A Hybrid Approach for Control Flow Graph Construction from Binary Code

Minh Hai Nguyen¹, Thien Binh Nguyen² and Thanh Tho Quan³

Faculty of Computer Science and Engineering
Hochiminh City University of Technology
Hochiminh City, Vietnam
¹55307910@hcmut.edu.vn, ²551105019@stu.hcmut.edu.vn, ³qttho@cse.hcmut.edu.vn

Mizuhito Ogawa
School of Information Science,
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
mizuhito@jaist.ac.jp

Abstract—Binary code analysis has attracted much attention. The difficulty lies in constructing a Control Flow Graph (CFG), which is dynamically generated and modified, such as mutations. Typical examples are handling dynamic jump instructions, in which destinations may be directly modified by rewriting loaded instructions on memory. In this paper, we describe a PhD project proposal on a hybrid approach that combines static analysis and dynamic testing to construct CFG from binary code. Our aim is to minimize false targets produced when processing indirect jumps during CFG construction. To evaluate the potential of our approach, we preliminarily compare results between our method and Jakstab, a state-of-the-art tool in this field.

Keywords: binary code analysis, static analysis, dynamic analysis, SMT, symbolic execution, control flow graph construction

I. INTRODUCTION

There are several reasons to choose binary code as a program analysis target. First, once source codes are lost or unavailable, we need to directly analyze binary codes. Third party modules and computer virus are such examples. Second, a serious issue emerges from compiling from source codes to binary codes. A compiler may remove certain behaviors of programs, hence altering its contents or even its semantics [1].

Recently, there are a lot of tools and prototypes introduced for analyzing binary code. BINCOA [2] offered a framework for binary code analysis. Its core technology is a refinement-based static analysis [6] by abstract interpretation [7]. IDA Pro [3] is commercial software, which has been used in many binary analysis platforms. Remarkably, Jakstab [4][5] is a state-of-the-art tool in the field of binary code analysis. It translates binary codes to a low level intermediate language in an on-the-fly manner and performs further analysis accordingly.

Fig.1 shows four major steps. The first step translates binary codes to disassembly codes. The second step builds an intermediate representation (IR) from the disassembly codes. The third step constructs the Control Flow Graph (CFG), whose vertices represent basic blocks of instructions and directed edges represent jumps in control flow [10]. Based on the constructed CFG, other analysis utilities like malware detection or security checking will be further provided.

In Fig.1, the CFG construction step plays an essential role. Whereas CFG construction at an imperative language level is a classic work, that at the binary code level still remains a challenging task due to the following obstacles.

The first challenge lies in Complex Instruction Set Computer (CISC) [11] architectures, such as x86. They have very rich instruction sets, e.g., hundreds of instructions and thousands of operands combination in x86 architecture [12]. All of them must be interpreted properly to construct a CFG. The second challenge lies in the lack of desirable properties of high level semantic structure. For instance, there are no function abstraction and/or type at binary code level. Moreover, the issues of code and data ambiguity, indirect branches and overlapping instructions [13] are also burden. Most of existing tools use static analysis with an over-approximation, resulting in a CFG with more false targets.

Inspired by [29], this paper proposes a hybrid approach which combines static analysis and dynamic testing for generating CFG from binary codes. We apply standard intra-procedural CFG generation until indirect jumps and/or function calls occur. Then, test data are generated to decide their precise destinations. Different from [29], we apply symbolic execution to generate appropriate test data. This hybrid method is neither sound nor complete, but will give a practically more precise CFG (even with mutation), compared to abstract interpretation based static analysis.

Fig. 2. CFG reconstructed by over-approximation abstraction
The rest of the paper is organized as follows. Section II briefly describes a motivating example which shows problems of an over-approximation approach. Section III illustrates the overview of our hybrid framework. Section IV discusses in more detail our running examples to clarify the advantages of our method. Section V illustrates our research challenges in the subsequent PhD project. Section VI shows our preliminary evaluation. Related works are presented in Section VII, and Section VIII concludes the paper.

### II. MOTIVATING EXAMPLE

Fig. 2 presents an example illustrating the drawback of the over-approximation approach. We consider a code fragment starting at *Instruction 0*, where variable *x* is given a value randomly picked up from a set of \{10, 15, 30\}. When we convert this program into an abstract form, a typical approach is to use an interval to represent possible values of variables. In this case, the abstract value of *x*, denoted as \(\alpha(x)\), is represented as an interval of \([10, 30]\).

The major problem occurs when value of *x* is used as the target address of an *indirect jump* instruction at line 6. In the abstract program, since *x* can take any values in the interval of \([10, 30]\), there are several other false branches which may be produced, illustrated as the dotted arrows in Fig. 2. They come from an over-approximation based abstract interpretation.

There are many approaches to give a better abstraction. The false branch problem of an over-approximation is inevitable, and this is crucial for CFG generation of binary codes. For instance, mutation tries to lead such false branching. This issue motivates us to consider a new hybrid approach.

### III. THE PROPOSED FRAMEWORK

Fig. 3 describes our framework, which consists of two phases: *Static Analysis* and Dynamic Analysis. They are executed alternatively until the CFG converges.

In this framework, a program to be analyzed is divided into *regions*. Each region is a block of instructions which contains no dynamic jump instructions. In the *Static Analysis* phase, we apply Symbolic Execution (SE) [14] to reconstruct execution paths in one region and create the corresponding sub-CFG. This process of SE is performed in forward manner until encountering an indirect branch.

When encountering a dynamic jump, we execute *Path Condition Solving* to solve path conditions associated with an execution path in the current region. Then, test-cases are generated to cover all execution paths. In the meantime, the sub-CFG of the current region will be updated.

Subsequently, the *Dynamic Analysis* phase will be executed. In this phase, firstly the *Model Construction* converts the CFG into an intermediate labeled transition system (LTS). The test-cases are executed in this intermediate LTS as *Test-case Execution* step. It allows us to verify real targets of dynamic jumps, which update the current CFG. If they jump into new areas, which are not explored yet, the Static Analysis phase is invoked again. Such combination of static and dynamic analysis is repeated until no new areas are discovered.

### IV. EXAMPLES

**Example I: Handling dynamic jump**

Fig 4 shows our first example, which starts at *start* and introduces an indirect jump at *Instruction 8*. By static analysis, two execution paths leading to this dynamic jump are easily determined, i.e., \(P_1 = (\text{start} \rightarrow 0 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8)\) and \(P_2 = (\text{start} \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8)\). For an initial value \(\alpha\) of register *eax*, the path conditions of \(P_1\) and \(P_2\) are evaluated to \((\alpha < 0)\) and \((\alpha \geq 0)\), respectively.

We apply a prover to generate two test-cases corresponding to these path conditions, say, \(a_1 = 1\) and \(a_2 = 2\). By executing the program with them, sound targets of indirect branches are determined to be *start* and *Instruction 6*. By continuing the path execution from *Instruction 6*, *Instruction 4* is discovered as a new target of the indirect jump at *Instruction 8*. The full CFG is illustrated in Fig 4, where the
dotted arrows indicate new edges discovered by Dynamic Analysis phase.

**Example II: Combination of multiple regions and handling dead code**

In this example, we extend Example I to illustrate a more complex scenario. Table 1 presents the program to be analyzed, which contains two indirect jumps at Instruction 7 and Instruction 16. In addition, this code uses an obfuscation by inserting dead code from Instruction 17 to Instruction 19.

This example describes our idea that each executed dynamic jump creates a new region in the program. Figure 5 illustrates our construction of CFG complying with this strategy. First, the CFG of Region 1 (corresponding to the code from Instruction 0 to Instruction 7) is extracted by the method as described in Example 1. By generating test-cases and executing the indirect jump at Instruction 7, we discover a new region starting at Instruction 8.

![Fig. 5. Inter-region strategy of CFG construction](image)

The next challenge is to handle path conditions associated to each execution path of source binary code during symbolic execution. Provers (SMT) solve the path conditions for test-case generation. The challenge encountered here is the computational limitation of provers. Current provers mostly cover only linear constraints for arithmetic. At binary code level, the types are arithmetic and the challenge lies in non-linear constraints, such as Z3 4.3 [20] and raSAT [18].

The next challenge is to infer loop invariants. This is a classic issue, and recently two methodologies (and their combinations) are popular. (1) Loop invariants in arithmetic. For a linear loop invariant, the technique based on Farkas' lemma [15] is common. For non-linear equational invariants, an algebraic method is known [33]. (2) Loop invariants in first-order logic. Craig Interpolation is known to be a good strategy to produce loop invariants [19].

The last challenge is to simulate the program execution by Dynamic Analysis on the current CFG. We intend to apply model checking, since the conversion from a CFG to a Labeled Transition System (LTS) is fairly straightforward. The key is, how to make model checking terminate, since test data are in an infinite domain, such as Integers. Currently, we use Promela of SPIN, which accepts arithmetic expressions and

### Table 1 – A binary code consisting of multiple dynamic jumps and dead code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: cmp eax, 0</td>
<td>0</td>
</tr>
<tr>
<td>1: jl then1</td>
<td>1</td>
</tr>
<tr>
<td>2: mov ax, offset start + 1</td>
<td>2</td>
</tr>
<tr>
<td>3: jmp lcont1</td>
<td>3</td>
</tr>
<tr>
<td>4: halt</td>
<td>4</td>
</tr>
<tr>
<td>5: mov eax, offset l1 + 4</td>
<td>5</td>
</tr>
<tr>
<td>6: sub eax, 3</td>
<td>6</td>
</tr>
<tr>
<td>7: sub eax, 1</td>
<td>7</td>
</tr>
<tr>
<td>8: jmp eax</td>
<td>8</td>
</tr>
<tr>
<td>9: jl lthen2</td>
<td>9</td>
</tr>
<tr>
<td>10: mov eax, offset lstart + 1</td>
<td>10</td>
</tr>
<tr>
<td>11: jmp lhalt</td>
<td>11</td>
</tr>
<tr>
<td>12: ret</td>
<td>12</td>
</tr>
<tr>
<td>13: mov eax, offset l2 + 6</td>
<td>13</td>
</tr>
<tr>
<td>14: sub eax, 5</td>
<td>14</td>
</tr>
<tr>
<td>15: sub eax, 1</td>
<td>15</td>
</tr>
<tr>
<td>16: jmp eax</td>
<td>16</td>
</tr>
<tr>
<td>17: cmp ebx,eax</td>
<td>17</td>
</tr>
<tr>
<td>18: jz l4</td>
<td>18</td>
</tr>
<tr>
<td>19: jmp eax</td>
<td>19</td>
</tr>
</tbody>
</table>
generates a model in an on-the-fly manner. We set the termination condition as to either reach to a final state or find a new target destination of an indirect jump. This does not guarantee termination, e.g., a loop that contains operations to increment. If it fails, we simply apply time-out. Alternatively, if an LTS gets stuck with a certain input value, i.e., it cannot determine the destination of a transition, the original CFG needs to be enlarged with that input value.

VI. SMALL EXPERIMENTS

Small experiments are performed to evaluate the feasibility of our method with 5 example programs1 (same shown in Table 2) under the following constraints: (i) the code contains indirect jumps and (ii) the loop conditions are linear, which allowed us to handle them using Farkas’ lemma. These experiments have been carried out in a Core i5-3340M computer with 4GB RAM. Our experiments are carried out in the following steps:

(1) From given assembly code, we produce an initial CFG that consists only program entry.

(2) We perform the intra-procedural CFG construction (or static CFG construction since we only process static jump instruction in this step) such that a CFG have program instructions as its vertices. There are 6 kinds of vertices defined, including Start, Exit, Condition, Join, Loop and Other Instruction. Other Instruction vertices cover the arithmetic instructions and move instructions of the assembly code.

We apply Jakstab to construct intra-procedural CFG. By default, Jakstab implements an on-the-fly method of static analysis on binary source code. Once encountering indirect branches, it applies abstract interpretation in order to resolve possible target addresses. Since Jakstab is an open source software, we automatically replace this step as to stop Jakstab when a dynamic jump is found, and apply step (3).

(3) We perform the symbolic execution on the current CFG. In order to do that, we build a simple symbolic execution framework to handle a subset of x86 instructions. For each edge in the CFG, we compute a symbolic condition, which is a necessary condition to have an execution path through this edge. For instance, Fig.6 illustrates the symbolic conditions generated when handling a Condition vertex of the CFG. Symbolic conditions for Start, Exit, Condition, Join, and Other Instruction vertices are straightforward. For Loop vertices, Farkas’ lemma is used to infer a loop invariant.

(4) We apply Z3 4.3 [20] to solve path conditions (given in step (3)) which are associated to paths reaching to the indirect jump vertices, and to generate test-cases to cover them.

(5) We use PAT [17] to generate a LTS from the current CFG and dynamically perform the test-cases on it. After PAT is performed, the outputs of PAT for estimating the targets of the indirect jumps. Then, the CFG is enlarged with the estimated targets.

(6) If the CFG is enlarged in step (5) with fresh vertices, return to step (2). Otherwise, the construction finished.

Fig.7 illustrates CFGs generated by Jakstab and our method in one testing program. The average runtime of Jakstab to process a program is less than one seconds. Although the test programs are just toy programs, Jakstab still fails to resolve the target addresses of dynamic jumps. For the program in Table 2, the CFG generated by Jakstab stopped at the indirect jump at location 21. Using our approach, the analysis process proceeds and achieves the full CFG.

Table 3 - Experimental results

Table 3 gives the summary of our experiments. In this table Inst implies the number of instructions in the original program, J-Inst and H-Inst are the numbers of instructions actually reachable using Jakstab and our approach respectively, for reader convenience, we also compute the value of Cvg., which implies the percentage of instruction coverage by the generated CFGs. One can observe that our
approach can significantly improve the quality of the generated CFG, as compared to Jakstab. However, our approach suffers from higher computational cost, thus consuming more execution time.

VII. RELATED WORKS

1. Hybrid approaches for program analysis

The approach of using hybrid method by combining static analysis and dynamic testing to analyze imperative programs has been considered in many related works. In the field of software testing, concolic testing [21][22][23][24] is a well-known technique, which combines symbolic execution and dynamic execution to generate test-case.

Table 2 – Source code of the experimental file

<table>
<thead>
<tr>
<th>.data</th>
<th>.code</th>
<th>.data</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter db 0h</td>
<td>18: inc counter</td>
<td>18: inc counter</td>
</tr>
<tr>
<td>.code</td>
<td>19: cmp counter, 2</td>
<td>19: cmp counter, 2</td>
</tr>
<tr>
<td>start: ; Entry point</td>
<td>20: je l3</td>
<td>20: je l3</td>
</tr>
<tr>
<td>0: jmp l2</td>
<td>21: jmp ebx</td>
<td>21: jmp ebx</td>
</tr>
<tr>
<td>1: inc edi</td>
<td>22: shr eax,31</td>
<td>22: shr eax,31</td>
</tr>
<tr>
<td>2: mov edi, 1</td>
<td>23: add eax, 401043h</td>
<td>23: add eax, 401043h</td>
</tr>
<tr>
<td>3: cmp al, 2</td>
<td>24: cmp ebx, eax</td>
<td>24: cmp ebx, eax</td>
</tr>
<tr>
<td>4: jle l1</td>
<td>25: je l4</td>
<td>25: je l4</td>
</tr>
<tr>
<td>5: nop</td>
<td>26: jmp eax</td>
<td>26: jmp eax</td>
</tr>
<tr>
<td>6: nop</td>
<td>27: nop</td>
<td>27: nop</td>
</tr>
<tr>
<td>7: nop</td>
<td>28: add eax,ebx</td>
<td>28: add eax,ebx</td>
</tr>
<tr>
<td>8: nop</td>
<td>29: sub ebx,eax</td>
<td>29: sub ebx,eax</td>
</tr>
<tr>
<td>9: cmp al, 2</td>
<td>30: jmp l5</td>
<td>30: jmp l5</td>
</tr>
<tr>
<td>10: jge l2</td>
<td>31: add ebx,eax</td>
<td>31: add ebx,eax</td>
</tr>
<tr>
<td>11: nop</td>
<td>32: sub eax,eax</td>
<td>32: sub eax,eax</td>
</tr>
<tr>
<td>12: nop</td>
<td>33: xor eax,eax</td>
<td>33: xor eax,eax</td>
</tr>
<tr>
<td>13: nop</td>
<td>34: invoke ExitProcess,0</td>
<td>34: invoke ExitProcess,0</td>
</tr>
<tr>
<td>14: nop</td>
<td>35:</td>
<td></td>
</tr>
<tr>
<td>15: shr ebx, 30</td>
<td>36:</td>
<td></td>
</tr>
<tr>
<td>16: shl ebx, 3</td>
<td>37:</td>
<td></td>
</tr>
<tr>
<td>17: add ebx, 401000h</td>
<td>38:</td>
<td></td>
</tr>
</tbody>
</table>

SLAM tool [25] is based on an automatic analysis of client code to validate a set of properties or find a counter-example showing a fail execution. DART [26] provides a new approach for completely automatic unit testing for software to avoid stubs that simulate the external environment of software. SYNERGY algorithm [27] presents a new approach to combine static and dynamic program analysis for property checking and test generation. DUALYZER is a dual static analysis tool [28], which is based on only over-approximation for both proving safety and finding real bugs.

OSMOSE [31] is a tool which also applies the concolic testing technique for automatic test-case generation from binary programs. This approach is quite close to our work, but it aims at generating test-cases, rather than CFG construction. Furthermore, OSMOSE involves solver to solve virtually all of path conditions of execution paths of the program. Meanwhile, our approach only invokes solver when handling execution paths leading to dynamic jumps, thus saving remarkable computational cost.

2. CFG construction from binary code

There are many methods of extracting CFG from binary source code. Gogul Balakrishnan and Thomas Reps introduced value-set analysis (VSA) [9]. By using numeric and pointer-analysis algorithms, VSA computes an over-approximation of the set of numeric values or addresses that every location may hold. This analysis technique was implemented in a tool called CodeSurfer [8][9], which is an extension from IDAPro [3].

Combining static and dynamic analyses for malwares is introduced in [29]. Regardless of over- or under-approximation, static analyses cannot resolve targets of indirect jumps when dynamic code modifications occur, e.g., mutations. Numerical abstract domains, such as intervals and k-sets, are used to handle targets of jumps, but hard to satisfy both accuracy and complexity. Recently, a refinement-based method is proposed based on k-sets [6]. Due to its cardinality bound, this method still remains certain limitations.

In BINCOA, a dynamic symbolic execution [14] and bit-vector constraint solving [30][31] are introduced. Meanwhile, IDA Pro relies on linear sweep decoding (the method of brute force decoding all addresses) and recursive traversal method [32] (decoding recursively until an indirect jump is found) for disassembly, which make it difficult to scale.

3. Binary analysis based on model-checking

Besides abstracting the memory addresses to reconstruct a CFG, another approach is to describe malicious behaviors of functions using temporal logic. This reduces virus detection to model checking. An extension CTPL of CTL (Computation Tree Logic) is proposed to specify certain obfuscation actions of a virus [36][37][38]. Further, Song and Tantau extend CTPL to SCTPL for better description on stack-based actions of viral behaviors [34]. Recently, LTL (Linear Temporal Logic) is suggested to replace CTL, and SLTPL is introduced [35]. They mostly consider the situation that a reasonably precise CFG are statically computed, say, without mutations.

VIII. CONCLUSION

This paper preliminarily reports a proposal for PhD work. The initial goal of this work is to produce a more precise CFG from binary codes. The difficulty to decide the precise destinations of indirect jumps remains as a major problem in the field. We propose a hybrid approach, which combines an over-approximation by static analyses and an under-approximation by dynamic testing to achieve practically more accurate CFGs. Initial results show that our method is quite promising. We expect that our approach not only resolves the issue of indirect jumps, but also improves efficiency of analyses on binary source codes.

ACKNOWLEDGEMENT

This research is funded by Vietnam National University Hochiminh City (VNU-HCM) under grant number 01/BK/2013/911VNUHCM-JAIST.
REFERENCES


