Journal of Software Engineering and Applications, 2016, 9, 80-90
Published Online March 2016 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2016.93007

Subgraph Isomorphism Based Intrinsic Function Reduction in Decompilation

Yanzhao Liu, Yinliang Zhao, Lei Zhang, Kai Liu

Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
Email: eqqlyz@163.com, zhaoy@mail.xjtu.edu.cn, zhangleivic@gmail.com, xjtulk@163.com

Received 29 December 2015; accepted 20 March 2016; published 23 March 2016

Abstract

Program comprehension is one of the most important applications in decompilation. The more abstract the decompilation result the better it is understood. Intrinsic function is introduced by a compiler to reduce the overhead of a function call and is inlined in the code where it is called. When analyzing the decompiled code with lots of inlined intrinsic functions, reverse engineers may be confused by these detailed and repeated operations and lose the goal. In this paper, we propose a method based graph isomorphism to detect intrinsic function on the CFG (Control Flow Graph) of the target function first. Then we identify the boundary of the intrinsic function, determine the parameter and return value and reduce the intrinsic function to a single function call in the disassembled program. Experimental results show that our method is more efficient at reducing intrinsic functions than the state-of-art decompilers such as Hex-Rays, REC and RD (Retargetable Decompiler).

Keywords

Program Comprehension, Decompilation, Graph Isomorphism, Intrinsic Function

1. Introduction

Intrinsic functions are inlined to reduce the overhead of function call and speed up the whole program. Most of the prevalent decompilers [1]-[4] do not deal with these intrinsic functions and treat their corresponding instructions as normal operations. Although Hex-Rays [5] has detected and abstracted some intrinsic functions using F.L.I.R.T [6] technology, the effect is still not satisfactory. In **Figure 1**, this is an example that Hex-Rays fails to abstract the *memcmp* intrinsic function.

In **Figure 1(a)**, the source code which invokes *memcmp* function is listed. Disassembled code of *memcmp* is listed in **Figure 1(b)** and the decompilation result of Hex-Rays is listed in **Figure 1(c)**. The disassembled

```
while( nNeedle<=nHaystack
      && memcmp(zHaystack, zNeedle, nNeedle)!=0 ){
  N++;
  do{
    nHaystack--;
    zHaystack++;
  }while( isText && (zHaystack[0]&0xc0)==0x80 );
}
```

(a)

```
.text:00451AD0 mov    ecx, eax              .text:00451AFC cmp    ecx, 0FFFFFFFDh
.text:00451AD2 mov    edx, esi              .text:00451AFF jz     short loc_451B48
.text:00451AD4 sub    ecx, 4                .text:00451B01 mov    al, [edx+1]
.text:00451AD7 jb     short loc_451AF1      .text:00451B04 cmp    al, [edi+1]
.text:00451AD9 lea    esp, [esp+0]          .text:00451B07 jnz    short loc_451B23
.text:00451AE0 loc_451AE0:                  .text:00451B09 cmp    ecx, 0FFFFFFFEh
.text:00451AE0 mov    eax, [edx]            .text:00451B0C jz     short loc_451B48
.text:00451AE2 cmp    eax, [edi]            .text:00451B0E mov    al, [edx+2]
.text:00451AE4 jnz    short loc_451AF6      .text:00451B11 cmp    al, [edi+2]
.text:00451AE6 add    edx, 4                .text:00451B14 jnz    short loc_451B23
.text:00451AE9 add    edi, 4                .text:00451B16 cmp    ecx, 0FFFFFFFFh
.text:00451AEC sub    ecx, 4                .text:00451B19 jz     short loc_451B48
.text:00451AEF jnb    short loc_451AE0      .text:00451B1B mov    al, [edx+3]
.text:00451AF1 loc_451AF1:                  .text:00451B1E cmp    al, [edi+3]
.text:00451AF1 cmp    ecx, 0FFFFFFFCh       .text:00451B21 jz     short loc_451B48
.text:00451AF4 jz     short loc_451B48      .text:00451B23 loc_451B23:
.text:00451AF6 loc_451AF6:                  .text:00451B23 inc    [esp+20h+var_8]
.text:00451AF6 mov    al, [edx]             .text:00451B48 loc_451B48:
.text:00451AF8 cmp    al, [edi]             .text:00451B48 cmp    [esp+20h+var_C], ebx
.text:00451AFA jnz    short loc_451B23      .text:00451B4C jle    short loc_451B52
```

(b)

```
   v12 = v7;
   v13 = v11 - 4;
   if ( v11 < 4 ) {
LABEL_27:
    if ( v13 == -4 ) break;
   }else{
    while ( *(_DWORD *)v12 == *(_DWORD *)v8 ){
     v12 = (char *)v12 + 4;  v8 = (char *)v8 + 4;
     v14 = (unsigned int)v13 < 4;   v13 -= 4;
     if ( v14 )  goto LABEL_27;
    }
   }
   if ( *(_BYTE *)v12 == *(_BYTE *)v8
     && (v13 == -3 || *((_BYTE *)v12 + 1) == *((_BYTE *)v8 + 1)
     && (v13 == -2|| *((_BYTE *)v12 + 2) == *((_BYTE *)v8 + 2)
     && (v13 == -1|| *((_BYTE *)v12 + 3) == *((_BYTE *)v8 + 3)))) )
    break;
```

(c)

Figure 1. Hex-Rays decompilation result. (a) Source code; (b) Disassembly program; (c) Decompile result of Hex-Rays.

program of *memcmp* shows that its control flow relationship is very complex and the decompilation result has *goto* statements, complicated conditional statements and nested control flow relationship. In fact, the disassembled program should just be translated into a single call statement v13 = *memcmp* (v12, v8, v11) where v12 points to the first buffer, v8 points to the second buffer and v11 is the number of bytes to be compared. According to our test, many inlined intrinsic functions are not handled properly by Hex-Rays. In order to improve the readability, we design the algorithm based graph matching [7] [8] to detect more such structures and improve the readability of decompilation results.

In this paper, we proposed a new decompilation architecture which first constructs the control flow graph of the disassembled program, then detects intrinsic functions on the structured disassembled program based on graph-subgraph isomorphism algorithm and update their representations, third we generate the immediate representation of this disassembled program, at last we do the data flow analysis, type analysis, structure analysis and code generation on the IR (Intermediate Representation).

Our contribution mainly lies in the following two aspects.

First, we rearrange the decompilation processes. Before generating the RTL (Register Transfer Language) [9] representation of the program, we recover the control flow graph of the program and the disassembled program is reserved in the subsequent optimization passes to offer more useful information, such as type information, memory access pattern etc.

Second, in order to raise the abstract level of the decompilation result, we utilize graph isomorphism algorithm on the CFG (Control Flow Graph) [10] to detect some inlined intrinsic functions or inlined built-in functions before RTL generation. And we believe this kind of optimization can be extended to other pattern detection and recovery, such as compound condition code detection, recovery of switch-case structure from binary tree representation and other memory access pattern detectionetc.

The rest of the paper is organized as follows. Section 2 describes previous work. Section 3 gives an overview of our system. Section 4 outlines our algorithm. Section 5 shows the experimental result. Section 6 and Section 7 show the discussion and the conclusion.

2. Previous Work

Decompilation techniques were initially used in the 1960s to aid the code migration from one platform to another. Since then, decompilation has been applied to program comprehension, source code recovery, debugging program and virus analysis. Some of the most prevalent decompilers and their main techniques are presented below.

2.1. Decompiler

Boomerang [1] constructs the RTL representation of a program with the help of SSL (Semantic Specification Language) [11] description and all the decompilation passes are operated on RTL. According to our test, Boomerang fails to detect the inlined intrinsic function.

Phoenix [12] utilizes BAP [13] to lift sequential x86 assembly instructions in the CFG into an intermediate language called BIL (BAP Intermediate Language) without detecting instruction idiom on both sequential x86 assembly instructions and BIL. But Phoenix uses an algorithm named Untiling algorithm which is similar to the tiling used in compiler to improve the decompilation readability. About 20 manually crafted untiling patterns are used to simplify these instructions emitted by gcc code generator. Phoenix shows great potential in *goto* elimination, structure recovery and abstract decompilation.

Hex-Rays [5] converts assembler instructions into detailed and precise microcode, typically each CPU instruction is converted into 5 - 15 microinstructions. Then it employs local optimization, global optimization, local variable allocation, structural analysis, pseudocode generation, pseudo code transformation and type analysis to raise the abstract level of the microinstructions. According to our tests, Hex-Rays can detect about one-third of the inlined intrinsic functions.

RD [2] detects the instruction idiom on the LLVM [14] IR code using a technique called peephole optimization [15] which is widely used in most optimizing compilers. But it's better to do the idiom analysis and intrinsic function detection on the assembly code rather than on the intermediate representation. The reasons are as following.

First, detecting idiom on IR is not universal because idioms vary greatly from different ISAs. So we need to construct different idiom patterns for different ISAs for the same idiom on IR which greatly reduces the generality of the IR analysis. While detecting the idiom earlier on the assembly code can improve the generality of the IR.

Second, analyzing the idioms earlier on the assembly code can reduce the complexity of many subsequent optimizations on the IR, such as the depletion in IR number and the simplicity of control flow, as the result, the complexity of data flow analysis is lowered.

Third, recovering the inlined intrinsic function earlier can provide type information for the succeeding type analysis. According to the result of [2], the idiom detection accuracy drops to 20% because of the instruction semantics of X86 is complex and the description of LLVM IR is so long which makes the idiom detection inefficient.

2.2. Graph Isomorphism

Graph isomorphism [16] is a well-studied topic which is widely used in compiler optimization [7], type checking [17] and semantic description [18] of functional programming language but not in decompiler. In fact graph isomorphism has many potential uses in decompiler as well; it's an area in need of being mined.

A graph *G*1 = (V1, E1) is isomorphic to a subgraph of a graph *G*2 = (V2, E2) if there exists a subgraph of G2, say *G*3 = (*V*3, *E*3), such that there is a bijection $\phi : V1 \to V3$. For each pair of vertices $vi, vj \in V1$, $(vi, vj) \in E1$ if and only if $(\phi(vi), \phi(vj)) \in E3$. In our case, as the instructions of a basic block play an important role so the graph is a labeled graph. Assume the vertex labeling for G1 and G3 is $\psi : V1 \to L$ and $\psi' : V3 \to L$ respectively. Then for each vertex $v \in V1$, $\psi(v)$ should be semantic compatible with $\psi'(\phi(V))$.

James Jianghai Fu [7] proposes an efficient algorithm to solve the rooted directed graph pattern matching problem using dynamic programming. Given a pattern graph P and a target graph T, its time and space complexity is $O(|E_P||V_T| + |E_T|)$. L.P. Cordella *et al.* [8] use SSR (State Space Representation) to describe a graph matching process and each state of the matching process represents a partial mapping solution. The algorithm does not rely on any special topological property of the graphs and can find all graph and graph-subgraph isomorphism between two given graphs. The worst temporal and spatial complexity of this algorithm is $O(N!N)$ and $O(N^2)$.

Based on the prevalence of code reuse, Wei Ming Khoo [19] proposes that decompilation can be done by search. He extracts instruction mnemonics, control-flow subgraphs and data constants from disassembled program to index source code from database which is constructed in advance. MotohiroKawahito *et al.* [7] propose to use topological embedding algorithm to detect hardware-assist instructions (such as TRT and TROT instructions on IBM zSeries) to accelerate delimiter search and character conversion. Tomáš Bílý [20] proposes to detect special cases of loops which can be transformed to *memset* or *memcpy* call to improve the runtime performance.

3. System Overview

ASMBoom is a decompiler which is constructed by us from the open source decompilation architecture named boomerang. **Figure 2** shows a high level overview of the approach that is taken by ASMBoom to decompile a

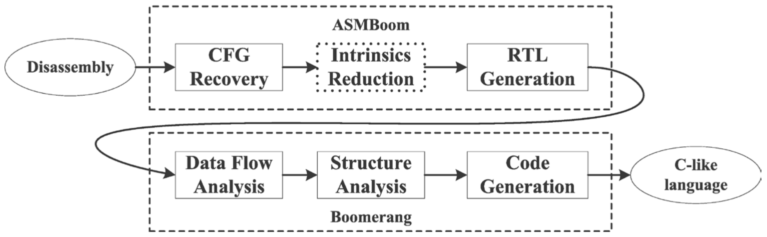


Figure 2. Decompilation flow.

target disassembled code. The frontend of the decompiler is called ASMBoom which consists of three parts CFG Recovery [21], Intrinsics Reduction and RTL Generation. The Backend of the decompiler is adapted from the Boomerang decompiler which also consists of three main parts Data Flow Analysis, Structure Analysis and Code Generation.

The biggest difference between our decompiler and the state-of-art decompilers is the decompilation sequence. Unlike the traditional decompilation process, before the generation of Intermediate Representation; we do intrinsics function detection which is highlighted using dotted box in the **Figure 2**.

The ASMBoom components consist of three main parts, they are CFG Recovery, Intrinsics Reduction and RTL Generation.

CFG Recovery is the process of converting the sequential disassembled program into graph representation based on control flow relationship. One of the greatest challenges in recovering the CFG is the indetermination of the branch address when coming across indirect jump instructions, jump table and indirect call instructions. In this paper we assume that the input disassembled code can be perfectly constructed so that we can focus on the intrinsic function detection.

Intrinsics detection procedure is fully based on pattern matching. We use graph isomorphism algorithm to detect some inlined intrinsic functions to raise the abstraction level of the decompilation result. Once the intrinsic function is detected, we modify the subgraphs which represent the inlined intrinsic functions in the original CFG and identify the parameter and return value. The details will be explained in Section 4.

After the intrinsic functions are detected, the original CFG is modified and the parameter and return value are identified, we need to generate the RTL representation of the assembly program which is completed by a simple traversal of each basic block of the CFG and generating IR for each instruction. Then the RTL of the input assembly is disposed by some of the processes in Boomerang, such as Data Flow Analysis, Structural Analysis and Code Generation. These are all well studied problems in modern compiler theory based decompiler which is out of the range of this paper.

4. Algorithm

The algorithm consists of three major parts. They are semantic compatibility of basic blocks, process of matching and determination of parameter and return value. Semantic compatibility of basic blocks, which assists the matching process, is employed to judge the similarity of two basic blocks. Most of the times, though two CFGs are isomorphic they can express different programs for the fact that the semantics of corresponding basic blocks are not compatible. Matching process determines which subgraphs of the target graph are isomorphic with the pattern graph. The basic blocks in the target graph which are matched with the pattern graph form a region [10]. This property of the in lined intrinsic function provides much more convenience to the analysis of parameter and return value of the intrinsic function. The CFG of the target function needs to be reconstructed before the parameter and return value are determined.

In **Figure 3** the intrinsic function reduction algorithm is outlined. The original recursive VF matching algorithm [8] is transformed into an iterative algorithm. Its time complexity is the same with the VF matching algorithm but its space complexity is greatly reduced for the recursive runtime stack is removed. The mapping M is expressed as the ordered pairs (n, m) each representing the mapping of a node n of graph $G1=(N1,B1)$ with a node m of $G2=(N2,B2)$:

$$M=\{(n,m)\in N_1\times N_2 \mid n \text{ is mapped onto } m\} \tag{4.1}$$

```
procedure intrinsic_function_reduction(target, template_list)
input: target function in CFG format; list of template template_list
output: target function with intrinsic function reduced to a single call
1.foreach template ∈ template_list
2. Construct match state S for target and template and set M(S) = ∅.
3.   Initiate the state stack state_stack with S.
4.while state_stack is not empty
5.     S = Pop (state_stack).
6. if M(S) covers all the nodes then
7.           Determine the parameter and return value of the inlinedfunction.
8.       else
9.           Compute the set P(S) of the pairs candidate for inclusion in M(S).
10.          foreach p ∈ P(S)
11.             if both the feasibility rules and semantic compatibility succeed
12. forthe inclusion of p in M(S)
13.             then
14.                Compute the state S' obtained by adding p to M(S).
15.                Push (state_stack, S').
16.             end if
17.          end foreach
18.       end if
19.    end while
20.end foreach
end procedure
```

Figure 3. Intrinsic function reduction algorithm.

State space representation is employed to effectively describe a graph matching process and each state S of the matching process represents a partial mapping solution. A partial mapping solution M(S) is the subsequence of M, *i.e.* contains only some components of M.

The details of semantic compatibility of two basic blocks, matching process and parameter and return value determination are explained below.

4.1. Semantic Compatibility of Two Basic Blocks

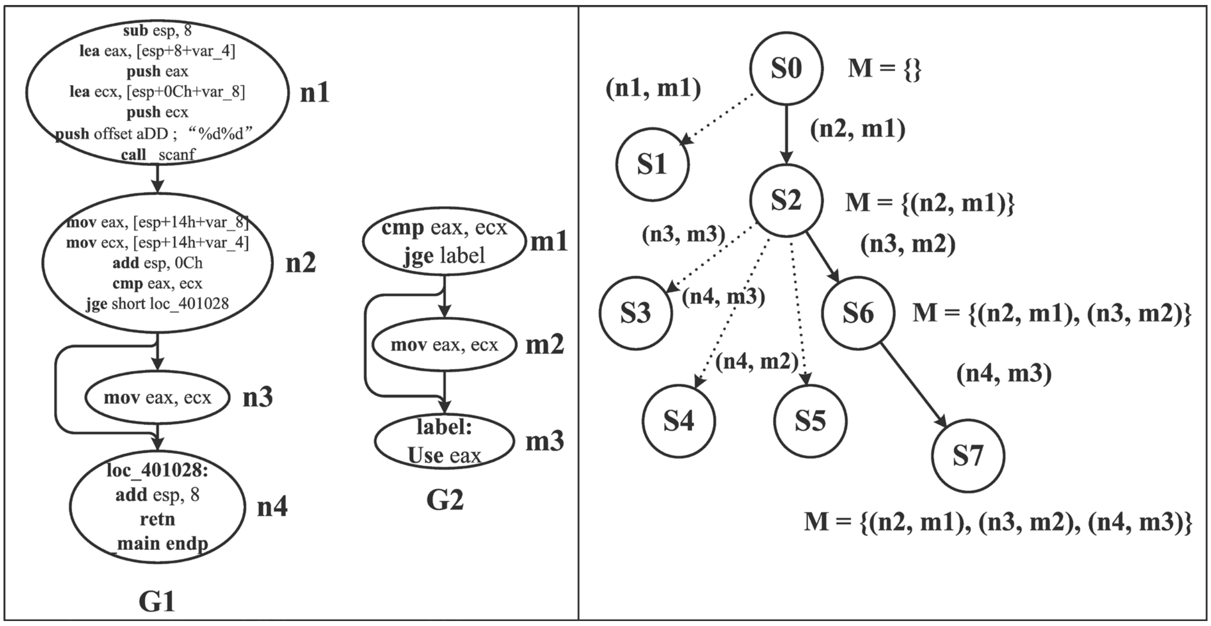
There are different ways to judge the semantic equivalence between two basic blocks. This topic is widely studied in compiler optimization and malware analysis. In [22], the authors use the Manhattan distance between the basic blocks' instruction mixes to judge their semantic equivalence. In [23], they use edit distance of the instructions to measure the semantic equivalence between two basic blocks. Paper [24] uses symbolic execution result to judge the semantic equivalence between basic blocks. Authors of [25] assign each basic block a color based on the instruction categories. They divide the instructions into 7 classes; data transfer, arithmetic, logic, test, stack, branch and call. Then the semantic compatibility is determined by the color.

However, all these methods are statistics based. In our system, we do not use the statistics based methods for the semantic of a basic block not only depends on the statistical features but also relies on the sequential feature and data flow. For instance, two basic blocks having the same statistical feature cannot guarantee that their semantics are equivalent because their instruction sequences and data flows may differ from each other.

Therefore, we use subsequence inclusion to judge the semantic compatibility of two basic blocks. That is basic block Pb of the intrinsic function template is compatible with basic block Tb of the target function if the mnemonics of Pb is a common sequence of the mnemonics of Tb. The reason why this method is employed is that sometimes basic blocks of the boundary of the intrinsic function is mixed up with adjacent basic blocks which do not belong to the intrinsic functions.

4.2. Matching Process

Matching process determines how the template is found in the target function using the modified VF matching algorithm in **Figure 3**. VF matching algorithm is a method to solve the subgraph isomorphism problem, which is a computational task in which two graphs G and H are given as input, and one must determine whether G contains a subgraph that is isomorphic to H. The feasibility rules are described in the above section. In our case, target graph is the decompiled assembly program expressed in CFG form. The pattern graph is the intrinsic function which is also in CFG form. In **Figure 4**, the target function consists of four basic blocks and the template



```
sub esp, 8
lea eax, [esp+8+var_4]
push eax
lea ecx, [esp+0Ch+var_8]
push ecx
push offset aDD ;  "%d%d"
call   scanf
```

```
mov eax, [esp+14h+var_8]
mov ecx, [esp+14h+var_4]
add esp, 0Ch
cmp eax, ecx
jge short loc_401028
```

```
mov eax, ecx
```

```
loc_401028:
add esp, 8
retn
_main endp
```

```
cmp eax, ecx
jge label
```

```
mov eax, ecx
```

```
label:
Use eax
```

Figure 4. Graph based pattern match process.

consists of three basic blocks. Target function is a simple main function to compute the max value of two input integers and template is the expansion of max macro. The searching tree is also illustrated on the right side of **Figure 4**. As we can see, many branches of the search tree are pruned based on some heuristic rules which is stated in the above section.

Figure 4 shows the matching process between target graph G1 and pattern graph G2. Initially, the matching solution M is empty and the SSR is s0. First, it tests node n1 in G1 with node m1 in G2. They do not match each other because their semantics are incompatible. So it continues to test node n2 in G1 with node m1 in G2, both of the two basic blocks have a *mov* instruction. Thus, they are semantic compatible, the matching procedure continues and the matching solution M becomes {(n2, m1)}. At the match state S2, each successor of n2 in G1 is compared with each successor of m1 in G2. When stage S2 tries to expand to stage S3, once n3 and m3 are not semantic compatible, the matching process returns. Then stage S2 tries to expand to stage S4 by comparing n4 and m3. At this time, as they are semantic compatible and the matching solution M becomes {(n2, m1), (n4, m3)}. When stage S4 tries to expand matching solution M, as there are no successors for either n4 or m3 and the matching solution M does not cover either G1 or G2 entirely, the matching process returns to stage S2. Then n4 of G1 is matched with m2 of G2 to extend the match state to S5, as the semantic is not compatible. So it returns to S2.

When matching node n3 in G1 and node m2 in G2, they match each other perfectly. As a result, it enters stage S6 and expands the matching solution M to {(n2, m1), (n3, m2)}. At state S6, each successor of n3 in G1 is compared with each successor of m2 in G2. As they do match each other, the stage is expanded. Matching solution M becomes {(n2, m1), (n3, m2), (n4, m3)}. Now matching solution M covers all the nodes of G2, so the matching process terminates and the matching solution M is returned as the final result which means that graph G2 match the subgraph of G1 which consists of {n2, n3, n4}. There exists a mapping {(n2, m1), (n3, m2), (n4, m3)} between graph G1 and graph G2. And the subgraphs of G1 made up of {n2, n3, n4} needs to be reduced to a basic block which consists of a call instruction.

4.3. Parameter and Return Value Determination

Before determining the parameter and return value of the inlined intrinsic function, the boundary of the intrinsic function needs to be identified and the CFG of the target function needs to be reconstructed.

For the entry basic block of the inlined intrinsic function, instructions in the root basic block of the matched subgraph are often mixed up with its predecessor's. If it is the case, we split the root basic block into two basic blocks with the first basic block containing instructions unrelated to the instructions in the template and the second basic block containing instructions related to the instructions in the template. Also, we add an edge from the first basic block to the second basic block, so as to denote the second basic block being the root basic block of the matched subgraph. Then we record all the incoming edges of the root basic block and set these edges as the incoming edges of the newly created basic block which contains a single call instruction with parameter and return value annotated.

For the exit basic blocks of the inlined intrinsic function, all the targeted basic blocks outside of the matched basic blocks need to be identified and recorded. These basic blocks will bethe targets of the newly created basic block.

Figure 5 illustrates the idea clearly. **Figure 5(a)** is the target function with a *strcmp* template embedded which is highlighted using dotted boxes. **Figure 5(b)** is the template of *strcmp*. **Figure 5(c)** is the target function after the intrinsic function *strcmp* is reduced. In **Figure 5**, there's a match {(B2, T1), (B3, T2), (B4, T3), (B5, T4), (B6, T5), (B7, T6)} between the target function and the *strcmp* template. So the basic block set {B2, B3, B4, B5, B6, B7} will be reduced to a single basic block B2′. All the incoming edges from outside of the basic block set {B2, B3, B4, B5, B6, B7} is {<B1, B2>}. All the outgoing edges to outside of the basic block set {B2, B3, B4, B5, B6, B7} is {<B6, B8>, <B7, B8>}. As the destination of the edges <B6, B8> and <B7, B8> are the same, so there should be only a single edge coming out from basic block B2′. Finally, the reconstructed target function will have only three basic blocks, namely B1, B2′ and B8. The edges will be <B1, B2′> and < B2′, B8>.

Van Emmerik [1] uses the following equations to determine the initial parameters and return values of a callee.

$$\begin{aligned} params(p) &= live\_on\_entry(p) \cap param\_filter \\ results(p) &= return\_filter(p) \cap live\_on\_exit(p) \end{aligned} \tag{4.2}$$

The parameter of the callee p, params (p), which are determined by the intersection of the live variables at call site live_on_entry (p) and the parameter filter param_filter. Results (p), the return results of the callee p, which are determined by the intersection of the return value definitions of procedure p return_filter (p) and the live variables at the exit of the call site live_on_exit (p).

Live_on_entry and live_on_exit can be computed by the live variable analysis algorithm. But for convenience, we assume live_on_entry and live_on_exit are all the full set as in our case the parameters and return values are determined by theparam_filter and the return_filter respectively.

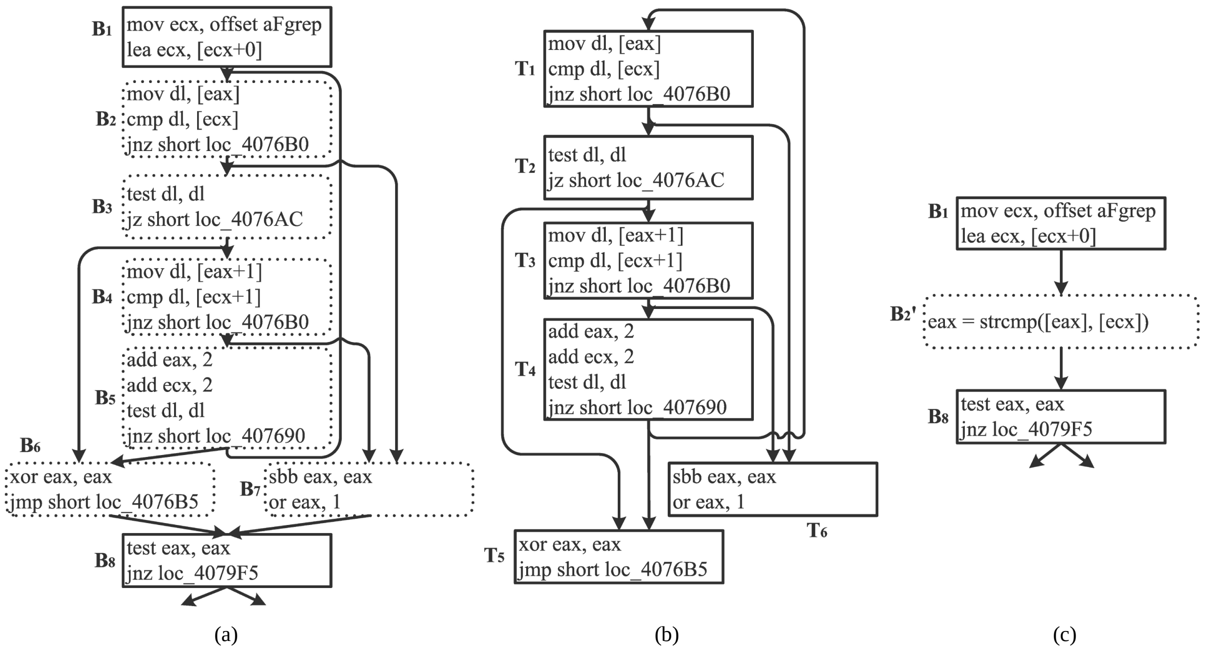


Figure 5. Intrinsic function reduction illustration. (a) Target function; (b) Strcmp template; (c) Intrinsic function reduced.

The param_filter and return_filter are determined by a triple <#BasicBlock, #Instruction, #Operand>, where #BasicBlock is the basic block number, #Instruction is the instruction number and # Operand is the operand number. For instance, in **Figure 5(b)**, the param_filter of the *strcmp* template is {<1, 1, 2>, <1, 2, 2>} which means the parameters of the *strcmp* intrinsic function can get from the 2nd operand of the 1st instruction in the 1st basic block and the 2nd operand of the 2nd instruction in the 1st basic block in turn. The return_filter of the *strcmp* template is {<6, 2, 1>} which means the return value of the *strcmp* intrinsic function can be got from the 1st operand of the 2nd instruction in the 6th basic block. To convert the triple representation into the variable representation, first the isomorphism mapping between the target function and the intrinsic function is used to convert the basic block number in template into the basic block number in the target function. In this case, the param_filter becomes {<2, 1, 2>, <2, 2, 2>} and the return_filter becomes {<7, 2, 1>} since T1 is matched with B2 and T6 is matched with B7. Second, the new triples in the param_filter and return_filter are used for indexing the operators in the target function. Finally, the param_filter becomes {[eax], [ecx]} and the return_filter becomes {eax}. Parameters of the inlined intrinsic function will be {[eax], [ecx]} and the return value of the inlined intrinsic function will be{eax}.

5. Experiment

To test the intrinsic function reduction algorithm, we collect some open source test suites and compile them with MSVC 2010 compiler with –*O*2 optimization option. We choose these test suites for the reason that lots of supported intrinsic functions illustrated in **Table 1** are contained in their source code. Xml [26] is the program which extracts and processes metadata from xml files. Awk [27] is the version of awk described in [28] by Al Aho *et al.* Grep [29] is a text search tool using regular expression. Sqlite [30] is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. We also compare our decompiler with three of the most popular decompilers Hex-Rays, RD and REC.

We manually craft 5 intrinsic function patterns which are all memory and string related method. These five patterns are extracted from the exes compiled by MSVC 2010 compiler with –*O*2 optimization option. **Table 1** shows the supported intrinsic function names and their functional descriptions.

Table 2 lists the contrast experimental result. Hex-Rays, a plug-in of IDA, is a very famous decompiler product which has a programmatic API to allow customers to implement their own analysis methods. REC is an interactive decompiler which supports different kinds of file formats and instruction sets. Although REC sources are not in the public domain its executable can be downloaded from the Internet. RD is a retargetable decompiler that can be utilized for source code recovery, static malware analysis. RD provides online decompilation service and has a very convenient UI.

Table 2 presents the comparison of intrinsic function reduction among decompilers. The dash line in the table means the data is not available for the decompiler fails to get the data. Call times show numbers of intrinsic funcions which are called in source code. We compare our decompiler ASMBoom with Hex-Rays, REC and RD decompiler based on the test suite xml, awk, grep and sqlite. Detected times show numbers which can be detected by the four decompilers.

6. Discussion

On average, ASMBoom outperforms other decompilers. It demonstrates that graph isomorphism based intrinsic function reduction technique is more efficient than the F.L.I.R.T method which is used in Hex-Rays decompiler

Table 1. Supported intrinsic function.

Function Name	Description
strlen	compute the length of a string
strcpy	copy a string of specific length from src to dst
memset	set memory of fixed length to a specific value
strcmp	compare two strings
memcmp	compare two blocks of memory with specific length

Table 2. Comparison of intrinsic function reduction among decompilers.

Test suite	**intrinsic function**	**Call times**	**Detected times**			
			Hex-Rays	REC	RD	ASMBoom
Xml	strlen	7	3	0	0	6
	strcpy	9	0	0	0	9
	memset	0	3	3	3	-
	strcmp	0	0	0	0	0
	memcmp	0	0	0	0	0
Awk	strlen	36	11	0	0	38
	strcpy	7	0	0	0	18
	memset	1	6	19	10	-
	strcmp	17	16	0	0	11
	memcmp	0	0	0	0	0
Grep	strlen	35	11	0	0	22
	strcpy	17	0	0	0	9
	memset	3	7	11	9	-
	strcmp	36	11	0	0	5
	memcmp	1	0	0	0	1
Sqlite	strlen	68	3	0	-	4
	strcpy	0	0	0	-	11
	memset	255	203	195	-	-
	strcmp	104	72	0	-	70
	memcmp	57	0	0	-	11

and the method used in RD [31]. Hex-Rays sometimes misinterpret some intrinsic function like code segment as intrinsic function, especially on the *memset* intrinsic function. Graph isomorphism based method is based on the details of semantic compatibility of two basic blocks, matching process. So it can be more efficient than other decompilers in detecting graphs in fixed pattern.

Our algorithm fails to detect all the inlined intrinsic functions owing to the incompleteness of the templates we provide. By tracking all the inlined *memcmp* intrinsic function in the sqlite test suite, we find there're different kinds of templates depending on its context. Most of the templates have redundant basic blocks inserted into the original template which turns the graph isomorphism problem into a graph topological embedding one.

7. Conclusions

Graph isomorphism based method to recover the inlined intrinsic function from structured assembly program is very efficient. We discover more inlined intrinsic functions than the state-of-art decompilers such as Hex-Rays, REC and RD.

The only shortcoming of our approach is that the intrinsic function template is compiler oriented. Intrinsic functions vary greatly from compilers both in topological structure and semantic of basic blocks. Constructing various templates for the same intrinsic function makes matching process a complex decision-making procedure.

In the future, topological embed should be employed to detect more intrinsic function and inlined user-defined functions. When trying to extend our idea to lift the abstract level of the decompilation result to other techniques, we need to mine the binaries or the assemblies to find some of the patterns that need to be abstracted. We can use some data mining methods, such as using frequent subgraphmining [32] to discover the unrolled loop, using frequent sequential pattern mining [33] to explore the array access pattern etc. Then we represent these patterns in graph format and use graph matching algorithm to discover these patterns and raise the abstract level of the RTL representation.

Fund

This work was supported by the science and technology project of Shaanxi province (2014K05-04).

References

[1] Van Emmerik, M.J. (2007) Static Single Assignment for Decompilation. The University of Queensland, Brisbane.

[2] Kroustek, J. and Pokorny, F. (2013) Reconstruction of Instruction Idioms in a Retargetable Decompiler. *Federated Conference on Computer Science and Information Systems* (*FedCSIS*), Kraków, 8-11 September 2013, 1519-1526.

[3] Chen, G., *et al.* (2013) A Refined Decompiler to Generate C Code with High Readability. *Software*: *Practice and Experience*, **43**, 1337-1358. http://dx.doi.org/10.1002/spe.2138

[4] Reverse Engineering Compiler. http://www.backerstreet.com/rec/rec.htm

[5] Guilfanov, I. (2008) Decompilers and Beyond. Black Hat USA.

[6] Hex-Rays, IDA F.L.I.R.T Technology: In-Depth. 2015. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

[7] Fu, J.J. (1997) Directed Graph Pattern Matching and Topological Embedding. *Journal of Algorithms*, **22**, 372-391. http://dx.doi.org/10.1006/jagm.1996.0818

[8] Cordella, L.P., *et al.* (1999) Performance Evaluation of the VF Graph Matching Algorithm. *Proceedings of International Conference on Image Analysis and Processing*, Venice, 1999, 1172-1177. http://dx.doi.org/10.1109/iciap.1999.797762

[9] Cifuentes, C. and Van Emmerik, M. (2000) UQBT: Adaptable Binary Translation at Low Cost. *Computer*, **33**, 60-66. http://dx.doi.org/10.1109/2.825697

[10] Aho, A.V., Ullman, J.D. and Sethi, R. (1986) Compilers, Principles, Techniques, and Tools. Addison-Wesley Pub. Co., Reading, MA, 796 p.

[11] Duke, R., Rose, G. and Smith, G. (1995) Object-Z: A Specification Language Advocated for the Description of Standards. *Computer Standards & Interfaces*, **17**, 511-533. http://dx.doi.org/10.1016/0920-5489(95)00024-O

[12] Brumley, D., *et al.* (2013) Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. USENIX, Washington DC, 353-368.

[13] Brumley, D., *et al.* (2011) BAP: A Binary Analysis Platform. In: *Proceedings of the* 23*rd International Conference on Computer Aided Verification*, Springer-Verlag, Snowbird, UT. http://dx.doi.org/10.1007/978-3-642-22110-1_37

[14] Lattner, C. and Adve, V. (2004) LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *IEEE International Symposium on Code Generation and Optimization*, 20-24 March 2004, 75-86. http://dx.doi.org/10.1109/cgo.2004.1281665

[15] Tanenbaum, A.S., Van Staveren, H. and Stevenson, J.W. (1982) Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), **4**, 21-36. http://dx.doi.org/10.1145/357153.357155

[16] Ullmann, J.R. (1976) An Algorithm for Subgraph Isomorphism. *Journal of the ACM* (*JACM*), **23**, 31-42. http://dx.doi.org/10.1145/321921.321925

[17] Katzenelson, J., Pinter, S.S. and Schenfeld, E. (1992) Type Matching, Type-Graphs, and the Schanuel Conjecture. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), **14**, 574-588. http://dx.doi.org/10.1145/133233.133247

[18] Holm, K.H. (1990) Graph Matching in Operational Semantics and Typing. In: *CAAP*'90, Springer, 191-205. http://dx.doi.org/10.1007/3-540-52590-4_49

[19] Khoo, W.M. (2013) Decompilation as Search. University of Cambridge, Cambridge.

[20] Bílý, T. (2006) Replacement Special Loop Form by a Call of Built-in Function. In: *Proceedings of the GCC Developers' Summit* 2006.

[21] Cooper, K.D., Harvey, T.J. and Waterman, T. (2002) Building a Control-Flow Graph from Scheduled Assembly Code.

[22] Demme, J. and Sethumadhavan, S. (2012) Approximate Graph Clustering for Program Characterization. *ACM Transactions on Architecture and Code Optimization*, **8**, 1-21. http://dx.doi.org/10.1145/2086696.2086700

[23] Cong, J., Hui, H. and Wei, J. (2010) A Generalized Control-Flow-Aware Pattern Recognition Algorithm for Behavioral Synthesis. *Design*, *Automation & Test in Europe Conference & Exhibition* (*DATE*), Dresden, 8-12 March 2010, 1255-1260. http://dx.doi.org/10.1109/date.2010.5456999

[24] Luo, L., *et al.* (2014) Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Hong Kong, China: ACM. http://dx.doi.org/10.1145/2635868.2635900

[25] Kruegel, C., *et al.* (2006) Polymorphic Worm Detection Using Structural Information of Executables. *Proceedings of the* 8*th International Conference on Recent Advances in Intrusion Detection*, Springer-Verlag, Seattle. http://dx.doi.org/10.1007/11663812_11

[26] Liu Zhangpei.xml Test Suite. https://github.com/livenowhy/xml

[27] Ted Nyman.awk test suite. https://github.com/tnm/awk. 2015 March.

[28] Aho, A.V., *et al.* (1988) The AWK Programming Language. Addison-Wesley, New York.

[29] Coapp-Packages. Grep Test Suite. https://github.com/coapp-packages/grep

[30] Sqlite Test Suite. http://www.sqlite.org/download.html

[31] Ďurfina, L. and Kolář, D. (2013) Generic Detection of the Statically Linked Code. *Proceedings of the Twelfth International Conference on Informatics* (*INFORMATICS*'13), SpišskáNováVes, SK, FEI TU in Košice, 157-161.

[32] Yan, X. and Han, J. (2002) gSpan: Graph-Based Substructure Pattern Mining. *Proceedings of IEEE International Conference on Data Mining*, Maebashi, 9-12 December 2002.

[33] Ayres, J., Gehrke, J., Yiu, T. and Flannick, J. (2002) Sequential Pattern Mining Using a Bitmap Representation. In: *Proceedings of the* 8*th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, Edmonton.