A New Algorithm for Identifying Loops in Decompilation*

Tao Wei, Jian Mao, Wei Zou**, and Yu Chen

Institute of Computer Science and Technology
Peking University
{weitao,maojian,zouwei,chenyu}@icst.pku.edu.cn

Abstract. Loop identification is an essential step of control flow analysis in decompilation. The Classical algorithm for identifying loops is Tarjan's interval-finding algorithm, which is restricted to reducible graphs. Havlak presents one extension of Tarjan's algorithm to deal with irreducible graphs, which constructs a loop-nesting forest for an arbitrary flow graph. There's evidence showing that the running time of this algorithm is quadratic in the worst-case, and not almost linear as claimed. Ramalingam presents an improved algorithm with low time complexity on arbitrary graphs, but it performs not quite well on "real" control flow graphs (CFG). We present a novel algorithm for identifying loops in arbitrary CFGs. Based on a more detailed exploration on properties of loops and depth-first search (DFS), this algorithm traverses a CFG only once based on DFS and collects all information needed on the fly. It runs in approximately linear time and does not use any complicated data structures such as Interval/Derived Sequence of Graphs (DSG) or UNION-FIND sets. To perform complexity analysis of the algorithm, we introduce a new concept called *unstructuredness coefficient* to describe the unstructuredness of CFGs, and we find that the unstructuredness coefficients of these executables are usually small (<1.5). Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation why those algorithms existed perform not quite well on real-world cases. The new algorithm has been applied to 11526 CFGs in 6 typical binary executables on both Linux and Window platforms. Experimental result has validated our theoretical analysis and it shows that our algorithm runs 2-5 times faster than the Havlak-Tarjan algorithm, and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm.

Keywords: Control flow analysis, Decompilation, Loop identifying, Unstructuredness coefficient.

1 Introduction

Decompilation is a key technique for static analysis in the field of reverse engineering. Decompilation was initially introduced for porting programs across

---

* Supported by The National High Technology Research and Development Program of China (No. 2006AA01Z402).

** Corresponding author.

platforms. It then had been widely used in areas such as software maintenance, re-engineering and comprehension of legacy systems. Since the 1990s, demand on decompilation from software security analysis community has been growing rapidly due to outbreaks of security vulnerabilities and malicious codes[1].

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

Loops are control structures used for repeating instructions. In control flow graphs (CFG) [2], loops are often nested within other loops. Such phenomenon induces a structure called "loop-nesting forest" [3][4]. Furthermore, although structured programming is well adopted by modern programmers, irreducible loops (loops with multientry) [5][6] still widely exist in executable codes due to optimizations performed by compilers. Identifying loops, in particular nested and irreducible ones, is a major challenge for the task of decompilation.

In 1970 F.E. Allen and J. Cocke pioneered the work on identifying loops by introducing the concept of *reducibility*[5][6] for control flow graphs. Since then both compiler and decompiler research communities have been investigating this problem. Influential pieces of work include those done by R.E. Tarjan [8], P. Havlak [7] and G. Ramalingam[3]. However, loop identification schemes proposed in these work are often based on multi-pass traversals and complicated data structures, such as *Interval/Derived sequence of graphs (DSG)*[5][6] and *UNION-FIND sets*[9]; these data structures often require complex operations, and such operations slow down the loop identification schemes [10].

This paper presents an innovative algorithm for identifying loops in binary executables. We explore some useful properties of loops and depth-first search (DFS) which make DFS collecting more information than simple forward/cross/backward edge information. Based on these properties, we give an algorithm which uses a one-pass DFS traversal to solve all loop problems. This algorithm does not use any complicated data structures, so it is simple and easy to implement.

We have applied our algorithm and other classic algorithms to 11526 CFGs in 6 typical binary executables on Windows XP and Linux. The experiments show that our algorithm runs 2-5 times faster than the Havlak-Tarjan algorithm [7], and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm [3].

Furthermore, as an interesting byproduct of complexity analysis of this algorithm, we introduce a new concept called ***unstructuredness coefficient***. This coefficient could describe the unstructuredness of CFG.

The statistics of experiments shows that while most real-world binary executables have irreducible CFGs, unstructuredness coefficients of CFGs are usually smaller than 1.5 and hardly correlated to the size of CFG.

Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation of why those algorithms with low time complexity for arbitrary graphs perform not quite well on "real" CFGs, esp. the Ramalingam-Havlak-Tarjan algorithm.

Besides decompilation, our algorithm could be used in many applications, such as computing the iterated dominance frontier for the SSA form and Sparse Evaluation

Graphs, constructing the dominator tree[4], and sequentializing program dependence graphs for code generation[12].

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 provides terminology and notations of identifying loops. Section 4 presents the algorithm for identifying loops. Section 5 introduces the concept of unstructuredness coefficient, and presents the complexity analysis of our scheme. Section 6 reports our experimental result and finding, in particular, the statistics of unstructured coefficient in real-world binary executables. Section 7 concludes this paper.

2 Related Work

Identifying loops is a well-built problem in control-flow analysis area. Loops in CFG have more attributes than simple cycles, such as nesting, multi-entry and irreducibility. Hence identifying loops in CFG is generally more challenging than detecting cycles. Research on identifying loops has a long history, starting from 1970 when F.E. Allen and J. Cocke introduced the concept of reducibility[5][6] for control flow graphs. Since then many researchers in both compiler and decompiler fields have studied this problem extensively.

Reducibility is an important property of CFG on its structuredness. In 1972 M.S.Hecht and J.D.Ullman showed that all and only the irreducible CFGs have a multi-entry-loop subgraph [11], as shown in Fig.1.

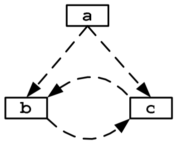

Fig. 1. The irreducible core

In CFG, loops are often nested within other loops according to their headers' positions. Such phenomenon induces a structure called "loop-nesting forest" [3][4]. Fig.2(A) shows a CFG with nested loops, and Fig.2(B) shows the corresponding loop-nesting forest.

There are various definitions of loop and loop-nesting forest. While there is a well-accepted one by Tarjan [8] of loops in a reducible graph, there is no consensus on how the loop nesting forest should be defined for CFGs with nested loops. B. Steensgaard [12], V.C. Sreedhar *et al* [13], Havlak [7] and Ramalingam [4] each provided a different definition.

Consider the CFG shown in Fig.3(A): The Sreedhar–Gao–Lee algorithm [13] and the Ramalingam algorithm [4] both identify a single loop ***{a,b,c,d}***; the Steensgaard algorithm [12] identifies two loops ***{a,b,c,d}*** and ***{b,c}***; the Havlak algorithm [7] identifies three loops ***{a,b,c,d}***, ***{b,c,d}*** and ***{c,d}***, as shown in Fig.3 (B).

In the context of decompilation, only the definition given by Havlak meets the requirements for rebuilding high level structures using single-entry loops and minimum ***goto*** (for re-entry edges) statements. Hence in this paper we adopt his definition.

Under Havlak's definition, the classical algorithm for identifying loops is Tarjan's interval-finding algorithm [8] proposed in 1974, which is restricted to reducible graphs. In 1997 Havlak presented an extension [7] to Tarjan's algorithm, which can handle arbitrary flow graphs.

The Havlak-Tarjan algorithm traverses a CFG twice: first a top-down traversal based on depth-first search, which collects information of forward edges, cross edges and back edges; then a bottom-up traverse based on the UNION-FIND operation, which propagates loop header information backward from loop tails.

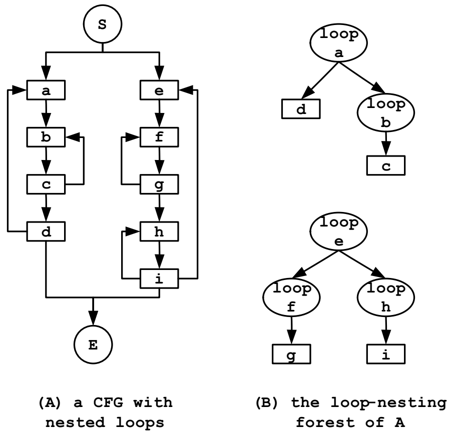

Fig. 2. A CFG with nested loops and its loop-nesting forest: loop ***b*** is nested in loop ***a***; loop ***f*** and loop ***h*** are nested in loop ***e***

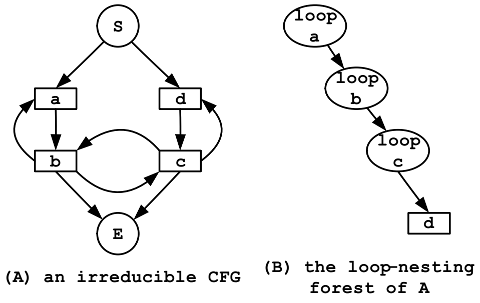

Fig. 3. A classic irreducible CFG and its loop-nesting forest based on the definition by Havlak

In 1999 Ramalingam showed that the running time of the Havlak algorithm is quadratic when the target CFG's multientry unstructuredness is very high; he then modified the algorithm to make it run in almost linear time [3]. However, the Ramalingam-Havlak-Tarjan algorithm needs extra procedures to solve least common ancestors and to mark irreducible loops, and these procedures need UNION-FIND operations.

On the other hand, the Steensgaard algorithm runs in quadratic time, and the Sreedhar-Gao-Lee algorithm runs in almost linear time, but the latter requires the dominator tree being built in advance.

In 2001, K.D. Cooper showed that, based on empirical evidence complex operations required by UNION-FIND slow down programs in practice; furthermore, simple algorithms with discouraging asymptotic complexities might be faster in handling real-world cases than those running in almost linear time, but containing complex operations [10].

3 Preliminaries

This section briefly describes some basic concepts in control flow analysis and definitions about loops, and introduces some important properties of loop and DFS.

3.1 Concepts in Control Flow Analysis

We present brief descriptions of concepts in control flow analysis as following. The detailed version can be found in [2] and [9].

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 single-root, 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.

A path from a node u to a node u' in a graph $G=(N,E,h)$ is a sequence $<v_0,v_1,v_2,...,v_k>$ of nodes such that $u=v_0$, $u'=v_k$, and $<v_{i-1},v_i> \in E$ for $i=1,2,...,k$.

A ***depth-first search (DFS)*** of a CFG $G=(N,E,h)$ visits all the nodes, marking them after they have been visited. The next node visited is an unmarked successor of the most recently visited node with such a successor. The time complexity of DFS is $O(N+E)$.

If a DFS traversal begins with h, and all other nodes are reachable, the edges followed define a ***depth-first spanning tree (DFST)*** of G.

$DFSP(N)$, the ***depth-first search path*** of node N, is the path from h to N in the DFST of G. Given a node N and a node M, and N is in $DFSP(M)$, then $DFSP(N,M)$ is the part of $DFSP(M)$ from N to M.

Besides creating a depth-first spanning tree, depth-first search also timestamps each node. Each node v has two timestamps: the ***first timestamp*** $d[v]$ records when v is first discovered, and the ***second timestamp*** $f[v]$ records when the search finishes examining v's adjacency list. ***Parenthesis theorem*** is an important property of DFS, and here is a short description: for two node u and v, the two sets $[d[u],f[u]]$ and $[d[v],f[v]]$ are either disjoint or nested.

In this paper, we use three edge types in terms of the DFST G_T produced by a DFS on G:

- ***Back edges*** are those edges $<u,v>$ connecting a vertex u to an ancestor v in G_T. Self-loops are considered to be back edges. An edge $<u,v>$ is a back edge if and only if $d[v] \leq d[u] < f[u] \leq f[v]$.
- ***Forward edges*** are those edges $<u,v>$ connecting a vertex u to a descendant v in G_T. A edge $<u,v>$ is a forward edge if and only if $d[u] < d[v] < f[v] < f[u]$.
- ***Cross edges*** are all other edges.
  A node u is in $DFSP(v)$ if and only if $d[u] \leq d[v] < f[v] \leq f[u]$.

Given a CFG $G=(N,E,h)$, a ***strongly connected region (SCR)*** is a nonempty set of nodes $S \subseteq N$, for which, given any $q,r \in S$, there exists a path from q to r and from r to q. A SCR is a ***maximal SCR*** if none of its proper supersets is a SCR.

3.2 Definitions About Loops

We present brief descriptions of definitions about loops. The detailed version can be found in [7].

Loops include ***outermost loops*** and ***inner loops***.

An ***outermost loop*** is a maximal SCR with at least one internal edge.

In any particular depth-first search, the first node of a loop L to be traversed is defined to be the ***header*** of the loop, i.e. for the header h, $d[h]$ is the minimum in all the nodes in L. The set of other nodes is defined to be the ***loop body***.

An ***inner loop*** nested inside a loop L with header h is an outermost loop with respect to the subgraph with node set $(L-\{h\})$.

Loop-nesting forest is a data structure that represents the containment relation between loops in a control flow graph.[4]

Given a loop L with header h and an edge $<q,r>$, $q \notin L$, $r \in L-\{h\}$, then r is called a ***re-entry*** of this loop, and $<q,r>$ is called a ***re-entry edge***.

For a node n in a loop body, its ***innermost loop*** is the smallest loop containing n., and the header of this loop is called n's ***innermost loop header***. The ***loop header list*** of n consists of its innermost loop header h_1, h_1's innermost loop header h_2, h_2's innermost loop header h_3, and so on. The loop header list of d in Fig.3 is $[c,b,a]$.

3.3 Properties of Loop and DFS

We discover that there are some interesting properties of DFS and loops defined in section 3.2, which are helpful in identifying loops.

Ancestor Property: For any node n, all of its loop headers must be in $DFSP(n)$.

Nesting Property: Two different loop headers x,y of node n must be nested, i.e. either x is a loop header of y, or y is a loop header of x.

Direct Transitive Property: Given that node m is a child of node n in the $DFST$, a loop header x of m is also a loop header of n if and only if $x \neq m$.

Indirect Transitive Property: Given that node m is a successor of node n and $m \notin DFSP(n)$, a loop header x of m is also a loop header of n if and only if $x \in DFSP(n)$.

All these properties can be proved, and here are some lemmas used during the proof:

Lemma 1: Given an edge $<n,m>$, if $d[n]<d[m]$, then $f[n]>f[m]$.

Lemma 2: Given an edge $<n,m>$, if it is not a back edge, then $f[n]>f[m]$.

Lemma 3: Given a re-entry $<n,m>$ of loop L, if the header of L is x, then $f[n]>f[x]>f[m]$.

4 Algorithm for Identifying Loops

Statement of the problem: Given a CFG $G=(N,E,h_0)$, for each node $n \in N$:

(1) Decide whether or not n is a loop header;
(2) Decide whether or not n is in a loop body; If yes, which node is its innermost loop header?
(3) Decide whether or not n is a re-entry; If yes, which edges are the re-entry edges?

Based on these properties given in section 3.3, we propose a new algorithm which contains two parts: traversing a CFG based on depth-first search, and tagging loop headers on demand.

In contrast with the multi-pass algorithms proposed by Tarjan, Havlak and Ramalingam, our algorithm collects and propagates loop header information during depth-first search based on these properties, so it doesn't need the second bottom-up traversal based on UNION-FIND operations to do the same thing.

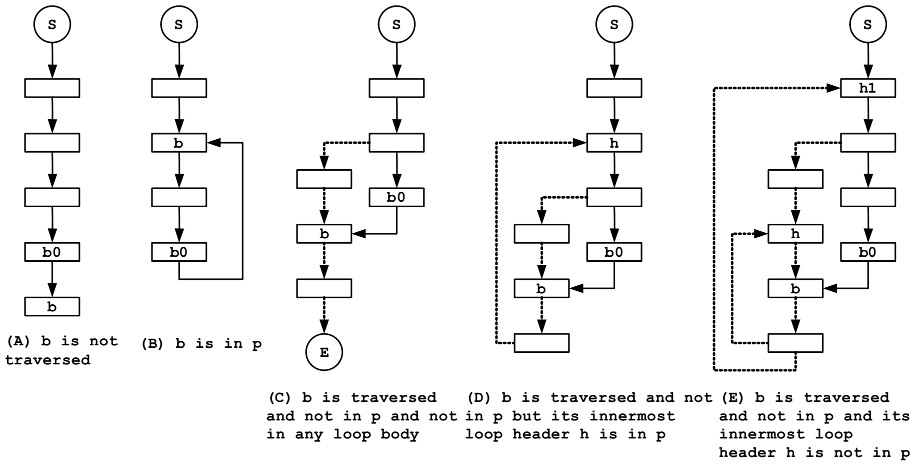
S
b0
b
(A) b is not traversed
b
b0
(B) b is in p
b0
b
E
(C) b is traversed and not in p and not in any loop body
h
b0
b
(D) b is traversed and not in p but its innermost loop header h is in p
h1
h
b0
b
(E) b is traversed and not in p and its innermost loop header h is not in p

Fig. 4. Cases during traversing

Traversing: The algorithm visits all the nodes in ***N***, starting from h_0 recursively in depth-first search order. When a node b_0 is visited, it is marked as ***traversed*** and let ***p*** be the current path from h_0 to b_0 in the depth-first spanning tree (i.e. ***DFSP(b_0)***). Each successor ***b*** of b_0 is checked in turn as followed:

(A) If ***b*** is a new node, i.e. ***b*** is not ***traversed*** yet, as shown in Fig.4(A): traverse it recursively; if it is found in a loop body after being traversed, tag ***b***'s innermost loop header as a loop header of b_0 if the header node is in ***p***;

(B) If ***b*** is ***traversed*** already, and it is in ***p***, as shown in Fig.4(B): mark ***b*** as a loop header, and tag ***b*** as a loop header of b_0;

(C) If ***b*** is ***traversed*** already, and it is not in ***p***, or any loop body, as shown in Fig.4(C): just skip it;

(D) If ***b*** is ***traversed*** already, and it is not in ***p***, but it is in a loop body whose innermost loop header ***h*** is in ***p***, as shown in Fig.4(D): tag ***h*** as a loop header of b_0;

(E) If ***b*** is ***traversed*** already, and it is not in ***p***, but it is in a loop body whose innermost loop header is not in ***p***, as shown in Fig.4(E): mark ***b*** as a re-entry node, and mark $\langle b_0, b\rangle$ as a re-entry edge. Find the innermost loop header h_1 of ***b*** in ***p*** if it exists, then tag h_1 as a loop header of b_0.

The pseudo code of traversing is shown as following:

```
procedure identify_loops(CFG G=(N,E,h0)):
   foreach(Block b in N): // init
      initialize(b); // zeroize flags & properties
   trav_loops_DFS(h0,1);

function trav_loops_DFS(Block b0, int DFSP_pos):
//return: innermost loop header of b0
   Mark b0 as traversed;
   b0.DFSP_pos := DFSP_pos;//Mark b0's position in DFSP
   foreach(Block b in Succ(b0)):
      if(b is not traversed):
         // case(A), new
         Block nh := trav_loops_DFS(b, DFSP_pos+1);
         tag_lhead(b0, nh);
      else:
         if(b.DFSP_pos > 0): // b in DFSP(b0)
            // case(B)
            Mark b as a loop header;
            tag_lhead(b0, b);
         else if(b.iloop_header == nil):
            // case(C), do nothing
         else:
            Block h := b.iloop_header;
            if(h.DFSP_pos > 0): // h in DFSP(b0)
               // case(D)
               tag_lhead(b0, h);
            else: // h not in DFSP(b0)
```

```
                // case(E), reentry
                Mark b and (b0,b) as re-entry;
                Mark the loop of h as irreducible;
                while(h.iloop_header!=nil):
                   h := h.iloop_header;
                   if(h.DFSP_pos > 0): // h in DFSP(b0)
                      tag_lhead(b0, h);
                      break;
                   Mark the loop of h as irreducible;
    b0.DFSP_pos := 0; // clear b0's DFSP position
    return b0.iloop_header;
```

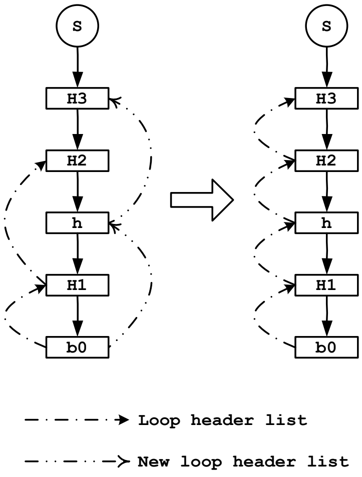

Fig. 5. Tagging loop headers

Tagging: When tagging ***h*** as a loop header of ***b_0***, weave ***h*** and its loop header list (if exists) into the current loop header list of ***b_0*** according to their positions in ***p***, as shown in Fig.5. The pseudo code of tagging loop headers is shown as following.

```
procedure tag_lhead(Block b, Block h):
   if(b == h or h == nil) return;
   Block cur1 := b, cur2 := h;
   while(cur1.iloop_header!=nil):
      Block ih := cur1.iloop_header;
      if(ih == cur2) return;
      if(ih.DFSP_pos < cur2.DFSP_pos):
         cur1.iloop_header := cur2;
         cur1 := cur2;
         cur2 := ih;
      else:
         cur1 := ih;
   cur1.iloop_header := cur2;
```

Based on properties given in section 3.3, it can be proved that the loop header of every node will be correctly tagged after this one-pass DFS traversing.

5 Complexity Analysis and Unstructuredness Coefficient

We now discuss the complexity of the algorithm given in section 4.

Given a CFG $G=(N,E,h_0)$, **trav_loops_DFS** is called recursively for each node $n \in N$ exactly one time, so it is called N times totally.

In one invocation of **trav_loops_DFS**, the ***foreach*** loop is executed for each out-edge of the current node. It follows that in all invocations of **trav_loops_DFS** for all nodes in G, the ***foreach*** loop is executed exactly one time for each edge in E, so it is executed E times totally.

In the i*-th* execution of the ***foreach*** loop ($i \in [1,E]$), only one of case (A), (B), (C), (D) and (E) can be chosen. Let x_i be the execution times of the ***while*** loop in **trav_loops_DFS**, and let y_i be the execution times of the ***while*** loop in **tag_lhead**. The complexity of case (A) is $O(1+y_i)$, except for the recursive call to **trav_loops_DFS** which is counted in the above already; the complexity of case (B) and (D) is $O(1+y_i)$; the complexity of case (C) is $O(1)$; the complexity of case (E) is $O(1+x_i+y_i)$. Notice that in case (A), (B), (C) and (D), $x_i=0$, and in case (C) $y_i=0$ too.

In summary, the total complexity of the algorithm is $O(N+E+\Sigma x_i+\Sigma y_i)$, $i \in [1,E]$. Let $k=1+(\Sigma x_i+\Sigma y_i)/E$, it follows that the total complexity can be expressed as $O(N+k*E)$.

In the following, we discuss the meaning of x_i, y_i and k behind these mathematical expressions.

As shown in Fig.6(A), the ***while*** loop in **trav_loops_DFS** is executed because the edge $<b_0,b>$ skips multi level loop headers and jumps directly into the (x_i+1)*th* inner loop. This situation is called multientry unstructuredness[14].

As shown in Fig.6(B), the ***while*** loop in **tag_lhead** is executed mainly because the back edges of loops overlap with each other, and y_i is the rough measurement of overlapping levels. This situation is called overlapping unstructuredness[14].

Multientry unstructuredness is irreducible, whereas overlapping unstructuredness is reducible. Multientry unstructuredness is caused by forward edges while overlapping unstructuredness is caused by backward edges. They both contribute to the total unstructuredness of a CFG.

$k=1+(\Sigma x_i+\Sigma y_i)/E$, it describes the ratio of the total unstructuredness, including both multientry unstructuredness and overlapping unstructuredness, to the size of a CFG. Hence we call k the ***unstructuredness coefficient***.

Please notice that the ***unstructuredness coefficient*** k is usually small: In today's binary executables, unstructuredness is introduced mostly by optimization compilers, not by programmers instead. The main reason is that structured programming has been well adopted already. In addition, unstructured code is hard to maintain correctness, even introduced by compilers. Therefore, although unstructuredness can be found in almost every binary code, the majority of binary code is well structured. Experiments in the next section validate such analysis.

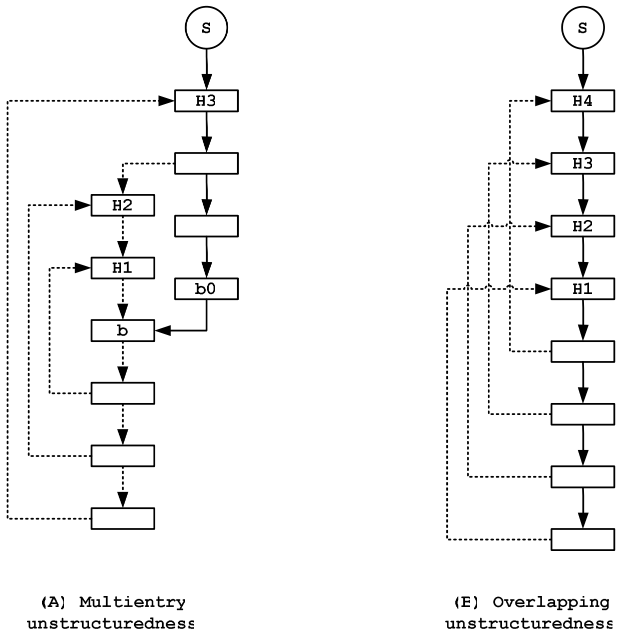
S
H3
H2
H1
b0
b
(A) Multientry unstructuredness
S
H4
H3
H2
H1
(E) Overlapping unstructuredness

Fig. 6. Unstructuredness of loops

6 Experimental Results

We analyze binary executables on different operating systems using ***BESTAR*** (Binary Executable Structurizer and Analyzer), our in-house decompiler which has implemented the algorithms described above, including our algorithm, the Havlak-Tarjan algorithm and the Ramalingam-Havlak-Tarjan algorithm.

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

Table.1 shows the statistics about loops in these instances identified by algorithms. There are totally 11526 CFGs in these instances. In these CFGs, there are 174 irreducible CFGs and 7841 loops. The experimental results of all these algorithms are the same, which validate the correctness of our algorithm and its implementation.

A phenomenon we have discovered from these results is that all these instances contain irreducible CFGs.

Another important phenomenon is that k is small in all these instances. Table.2 and Fig.7 show statistics of k with respect to the number of nodes of CFGs in these instances. The statistics show that in these real-world instances the unstructuredness coefficient is usually smaller than 1.5 and its average value is hardly correlated to the size of CFG. We call this phenomenon **"low-unstructuredness"** property of CFGs, and this property distinguishes real-world CFGs from general single-root connected directed graphs.

Table 1. Statistics of loops

	kernel32	user32	explorer	samba 3.0.23d	sendmail 8.13.8	vsftpd 2.0.5
CFGs	1488	1711	1380	5946	642	359
irreducible CFGs	5	8	2	81	71	7
loop headers	1134	636	354	4112	1440	165
Avg(k)	1.01	1.01	1.01	1.01	1.05	1.01
Max(k)	1.35	1.29	1.31	1.40	1.41	1.31
Min(k)	1.00	1.00	1.00	1.00	1.00	1.00

Table 2. Statistics of k to N

N	Max(k)	Avg(k)	Min(k)
0-49	1.33	1.01	1
50-99	1.34	1.03	1
100-149	1.41	1.05	1
150-199	1.28	1.06	1
200-249	1.3	1.06	1
250-299	1.25	1.05	1
300-349	1.13	1.05	1
350-399	1.09	1.03	1

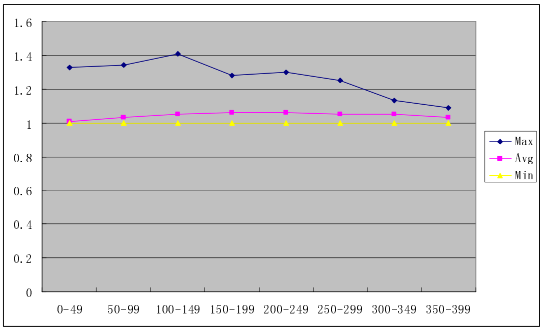

Fig. 7. Statistics of k to N

For performance comparison, we ran all these algorithms on an unloaded 2.6GHz AMD Opteron Server, with each implementation properly optimized. Table.3 and Fig.8 show the time spent during processing these instances. The results show that our algorithm is 2-5 times faster than the Havlak-Tarjan algorithm, and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm.

Table 3. Time of algorithms(in μsec, lower is better)

	kernel32	user32	explorer	samba 3.0.23d	sendmail 8.13.8	vsftpd 2.0.5
Our	0.22	6.6	0.14	0.36	1.1	0.2
Havlak	0.44	28	0.28	1.6	5.3	1.1
Ramalingam	0.51	54	0.32	2.3	9.6	1.4

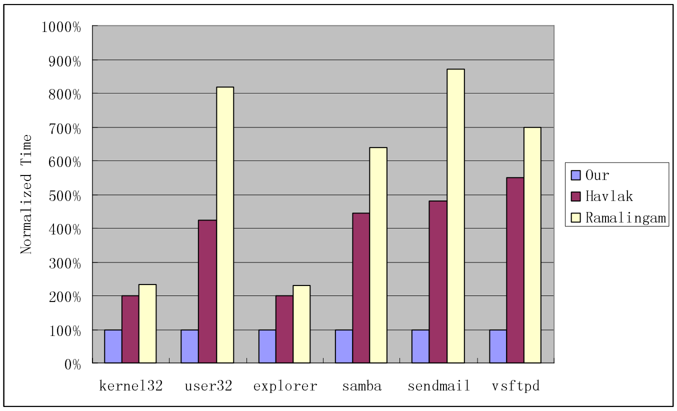

Fig. 8. Time comparison of algorithms (lower is better)

Here is another interesting result: although the Ramalingam-Havlak-Tarjan algorithm is an "improved" version of the Havlak-Tarjan algorithm, its performance is even worse than the latter. Based on the "low-unstructuredness" property of CFGs, this phenomenon can be easily explained: in order to reach almost linear time complexity when the target CFG's multientry unstructuredness is very high, Ramalingam adds extra procedures to solve least common ancestors and to mark irreducible loops, and both need UNION-FIND operations; however, for real-world CFGs, the unstructuredness is low, so the extra procedures contribute little to the performance and slow down the whole process instead.

7 Conclusion

This paper presents an innovative method for identifying loops in binary executables. First, we explore some useful properties of loops and DFS which make DFS collecting more information than simple forward/cross/backward edge information. Then, we propose the algorithm building on a one-pass DFS traversal and these properties. It does not use any complicated data structures such as Interval/DSG or UNION-FIND sets, so it is simpler and easier to implement than classical multi-pass traversal algorithms.

The complexity of our method is $O(N+k*E)$, where k is the ***unstructuredness coefficient***, a new concept proposed in this paper to describe the unstructuredness of CFGs.

The ***unstructuredness coefficient*** k is usually small, because structured programming has been well adopted, and unstructured code is hard to maintain its correctness, even introduced by compilers. Hence although unstructuredness can be found in almost every binary code, the majority of binary code is well structured. In fact, we found that in real-world binaries the average value of k is usually smaller than 1.5 and hardly correlated to the size of CFGs. Such "low-unstructuredness" property distinguishes these CFGs from general single-root connected directed graphs, and it offers an explanation of why those algorithms with low time complexity on arbitrary graphs perform not quite well on "real" CFGs.

Using ***BESTAR*** (Binary Executable Structurizer and Analyzer), our in-house decompiler, we have applied the algorithm and classical algorithms to 11526 CFGs in 6 typical binary executables on Windows XP and Linux. Due to the simplicity of our algorithm and the "low-unstructuredness" property of real-world binaries, our algorithm is 2-5 times faster than the Havlak-Tarjan algorithm[7], and 2-8 times faster than the Ramalingam-Havlak-Tarjan algorithm[3].

Due to its remarkable performance, our algorithm could also be used in other applications, besides general decompilation, such as computing the SSA form or sequentializing program dependence graphs during just-in-time compilation.

References

[1] http://www.program-transformation.org/Transform/HistoryOfDecompilation1
[2] Muchnick, S.S.: Advanced Compiler Design and Implementation. Elsevier Science, Amsterdam (1997)
[3] Ramalingam, G.: Identifying loops in almost linear time. ACM Transactions on Programming Languages and Systems 21(2) (1999)
[4] Ramalingam, G.: On loops, dominators, and dominance frontiers, ACM Transactions on Programming Languages and Systems 24(5) (2002)
[5] Allen, F.E.: Control flow analysis. SIGPLAN Notices 5(7), 1–19 (1970)
[6] Cocke, J.: Global common subexpression elimination. SIGPLAN Notices 5(7), 20–25 (1970)
[7] Havlak, P.: Nesting of reducible and irreducible loops. ACM Transactions on Programming Languages and Systems 19(4) (1997)
[8] Tarjan, R.E.: Testing flow graph reducibility, J. Comput. Syst. Sci. 9 (1974)
[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2001)
[10] Cooper, K.D., Harvey, T.J., Kennedy, K.: A Simple Fast Dominance Algorithm, Software Practice and Experience (2001)
[11] Hecht, M.S., Ullman, J.D.: Flow graph reducibility. SIAM Journal of Computing 1(2), 188–202 (1972)
[12] Steensgaard, B.: Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Wash (1993)
[13] Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using DJ graphs. ACM Transactions on Programming Languages and Systems 18(6) (1996)
[14] Cifuentes, C.: Reverse compilation techniques. PhD Thesis, Queensland University of Technology (1994)