Software Mining and Re-engineering

Reverse-engineering from binary code

Master M2 MoSiG (AISSE)

Academic Year 2018 - 2019
About this part of the SMRe course

Objectives

▶ a brief overview on binary code reverse engineering: motivation, challenges, techniques and tools . . .
▶ how to protect your code from being “reversed”: obfuscation and de-obfuscations techniques . . .

Organisation

▶ 2 lectures (Dec. 13th and 20th, 11.30 am, room H201)
▶ 2 “labs” (Dec. 13th and 20th, 2 pm, room E212)
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools . . .
Software = several knowledge/information levels

- (formal) models: overall architecture, component behaviors
- specifications, algorithms, abstract data structures
- source code
  - objects, variables, types, functions, control and data flows
- possible intermediate representations: Java bytecode, LLVM IR, etc.
- assembly
- binary code (relocatable / shared object / executable)

Some reverse-engineering settings:
- source level → model level . . .
- de-compiling: binary → source level
- disassembling: binary → assembly level
- etc.
Why and when bothering with binary code? (1)
Why and when bothering with binary code? (1)

→ when the source code is not/no longer available

- updating/maintaining legacy code
- “off-the-shell” components (COST), external libraries
- dynamically loaded code (applets, plugins, mobile apps)
- pieces of assembly code in the source
- suspicious files (malware, etc.)
Why and when bothering with binary code? (2)

→ when the source code is not sufficient


▶ untrusted compilation chain

▶ low-level bugs, at the HW/SW interface

▶ security analysis
  going beyond standard programming language semantics (optimization, memory layout, undefined behavior, protections, etc.)
Why and when bothering with binary code? (2)

→ when the source code is not sufficient


► untrusted compilation chain

► low-level bugs, at the HW/SW interface

► security analysis
  going beyond standard programming language semantics
  (optimization, memory layout, undefined behavior, protections, etc.)

Beware! Reverse-engineering is restricted by the law . . .
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools . . .
Example 1: Java ByteCode (stack machine)\(^1\)

```java
public static int main() {
    int x, r;
    x=42 ; r=1 ;
    while (x>0) {
        r = r*x;
        x = x-1;
    }
    return r ;
}
```

```sqlite
public static int main(java.lang.String[]);
Code:
0: bipush 42
2: istore_1
3: iconst_1
4: istore_2
5: iload_1
6: ifle 20
9: iload_2
10: iload_1
11: imul
12: istore_2
13: iload_1
14: iconst_1
15: isub
16: istore_1
17: goto 5
20: iload_2
21: ireturn
```

\(^1\)use javap -c to produce the bytecode
Example 2: LLVM IR (machine à registre)

int main() {
    int x, r;
    x = 42; r = 1;
    while (x > 0) {
        r = r * x;
        x = x - 1;
    }
    return r;
}

CFG for 'main' function

%0:
%1 = alloca i32, align 4
%x = alloca i32, align 4
%r = alloca i32, align 4
store i32 0, i32* %1
store i32 42, i32* %x, align 4
store i32 1, i32* %r, align 4
br label %2

%2:
%3 = load i32* %x, align 4
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %11

%5:
%6 = load i32* %r, align 4
%7 = load i32* %x, align 4
%8 = mul nsw i32 %6, %7
store i32 %8, i32* %r, align 4
%9 = load i32* %x, align 4
%10 = sub nsw i32 %9, 1
store i32 %10, i32* %x, align 4
br label %2

%11:
%12 = load i32* %r, align 4
ret i32 %12
Example 3: assembly code (x86-64)

```c
int main() {
    int x, r;
    x=42 ; r=1 ;
    while (x>0) {
        r = r*x;
        x = x-1;
    } ;
    return r ;
}
```

```
main:
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], 42
mov DWORD PTR [rbp-8], 1
jmp .L2
.L3:
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-4]
mov DWORD PTR [rbp-8], eax
sub DWORD PTR [rbp-4], 1
.L2:
cmp DWORD PTR [rbp-4], 0
jg .L3
mov eax, DWORD PTR [rbp-8]
pop rbp
ret
```

\(^2\text{see https://godbolt.org/}\)
Memory layout at runtime (simplified)

Executable code = (binary) file produced by the compiler
→ need to be loaded in memory to be executed (using a loader)

However:

▶ no absolute addresses are stored in the executable code
  → decided at “load time”
▶ not all the executable code is stored in the executable file
  (e.g., dynamic libraries)
▶ data memory can be dynamically allocated
▶ data can become code (and conversely . . . )
▶ etc.

→ the executable file should contain all the information required . . .
Memory layout at runtime (simplified)

Executable code = (binary) file produced by the compiler
→ need to be *loaded* in memory to be executed (using a *loader*)

**However:**

- no absolute addresses are stored in the executable code
  → decided at “load time”
- not all the executable code is stored in the executable file
  (e.g., dynamic libraries)
- data memory can be dynamically allocated
- data can become code (and conversely . . . )
- etc.

→ the executable file should contain all the information required . . .

∃ standards executable formats: ELF (Linux), PE (Windows), etc.
  - header
  - sections: text, initialized/uninitialized data, symbol tables, relocation tables, etc.

**Rks:** stripped (no symbol table) vs verbose (debug info) executables . . .
**Example 1: Linux Elf**

**ELF object file format**

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
</tr>
<tr>
<td>.text</td>
</tr>
<tr>
<td>.data</td>
</tr>
<tr>
<td>.rodata</td>
</tr>
<tr>
<td>.bss</td>
</tr>
<tr>
<td>.sym</td>
</tr>
<tr>
<td>.rel.text</td>
</tr>
<tr>
<td>.rel.data</td>
</tr>
<tr>
<td>.rel.rodata</td>
</tr>
<tr>
<td>.line</td>
</tr>
<tr>
<td>.debug</td>
</tr>
<tr>
<td>.strtab</td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>

---
Example 2: Windows PE

PE File Format

- MS-DOS: MZ Header
- MS-DOS Real-Mode Stub Program
- PE File Signature
- PE File Header
- PE File Optional Header
- text Section Header
- bss Section Header
- rdata Section Header
x86 (32) assembly language in one slide

Registers:
- stack pointer (ESP), frame pointer (EBP), program counter (EIP)
- general purpose: EAX, EBX, ECX, EDX, ESI, EDI
- flags

Instructions:
- data transfer (MOV), arithmetic (ADD, etc.)
- logic (AND, TEST, etc.)
- control transfer (JUMP, CALL, RET, etc)

Adressing modes:
- register: mov eax, ebx
- immediate: mov eax, 1
- direct memory: mov eax, [esp+12]
Stack layout for the x86 32-bits architecture

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
ABI (Application Binary Interface)

to “standardize” how processor resources should be used
⇒ required to ensure compatibilities at binary level

▶ sizes, layouts, and alignments of basic data types

▶ **calling conventions**
  argument & return value passing, saved registers, etc.

▶ system calls to the operating system

▶ the binary format of object files, program libraries, etc.

![Table of Calling Conventions](image)

**Figure**: some calling conventions
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools ...
Understanding and analysing binary code?

<table>
<thead>
<tr>
<th>01010100</th>
<th>01101000</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>01101110</td>
</tr>
<tr>
<td>01101111</td>
<td>00100000</td>
</tr>
<tr>
<td>01100100</td>
<td>01101001</td>
</tr>
<tr>
<td>01100110</td>
<td>01100110</td>
</tr>
<tr>
<td>01100101</td>
<td>01110010</td>
</tr>
<tr>
<td>01100101</td>
<td>01101110</td>
</tr>
<tr>
<td>01110100</td>
<td>00101110</td>
</tr>
</tbody>
</table>
Understanding and analysing binary code?

Disassembling!

statistically:

disassemble the **whole** file content *without executing it* . . .

dynamically: disassemble the **current** instruction path during execution/emulation . . .
Static Disassembling (1)

Assume “reasonnable” (stripped) code only
→ no obfuscation, no packing, no auto-modification, . . .

Enough pitfalls to make it undecidable . . .

**main issue:** distinguishing code vs data . . .

- interleavings between code and data segments
- dynamic jumps (\texttt{jmp <register>})
- possible variable-length instruction encoding, # addressing modes, . . .
  e.g, $>$ 1000 distinct x86 instructions
  1.5 year to fix the semantics of x86 shift instruction at CMU
Static Disassembling (1)

Assume “reasonnable” (stripped) code only
→ no obfuscation, no packing, no auto-modification, . . .

Enough pitfalls to make it undecidable . . .

**main issue:** distinguishing code vs data . . .

- interleavings between code and data segments
- dynamic jumps (`jmp <register>`) 
- possible variable-length instruction encoding, # addressing modes, . . .
  
  e.g, > 1000 distinct x86 instructions 
  
  1.5 year to fix the semantics of x86 shift instruction at CMU

→ much worse when considering **self-modifying code, packers**, etc.

Example: x86 instruction format
Static Disassembling (2)

Classical static disassembling techniques

- linear sweep: follows increasing addresses (ex: `objdump`)  
  → pb with interleaved code/data ?
- recursive disassembly: control-flow driven (ex: `IDAPro`)  
  → pb with dynamic jumps ?
- hybrid: combines both to better detect errors ...

Some existing tools

- IDA Pro  
  a well-known commercial disassembler, # useful features
- On Linux plateforms (for ELF formats):  
  - `objdump (-S for code disassembling)`  
  - `readelf`
- and many others (Capstone, Miasm, etc.)

Rk: may produce assembly-level IR instead of native assembly code  
→ simpler language (a few instruction opcodes), explicit semantics (no side-effects), share analysis back-ends
Static disassembly (cont’d)

See some Emmanuel Fleury slides . . .
Indirect Jumps

BRANCH $R_i$

(branch address computed at runtime and stored inside register $R_i$)

⇒ A critical issue for static disassemblers/analysers . . .

Occurs when compiling:

- some switch statements
- high-order functions (with function as parameters and/or return values)
- pointers to functions
- dynamic method binding in OO-languages, virtual calls
- etc.
Example of Indirect Jump

(borrowed from E. Fleury)

Source code example:

```c
enum {DIGIT, AT, BANG, MINUS}
int f (char c) {
    switch(c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9': return DIGIT ;
    case '@': return AT ;
    case '!': return BANG ;
    case '-': return MINUS ;
    }
    }
```

\(^3\)See https://godbolt.org/
Example of Indirect Jump

(borrowed from E. Fleury)

Source code example:

enum {DIGIT, AT, BANG, MINUS}
f (char c) {
switch(c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9': return DIGIT ;
case '@': return AT ;
case '!': return BANG ;
case '-': return MINUS ;
}
}

Code produced with x86-64 gcc8.2

f:
    push rbp
    mov rbp, rsp
    mov eax, edi
    mov BYTE PTR [rbp-4], al
    movsx eax, BYTE PTR [rbp-4]
    sub eax, 33 ; Ascii for ’!’
    cmp eax, 31 ; 64 is Ascii for ’@’
    ja .L2 ; out of bounds ...
    mov eax, eax
    mov rax, QWORD PTR .L4[0+rax*8] ; offset in a jump table
    jmp rax

3See https://godbolt.org/
Dynamic disassembly

Main advantage: disassembling process **guided by** the execution

- ensures that instructions only are disassembled
- the whole execution context is available (registers, flags, addresses, etc.)
- dynamic jump destinations are resolved
- dynamic libraries are handled
- etc.

However:

- only a **(small) part** of the executable is disassembled
- need some suitable **execution platform**, e.g.:
  - emulation environment
  - binary level code instrumentation
  - (scriptable) debugger
  - etc.
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools ...
Objectives

When the code has been (partially !) disassembled . . .

. . . how to retrieve useful source-level information ?
(e.g.: variables, types, functions, control and data-flow relations, etc.)

Challenges

Still a gap between assembly and source-level code . . .

▶ basic source elements lost in translation:
  functions, variables, types, (conditionnal) expressions, . . .
▶ pervasive address computations (addresses = values)
▶ etc.

Rk: ̸= between code produced by a compiler and written by hand
     (structural patterns, calling conventions, . . .)

Again, ∃ static and dynamic approaches . . .
Function identification

Retrieve functions boundaries in a stripped binary code?

Why is it difficult?

▶ not always clean call/ret patterns: optimizations, multiple entry points, inlining, etc.
▶ not always clean code segment layout: extra bytes (∉ any function), non-contiguous functions, etc.

Possible solution . . .

▶ from pattern-matching on (manually generated) binary signatures
  ▶ simple ones (push [ebp]) or advanced heuristics as in [IDAPro]
  ▶ standard library function signature database (FLIRT)
▶ . . .
▶ to supervised machine learning classification . . .

→ no “sound and complete” solutions . . .
Variable and type recovery

2 main issues

- retrieve the memory layout (stack frames, heap structure, etc.)
- infer size and (basic) type of each accessed memory location
Variable and type recovery

2 main issues
- retrieve the memory layout (stack frames, heap structure, etc.)
- infer size and (basic) type of each accessed memory location

Memory Layout
“addresses” of global/local variables, parameters, allocated chunks
- static basic access patterns \((epb+offset)\) [IDAPro]
- Value-Set-Analysis (VSA)
Variable and type recovery

2 main issues
▶ retrieve the memory layout (stack frames, heap structure, etc.)
▶ infer size and (basic) type of each accessed memory location

Memory Layout
“addresses” of global/local variables, parameters, allocated chunks
▶ static basic access patterns (\texttt{epb+offset}) [IDAPro]
▶ Value-Set-Analysis (VSA)

Types
▶ dynamic analysis:
  type chunks (library calls) + loop pattern analysis (arrays)
▶ static analysis: VSA + Abstract Structure Identification
▶ Proof-based decompilation relation inference
  type system + program witness [POPL 2016]
Static variable recovery

Retrieve the **address** (and size) of each program “variable”?  

**Difficult because:**

- addresses and other values are not distinguishable
- **address ↦ variable** is not one-to-one
- address arithmetic is pervasive
- both direct and indirect memory addressing

---

Memory regions + abstract locations

A memory model with 3 distinct regions:

- **Global:** global variables
- **Local:** local variables + parameters (1 per proc.)
- **Dynamic:** dynamically allocated chunks

Registers → associates a relative address to each variable (**a-loc**)
Static variable recovery

Retrieve the **address** (and size) of each program “variable”? 

**Difficult because:**
- addresses and other values are not distinguishable
- address $\leftrightarrow$ variable is not one-to-one
- address arithmetic is pervasive
- both direct and indirect memory addressing

**Memory regions + abstract locations**
A memory model with 3 distinct regions:
- Global: global variables
- Local: local variables + parameters (1 per proc.)
- Dynamic: dynamically allocated chunks
- Registers
$\leftrightarrow$ associates a relative address to each variable (**a-loc**)
The so-called “naive” approach (IDAPro)

Heuristic
Adresses used for direct variable accesses are:
  ▶ absolute (for globals + dynamic)
  ▶ relative w.r.t frame/stack pointer (for globals)
→ can be statically retrieved with simple patterns . . .

Limitations
  ▶ variables indirectly accessed (e.g., [eax]) are not retrieved (e.g., structure fields)
  ▶ array = (large) contiguous block of data

⇒ Fast recovery technique, can be used as a bootstrap
But coarse-grained information, may hamper further analyses . . .
typedef struct
    {int i ; char c ;} S ;

int main() {
    S x, a[10] ;
    char *p1 ; int *p2 ;
    p1 = &(a[9].c) ;
    p2 = &(x.i) ;
    return 0 ;
}

<table>
<thead>
<tr>
<th>a</th>
<th>-60</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.i</td>
<td>-10</td>
</tr>
<tr>
<td>p2</td>
<td>-8</td>
</tr>
<tr>
<td>p1</td>
<td>-4</td>
</tr>
</tbody>
</table>

var_60= byte ptr -60h
var_10= byte ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

push   ebp
mov    ebp, esp
sub    esp, 60h
lea    eax, [ebp+var_60]
add    eax, 4Ch
mov    [ebp+var_4], eax
lea    eax, [ebp+var_10]
mov    [ebp+var_8], eax
mov    eax, 0
leave
retn
main endp
Going beyond: Value Set Analysis (VSA)

Compute the contents of each a-loc at each program location . . .

. . . as an over-approximation of:

- the set of (integer) values of each data at each prog. loc.
- the addresses of “new” a-locs (indirectly accessed)

→ combines simultaneously numeric and pointer-analysis

Rk: should be also combined with CFG-recovery . . .

⇒ Can be expressed as a forward data-flow analysis . . .
Going beyond: Value Set Analysis (VSA)

Compute the contents of each a-loc at each program location . . .

. . . as an **over-approximation** of:

- the set of (integer) values of each data at each prog. loc.
- the addresses of “new” a-locs (indirectly accessed)

→ combines simultaneously numeric and pointer-analysis

**Rk:** should be also combined with CFG-recovery . . .

⇒ Can be expressed as a forward data-flow analysis . . .

**A building block for many other static analysis . . .**

- function “signature” (size and number of parameters)
- data-flow dependencies, taint analysis
- alias analysis
- type recovery, abstract structure identification
- etc.
Example: data-flow analysis

Does the value of $y$ depend from $x$?

```c
int x, *p, y;
x = 3;
p = &x;
...
y = *p + 4; // data-flow from x to y?
```

At assembly level:

1. needs to **retrieve** $x$ address
2. needs to **follow** memory transfers from $x$ address ...

```assembly
mov [ebp-4], 3  /* x=3; */
lea eax, [ebp-4]
mov [ebp-8], eax  /* p = &x; */
mov eax, [ebp-8]
...
  /* follow operations on eax ... */
mov eax, [eax]  /* y = *p+4; ??? */
add eax, 4
mov [ebp-12], eax
```
Main issue
handling dynamic jumps (e.g., jmp eax) due to:

- switch statements ("jump table")
- function pointers, trampoline, object-oriented source code, . . .

Some existing solutions

- heuristic-based approach ("simple" switch statements) [IDA]
- abstract interpretation: interleaving between VSA and CFG expansion
  - use of dedicated abstract domains
  - use of under-approximations . . .

**Rk:** may create many program “entry points” ⇒ many CFGs . . .
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools ...
An (ultra) lightweight dynamic technique

Starting from a binary code . . .

- without source, debug information, symbol table
- but those architecture and calling convention is known
- and which can be instrumented & executed

. . . retrieve function-level information

- function arity and signatures
- quantified coarse grain data-flow information between functions

→ within a **single code execution**
General approach

A 3-steps process

1. a lightweight dedicated binary code instrumentation to collect runtime information
2. the one trace execution step to generate a log file
3. an offline log analysis to produce the results . . .

Relying on aggressive heuristics to approximate the notion of parameter, type and data-flow . . .
Main heuristics

parameter definition

a memory location read before written is a input parameter
(holds also across function boundaries)
Main heuristics

parameter definition
a memory location read before written is a input parameter
(holds also across function boundaries)

type definition

- $ADDR$ types can be deduced from load/store operations
- once an $ADDR$, always an $ADDR$
- non $ADDR$ values are of type $NUM$
Main heuristics

parameter definition
a memory location read before written is a input parameter
(holds also across function boundaries)

type definition

- $ADDR$ types can be deduced from load/store operations
- once an $ADDR$, always an $ADDR$
- non $ADDR$ values are of type $NUM$

data-flow definition

- consider only $ADDR$ flows
- $ADDR$ collisions are not fortuitous:
  ADRRR value $a$ produced by $foo$ and consumed by $bar \Rightarrow$ data-flow from $foo$ to $bar$ …
Implementation

**SCAT, open source:**  [https://github.com/Frky/scat](https://github.com/Frky/scat)

- dynamic code instrumentation using PIN
  → function detection based on call/ret instructions

- minimize the size of the instrumentation code
  → extra implementation level heuristics
  (e.g., a value between two ADDR is an ADDR)

- user given MIN_CALL threshold

- embeds an oracle\(^4\) for function signatures

Experiments:
- coreutils (> 100 pgms)
- 10 common Linux pgms: git, grep, mupdf, objdump, openssl, etc.

\(^4\)based on clang
Experimental results: arity

- EVALUATION - PARAMETERS -

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>accuracy</th>
<th>fn</th>
<th>fp</th>
<th>total</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8cc</td>
<td>98 %</td>
<td>4</td>
<td>2</td>
<td>283</td>
<td>5.43</td>
</tr>
<tr>
<td>bash</td>
<td>95 %</td>
<td>8</td>
<td>7</td>
<td>283</td>
<td>1.92</td>
</tr>
<tr>
<td>coreutils</td>
<td>93 %</td>
<td>154</td>
<td>15</td>
<td>2515</td>
<td>3.67</td>
</tr>
<tr>
<td>git</td>
<td>96 %</td>
<td>15</td>
<td>6</td>
<td>492</td>
<td>2.37</td>
</tr>
<tr>
<td>grep</td>
<td>91 %</td>
<td>6</td>
<td>2</td>
<td>86</td>
<td>3.46</td>
</tr>
<tr>
<td>mupdf</td>
<td>94 %</td>
<td>13</td>
<td>7</td>
<td>348</td>
<td>4.55</td>
</tr>
<tr>
<td>objdump</td>
<td>89 %</td>
<td>7</td>
<td>7</td>
<td>132</td>
<td>4.10</td>
</tr>
<tr>
<td>openssl</td>
<td>96 %</td>
<td>4</td>
<td>3</td>
<td>194</td>
<td>3.04</td>
</tr>
<tr>
<td>opusenc</td>
<td>94 %</td>
<td>2</td>
<td>0</td>
<td>36</td>
<td>10.97</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>TOTAL</td>
<td>93 %</td>
<td>248</td>
<td>61</td>
<td>4546</td>
<td>3.76</td>
</tr>
</tbody>
</table>

(average) (sum) (sum) (sum) (average)

October 20th, 2017
Experimental results: types

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>accuracy</th>
<th>fn</th>
<th>fp</th>
<th>total</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8cc</td>
<td>98 %</td>
<td>4</td>
<td>3</td>
<td>307</td>
<td>6.37</td>
</tr>
<tr>
<td>bash</td>
<td>96 %</td>
<td>14</td>
<td>2</td>
<td>374</td>
<td>1.75</td>
</tr>
<tr>
<td>coreutils</td>
<td>92 %</td>
<td>208</td>
<td>55</td>
<td>3299</td>
<td>2.51</td>
</tr>
<tr>
<td>git</td>
<td>94 %</td>
<td>26</td>
<td>4</td>
<td>530</td>
<td>2.12</td>
</tr>
<tr>
<td>grep</td>
<td>95 %</td>
<td>4</td>
<td>3</td>
<td>129</td>
<td>2.99</td>
</tr>
<tr>
<td>mupdf</td>
<td>96 %</td>
<td>17</td>
<td>11</td>
<td>746</td>
<td>7.12</td>
</tr>
<tr>
<td>objdump</td>
<td>94 %</td>
<td>3</td>
<td>12</td>
<td>231</td>
<td>3.59</td>
</tr>
<tr>
<td>openssl</td>
<td>95 %</td>
<td>9</td>
<td>5</td>
<td>308</td>
<td>2.99</td>
</tr>
<tr>
<td>opusenc</td>
<td>98 %</td>
<td>0</td>
<td>1</td>
<td>53</td>
<td>7.23</td>
</tr>
<tr>
<td>TOTAL</td>
<td>96 %</td>
<td>296</td>
<td>101</td>
<td>6294</td>
<td>3.53</td>
</tr>
</tbody>
</table>
Outline

Introduction

Low-level code representations

Disassembling

Retrieving source-level information

Bonus: Dynamic source-level information recovery

Some Tools . . .
IDA Pro [HexRays]

* A swiss-knife for reverse engineering . . .

- Commercial disassembler and debugger
- Supports 50+ processors (intel, ARM, .NET, PowerPC, MIPS, etc.)
- Recognizes library functions FLIRT (C/C++ only)
- Builds call graphs and CFGs
- Tags arguments/local variables
- Rename labels (variables names etc.)
- Provides scripting environment (IDC, Python) and debugging facilities
Script example

```c
#include <idc.idc>
/* this IDA pro script enumerate all functions and prints info about them */
static main()
{
    auto addr, end, args, locals, frame, firstArg, name, ret;
    addr=0;
    for ( addr=NextFunction(addr); addr != BADADDR; addr=NextFunction(addr) )
    {
        name=Name(addr);
        end= GetFunctionAttr(addr,FUNCATTR_END);
        locals=GetFunctionAttr(addr,FUNCATTR_FRSIZE);
        frame=GetFunctionAttr(aiddr,FUNCATTR_FRAME);
        ret=GetMemberOffset(frame, " r");
        if (ret == -1) continue;
        firstArg=ret +4;
        args=GetStrucSize(frame) -firstArg;
        Message("function %s start at %x, end at %x\n",name, addr, end);
        Message("Local variables size is %d bytes\n",locals);
        Message("arguments size %d (%d arguments)\n",args, args/4);
    }
}
```
PIN [Intel]

A swiss-knife for binary-level dynamic analysis . . .

A dynamic code instrumentation framework

- run time instrumentation on the binary files
- provides APIs to define insertion points and callbacks (e.g., after specific inst., at each function entry point, etc.)
- Free for non-commercial use, works on Linux and windows
Example: instruction counting

```c
#include "pin.h"
UINT64 icount = 0;
void docount() { icount++; }

void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```