h′∈N′***;

(2) ***N′*** is the set of branch blocks in S, which satisfies: ***N′⊂N***; for ∀ ***n∈N′***, ***n*** is an branch block; for ∀ ***n′ ∈N′-{h′}***, ***|Pred(n′)|=1***, ***Pred(n′) ⊂N′***, and ***n′*** is a branch judgment block of ***h′*** i.e. the only role of ***n′*** is to extend the judgment of ***h′***;

(3) ***E′*** is the set of edges in ***S***, i.e. ***E′={<n_1,n_2>|n_1∈N′, n_2∈N′-C, <n_1,n_2>∈E}***;

(4) ***C*** is the set of "**case**" exit nodes in ***S***, i.e. ***C⊆(N-N′)∪{h′}***; in the graph ***G***, for ∀ ***n∈N′***, ***Succ(n) ⊆ N′ ∪ C***, and for ∀ ***c ∈ C***, ***Pred(c)∩N′≠∅***; ***|C|>2***;

(5) There is no cycle in ***S***.

Because ***S*** is acyclic, the restriction of ***<<*** on nodes in ***N′*** satisfies the transitive relation.

Cascade branch subgraph is the topological representation of **switch** with small number of **case**; Notice that translating cascade 2-way branches into **switch** needs data flow analysis to identify branch judgment blocks, which is beyond the scope of this paper.

5 Algorithm for structuring compound branch subgraph

Based on definitions given above, the problem of structuring conditionals and loop conditionals is then transformed into one of finding compound branch subgraphs in a flow graph: given an expanded CFG ***G=(N,E,h)*** and a branch block ***h′***, find the entire compound branch subgraph ***T=(N′,E′,h′,t,f)*** rooted by ***h′***.

We propose a 2-phase algorithm as the following.

5.1 Expansion phase

Step 1: Let ***N′={h′}***;

Step 2: For any node n_i in ***G*** successive to a node in ***N′***, if n_i is a branch block, $n_i \notin N'$, and ***Pred(n_i)*** $\subseteq N'$, add it in ***N′***. Observe that such n_i will not introduce new cycles, and n_i is the largest one in ***N′*** according to the order defined by ***<<***;

Step 3: Repeat step 2, until ***N′*** is not changed any more.

Fig.7 shows the expansion phase of branch subgraph provided in Fig. 6. The parts with dashed line identify the exit nodes.

5.2 Contraction phase

Step 1: Compute the set of exit nodes $O=\{c \mid c \notin N'-\{h'\}, \exists b \in N', c \in Succ(b)\}$;

Step 2: If the number of nodes in O is not exactly TWO, choose nodes to remove from N'. These nodes should be largest nodes in the partial order defined by $<<$;

Step 3: Repeat step 1 and 2 until it stops, i.e. there are exactly TWO exit nodes left. Let these two nodes as t and f, and compute E' according to **Def.8(2)**, and then $T=(N',E',h',t,f)$ is the solution.

The contraction phase will stop in finite steps, for the minimum compound branch subgraph has only the node h' and its exit nodes are just h''s two successors.

Fig.8 shows the contraction phase of branch subgraph provided in Fig. 6.

The pseudo code of this algorithm is shown in Fig.9 and Fig.10.

6 Algorithm for structuring cascade branch subgraphs

Statement of the Problem: Given a expanded CFG $G=(N,E,h)$, and a branch block h', find the entire cascade branch subgraph $S=(N',E',h',C)$ rooted by h'.

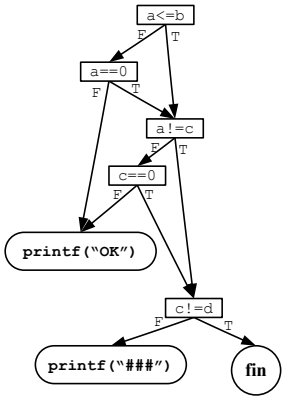

Figure 6. An example CFG

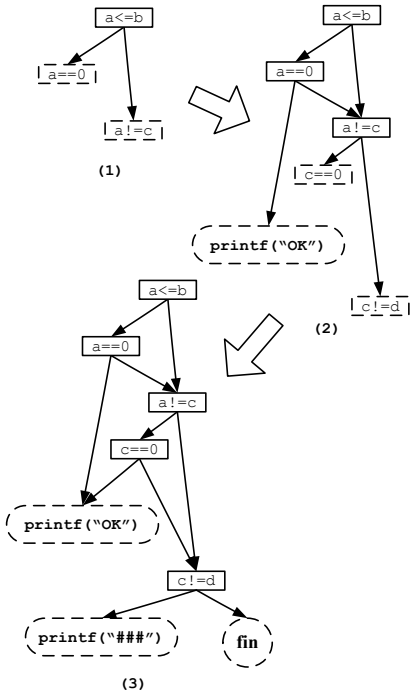

Figure 7. Expansion phase

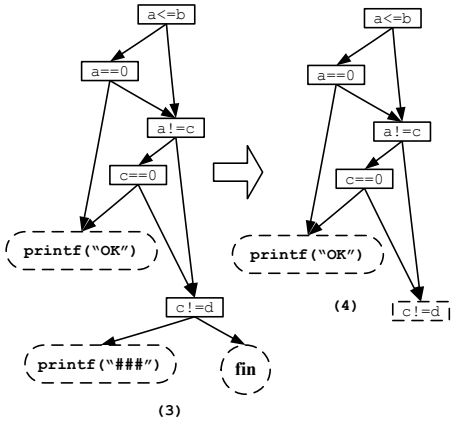

Figure 8. Contraction phase

Step 1: Let $N'=\{h'\}$;

Step 2: For any node n_i in G successive to a node in N', add it into N', provided that n_i is a branch judgment block of h', $n_i \notin N'$, $Pred(n_i) \subseteq N'$, and $|Pred(n_i)|==1$. Notice that such n_i will not introduce any new cycle;

Step 3: Repeat step 2, until N' is not changed any more. Compute C and E' according to **Def.9**;

Step 4: If $|C|>2$, $S=(N',E',h',C)$ is the solution; otherwise h' is not a root node of a cascade branch subgraph.

The pseudo code of this algorithm is showed in Fig.11.

7 Theoretic analysis

The method suggested by C. Cifuentes is based on heuristic subgraph matching and node binding. It can be proven by mathematical induction that all subgraphs which can be structured by this approach are strictly ***compound branch subgraphs***. Furthermore, C. Cifuentes's method can not detect all compound branch subgraphs.

The main idea of the proof is briefly described below: Observing the reverse operation of the approach by C. Cifuentes, begin from the last one node generated by the matching and binding process, and repeat splitting these bond node into their original component nodes reversely until the original CFG is recovered. At any time in the process of splitting, the subgraph has only one entry node and only two exit nodes, and all nodes in it are branch nodes and accord with the partial order **<<**, i.e. the subgraph is one compound branch subgraph. On the other side, this approach can not detect all compound branch subgraphs, such as the one shown in Fig.3, which could be recognized correctly by our algorithm.

Compared with the approach suggested by E. Moretti[4], compound branch subgraph imposes more accurate restrictions on the structure of conditionals. Therefore, our algorithm can recognize loop header conditionals shown in Fig.13D. In addition, it will never misrecognize the optimized adjacent conditionals as one conditional, as is shown in Fig.5.

We now discuss the complexity of our algorithms. 1) Given an expanded CFG $G=(N,E,h)$, a compound branch subgraph $T=(N',E',h',t,f)$, the biggest subgraph $T''=(N'',E'',h')$ generated at expansion phase, then the complexity of expansion phase is $O(N'')+O(E'')$, and the complexity of contraction phase is $O(N'')$, in sum, the total complexity to structure one compound branch subgraph is $O(N'')+O(E'')$; 2) Given an expanded CFG $G=(N,E,h)$, a cascade branch subgraph $S=(N',E',h',C)$, similar to the expansion phase above, the complexity to structure one cascade branch subgraph is $O(N')+O(E')$.

```
// Input:
//    h1: header node
//    For any Block b, b.in_comp is
//      False initially.
// Output:
//    N1: set of innder nodes
//    O: set of exit nodes
void trav_comp(List& N1, Set& O, Block h1)
{
    // Expansion phase
    N1=[h1];
    h1.in_comp=True
    O={Succ(h1)[0], Succ(h1)[1]};
    List pipe=Succ(h1);
    Dict out_ref={Succ(h1)[0]:1, Succ(h1)[1]:1};
    while(pipe in not null){
        Block b=pipe.pop(0);
        if(b is not a branch block || b.in_comp
           || Pred(b) is not a subset of N1)
            continue;
        N1.append(b);
        b.in_comp=True;
        pipe=pipe+Succ(b);
        O.erase(b);
        add_out(O, out_ref, Succ(b)[0]);
        add_out(O, out_ref, Succ(b)[1]);
    }
    // Contraction phase
    while(len(O)!=2){
        Block b=N1.pop();
        b.in_comp=False;
        O.add(b);
        erase_out(O, out_ref, Succ(b)[0]);
        erase_out(O, out_ref, Succ(b)[1]);
    }
}
```

Figure 9. Pseudo code of structuring compound branch subgraphs

```
void add_out(Set& O, Dict& out_ref, Block b)
{
    if(b in out_ref)
        out_ref[b]=out_ref[b]+1;
    else{
        out_ref.insert(b,1);
        O.add(b);
    }
}

void erase_out(Set& O, Dict& out_ref, Block b)
{
    if(out_ref[b]>1)
        out_ref[x]=out_ref[b]-1;
    else{
        out_ref.erase(b);
        O.erase(b);
    }
}
```

Figure 10. Assistant pseudo code functions

```
// Input:
//    h1: header node
//    For any Block b, b.in_cas is
//      False initially.
// Output:
//    N1: set of innder nodes
//    C: set of "case" exit nodes
// Return:
//    True: h1 is a valid header
//    False: h1 is not a valid header
bool trav_cas(List& N1, Set& C, Block h1)
{
    if(h1 is not a branch block)
       return False;
    N1=[h1];
    h1.in_cas=True;
    C={Succ(h1)[0], Succ(h1)[1]};
    List pipe=Succ(h1);
    while(pipe in not null){
        Block b=pipe.pop(0);
        if(b is not a branch block ||
           b.in_cas || |Pred(b)|>1 ||
           b is not a branch judgment
             block of h1)
            continue;
        N1.append(b);
        b.in_cas=True;
        pipe=pipe+Succ(b);
        C.erase(b);
        C=C+Succ(b);
    }
    return |C|>2;
}
```

Figure 11. Pseudo code of structuring cascade branch subgraphs

The approach by C. Cifuentes is a heuristic method, which need to traverse the CFG unpredictable times to match and to bind nodes, so the complexity is ***O(k*N)***. The approach by E. Moretti firstly needs to compute ***Interval/DSG*** which is not a trivial work, and the successive traversal scans the header nodes, branch nodes and following nodes while our algorithm only needs to scan header nodes, so the complexity is much higher than ***O(N″)+O(E″)***. In summary, the efficiency of our algorithm is notably improved than classic algorithms.

8 Experimental results

In what follows, we analyze binary executables from different operating systems using algorithms demonstrated above.

The selected instances include: 1). System binary executables of Windows XP, including **kernel32.dll**, **user32.dll**, **explorer.exe**; 2) Well-known applications on Linux, including **samba 3.0.23d**、**sendmail 8.13.8**、**vsftpd 2.0.5**, which are compiled by "**gcc –O2**".

Table 1 shows the statistics of compound branch subgraphs of these instances, e.g. there are 7991 1-inner-node compound branch subgraphs and 1272 2-inner-node compound branch subgraphs in **kernel32.dll**.

Fig.12A, 12B, 12C, 12D show the typical structures of compound branch subgraphs in these instances recognized by our algorithms. C. Cifuentes' approach can't recognize structures showed in Fig.12B and 12C, and E. Moretti's approach can not recognize the structure showed in Fig.12D. We believe that our algorithms provide a more accurate and efficient way to structure 2-way branches in binary executables.

Table 2 shows the statistics of cascade branch subgraphs of these instances. Fig.13A, 13B show the typical structures of cascade branch subgraphs in these instances recognized by our algorithms. We randomly chose 50 samples from the results, checked them, and have not found any false recognition; these samples are all valid **switch** statements. Such result demonstrates that our method provides a satisfying solution to this problem.

inner nodes \ instance (num of subgraphs)	kernel32	user32	explorer	samba 3.0.23d	sendmail 8.13.8	vsftpd 2.0.5
1	7991	7374	3354	30601	7778	930
2	1272	1195	556	5412	1202	147
3	290	337	156	1485	309	42
4	79	105	47	1743	105	16
5	41	46	25	189	32	1
6	34	27	9	122	22	3
7	11	7	1	61	7	–
8	7	3	4	40	2	–
9	4	4	1	16	3	–
…	…	…	…	…	…	…

Table 1. Statistics of compound branch subgraph size

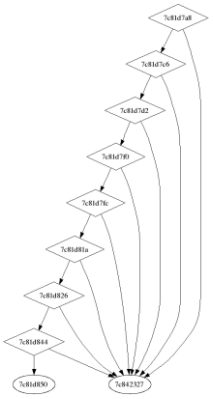

Figure 12A. Sequence

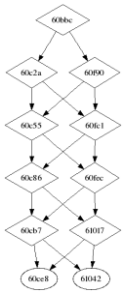

Figure 12B. Cross

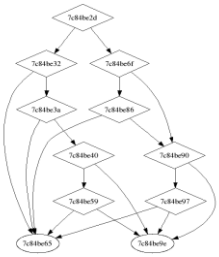

Figure 12C. Hybrid

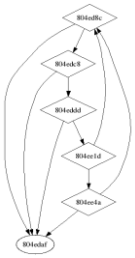

Figure 12D. Loop header

inner nodes \ num of subgraphs \ instance	kernel32	user32	explorer	samba 3.0.23d	sendmail 8.13.8	vsftpd 2.0.5
2	272	167	54	468	188	34
3	93	67	21	59	37	3
4	25	25	21	57	31	2
5	33	18	9	32	7	–
6	12	5	4	5	5	1
7	12	12	3	2	1	–
8	4	4	–	3	1	–
9	3	13	1	4	–	–
…	…	…	…	…	…	…

Table 2. Statistics of cascade branch subgraph size

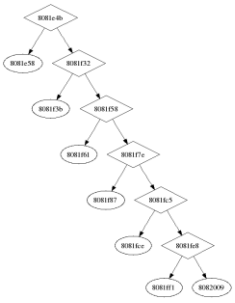

Figure 13A. Sequence

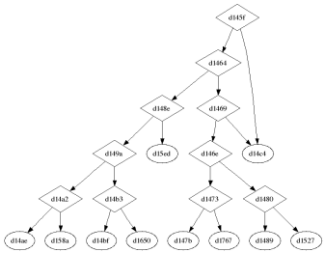

Figure 13B. Binary

9 Conclusions

This paper proposes a method for structuring 2-way branches in binary executables to conditionals, loop conditionals and switches.

Our contribution is based on strict definitions based on graph theory, in particular, definitions of ***compound branch subgraph*** and ***cascade branch subgraph*.**

In this work, a novel 2-phase algorithm is provided for structuring 2-way branches into conditionals and loop conditionals based on compound branch subgraph. The approach can correctly recognize normal, crossover and loop header structures, and it would not recognize adjacent optimized conditionals as one conditional. Most classical works can not achieve all these goals. The complexity of this approach is notably reduced compared with previous approaches.

Another innovative algorithm is given to structure 2-way branches into switches based on cascade branch subgraph. As far as we know, the algorithm is the first publicized one. It is concise and fast, and no false-identification has been found in its results.

We have applied these algorithms to typical binary executables on Windows XP and Linux. The statistics of 2-way branch structures in these instances are given. Experiment results validate our theoretical analysis. It is shown that the methods can handle every situation correctly, and they are more accurate than typical current approaches.

References

[1] http://www.program-transformation.org/Transform/HistoryOfDecompilation1.

[2] C. Cifuentes. Reverse compilation techniques. PhD Thesis, Queensland University of Technology, July 1994.

[3] C. Cifuentes. Structuring decompiled graphs. *International Conference on Compiler Construction*, *Lecture Notes in Computer Science 1060*, pages 91-105, 24-26 April 1996.

[4] Eric Moretti, Gilles Chanteperdrix, Angel Osorio, "New Algorithms for Control-Flow Graph Structuring," csmr, p. 184, *Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[5] F.E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7)1:19, July 1970.

[6] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20-25, July 1970.

[7] K. Kaspersky, *Hacker Disassembling Uncovered*, A-List LLC, P384, 2004.

[8] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Elsevier Science (USA), 2005.