BINGO-E

講者：zeze

目錄

介紹

BINGO-E

- Title：Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation

- Author：Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, Yang Liu, Chia Yuan Cho

- Publish：IEEE 2017

前提

- 不同的 architecture 會產生不同的 binary
- 不同的 compiler 會產生不同的 binary
- 不同的 compiler option 會產生不同的 binary
- 不同的 OS 會產生不同的 binary

目標

- P1. 要能彈性使用於不同的 compiler, architecture, os
- P2. 不管 compiler option 為何，都要能精確抓取語意(semantic)
- P3. 在 binary 很大時仍然不會 overhead

目前的狀況

- static: 靜態分析是找 syntatic 和 structural 的資訊，速度比較快，所以可以解決 P3(速度夠快)，但是無法解決 P1(在不同環境下都可使用), P2(就算優化也可以抓取語意)

- dynamic: 動態分析是透過 input/output 和 runtime 中的值，可以解決 P1, P2，但是無法解決 P3

四種特徵值

- low-level semantic feature: 主要是 runtime 中的資訊，例如 register, CPU flags
- high-level semantic feature: 主要是針對 text 段取得的資訊，例如 opcode, function call
- structural feature: 取得 CFG 中 basic block(BB) 的資訊，例如 outgoing edge, the number of instructions in the BB
- syntatic feature: 從 data, bss 段取得的資訊，例如 function name, string leteral

function inline

- 只有 structural 和 syntatic feature 解決不了 P1(在不同環境下都可使用)，所以要用 semantic feature
- 就算有 semantic feature，由於分析時不會把 callee 的內容當作這個 function 的一部分，因此解決不了 P2(就算優化也可以抓取語意)，所以要有適當的 function inlining
- sol: 用 selective inlining，選擇相關的 function 做 inline(這部分後面投影片有)

Semantic-based matching - challenge

- C1. 利用 basic block(BB) 來比對，會對 structure 太敏感，只要其中有些不同就會配對失敗
- C2. 各 function 視為獨立，但是如果其中 call 了會影響語意的 function，就會配對失敗
- C3. 效能問題

實作過程

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 1. Preprocessing

用 angr

1. Disassemble
2. build CFG

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 2. Feature Extraction

取出:

1. high-level semantic features
2. structural features
3. low-level semantic features

High-level Semantic Feature

取 6 種:

1. op type: opcode 的類型
2. system call tag: syscall 的類型
3. function call sequence: syscall 順序
4. function parameter: 參數
5. local variable: 有當作目的地位址的區域變數
6. opcode: opcode 順序

```
push  ebp
mov   ebp, esp
sub   esp, 56
sub   esp, 8
lea   eax, [ebp - 16]
push  eax
push  OFFSET FLAT : .LC0
call  scanf
add   esp, 16
mov   eax, DWORD PTR[ebp - 16]
sub   esp, 12
push  eax
call  strlen
add   esp, 16
mov   DWORD PTR[ebp - 12], eax
cmp   DWORD PTR[ebp - 12], 9
jg    .L2
```

(a) basic block B_1

```
mov   eax, 0
leave
ret
```

(b) basic block B_2

```
mov   edx, DWORD PTR[ebp - 12]
mov   eax, DWORD PTR[ebp - 16]
sub   esp, 4
push  edx
push  eax
lea   eax, [ebp - 56]
push  eax
call  memcpy
add   esp, 16
mov   eax, 10
sub   eax, DWORD PTR[ebp - 12]
mov   ecx, eax
mov   eax, DWORD PTR[ebp - 12]
lea   edx, [0 + eax * 4]
lea   eax, [ebp - 56]
add   eax, edx
sub   esp, 4
push  ecx
push  32
push  eax
call  memset
add   esp, 16
```

(c) basic block B_3

Op. type:
$data\_transfer(13), stack(10),$
$arithmetic(11), call(4), logical(1),$
$control\_transfer(1), misc(1),$
System call tags:
$io, string, mem, mem$
Function call sequence:
$scanf \rightarrow strlen \rightarrow$
$memcpy \rightarrow memset$
Function parameter: Null
Local variable:
$ebp - 12(mov[ebp - 12], eax)$
Op. code:
$push \rightarrow mov \rightarrow \ldots$
$\ldots \rightarrow leave \rightarrow ret$

(d) High-level semantic features of B_1, B_2 and B_3

```
int foo2()
{
 char * buf;
 char * p[10];
 scanf("%s", &buf);
 int len;
 len = strlen(buf);
 if(len < 10){
  memcpy(p, buf, len);
  memset(p + len, ' ', 10 - len);
 }
 return0;
}
```

(e) the source code of these binary code segments

Structural Feature

建造 3D-CFG，每個 node 都是一個 3-tuple (x,y,z)，以 BB 為單位

x: sequential

y: number of outgoing edges

z: loop depth

w: number of instructions

centroid c:

Definition 1. The centroid is a 4-tuple (c_x, c_y, c_z, w) [31]:

- $w = \sum_{e(p,q)\in 3D-CFG}(w_p + w_q)$,
- $c_i = \frac{\sum_{e(p,q)\in 3D-CFG}(w_p i_p + w_q i_q)}{w}$, where $i \in \{x, y, z\}$

centroid c'=(c'x, c'y, c'z, w'), w' = w + N, N=number of function call

Structural Feature

以 function 為單位

Centroid Difference Degree (CDD)

$$CDD(\vec{c_1}, \vec{c_2}) = max(\frac{|c1_x - c2_x|}{c1_x + c2_x}, \frac{|c1_y - c2_y|}{c1_y + c2_y}, \frac{|c1_z - c2_z|}{c1_z + c2_z}, \frac{|w_1 - w_2|}{w_1 + w_2})$$

$$FDD(\vec{f_1}, \vec{f_2}) = max(CDD(\vec{c_1}, \vec{c_2}), CDD(\vec{c_1}', \vec{c_2}'))$$

Function Difference Degree (FDD)
FDD 越大代表越不相似

Low-level Semantic Feature

pre-state x = <mem, reg, flag>

post-state x' = <mem', reg', flag'>

因為不同的 architecture 的 register 不一樣，所以只能丟 CPU flags, memory addresses 當 input

為了計算相似度，可以用 constraint solving 算 semantic equivalence，但是非常慢。

sol：用動態的方式直接跑

Other Feature

其他 features:

v1. value read from the program heap

v2. value written to the program heap

v3. value read from the program stack

v4. value written to the program stack

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 3. Primary Matching - High-level

High-level semantic features

- Jaccard containment similarity

$$\mathcal{SIM}_{\mathcal{H}}(sig, tar) = \frac{\mathcal{H}_{sig} \bigcap \mathcal{H}_{tar}}{\mathcal{H}_{sig}}$$

- overall

6 種 features 的 Jaccard distance 平均值

workflow - 3. Primary Matching - structural

Structural features

$$\begin{aligned} \mathcal{SIM}_{S}(sig, tar) &= Convert(FDD(sig, tar)) \\ &= 1 - FDD(sig, tar) \end{aligned}$$

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 4. Selective inlining

a) caller call libcall: inline
b) caller recursively call ud callee: liline
c) many caller call ud callee, calee call libcall and ud callee: 不一定
d) caller call ud callee, calee call termination libcall and other libcall: inline
e) caller call ud callee, callee only call termination libcall: not inline
f) many caller call ud callee, callee call many ud callee: 不一定

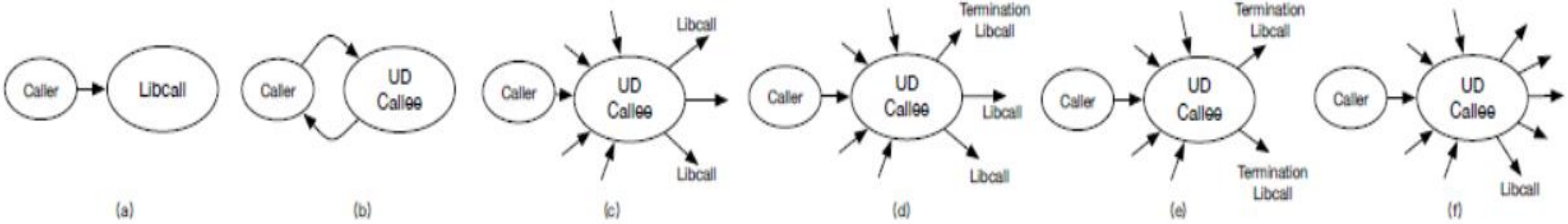

Fig. 7: Commonly observed function invocation patterns, where 'UD' denotes user-defined callee function. Here, all the incoming and outgoing edges (or calls) represent user-defined functions, unless specified as *Libcall* or *Termination Libcall* next to them

workflow - 4. Selective inlining

- 由於 (c) 和 (f) 不一定會 inline，所以有個 Algorithm 決定是否 inline
- (c) : 如果 callee 有呼叫 UD function，就看那個 UD function 的語意，沒有就直接 inline
- (f) : 如果 (callee 呼叫的 UD function)/(呼叫 callee 的 UD function + callee 呼叫的 UD function) 大於 threshold 就 inline

Algorithm 1: Selective inlining algorithm

Data: caller $\mathcal{F}$, set of callee functions $\mathcal{C}$, set of termination lib. func. $\mathcal{L}_t$, set of inlining lib. func. $\mathcal{L}_s$

Result: inlined function $\mathcal{F}^I$

Algorithm `SelectiveInline` $(\mathcal{F}, \mathcal{C}, \mathcal{L}_s, \mathcal{L}_t)$
- **foreach** function f in $\mathcal{C}$ **do**
  - // inline selected library functions
  - **if** $f \in \mathcal{L}_s$ **then**
    - $\mathcal{F}^I \longleftarrow \mathcal{F}.\mathtt{inline}(f)$
    - **return** $\mathcal{F}^I$
  - **else if** $f \notin \mathcal{L}_s$ && $\mathtt{isLibCall}(f)$ **then**
    - **return** null
  - // for all other user-defined callee functions
  - $I_u^f \longleftarrow \mathtt{getIncomingCalls}(f)$
  - $O_u^f, O_l^f \longleftarrow \mathtt{getOutgoingCalls}(f)$
  - $O_u^f \longleftarrow O_u^f \setminus \mathcal{F}$ // remove recursion
  - **if** $|O_u^f| == 0$ && $(|O_l^f \cap \mathcal{L}_s| - |O_l^f \cap \mathcal{L}_t|) \leqslant 0$ **then**
    - **return** null
  - **else**
    - $\alpha = \lambda_e/(\lambda_e + \lambda_a)$ *where* $\lambda_a = |I_u^f|, \lambda_e = |O_u^f|$
    - // lower the α, function f is likely to be inlined
    - **if** $\alpha >$ threshold t && $\mathtt{notRecursive}(\mathcal{F}, f)$ **then**
      - **return** null
    - **else**
      - $\mathcal{F}^I \longleftarrow \mathcal{F}.\mathtt{inline}(f)$
      - **if** $|O_u^f| > 0$ **then**
        - $\mathtt{SelectiveInline}(f, O_u^f, \mathcal{L}_s, \mathcal{L}_t)$
      - **else**
        - **return** $\mathcal{F}^I$
- **return** $\mathcal{F}^I$

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 5. Function Model Generation

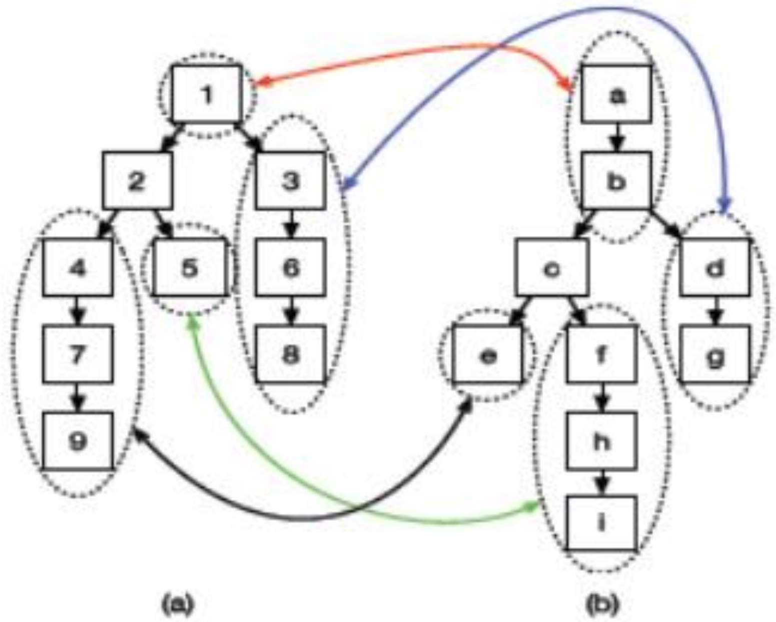

Fig. 9: A sample signature (a) and target (b) functions, where the lines indicate the matched partial traces and the nodes refer to the basic-blocks.

$$\mathcal{M}_{\mathtt{sig}} : \{\langle 1\rangle, \langle 2\rangle, \ldots, \langle 1,2\rangle, \langle 2,5\rangle, \ldots, \langle 1,2,4\rangle, \langle 2,4,7\rangle, \ldots\}$$
$$\mathcal{M}_{\mathtt{tar}} : \{\langle \mathtt{c}\rangle, \langle \mathtt{b}\rangle, \ldots, \langle \mathtt{a},\mathtt{b}\rangle, \langle \mathtt{b},\mathtt{c}\rangle, \ldots, \langle \mathtt{a},\mathtt{b},\mathtt{c}\rangle, \langle \mathtt{b},\mathtt{c},\mathtt{f}\rangle, \ldots\}$$

workflow - 5. Function Model Generation - result

$$SIM_{\mathcal{L}}(sig, tar) = \frac{\mathcal{M}_{sig} \bigcap \mathcal{M}_{tar}}{\mathcal{M}_{sig}}$$

其中 M sig 和 M tar 分别是 signature 和 target function 的 function model

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 6. Emulation

用 Unicorn 跑在 qemu 上模擬，將 low-semantic features 跑出來並且做比對

workflow

First Phase (如果這邊的配對程度高於某門檻，就可以不用做 second phase)

1. Pre-processing
2. Feature Extraction
3. Primary Matching

Second Phase

1. Selective Inlining
2. Function Model Generation
3. Emulation
4. Final Matching

workflow - 7. Final Matching

- same code base

$$\mathcal{SIM}^{*}(sig, tar) = (1/2) * \mathcal{SIM}'(sig, tar) + (1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar)$$

- different code base

$$\mathcal{SIM}^{*}(sig, tar) = (1/2) * \mathcal{SIM}_{\mathcal{H}}(sig, tar) + (1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar)$$

測試結果

TABLE 6: In BinGo-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for `coreutils`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3.

	C0	**C1**	**C2**	**C3**	**G0**	**G1**	**G2**	**G3**
C0	-	0.865	0.791	0.785	0.799	0.840	0.836	0.701
C1	0.840	-	0.838	0.833	0.891	0.913	0.887	0.756
C2	0.814	0.889	-	0.996	0.862	0.926	0.858	0.757
C3	0.809	0.885	0.997	-	0.857	0.924	0.855	0.759
G0	0.801	0.888	0.853	0.846	-	0.884	0.856	0.754
G1	0.786	0.878	0.857	0.852	0.850	-	0.928	0.857
G2	0.806	0.882	0.814	0.808	0.848	0.963	-	0.732
G3	0.732	0.773	0.754	0.753	0.770	0.863	0.751	-

TABLE 12: Matching results between functions of BusyBox 1.20.0 and functions of BusyBox 1.21 using BINGO-E, TRACY and BINDIFF

All 2406 Functions	Rank 1	Rank 1-3	Rank 1-10
BINGO-E	95.87%	97.69%	98.38%
TRACY (k=1)	83.33%	85.95%	89.61%
BINDIFF	96.81%	N.A.	N.A.

1348 Large Functions	Rank =1	Rank 1-3	Rank 1-10
BINGO-E	99.11%	99.63%	99.85%
TRACY (k=4)	97.18%	98.00%	98.81%

Thanks For Listening