ByteWeight:
Learning to Recognize Functions in Binary Code

Tiffany Bao
Jonathan Burket
Maverick Woo
Rafael Turner
David Brumley
Carnegie Mellon University

USENIX Security ’14
Can we *automatically* and *accurately* recover function information from stripped binaries?
#include <stdio.h>

int fac(int x){
    if (x == 1)
        return 1;
    else
        return x * fac(x - 1);
}

void main(int argc, char **argv){
    printf("%d", fac(10));
}
Example: GCC

08048443 <main>:
push %ebp
mov %esp,%ebp
and $0xffffffff0,%esp
sub $0x10,%esp
...

0804841c <fac>:
push %ebp
mov %esp,%ebp
sub $0x18,%esp
cmpl $0x1,0x8(%ebp)
jne 804842f <fac+0x13>
mov $0x1,%eax
...

-O0: Default
Example: GCC

0804841c <fac>:
push  %ebx
sub   $0x18,%esp
mov   0x20(%esp),%ebx
mov   $0x1,%eax
cmp   $0x1,%ebx
...

08048330 <main>:
mov    $0x1,%edx
mov    $0xa,%eax
lea    0x0(%esi),%esi
...
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x10,%esp
...

-O1: Optimize
-O2: Optimize Even More
Current Industry Solution: IDA

```c
#include<stdio.h>
#include<string.h>
#define MAX 128

void sum(char a[MAX], char b[MAX]){
    printf("%s + %s = %d\n", a, b, atoi(a) + atoi(b));
}

void sub(char a[MAX], char b[MAX]){
    printf("%s - %s = %d\n", a, b, atoi(a) - atoi(b));
}

void assign(char a[MAX], char b[MAX]){
    char pre_b[MAX];
    strcpy(pre_b, b);
    strcpy(b, a);
    printf("b is changed from %s to %s\n", pre_b, b);
}

int main(){
    void (*funcs[3]) (char x[MAX], char y[MAX]);
    int f;
    char a[MAX], b[MAX];
    funcs[0] = sum;
    funcs[1] = sub;
    func[2] = assign;
    scanf("%d %s %s", &f, &a, &b);
    (*funcs[f])(a, b);
    return 0;
}
```
Function Identification Problems

Given a *stripped* binary, return

1. A list of function start addresses
   - “Function Start Identification (FSI) Problem”

2. A list of function (start, end) pairs
   - “Function Boundary Identification (FBI) Problem”

3. A list of functions as sets of instruction address
   - “Function Identification (FI) Problem”
ByteWeight

A *machine learning* + *program analysis* approach to function identification

Training:
- 1. Creates a model of function start patterns using supervised machine learning

Usage:
- 1. Use trained models to match function start on stripped binaries — Function Start Identification
- 2. Use program analysis to identify all bytes associated with a function — Function Identification
- 3. Calculate the minimum and maximum addresses of each function — Function Boundary Identification
Function Start Identification

1. Previous approaches
2. Our approach
Previous Work: Rosenblum et al.\textsuperscript{[1]}

Method: Select instruction idioms up to length 4; learn idiom parameters; label test binaries

“Feature (idiom) selection for all three data sets (1,171 binaries) consumed over 150 compute-days of machine computation”

ByteWeight: Lighter (Linear) Method

Extraction
- Training Binaries
- Extracted Sequences

Weight Calculation
- Weighted Sequences

Tree Generation
- Weighted Prefix Tree

Training

Testing Binary

Function Start

CFG Recovery

RFCR
Classification
Function Bytes

Function Identification
Function Boundary

Identification
Step 1: Extract All ≤ K-length Sequences

Bytes

• 55
• 55 48
• 55 48 89
• 55 48 89 e5
• ...

Instructions

• push %rbp
• push %rbp; mov %rsp,%rbp
• push %rbp; mov %rsp,%rbp; sub $0x10,%rsp
• push %rbp; mov %rsp,%rbp; sub $0x10,%rsp; mov %edi,-0x4(%rbp)
• ...

00000000100000e3b <func_1>:
push %rbp
mov %rsp,%rbp
sub $0x10,%rsp
mov %edi,-0x4(%rbp)
mov %esi,-0x8(%rbp)
mov -0x8(%rbp),%edx
mov -0x4(%rbp),%eax
mov %eax,%esi
lea 0xc0(%rip),%rdi
mov $0x0,%eax
callq 100000ee8
leaveq
retq
### Step 2: Weight Sequences

#### score:
\[
\frac{2}{(2 + 2)} = 0.5
\]

#### Code Snippet

```
000000000100000e3b <__func_1>:
  55      push  %rbp
  48 89 e5  mov  %rsp,%rbp
  48 83 ec 10 sub  $0x10,%rsp
  89 7d fc  mov  %edi,-0x4(%rbp)
  89 75 f8  mov  %esi,-0x8(%rbp)
  8b 55 f8  mov  -0x8(%rbp),%edx
  8b 45 fc  mov  -0x4(%rbp),%eax
  89 c6   mov  %eax,%esi
  48 8d 3d c0 00 00 00 lea  0xc0(%rip),%rdi
  b8 00 00 00 00 mov  $0x0,%eax
  e8 86 00 00 00 callq 100000ee8
  c9     leaveq
  c3     retq
```

```
000000000100000e64 <__func_2>:
  55      push  %rbp
  48 89 e5  mov  %rsp,%rbp
  48 83 ec 16 sub  $0x16,%rsp
  89 7d fc  mov  %edi,-0x4(%rbp)
  89 75 f8  mov  %esi,-0x8(%rbp)
  8b 55 f8  mov  -0x8(%rbp),%edx
  8b 45 fc  mov  -0x4(%rbp),%eax
  89 c6   mov  %eax,%esi
  48 8d 3d a6 00 00 00 lea  0xa6(%rip),%rdi
  b8 00 00 00 00 mov  $0x0,%eax
  e8 5d 00 00 00 callq 100000ee8
  c9     leaveq
  c3     retq
```
Step 2: Weight Sequences

push %rbp; mov %rsp,%rbp
→ 55 48 89 e5

score:
2 / (2 + 0) = 1.0
Step 3: Generate Weighted Prefix Tree

2 / (2 + 2) = 0.5
push %rbp
(55)

2 / (2 + 0) = 1.0
mov %rsp, %rbp
(48 89 e5)

1 / (1 + 0) = 1.0
sub $0x10, %rsp
(48 83 ec 10)

push %rbp
→ 2/(2+2)=0.5
push %rbp; mov %rsp, %rbp
→ 2/(2+0)=1.0
...

1 / (1 + 0) = 1.0
sub $0x16, %rsp
(48 83 ec 16)
Classification

```
Classifica5on
push %rbp (55)
...  0.0
mov %rsp,%rbp (48 89 e5)
...  1.0
sub $0x10,%rsp (48 83 ec 10)
...  1.0
sub $0x16,%rsp (48 83 ec 16)
...  1.0
mov %rsp,%rbp (48 89 e5)
...  0.5
push %rbp (55)
...  0.0
```

Test Binary

```
00 00 00 00 e8 5d 00 00 00 c9
c3 55 48 89 e5 48 83 ec 60 48
c3 48 8d 05 9f ff ff ff 48 89 45 b8
c0 48 8d 05 bd ff ff ff ff 48 89 45
8d 45 a4
```
Normalization (Optional)

```
sub $0x60,%rsp
00 00 00 00 e8 5d 00 00 00 c9
c3 55 48 89 e5 48 83 ec 60 48
8d 05 9f ff ff ff 48 89 45 b8
48 8d 05 bd ff ff ff 48 89 45
c0 48 8d 4d a8 48 8d 55 ac 48
8d 45 a4
```

```
sub $0x[1-9a-f][0-9a-f]*,%rsp
```

```
push %rbp
(55)
```

```
mov %rsp,%rbp
(48 89 e5)
```

```
sub $0x10,%rsp
```

```
(48 83 ec 10)
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Function (Boundary) Identification

Identify all bytes associated with a function, and extract the lowest and highest addresses
ByteWeight: Function (Boundary) Identification

1. Recursive disassembly, using Value Set Analysis\cite{Balakrishan:2007:WYSINWYX} to resolve indirect jumps.

2. Recursive Function Call Resolution—add any call target as a function start.

ByteWeight: Function (Boundary) Identification

F1

instr1, instr2, instr3, instr6, instr10, instr12, ..., instr100, instr101.

[instr1, instr101]

Control Flow Graph Recovery

Testing Binary

Function Start

Function Bytes

RFCR

Function Identification

Function Boundary

Function Boundary Identification

Experiment Results

Compilers: GCC, ICC, and MSVS
Platforms: Linux and Windows
Optimizations: O0(Od), O1, O2, and O3(Ox)
Training Performance

ByteWeight:
- 10-fold cross-validation, 2200 binaries
- 6.1 days to train from all platforms and all compilers including logging

Rosenblum et al.:
- ??? (They reported 150 compute days for one step of training, but did not report total time, or make their training implementation available.)
  - training data and code both unavailable
Precision and Recall

\[
\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}
\]
Function Start Identification: Comparison with Rosenblum et al.

**Precision**

- GCC
- ICC

**Recall**

- GCC
- ICC
Function Start Identification: Existing Binary Analysis Tools

**Precision**

- **Naïve**
- **Dyninst**
- **BAP**
- **IDA**
- **ByteWeight (no RFCR)**
- **ByteWeight**

**Recall**

- **Naïve**
- **Dyninst**
- **BAP**
- **IDA**
- **ByteWeight (no RFCR)**
- **ByteWeight**
Function Boundary Identification: Existing Binary Analysis Tools

**Precision**

- Naïve
- Dyninst
- BAP
- IDA
- ByteWeight (no RFCR)
- ByteWeight

**Recall**

- Naïve
- Dyninst
- BAP
- IDA
- ByteWeight (no RFCR)
- ByteWeight
Summary: ByteWeight

Machine-learning based approach
– Creates a model of function start patterns using supervised machine learning
– Matches model on new samples
– Uses program analysis to identify all bytes associated with a function
– Faster and more accurate than previous work
Thank You

Our experiment VM is available at:

http://security.ece.cmu.edu/bytesweight/

Tiffany Bao
tiffanybao@cmu.edu