Int. J. Nonlinear Anal. Appl. (10) No. 1, 9-13
ISSN: 2008-6822 (electronic)
http://www.ijnaa.semnan.ac.ir

Analysis of disassembled executable codes by abstract interpretation

Mohammadhadi Alaeiyan[a], Saeed Parsa[a,*]

[a]School of Computer Engineering, Iran University of Science and Technology, Narmak, Tehran 16846, Iran

(Communicated by Madjid Eshaghi Gordji)

Abstract

The aim of this paper is the definition of the abstract domain, abstract operator, abstract semantic, environments and states of disassembled executable codes. They help us to analysis the disassembled executable codes. Static analysis on the disassembled code is a popular task and reverse engineers and malware analyzers leverage this technique to apply a fast scan on the codes. In this paper, we perform a summarization on the requirements of abstract interpretation and present an algorithm to approximate the range value of the code variables.

Keywords: Static analysis, Abstract domain; Abstract operator; Abstract semantic; Environment and state of abstract interpretation
2010 MSC: 11Y16, 37C25.

1. Introduction

According to a widely recognized definition: "Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems" [3]. Abstract interpretation presents an idea to approximate the concrete semantics of a program by a sound or complete abstract semantics. Thus, the results of abstract execution perform information about the actual computations of a program. The concrete and abstract domains are related through a Galois connection [8] which defines the soundness and completeness of the information. the information is completeness if there are not different between concrete and abstract domain. Otherwise, the information is soundness.

Static analysis by abstract interpretation is a technique to collect information about the code. This technique uses on optimization [4], software protection, security [6, 7, 8], software testing [2].

*Corresponding author
Email addresses: hadi_alaeiyan@comp.iust.ac.ir (Mohammadhadi Alaeiyan), Parsa@iust.ac.ir (Saeed Parsa)

Received: October 2018 *Revised:* June 2019

However, this technique almost relies on the source code. A disassembled executable code of a program is not complete source code. Therefore, determining a complete abstract domain, abstract operator, abstract semantic, environments and states, will be helpful to interpret the disassembled executable code. While Cousots, in paper [1] defines a general interval abstract domain and its operators, it lacks prominent information about disassembled executable codes. Subsequently, we will describe that performing static analysis needs an interval CPU simulator and an interval analyzer to fulfill this analysis. The disassembled code analysis has some crucial problems like indirect jumps and indirect calls. In this paper, we attempt to clarify how these problems can partly be solved. In addition, various types of instructions and data such as stack, heap, and memory management are other challenge-able issues that in the following will be discussed.

The remainder of this paper is organized as follows: domain, operator, semantic are three variables of abstract interpretation that their concern and abstracts are defined in sections 2, 3, 4. In addition, Section 5 is included the environment and state of the disassembled executable code and presented a way to analyze the disassembled executable code.

2. Abstract Domain

In general, CPUs have a 64-bit flags register, 16 general 64-bit registers, 16 `SIMD` 256-bit registers, 8 floating-point 80-bit registers and memory that contains stack and heap [5]. Programs leverage registers to perform their task. The best abstract domain to analyze instruction of the disassembled code is interval abstract domain. Because not only it satisfies our requirements by solving disassembly problems, but also offers a lattice of possible values for program variables. The possible values can be modeled by an interval.

The first problem of variable interval analysis is the variable conjunction. In assembly, a variable could be split into several sub-variables. As a case point, there are 64-bits for each register such as `rax` that contains five variables {`rax, eax, ax, ah, al`}. It means that if `al` register is changed, then other related registers will change automatically. Consequently, the abstract domain has the flexibility to separate each of rax, eax, ax, ah, al from its interval. In compassion to the interval of the abstract domain, other abstract domains such as octagon cannot fulfill this analysis because of data relationship amongst domains. Thus, it is impossible that these domains split into their smaller parts.

Moreover, a `SIMD` 256-bits register can be split into two 128-bits which they also can be split into four 64-bits and so on. Therefore, the intervals can easily cover this problem. Two objects `__m256i` is used for implementation at C++ language. They are leveraged for keeping the upper and lower bound of the interval. Let L be a lattice. A subset I of L is called a lattice interval if there exist elements $a, b \in L$ such that $I = \{t \in L | a \leq t \leq b\} = [a, b]$. The elements a, b are called the endpoints of I. Clearly $a, b \in I$. Therefore, our abstract domain is a lattice interval. A variable is able to keep its data in a range $[a, b]$ for each instruction.

3. Abstract Operator

To simulate the CPU instructions, we disassemble programs by leveraging BeEngine. Thus, CPU instructions are the concern operators of the abstract interpretation. Therefore, the abstract operator is a simulation of CPU instructions that each instruction works on interval values. That means each input and output of each instruction is an interval. Thus, around 500 instructions must be simulated.

The accuracy of the simulator shows the soundness and completeness of the abstract operators. The operators add, subtract, multiplication and divide are soundness and completeness in interval

domains. Therefore, whether bit-operators are completeness and soundness? For instance, $[a, b] + [c, d] = [minimum(a + c, a + d, b + c, b + d), maximum(a + c, a + d, b + c, b + d)]$ and the result is completeness. But, computers suffer from the overflow problem which it makes problems for abstract operation. Now if we pay attention to the `XOR` operator. The formula that exists for add operator, it does not have complete support for the `XOR` operator. In particular, suppose that the value of register `eax` is $[8, 10]$ and the register `ebx` is $[20, 30]$ for the instruction `xor eax, ebx`. The formula gives the result $[20, 30]$. This interval does not false, but its accuracy is low. This abstract operator has soundness but it does not have completeness. It is possible to achieve the completeness with paying attention to $[8\, xor\, 20, 10\, xor\, 30]$ therefore the result is $[28, 20]$. Know we break down the definition of intervals.

To continue, suppose the next instruction is `add eax, ebx` and value of `eax` is $[28, 20]$. Thus, its output is $[48, 50]$ that its accuracy is better than $[40, 50]$. Experimentally, we achieve that the results will be fixed when the next instruction fulfills its action.

4. Abstract Semantic

As observed earlier, one interest of abstract interpretation theory is the systematic design of semantic approximation of programs. As you understand, concern semantic of this analysis is the semi-language of assembly language. Therefore, abstract semantic is a brief grammar of this language by containing the abstract semantics. Abstract semantic is a collection of intervals which is dependent on variables of each code instruction.

5. Environments and States and Analysis

Each environment keeps information about each state of the code instruction. In this issue, each instruction has a state. The information that needs to keep are all registers (contain floating-point and SIMD registers), flags and memory heap, stack, local and global variables. The state keeps the information environment and the instruction of the disassembled code. In addition, it keeps children and parent states.

After conducting a perfect disassembly, the process continues with providing a control flow graph (CFG). Since advancing CFG some time is impossible due to problems such as indirect jumps or calls. Therefore, this process mixed with analysis to make a better CFG. In the beginning, the entry point of code should find. Table 1 shows the algorithm that does abstract interpretation and makes CFG. Two functions that exist in this algorithm are important for us. `AI_static_analysis` and `getItsJumpLocation`.

P. Cousot et al. [1] perform static analysis. However, there are some different notes that satisfy our requirements. Function calls, system calls, stack management, and global memory makes some problem in our analysis. Hence, calls need more details, calls need two environments, one environment for the output and another environment for states of is called function how keeps information of all time that the function called. Table 2 shows an example of this issue, INC function called two times and each time has an environment for output and one environment for keeping the union of two times.

In addition, calls have some inputs, outputs, and local variables. All of them are stored in the stack. Therefore, every time every point of the stack can use. For an example in C++ compiler, after calls, separate some stack space for local variables and the analyzer does not know that what the data types of this local variable are to separate and determine these local variables.

System calls are another problem that decreases the number of fixed points as well as decrease the accuracy of the analysis. The idea is to simulate some general system calls. In windows OS,

Table 1: Disassembled executable code analysis algorithm

```
A recursive algorithm for disassembled executable code analysis
Inputs : FPath, the text file of disassembled executable code.
EPOINT = Entry point label.
Output: CFG, control flow graph of disassembled executables code with some analysis
information.
Begin
    Size = 0.
    CFG cfg().
    Instruction ini.
    While(true)
    Begin
        ini = read(EPOINT);
        if (finish(ini)) then break;
        AI_static_analysis(ini); // fulfill abstract interpretation on the instruction ini.
        If(isjump(ini)) then
        Begin
            Address [] REIP = getItsJumpLocation(Code, out Size);
            for( int i=0, i¡ Size;i++)
                ncfg = Algorithm_analysis_of_disassembled_executable_code(FPath,REIP[i])
                cfg.add(ncfg);
        End //end if (isjump(ini))
        Else
            EPOINT = label_Next_code();
            cfg.add(Code);
        End // else
        End // end if (finish(ini)).
    End // end while.
End
```

there are around 500 system calls that are important for this analysis. However, there are some `Dll`s or other files. The idea is the disassembling of `Dll`s and analyzing them too.

References

[1] Patrick Cousot and Radhia Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of the 4th ACM Symp. on Principles of Programming Languages (POPL 77), New York, (1977).

[2] Patrick Cousot and Radhia Cousot. *Abstract Interpretation Based Program Testing*, Proceedings of the SSGRR 2000 Computer and eBusiness International Conference, (2000).

[3] Patrick Cousot, *Abstract interpretation*, ACM Computing Surveys, 28 (1996) 324-328.

[4] Saumya Debray, *Abstract interpretation and Low-Level Code Optimization*, Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. ACM, (1995).

[5] Intel. Intel 64 and IA-32 Architectures Software Developers Manual. Intel. (2013).

[6] Mila Dalla Preda, *Code Obfuscation and Malware Detection by Abstract Interpretation.* Verona: Universit'a di Verona. (2005).

[7] Mila Dalla Preda, and Roberto Giacobazzi. *Semantic-Based Code Obfuscation by Abstract Interpretation*, Automata, Languages and Programming. Springer Berlin Heidelberg, 3580 (2005) 1325-1336.

Table 2: An example of disassembled executable code analysis

Address	Instruction	Environment
	// function inc	
04124320	pop ebx	$ebx^1 = [10, 10]$, $ebx^2 = [20, 20]$. $\Rightarrow [10, 20]$
	inc ebx	$ebx^1 = [11, 11]$, $ebx^2 = [21, 21]$. $\Rightarrow [11, 21]$
	push ebx	Top of $Stack^1 = [11, 11]$,
		Top of $Stack^2 = [21, 21]$. $\Rightarrow [11, 21]$
	ret	
	// function main	
04125320	mov eax, 10	eax = [10, 10].
	push eax	Top of Stack = [10, 10].
	call 04124320	
	pop ebx	ebx = [11, 11].
	mov eax, 20	eax = [20, 20].
	push eax	Top of Stack = [20, 20].
	call 04124320	
	pop ebx	ebx = [21, 21].

[8] Ying Zeng and Fenlin Liu. *Abstract Interpretation-based Formal Description of Data Obfuscation*, International Conference on Electonic and Mechanical Engineering and Information Technology, (2011).