IL Optimization: Detecting and Eliminating Redundant Eflags by Flag Relevant Chain

Xue Lei[1,*], Wenqing Fan[2], Wei Huang[2], Yixian Yang[1] and Zhongxian Li[3]

[1]*Information Security Center, Beijing University of Posts and Telecommunications National Engineering Laboratory for Disaster Backup and Recovery, Beijing, China*

[2]*Communication University of China, Beijing, China*

[3]*National Cybernet Security Ltd, Beijing, China*

Abstract: In this paper, we proposed a systematic approach for automatically detecting and elimination redundant Eflags to optimize intermediate language (IL). We analyzed a broad spectrum of different IL and found that a number of IL expose all side effects explicitly by default and that not all the Eflags are relevant with subsequent analysis. Therefore, we proposed a unified approach, invertible analysis, to reduce the volume of IL. Our approach does not rely on any concrete IL, and thus can identify redundant Eflags in the IL. Moreover, we devised a method using flag relevant chain dependency analysis to remove redundant Eflags and shrink the IL. We developed a prototype, and conducted extensive experiments using representative samples from various categories. We demonstrated that our approach could diminish the volume of Vine IL obviously, and provide accurate representation about the assembly code.

Keywords: Intermediate language, Redundant Eflags, Invertible analysis, IL optimization.

1. INTRODUCTION

Intermediate representation (IR) of the program has long been an important task in program analysis research and practice. IR is a key issue both for compilers as well as for reverse engineering tools to enable efficient analyses. Intermediate language (IL) is designed to describe the intermediate representation of the program. From an analyst's perspective, an appropriate IL can achieve a multiplier effect. In the field of program analysis technology, the importance of a suitable IL has long been recognized. The design of an IL is a key factor for the efficiency of analyses for code optimization and has been extensively studied. With IL program analyses can be written in an architecture independent fashion and do not need to directly deal with the complexity of an instruction set such as x86 [1]. This design also provides extensibility, so users can easily write their own analysis on the IL by building on top of the core utilities provided through some program analysis platforms. Every platform such as CodeSurfer/x86 [2], McVeto [3], Phoenix [4], and Jakstab [5], BAP [6] defines its IL and then performs analysis at the IL level. For example, Vine that is the static analysis component of BitBlaze [7] disassembles binary code into assembly instructions, lifts the instructions to an IL and enables all subsequent analyses to be written in a syntax-directed fashion.

*Address correspondence to this author at the Beijing University of Posts and Telecommunications, Beijing, 100876, China;
E-mail: leixue@bupt.edu.cn

Machine language and assembly language have side effects such as that EFLAGS set by x86 instructions is implicit. Those side effects are the keys to determine which branches will be taken. In order to explain the side effects, consider the two-line assembly program below. The arithmetic operation (shr) sets up to six status flags, and control flow in assembly depends upon the values of those flags.

```
1 shr ebx, cl # ebx=ebx>>cl(sets OF, SF, ZF, AF, PF, CF)
2 jc target # jump to target if carry flag is set
```

Therefore, a couple of IL are designed to address the problem by explicitly exposing all side effects. The result is that subsequent analyses and verification could rely upon the IL syntax alone. In the field of compiler optimization and program analysis, the most common IL belongs to static single assignment (SSA) form. Informally, the IL is said to be in SSA form if it meets two criteria [8, 9]: 1) each name has exactly one definition point, and 2) each use refers to exactly one name. Comparing the IL and assembly instruction, the advantages of IL are three-fold: 1) each instruction represents only one operation, 2) instruction set does not consist of control flow information, and 3) the number of registers defined in IL is unlimited by the number of registers defined in the operating system. However, the volume and amount of IL has to be considered when the target application is large and complex.

Applications are getting larger and more complex due to increasing functionality, a more sophisticated software stack, and new abstractions and concepts that simplify development. When some complex application is translated to IL, the volume and amount of IL is inevitably infinite. For instance, dynamic symbolic analysis will only run one path at a time while the complex application always has countless paths. If every result of dynamic symbolic analysis is lifted to IL, the volume of these IL files is unimaginable. So infinite number of IL files leads to tremendous need for storage space, and if so, it is a challenge to the stable and reliable performance of the computer. Moreover, it is hard for subsequent analyses to deal with these IL files in high efficiency. In this paper, we propose the first systematic approach to address this research problem.

The motivation of our approach is that the first step of program analysis is to represent the program by IL, and the volume and number of IL files is enormous, of which impacts eventually affects the performance of the program analysis system and the whole computer. The motivation of our approach is to optimize the volume and complexity of IL and facilitate later analysis. As assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. In order to describe our algorithm, we use assembly language as the source code and Vine IL [7] as the intermediate language. The semantics of Vine IL are designed to be faithful to assembly languages. Although some assembly instruction affects a couple of flags, maybe there is only one (or a few) flag/flags is relevant with the next instruction in practice. If we can eliminate those irrelevant flags, the efficiency of later analysis maybe multiplied several times. To verify this idea, we present the novel approach. We primarily disassemble binary code to assembly code, then inversely analyze the assembly code to construct the flag relevant chain (FRC) for every instruction. According to FRC, when assembly codes are lifted to Vine IL, it omits those irrelevant flags. Finally, we evaluate the feasibility of the algorithm with six programs. In the experiment, our approach could diminish the volume of Vine IL obviously. The efficiency and effectiveness of the approach makes it possible to automatically eliminate the redundancy of IL from the large application.

In summary, this paper makes the following contributions:

1. we introduce a new approach to optimize IL, the FRC which represents the relationship between flags and instructions by inverse analyses of assembly code;
2. we give algorithms for building the FRC given only a stripped binary program and optimizing of IL based on FRC;
3. We have conducted extensive experiments with large program from various categories, and demonstrated that the approach could instantly optimize IL, and provide accurate representation about the assembly code.

The paper is structured as follows. The next section gives an overview of our approach. Section 3 describes details on the design and implementation of the approach. Section 4 presents the experimental results. Section 5 provides some additional limitation of our approach. Section 6 surveys related work and Section 7 concludes the paper.

2. PROBLEM STATEMENT AND OUR APPROACH

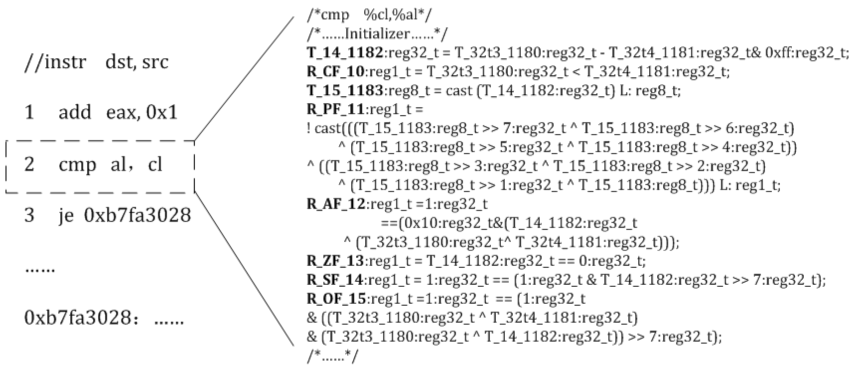
In this section, we formalize the problem of IL analysis and optimization, and give a brief overview of our approach.

2.1. Problem Statement

Background: intermediate language. To eliminate side effects of machine language or assembly language, intermediate language is designed to describe the intermediate representation of the program. The semantics of the IL must be well-defined and it must exactly describe the constructs of the modeled programming languages which are necessary for an exact analysis. Likewise, the IL should be linear in size to the length of the source code. This property is particularly important for global analyses of large programs. In addition, the IL should allow efficient control and data flow analysis. Finally, to handle large systems in a reasonable time, IL should be constructed efficiently.

The challenge of intermediate language with redundancy. Although analyzers are trying to design syntax of IL clearly and explicitly, in the face of large applications, some parts of IL are obviously redundant. In this paper, we use the term “instruction” to refer to an assembly-level instruction, and the term “statement” to refer to instructions within Vine IL. In order to explain the redundancy of statement, consider the three line assembly program in Fig. (**1a**) below. The statement of the first instruction *add* on line1 that computes the sum of *eax* and integer *0x1* and affects the flags including *OF, SF, ZF, AF, PF*, and *CF*. Then the second instruction *cmp* on line2 affects the flags as well as the first instruction. The third instruction *je* on line3 does not affect any flag, but it is based on the result of a *cmp* instruction on line2 that is to compare *al* with *cl* and updates those flags according to the result. We find that except zero flag from line2, all the other flags are irrelevant with the line3. That is to say, there is no need to state flags of both line1 and line2 except *ZF* of line2. Translated a single typical instruction has all side effects explicitly exposed as statement. Because of the limitation of length, we choose line2 in Fig. (**1a**) to be translated as a sequence of statement, as illustrated in Fig. (**1b**).

Fig. (**1b**) only shows parts of the statement of the instruction *cmp* and it is obvious that the size of statement is worth our attention. We can find that whether the instruction *je* jumps to the address *0x7fa3028* based on the result of instruction *cmp*. If *al* is equal to *cl*, the flag *ZF* is set to one. Therefore, the instruction je only relies on the flag *ZF* of *cmp*. When an analyst is faced with a large application with millions of lines of instructions, the first step is to translate these instructions into statement. With so large amount of statement, it is a headache. If there were only a few instructions, perhaps the complexity would not be too onerous. For instance, if the target program has millions of lines of instructions and *N* instructions affect flags, the statement are at least *6*N* lines more than the original. In dynamic analysis, if the target program has *M* branches, it will generate *M* IL files. Due to large quantities of statement, the processor needs more time to create them, which is a key to efficiency of the program analysis tool. Furthermore, if a large application has thousands (even millions) of branches, it needs more space



```
//instr   dst, src

1   add   eax, 0x1

2   cmp   al，cl

3   je   0xb7fa3028

……

0xb7fa3028：……
```

```
/*cmp   %cl,%al*/
/*……Initializer……*/
T_14_1182:reg32_t = T_32t3_1180:reg32_t - T_32t4_1181:reg32_t& 0xff:reg32_t;
R_CF_10:reg1_t = T_32t3_1180:reg32_t < T_32t4_1181:reg32_t;
T_15_1183:reg8_t = cast (T_14_1182:reg32_t) L: reg8_t;
R_PF_11:reg1_t =
! cast(((T_15_1183:reg8_t >> 7:reg32_t ^ T_15_1183:reg8_t >> 6:reg32_t)
     ^ (T_15_1183:reg8_t >> 5:reg32_t ^ T_15_1183:reg8_t >> 4:reg32_t))
^ ((T_15_1183:reg8_t >> 3:reg32_t ^ T_15_1183:reg8_t >> 2:reg32_t)
     ^ (T_15_1183:reg8_t >> 1:reg32_t ^ T_15_1183:reg8_t))) L: reg1_t;
R_AF_12:reg1_t =1:reg32_t
            ==(0x10:reg32_t&(T_14_1182:reg32_t
      ^ (T_32t3_1180:reg32_t^ T_32t4_1181:reg32_t)));
R_ZF_13:reg1_t = T_14_1182:reg32_t == 0:reg32_t;
R_SF_14:reg1_t = 1:reg32_t == (1:reg32_t & T_14_1182:reg32_t >> 7:reg32_t);
R_OF_15:reg1_t =1:reg32_t  == (1:reg32_t
& ((T_32t3_1180:reg32_t ^ T_32t4_1181:reg32_t)
& (T_32t3_1180:reg32_t ^ T_14_1182:reg32_t)) >> 7:reg32_t);
/*……*/
```

(**a**) (**b**)

Fig. (1). A simplified example of a three line assembly program is shown on the left (**a**). The right (**b**) illustrates the statement of the second instruction *cmp*.

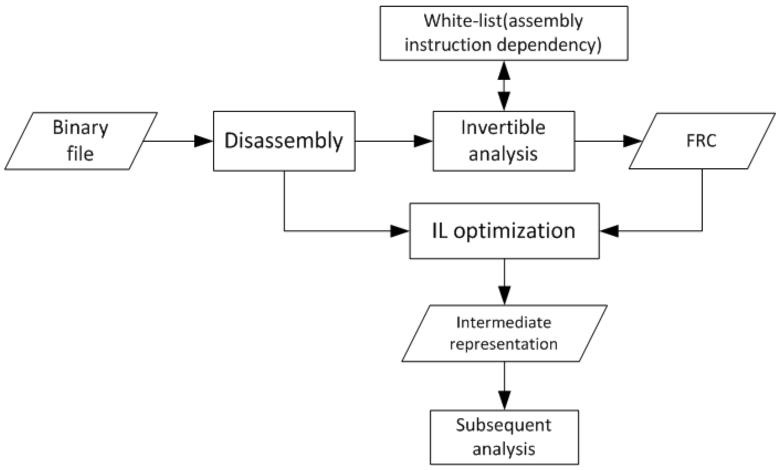


Fig. (2). IL optimization design.

to store these IL files. If we reduce the redundancy of statement, maybe it increases the efficiency of the successor analysis. To verify the hypothesis, we propose our novel approach.

2.2. Our Approach

In this section, we give an overview of our approach that is an invertible analysis based on instruction to create FRC to guide statement optimization during statement generation. We first discuss the intuition behind it, outline the steps involved, and then explain how it applies to optimize statement. Fig. (**2**) shows the overall flow of our approach.

Intuition. The insight behind our approach is that it is possible to avoid the problem caused by statement redundancy, via constructing FRC and using it to guide statement optimization. For instance in the Fig. (**1a**), some flags are redundancy. Our approach can verify that flags of first instruction are not necessary to be translated, so is the second instruction except the zero flag. For a binary-only program, we disassemble it and invertibly analyze instructions from end to beginning to create FRC which is guiding to omit redundant flags. We hope our work will spur discussions on the implications and applications of IL optimization.

Inversed analysis. In outline, our approach proceeds as follows. As a first phase, our approach disassemble the binary file by off-the-shelf disassemblers such as IDA Pro [10], a commercial disassembler and a research disassembler from Kruegel *et al.* [11] that can disassemble x86 obfuscated code. Then in the second phase, our approach proposes inversely linear-sweep instructions of code segment, constructing FRC by context. According to x86 instruction set, we summarize the instructions that depend upon flags into an instruction dependency table. As in Fig. (**1a**), we inversely analyze the three line assembly program and construct FRC. We first check line3 and find that it is a conditional transfer instruction only relying on the zero flag of line2. Thus, we mark line3 with zero flag, and then we check line2 and mark it with 0 because it is not dependent on any flag. We decide the dependent flags of the instruction by comparing it with the white-list into which we categorize flag dependent instructions. We detail our construction of FRC approach in Section 3.3.

Optimization IL. The goal of our approach is to deduce redundancy of IL and optimize it. During the invertible analysis, we record into FRC the details about which flags are relevant with subsequent instructions in the program. Therefore, from the FRC corresponding to the invertible analysis, we can perform IL optimization according to the FRC. In view of those instructions that do not operate on data in the Eflags register are translated to statement normally. On the contrary, to these instructions affecting flags we identify weather if there are any relevant flags by looking up the FRC, and if so, we translate the relevant flags of the instruction recorded in FRC while omit those irrelevant flags and vice versa. We detail our optimization IL approach based on FRC in Section 3.4. We further discuss the availability of FRC in Section 5.

3. SYSTEM DESIGN AND IMPLEMENTION

In this section, we describe our approach to optimize IL: first some infrastructure details (Section 3.1), then techniques for categorizing instructions based on dependent flags into a white-list (Section 3.2), inversely analyzing instructions to constructing FRC (Section 3.3), and optimizing IL according to FRC (Section 3.4).

3.1. Infrastructure

To describe our algorithm, we use assembly language as the source code and Vine IL as the intermediate language. To construct a FRC via static analysis, we take an invertible analysis, which lets us easily identify relevant flags of instructions in context and evaluate them with concrete examples.

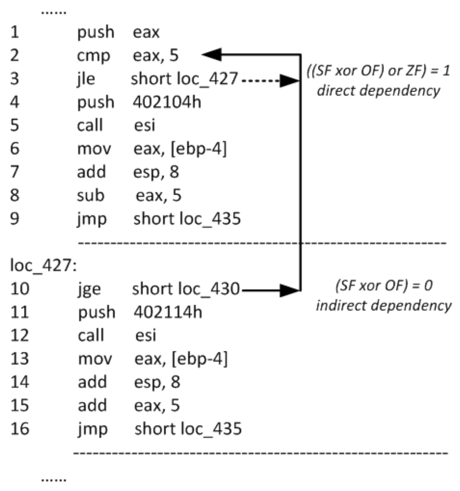
We implement our approach on top of Vine [12], which is the static analysis component of BitBlaze project [13]. Vine supports translating x86 [14] and ARMv4 [15] to the Vine IL. It not only interfaces with disassemblers such as IDA [10], and also uses a set of third-party libraries to parse different binary formats and produce assembly. The assembly is then translated into the Vine IL in a syntax-directed manner.

Our modular prototype interfaces with disassembler IDA (for disassembly binary files). It includes a white-list that lists all the flag dependent instructions, FRC that consists of flags that each instruction relies on, and an optimization module that omits irrelevant flags based on FRC.

3.2. Assembly Instruction Dependency

One approach primarily disassembles binary code into a sequence of assembly instructions, and then constructs FRC to guide IL optimization. The performance of program analysis on assembly is naive because each analysis would have to individually understand the semantics of the assembly, which is difficult. Since Vine IL interface with disassembler-IDA Pro, we use it to translate a binary file to assemble instructions.

In our implementation, we categorize the architecture x86 instruction set which has well over 300 instructions (which are documented in over 11 lbs of manuals [14]). Our goal is to summarize flag dependency instructions



```
   ......
1       push   eax
2       cmp    eax, 5
3       jle    short loc_427
4       push   402104h
5       call   esi
6       mov    eax, [ebp-4]
7       add    esp, 8
8       sub    eax, 5
9       jmp    short loc_435
   ----------------------------------------------------------
loc_427:
10      jge    short loc_430
11      push   402114h
12      call   esi
13      mov    eax, [ebp-4]
14      add    esp, 8
15      add    eax, 5
16      jmp    short loc_435
   ----------------------------------------------------------
   ......
```

Fig. (3). An example of invertible analysis of some part of instructions from a binary file.

(e.g.,cmovz, je, rep, etc.) by a hash table. The instruction name is key value *k*, and with the hash function *f* the index of the instruction is *f (k, array_size)*.

In our research, flag dependency instructions can be divided into three main groups that are: 1) conditional data transfer instructions, 2) conditional control transfer instructions, 3) conditional loop operation instructions, 4) repeated prefixes of string instructions, and 5) byte set on condition instructions. These instructions do not affect flags but rely on the state of selected status flags in the Eflags register. Conditional data transfer instructions in the form of *CMOVcc* move data between memory and the general-purpose and segment registers according to some flags. Conditional control transfer instructions provide conditional jump in the form of *Jcc* and loop in the form of *LOOPcc* to control program flow. The repeated prefixes of string instructions in the formation of *REPcc* operate on strings of bytes, allowing them to be moved to and from memory. A *REPZ* prefix essentially turns an instruction into a single instruction *loop* under the condition that zero flag is set. Byte set on condition instructions in the form of *SETcc* set the value of a byte operand to indicate the status of flags in the EFLAGS register.

3.3. Invertible Analysis

The most important technique for reducing the redundancy of statement must be the construction of FRC to guide optimization of IL by eliminating irrelevant flags.

FRC is a single list structure that stored information from the results of invertible analysis. Whenever a flag dependency instruction is verified, information about the instruction is written into a special defined data structure as a new node of FRC. During our research, we discovered that flag dependency instructions could generally be divided into two categories: direct and indirect, as illustrated in Fig. (**3**).

The direct is defined that the flag dependency instruction depends upon flags of its previous instruction, as implied by dotted arrow in Fig. (**3**). For instance, the instruction *jle* on line3 tests to see if the expression ((*SF xor OF) or ZF*) is equal to one, and if so, jump to the address of module *loc_427*. Our goal is to find which instruction decides the three flags *SF, OF* and *ZF*. In the same module divided by the control flow graph, the precious instruction *cmp* on line2 computes the difference between *eax* and integer 5 and updates the *OF, SF, ZF, AF, PF,* and *CF* flags according to the result. We can conclude that the instruction *jle* directly relies on its previous instruction *cmp* and describe it as the following formula:

$$\exists_n : I_{fdi}(n) \xrightarrow{Dd} I_{fri}(n+1) \tag{1}$$

Formula (1). The formula of direct dependency.1)I_{fdi}: the flag dependency instruction, 2)I_{fri}: the flag relevant instruction, 3)D_d:direct dependency.

The indirect is defined that the flag dependency instruction relies on flags of the instruction in other module and generally both instructions are separated by several instructions, as implied by solid arrow in Fig. (**3**). In this example, the instruction *jge* on line10 tests to see if the expression (*SF xor OF*) is equal to zero, and if so, jump to the address of module *loc_430*. The instruction *jge* is the first instruction of this module indicating that it may rely on some flag relevant instruction somewhere in other module. That is to say, we must find the place where the module *loc_430* is referenced. Throughout invertible linear-sweep, the entry of *loc_430* is the target address of the instruction *jle* on line3. The challenge is converted to search for the flag relevant instruction of *jle* and the result is obvious. In conclusion, the instruction *jge* indirectly rely on the instruction *cmp* with *m* intervals, illustrated as the following formula:

$$\exists_{n,m} : I_{fdi}(n) \xrightarrow{Did} I_{fri}(n+m) \tag{2}$$

Formula (2). The formula of indirect dependency.1)D_{id}:indirect dependency, 2)m: the number of intervals between I_{fdi} and I_{fri}.

In order to construct FRC, our approach performs two major steps. First, the instructions produced by the disassembler are invertible linear-sweep by the scanner from the last instruction, in order to determine the relationship between the flag dependency instruction I_{fdi} and the flag relevant instruction I_{fri}. When scanner detects an instruction, it looks up the hash table T_{hash} of all the flag dependency instructions to verify if it is a I_{fdi}, and if so, records its relevant flags F_{ins} and its address A_{ri}, and then the scanner goes up to the precious instruction and verify the dependent manner (direct D_d and indirect D_{id} dependency) detailed in the first two paragraphs. Further, we create a new node to record related information and check if there has been a node about I_{fri} in FRC already, and if not, we insert the node to the FRC. On the contrary, as the example shown in Fig. (**2**), the instruction *jge* and *jle* have the same relevant flags instruction I_{fri}, when we detect there has been already a node about *cmp* in FRC, we take intersection of the two F_{ins} and update the node information. Until the number of instruction I_{No} refers to the last one which means the invertible analysis is complete, loop is over. A pseudocode description of this algorithm showing how the inverse analysis is performed to generate FRC is in algorithm1.

Algorithm 1: Our FRC generation algorithm

```
    input : src: a binary file
    output: FRC: the flag relevant chain
1   ($I_{ida}$, $I_{sum}$) = Disassembler(src);
2   FRC = Create-FRC();
3   While ($I_{No} \neq end$) do
4       ($I_{fdi}$, $F_{ins}$) = Look-table($I_{ida}$, $T_{hash}$) ;
5       if $I_{fdi} \wedge D_d$ then
6           $I_{fri} = I_{No}$ ++;
7       if $I_{fdi} \wedge D_{id}$ then
8           $I_{fri}$ = Find-Ins($I_{No}$, $A_{ri}$ );
9       $N_{ins}$ = Node-Gen($I_{fdi}$, $I_{fri}$, $F_{ins}$);
10      FRC = Insert-FRC($N_{ins}$);
11      $I_{No}$ --;
12  return FRC;
```

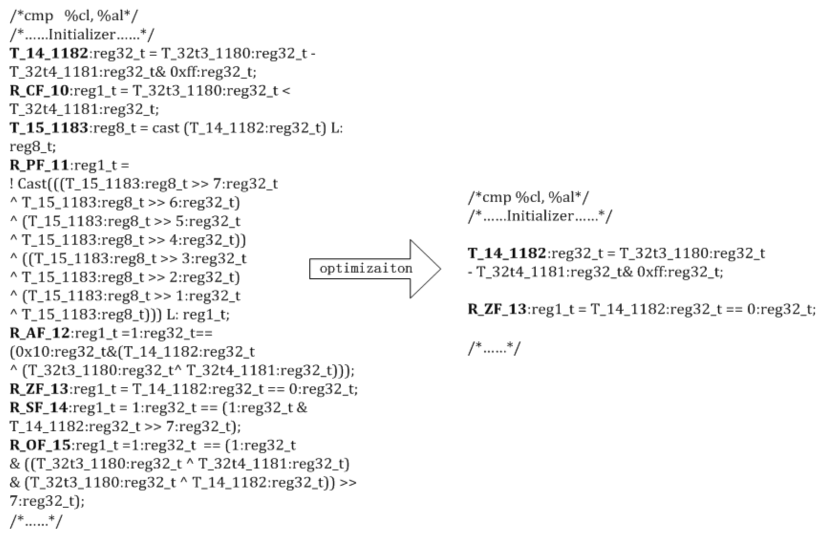
3.4. IL Optimization

To omit redundant flags from IL, we reference FRC obtained by the precious invertible analysis. Comparing with FRC may seem strange since there are more checks to perform at each instruction during lifting. However, the shrinking of the statement-imposed by the FRC outweighs subsequent analyses overhead for the IL. The advantages of IL optimization are best demonstrated via example. Consider the program shown in Fig. (**1**). Firstly, suppose that the instruction add on line1 is not relevant with any flag dependency instruction. In other words, although the instruction add affects flags including *OF, SF, ZF, AF, PF,* and *CF*, no subsequent instructions rely on them. Thus, there is no information about the instruction *add* in FRC, we can omit the flags of the instruction *add*. Secondly, by checking the *FRC* in the instruction *cmp* on line2 it is only relevant with *ZF* of the instruction *je* on line3, meaning that we can only express the zero flag. Fig. (**4**) depicts the optimization of the instruction *cmp* visually.

In summary, the optimization based on FRC goes through the following analysis steps. Those instructions that do not operate on data in the Eflags register are translated into statement normally. In contrast, to those instructions affecting flags (I_{fri}) we identify weather if there are any relevant flags (F_{ins}) by looking up the FRC, and if not, we set F_{ins} null and do not translate the flags of this instruction to statement. Otherwise, in order to optimize statement, we translate the relevant flags of this instruction recorded in FRC and omit those irrelevant flags. Algorithm 2 shows how the FRC guides to optimize statement.

4. EVALUATION

In this section, we present details on the experimental results of our approach, by shrinking the statements of common applications. We first describe the environment in which we conducted our experiments. Then, we show the effectiveness of our approach to construct FRC by summa-



```
/*cmp   %cl, %al*/
/*......Initializer......*/
T_14_1182:reg32_t = T_32t3_1180:reg32_t -
T_32t4_1181:reg32_t& 0xff:reg32_t;
R_CF_10:reg1_t = T_32t3_1180:reg32_t <
T_32t4_1181:reg32_t;
T_15_1183:reg8_t = cast (T_14_1182:reg32_t) L:
reg8_t;
R_PF_11:reg1_t =
! Cast(((T_15_1183:reg8_t >> 7:reg32_t
^ T_15_1183:reg8_t >> 6:reg32_t)
^ (T_15_1183:reg8_t >> 5:reg32_t
^ T_15_1183:reg8_t >> 4:reg32_t))
^ ((T_15_1183:reg8_t >> 3:reg32_t
^ T_15_1183:reg8_t >> 2:reg32_t)
^ (T_15_1183:reg8_t >> 1:reg32_t
^ T_15_1183:reg8_t))) L: reg1_t;
R_AF_12:reg1_t =1:reg32_t==
(0x10:reg32_t&(T_14_1182:reg32_t
^ (T_32t3_1180:reg32_t^ T_32t4_1181:reg32_t)));
R_ZF_13:reg1_t = T_14_1182:reg32_t == 0:reg32_t;
R_SF_14:reg1_t = 1:reg32_t == (1:reg32_t &
T_14_1182:reg32_t >> 7:reg32_t);
R_OF_15:reg1_t =1:reg32_t  == (1:reg32_t
& ((T_32t3_1180:reg32_t ^ T_32t4_1181:reg32_t)
& (T_32t3_1180:reg32_t ^ T_14_1182:reg32_t)) >>
7:reg32_t);
/*......*/
```

```
/*cmp %cl, %al*/
/*......Initializer......*/

T_14_1182:reg32_t = T_32t3_1180:reg32_t
- T_32t4_1181:reg32_t& 0xff:reg32_t;

R_ZF_13:reg1_t = T_14_1182:reg32_t == 0:reg32_t;

/*......*/
```

Fig. (4). A simplified example of the optimization of the statement of *cmp*.

Table 1. Summary of the Experimental Results. The first column of the table lists the samples and the second column shows the size of the samples in assembly instructions. The third and fourth columns give the number of flag dependency instructions and flag relevant instructions respectively. The fifth column shows the original statement for each sample. The last column shows the results of statement after optimization.

Sample	Assembly Instruction	Flag Relevant Instructions	Flag Dependency Instructions	Original Statement		Statement after Optimization	
	Line	Line	Line	Size	Line	Size	Line
Gcov	3846	731	349	2.8MB	94255	1.6MB	52838
Gcc	15218	2957	1732	11.3MB	382888	5.9MB	217595
Bzip2	18584	4364	1532	16.2MB	532957	7.6MB	281841
Bzip2recover	1068	187	75	759.3KB	24095	431KB	15627
Nhmmer	272425	57416	22207	230.9MB	7586043	115.6MB	4296532
Hmmemit	212880	45511	17653	181.6MB	5966370	90.5MB	3359281

rizing the information of flag dependent instructions and flag relevant instructions. Last, we evaluate the proportion of eliminating redundant Eflags by our approach.

4.1. Overview

Our approach is written in a mixture of C and Python and consists of 3 major components: assembly instruction dependency, invertible analysis, and IL optimization. We chose Vine IL as our intermediate language, and added about 3000 lines of code to implement our algorithms and heuristics as well as to add in support for other IL. We used STP for constraint solving and chose IDA pro 5.2 as our disassembler from which we took code segment of each sample to analyze.

We evaluated our algorithms on a 4GHz Intel(R) Core 5 Duo Linux workstation with 4GB of RAM running Ubuntu 10.04.We measure the effectiveness of our approach on statement optimization with SPECint 2006[16]. Our sample set presented in the paper are consisted of six samples, which are unmodified applications that people use and can be downloaded from the Internet. The results of the evaluation are summarized in Table **1** and described in more detail in the remainder of the section.

4.2. Detailed Analysis

In the experiment, our approach has successfully optimized statement for all the samples. We summarize the results in Table **1**. In the second column of Table **1**, we list the line number of assembly instructions from IDA pro for each sample excluding the comment and blank lines. The line number of assembly instructions varies from 1068 to 272425 and almost includes the four groups of flag dependency instructions as illustrated in Section 3.2.

The third column gives the number of flag relevant instructions for each sample. We found that flag relevant instructions can be divided into two categories according to whether it is related to any flag dependency instruction. That is to say, redundant Eflags have two sources: the I_{fri} not related to any I_{fdi} (all the Eflags are irrelevant with subsequent instructions), the I_{fri} related to some I_{fdi} but only a few Eflags are used by the I_{fdi}. In particular, the line number difference between I_{fri} and I_{fdi}, to a large extent, determines the efficiency of optimization. That the two programs have roughly the same number of assembly instructions, the more the I_{fri} is not related to any I_{fdi}, the more obvious optimization is. As is shown in the Table **1**, the program gcc and bzip2, the difference between I_{fri} and I_{fdi} of the later is more than the former, thus the optimization proportion of the later is 53.1% more than the former i.e. 47.8%.

The fourth column shows the number of flag dependency instructions for each sample. In our implementation, we first construct a hash table to categorize the flag dependency instructions and record Eflags that are referenced, which is the basis for FRC that consists of flags that each instruction relies on. We use the instruction name as key value *k*, and with the hash function *f* the index of the instruction is *f(k, array_size)*. Each flag dependency instruction corresponds to a node in FRC and the node consists of concrete information about Eflags including its number and its belonging to which flag relevant instruction.

The fifth column lists the fundamental information about each sample after translating to IL. It consists of two subcolumns: the size and the line number of the statement from Vine. As we can see, the maximum line number in the table is 7586043 while its line number of assembly instruction is 272425, which shows the volume of the statement is so enormous and a large number of IL files will challenge the storage performance. Therefore, statement optimization seems particularly important.

The last column gives the size and the number of the statement optimized by our approach. The size of statement after eliminating redundant Eflags varies from 1.2 MB to 91.1MB, depending on the FRC size, the number of flag relevant instructions and other factors. In total, our approach decreases by 43% line on a sample during the evaluation, which is significant compared to original statement that has millions of lines.

5. LIMITATION AND FUTURE WORK

The first, and the most obvious, limitation is the complete support of the Vine IL to assembly instruction. In our experiment, not all the assembly instruction can be translated into the Vine IL, when we lifted the assembly instructions of the program nhmmer to Vine IL, it failed to handle floating point and privileged instruction. Though we attempted to use a newer version of VEX library that is responsible for this problem, yet the problem persisted. We thus had to manually locate these instructions. It is not possible to prove the correctness of statement if the semantics of the x86 ISA are not formally defined.

The second limitation of our approach is the efficiency of construction of FRC. Currently, our implementation of invertible analysis follows linear-sweep by the scanner from the last instruction. A major challenge of analyzing real-world applications is that we need to process a huge amount of assembly instruction. To very large and complex applications, the number of flag dependent instruction is also enormous and thus we need more time to deduce the relationship between flag dependent instructions and corresponding flag relevant instructions. Our paper is a step forward in optimizing IL to enhance the efficiency of subsequent analysis, but obviously, much more work remains to be done.

The third limitation of our approach is from Vine itself. Through trial and error, we found that Vine failed to translate enormous assembly instructions into statement due to an out-of-memory error. With this problem, we were unable to use large and complex applications generally with millions of assembly instructions as samples. To address this problem, we need to modify the source code of Vine, and reason about the procedure of IL translation. In the future, we are going to develop a more robust and secure IL

6. RELATED WORK

Intermediate language is designed to describe the intermediate representation of the program. Several program analysis platforms have their intermediate languages. Vine is the static analysis component of BitBlaze Binary Analysis Platform and has its intermediate language Vine IL that is the base of subsequent analysis [12]. Vine IL is independent of the platform and the semantics of the IL are designed to be faithful to assembly languages. Vine used VEX which is part of the Valgrind dynamic instrumentation tool [17] and similar to a RISC-based language to provide a rough IL for each instruction and augments it to expose all otherwise-implicit side effects. Vine IL makes all side effects explicit by default, many of which may not matter for a particular analysis. However, the optimization removes code that does not affect the program results or unreachable code that can never be executed.

Since Vine IL lacked a formal semantics for the IL itself, and did not handle bi-endian architectures such as ARM correctly. BAP is a complete redesign of Vine that encompasses lessons learned from previous work on binary analysis and also a binary analysis platform developed by Carnegie Mellon University[18].The core of BAP is the BAP intermediate language, called BIL[6]. BIL only has a few language constructs, which makes it easy to analyze. In addition, BIL also explicitly represents side effects such as flag computations. To address the issue of redundant flags, BAP uses dead code elimination to remove irrelevant OF, SF, ZF, AF and PF. Nevertheless, the optimization is only applicable to BIL and does not work on other intermediate languages.

Phoenix is a program analysis environment developed by Microsoft as part of their next generation compiler [4]. One of the Phoenix tools allows code to be raised up to a register transfer language (RTL). A RTL is a low-level IR that resembles an architecture-neutral assembly. Phoenix lifts assembly to a low-level IR that does not expose the semantics of complicated instructions, e.g., register status flags, as part of the IR [19]. Phoenix optimizes RTL based on the logic of the program, thus is not suitable for our research purposes.

In comparison, our approach captures the intrinsic characteristics of IL redundancy: not all the Eflags are relevant with a particular analysis. Therefore, we can abstract a general approach for various IL that makes all side effects explicit to eliminate redundant flags. Moreover, it also provides insights about the IL optimization.

CONCLUSION

In this paper, we presented a novel invertible analysis approach, eliminating redundant Eflags based on the FRC, to optimize IL. Through analyses with a broad spectrum of different IL, we found that a couple of IL represent all side effects explicit by default and not all the Eflags are relevant with subsequent analysis. Thus, we give algorithms for building the FRC given only a stripped binary program and remove redundant Eflags based on the FRC to shrink the IL. The experimental results demonstrated that our approach could diminish the volume of Vine IL obviously, and provide accurate representation of the assembly code.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

ACKNOWLEDGEMENTS

We thank Quanchen Zou and Yu Han for suggestions and help related to the experiments and previous papers. This work is supported by National Natural Science Foundation of China (No. 61121061), National Key Technology R&D Program (2012BAH37B05) and National Key Technology R&D Program (2012BAH06B02). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Natural Science Foundation of China or other supporters.

REFERENCES

[1] K. Anshumali, T. Chappell, and W. Gomes, "Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes1-5 (April 2008)," *Intel Technology Journal*, vol. 14, pp. 104–127, 2010.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—a platform for analyzing x86 executables," in *Compiler Construction*, pp. 250-254, 2005.

[3] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," in *Computer Aided Verification*, pp. 288-305, 2010.

[4] "Microsoft. Phoenix framework. [Online]. Available:http://research.microsoft.com/phoenix/. URL checked 4/21/2011.

[5] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Computer Aided Verification*, pp. 423-427, 2008.

[6] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, pp. 463-469, 2011.

[7] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Information systems security*, Springer, pp. 1-25, 2008.

[8] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Software-Practice and experience*, vol. 28, no. 8, pp. 859-882, 1998.

[9] A. W. Appel, "SSA is functional programming," *SIGPLAN notices*, vol. 33, no. 4, pp. 17–20, 1998.

[10] "DataRescue. IDA Pro.(Page checked 7/31/2008)." [Online]. Available: http://www.datarescue.com.

[11] "Static Disassembly of Obfuscated Binaries." [Online]. Available: http://static.usenix.org/event/sec04/tech/full_papers/kruegel/kruegel_html/disassemble.html.

[12] "Vine: The BitBlaze Static Analysis Component." [Online]. Available: http://bitblaze.cs.berkeley.edu/vine.html.

[13] Dawn Song, David Brumley, Heng Yin, "BitBlaze: A New Approach to Computer Security via Binary Analysis," Proc. 4th International Conference, ICISS 2008, Hyderabad, India, pp. 1-25, 2008.

[14] B. Team, "Vine installation and user manual," August 26th, 2009..

[15] ARM. *ARM Architecture Reference Manual* (2005) Doc. No. DDI-0100I. .

[16] "Standard Performance Evaluation Corporation, 'SPEC CPU2006.'"[Online]. Available: http://www.spec.org/cpu2006/CINT2006/.

[17] N. Nethercote, "Dynamic binary analysis and instrumentation," PhD thesis, University of Cambridge, 2004.

[18] "The Binary Analysis Platform from Carnegie Mellon University." [Online]. Available: http://bap.ece.cmu.edu/.

[19] "Microsoft. Phoenix project architect posting (Page checked 7/31/2008) (July 2008)," [Online]. Available: http://forums.msdn.microsoft.com/en-US/phoenix/thread/ 90f5212c-05a-4aea-9a8f-a5840a6d101d.

Received: September 16, 2014 Revised: December 23, 2014 Accepted: December 31, 2014