Combine Static Analysis and Dynamic Testing to analyze Binary Code

April, 22th 2013
Nguyễn Minh Hải
Table of Contents

- Motivation
  - Binary Code: benefits and challenges
  - Binary Analysis Frameworks
- Goal
  - Architecture
  - Example
  - Project of integration of symbolic execution and linear invariant to Jakstab
- Summary
Binary Code

- Binary code or Source code?

Benefits:
- Source code is unavailable
- Binaries are reliable

Challenges:
- Complexity
- Lack of higher-level semantics
- Obfuscation of malicious code
Frameworks of analyzing binaries

- BINCOA
- IDA Pro
- Jakstab

..........
Architecture

Binary Code → Disassembly → Intermediate Representation → CFG → Analysis Tool
Shortcomings

- IDA Pro
  - Closed source software
  - Recursive traversal disassembler decodes binaries by Depth First Search algorithm.
  - Problem of indirect jump

- Jakstab
  - Abstract interpretation with data flow analysis
  - Abstract interpretation can produce imprecision
Table of Contents

- Motivation
  - Binary Code: benefits and challenges
  - Binary Analysis Frameworks

- Goal
  - Architecture
  - Example
  - Project of integration of symbolic execution and linear invariant to Jakstab

- Summary
Goal

- Create the framework of combining static analysis and dynamic testing to analyze binary files.
- Apply this framework to detect malicious code.
Architecture

Static Analysis Component

Mixed Symbolic Analysis Component

Dynamic Analysis Component
Architecture (tt)

- Static analysis component (SAC)
  - Translate assembly to intermediate representation (IR)
  - Provide the core functions of static analysis, such as data flow analysis, abstract interpretation ...

- Dynamic analysis component (DAC)
  - Provide the core functions of implementing the binary code
  - Allow to monitor the behavior of executable

- Mixed symbolic analysis component (MSAC)
  - Combine symbolic execution and the functions of SAC and DAC
  - Provide the functions of SMT Solvers to resolve the path condition
Example

Path condition: $x < 8$

$x = 7$
Project

- Integrate symbolic execution and linear invariant to Jakstab
  - When the instructions can be disassembled, apply symbolic execution to find the environment and path condition.
  - When unresolved address found, instead of using abstraction interpretation, stop and use calculated condition to generate a test-case.
    - Dynamic executing the code with given test-case.
    - Find the branch destination.
    - Resume.
DEMO

`<loop.asm>`

```
start:
    mov ecx, 1000
    mov eax, 0
loop1:
    inc eax
    loop loop1
ret;
```

[masm32 assembler]

`<loop.exe>`
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
</table>

DEMO
PE Header

- Address of entry point
- Program information
DEMO

0 1 2 3 4 5 6 7 8 9 10 11 12 ...

`mov 1000, %ecx`

2: `mov 1000, %ecx`
DEMO

mov 0, %eax

2: mov 1000, %ecx
3: mov 0, %eax
DEMO

0 1 2 3 4 5 6 7 8 9 10 11 12 ...

jmp 7

2: mov 1000, %ecx
3: mov 0, %eax
4: jmp 7
DEMO

0 1 2 3 4 5 6 7 8 9 10 11 12 ...

inc %eax

2: mov 1000, %ecx
3: mov 0, %eax
4: jmp 7
7: inc %eax
DEMO

[%ecx != 0]

[0 1 2 3 4 5 6 7 8 9 10 11 12 …

2: mov 1000, %ecx
3: mov 0, %eax
4: jmp 7

7: inc %eax
8: loop 7

[&%ecx = 0]

loop 7
[ Invariant: $\%ecx + \%eax = 1000$ ]

2: mov 1000, \%ecx

3: mov 0, \%eax

4: jmp 7

7: inc \%eax

[\%ecx != 0]

[\%ecx = 0]
DEMO

STACK INFORMATION MISSING

2: mov 1000, %ecx
3: mov 0, %eax
4: jmp 7
5: [\%ecx != 0]
6: [\%ecx + %eax = 1000]
7: inc %eax
8: loop 7
9: ret
10: [\%ecx != 0]
11: [%ecx = 0]
12: [%ecx = 1000]
Summary

- To analyze binary code is imperative and challenging

- Our work is to create a framework of combining dynamic testing and static analysis
  - Static analysis component
  - Dynamic analysis component
  - Mixed symbolic analysis component
Reference


Q&A