# ByteWeight: Learning to Recognize Functions in Binary Code

USENIX 2014, 2014. 8
Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, David Brumley

김영철

2018. 11. 15.

# Introduction

- 바이너리 분석은 다양한 분야에서 필수.
- 분석하기 전에 보통 내부에 있는 함수를 식별하는 작업을 함.
  - CFI가 적용된 바이너리에 대해서는 정확한 식별 작업이 필요함.

- IDA, BAP, Bit Blaze, Dyninst 등 여러 바이너리 분석 도구들은 CFI 적용된 바이너리를 잘못 분석하는 경우가 많음.
  - ex) 0x55와 같은 시그니처를 이용하여 함수 시작 주소 식별
- 기계학습을 이용한 방법도 있지만 매우 느림.

➔ ByteWeight는 Weighted Prefix Tree와 Value Set Analysis를 이용하여 더 빠르고 정확한 함수 식별 도구를 제작함.

# Running Example

- 간단한 C 프로그램을 IDA로 분석.
  - x86-64 gcc -O3
  - strip을 이용하여 디버그 심볼 삭제

- IDA는 sum, sub, assign 함수를 식별하지 못했음.

*우분투 18.04에서 같은 옵션으로 수행해본 결과 식별하지 못했음.

```
1   #include <stdio.h>
2   #include <string.h>
3   #define MAX 10
4   void sum(char *a, char *b)
5   {
6       printf("%s + %s = %d\n",
7              a, b, atoi(a) + atoi(b));
8   }
9   void sub(char *a, char *b)
10  {
11      printf("%s - %s = %d\n",
12             a, b, atoi(a) - atoi(b));
13  }
14  void assign(char *a, char *b)
15  {
16      char pre_b[MAX];
17      strcpy(pre_b, b);
18      strcpy(b, a);
19      printf("b is changed from %s to %s\n",
20             pre_b, b);
21  }
22  int main(int argc, char **argv)
23  {
24      void (*funcs[3])(char *x, char *y);
25      int f;
26      char a[MAX], b[MAX];
27      funcs[0] = sum;
28      funcs[1] = sub;
29      funcs[2] = assign;
30      scanf("%d %s %s", &f, a, b);
31      (*funcs[f])(a, b);
32      return 0;
33  }
```

<source code>

```
1       00400660 <assign>:
2       mov     %rbx,-0x10(%rsp)
3       mov     %rbp,-0x8(%rsp)
4       sub     $0x28,%rsp
5       mov     %rdi,%rbp
6       lea     0xf(%rsp),%rdi
7       ...
8       004006b0 <sub>:
9       mov     %rbx,-0x18(%rsp)
10      mov     %rbp,-0x10(%rsp)
11      mov     %rsi,%rbx
12      mov     %r12,-0x8(%rsp)
13      xor     %eax,%eax
14      sub     $0x18,%rsp
15      ...
16      00400710 <sum>:
17      mov     %rbx,-0x18(%rsp)
18      mov     %rbp,-0x10(%rsp)
19      mov     %rsi,%rbx
20      mov     %r12,-0x8(%rsp)
21      xor     %eax,%eax
22      sub     $0x18,%rsp
23      ...
```

<result of compilation>

# Problem Definition and Challenges

1. Notation and Definitions

- B : binary string
  B[i] : i번째 byte
  B[i:i+j] : i번째 byte부터 연속되는 길이 j의 연속된 byte
- F : 함수, byte 집합 ➔ F = {B[i], B[j], … B[k]}
  FUNCS(B) : 존재하는 함수 집합 ➔ FUNCS(B) = $\{F_1, F_2, \ldots F_k\}$
- Function Oracle ($O_{func}$) : 함수 리스트
  Boundary Oracle ($O_{bound}$) : 함수 시작 주소, 마지막 주소 쌍 리스트
  Start Oracle ($O_{start}$) : 함수 시작 주소 리스트

# Problem Definition and Challenges

2. Problem Definition

   Definition 1. *Function Start Identification* (FSI) Problem

   Definition 2. *Function Boundary Identification* (FBI) Problem

   Definition 3. *Function Identification* (FI) Problem

- ByteWeight의 알고리즘 결과로 얻어진 것과 Oracle 셋과 단순히 비교하기만 하면 됨.
- FI problem은 함수를 구성하는 instruction 하나 하나를 비교해야 함.

# Problem Definition and Challenges

## 3. Challenges

- Not every byte belongs to a function
  - alignment or padding으로 인해 생겨나는 공간

```
1    <_func1>:
2    100000e20:    push    %rbp
3    100000e21:    mov     %rsp,%rbp
4    100000e24:    lea     0x69(%rip),%rdi
5    100000e2b:    pop     %rbp
6    100000e2c:    jmpq    100000e5e <_puts$stub>
7    100000e31:    nopl    0x0(%rax)
8    100000e38:    nopl    0x0(%rax,%rax,1)
9    <func2>:
```

7, 8번 라인은 함수에 포함되지 않음.

# Problem Definition and Challenges

3. Challenges

- Functions may be non-contiguous
  - 하나의 함수 내부에 다른 함수의 chunk code가 존재할 수 있음.

```
1   <ConvertDefaultLocale>
2     7c8383ff:   mov     %edi,%edi
3     7c838401:   push    %ebp
4     ...
5     7c83840c:   jz      7c848556
6     7c838412:   test    %eax, %eax
7     7c838414:   jz      7c83965c
8     7c83841a:   mov     $1024,%ecx
9     7c83841f:   cmp     %ecx,%eax
10    7c838421:   jz      7c83965c
11    7c838427:   test    $252,%ah
12    7c83842a:   jnz     7c838442
13    7c83842c:   mov     %eax,%edx
14    ...
15    7c838442:   pop     %ebp
16    7c838443:   ret     4
17    ; chunk of different function FindNextFileW
18    7c838446:   push    6
19    7c838448:   call    sub_7c80935e
20    7c83844d:
21    ; end of chunk
22    ...
23    7c83965c:   call    GetUserDefaultLCID
24    7c890661:   jmp     7c838442
25    ...
26    7c848556:   mov     $8,%eax
27    7c84855b:   jmp     7c838442
```

# Problem Definition and Challenges

3. Challenges

- Functions may not be reachable
  - 바이너리 코드 내에 함수가 존재해도 호출이 안될 수 있음.
  - 최적화 옵션을 적용하여 컴파일 타임에 미리 계산하는 경우.

- Functions may have multiple entries

# Problem Definition and Challenges

3. Challenges

- Functions may be removed
  - function inlining에 의해 위치가 이동될 수 있음.
  - function call overhea가 작아짐.

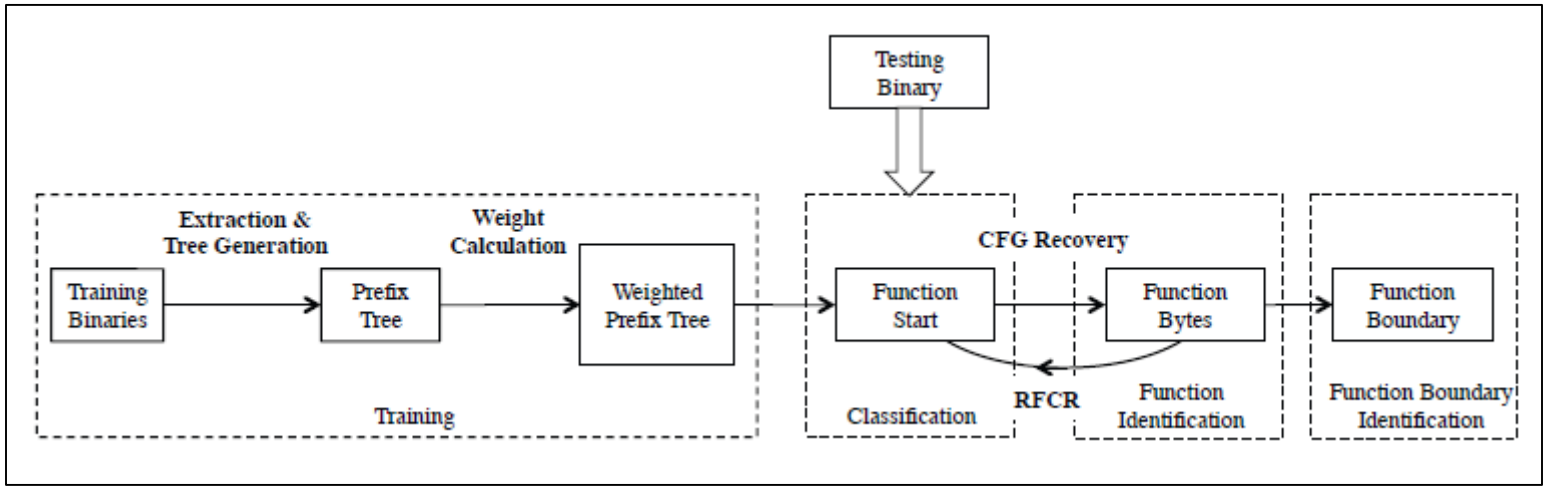
# Problem Definition and Challenges

3. Challenges

- Each compilation is different
  - 사용하는 컴파일러나 적용하는 옵션에 따라 바이너리가 매우 달라짐.
  - 특히, 함수 시작은 *push ebp* 으로 간주할 수 있는데 최근에는 생략 옵션도 있어서 시그니처 기반의 탐지 방법이 무용지물이 됨.

# ByteWeight

- FSI 문제를 바이너리의 각 byte에 대해 함수 시작 주소인지 분류하는 문제로 생각함.
- 학습 과정은 함수 시작 주소 정보를 갖도록 소스 코드를 컴파일하고 생성된 바이너리의 바이트 시퀀스를 Tree로 만드는 과정




<ByteWeight System>

# ByteWeight

- 함수 시작 주소 분류 과정은 Tree를 이용한다.
- terminal node가 임계값을 넘으면 함수 시작 주소로 간주.
  - terminal node : 주어진 시퀀스가 Tree에 존재하는 path와 완전히 같은 경우에는 그 path의 마지막 node. 완전히 같지 않은 경우에는 시퀀스에서 마지막으로 매칭된 node.
- 함수 시작을 찾으면 CFG 복원 알고리즘을 통해 함수를 구성하는 바이트를 추론, 이 과정에서 VSA를 이용하여 register 값을 이용한 indirect jump를 처리함. (over approximation)

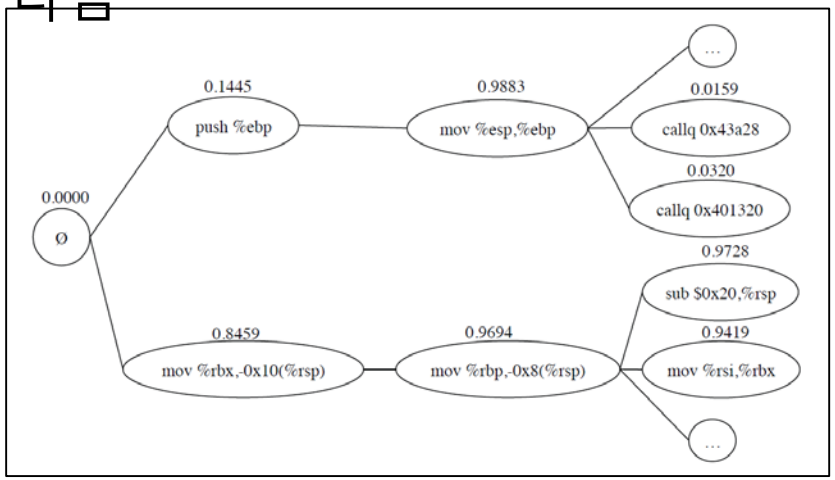
# ByteWeight

1. Learning Phase
   - input : 학습 시킬 바이너리 T의 일부분, 학습 가능한 최대 길이 l은 Tree의 최고 높이
   - 먼저 소스 코드를 디버그 정보를 포함하도록 컴파일하여 $O_{bound}$를 생성

# ByteWeight

1. Learning Phase (continue)
   - step1) 각 함수에서 첫 l개의 바이트를 추출
     - 함수가 l보다 작은 수의 바이트로 구성되어 있으면 모두 추출
     - 즉, B[start : start + l]을 말함.
   - step2) prefix tree 생성
     - 모든 노드는 한 바이트와 관련됨(root 제외)
     - 노드 n은 root부터 노드 n까지의 경로를 나타냄
     - normalization을 통해 성능 상승

```
push   %ebp            ;saved stack pointer
mov    %esp,%ebp       ;establish new frame
callq  0x43a28         ;call another function
```



# ByteWeight

1. Learning Phase
   - step3) Tree의 Weight 계산하기
     - Tree는 결국 I개의 요소로 가능한 함수 시작 시퀀스를 나타냄.
     - 각 노드까지 도달하는 경로가 함수 시작일 가능성이 가중치가 됨.
     - Tree 가지치기 (…)