CS 6V81-05: System Security and Malicious Code Analysis
Understanding the Binary Representation

Zhiqiang Lin
Department of Computer Science
University of Texas at Dallas

January 23rd, 2012

Outline
1 x86 Architecture
2 Data Representation
3 Code Representation
4 Summary

Outline
1 x86 Architecture
2 Data Representation
3 Code Representation
4 Summary

Outline
x86 Architecture

1 x86 Architecture
2 Data Representation
3 Code Representation
4 Summary
Von Neumann Architecture

**Data**
- Software code is loaded memory
- Program data is also in the memory

**Attacker's goal**
- Control the data
- The data could be interpreted as code

Architecture Approach to stop attack

Separating Code and Data
### Binary Data In Memory and Disk

**Binary Representations**

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000010</td>
<td>02 00 3E 00 01 00 00 00 90 24 40 00 00 00 00 00</td>
</tr>
<tr>
<td>00000020</td>
<td>00 00 00 40 00 38 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000030</td>
<td>00 00 03 00 96 01 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000040</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000050</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000060</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000070</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000080</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000090</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

#### Data Type
- u/int_8 (char), u/int_16, u/int_32, pointers
- float, double(8)
- C-String, Unicode-String

### Memory Organization

- **Array of Data**
  - Virtual Addresses (Array Index)
  - OS provides address space private to particular “process”
  - Program can clobber its own data, but not that of others
  - Compiler + Run-Time System Control data allocation/de-allocation

### Machine Words

- **Machine Has “Word Size”**
  - Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Machines support multiple data formats
Word-Oriented Memory Organization

Addresses Specify Byte Locations
- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

Byte Ordering

How should bytes within a multi-byte word be ordered in memory?
- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address

Examples

Big Endian: Sun, PPC Mac, Internet
- Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address

0x01234567 at address 0x100

Reading Byte-Reversed Listings

Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

Assembly Example

401e50: ff 35 9a 71 21 00 pushq 0x21719a(%rip)
401e56: ff 25 9c 71 21 00 jmpq *0x21719c(%rip)
401e5c: 0f 1f 40 00 nopl 0x0(%rax)
Representing Integers

1. int A = 15213;
2. int B = -15213;

Question?
- Will always $x^2 > 0$?
- Integer overflow

Arbitrary-precision Arithmetic

In computer science, arbitrary-precision arithmetic indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system.

Using Arrays

Several modern programming languages have built-in support for bignums, and others have libraries available for arbitrary-precision integer and floating-point math. Rather than store values as a fixed number of binary bits related to the size of the processor register, these implementations typically use variable-length arrays of digits.


Representing Strings

Strings in C
- Represented by array of characters
- Each character encoded in ASCII format
- Standard 7-bit encoding of character set
- Character “0” has code 0x30
- Digit i has code 0x30+i
- String should be null-terminated
- Final character = 0

Representing of Float Point

float: IEEE 754

A numerical value $n$ for a float variable is:

$$ n = (1 - 2s) \times (1 + m \times 2^{-23}) \times 2^{e-127}, $$

where $s$ is a sign bit (zero or one), $m$ is the significand (i.e., the fraction part), and $e$ is the exponent.
### How to represent $\pi$

$$\pi = 6 \arcsin \frac{1}{2} = 3 \sum_{n=0}^{\infty} \frac{\binom{2n}{n}}{16^n (2n+1)}$$

$$= 6 \left( \frac{1}{1} + \frac{1}{2} \cdot \frac{1}{2^3} + \frac{1}{2^4} \cdot \frac{3}{2^5} + \frac{1}{2^6} \cdot \frac{3}{2^7} + \cdots \right)$$

$$= 3 \left( \frac{1}{8} + \frac{9}{64} + \frac{15}{768} + \frac{495}{98304} + \frac{3465}{2883584} + \frac{54552562}{1567772160} + \cdots \right)$$

### Sign Extension

**Task**
- Given $w$-bit signed integer $x$
- Convert it to $w+k$-bit integer with same value

**Rules**
- Make $k$ copies of sign bit

**Example**

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

### Complement & Increment (Integer Overflow)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>3B</td>
<td>00111011 11010110</td>
</tr>
<tr>
<td>-15213</td>
<td>C4</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>15213+1</td>
<td>C4</td>
<td>11000110 10010011</td>
</tr>
<tr>
<td>-15213</td>
<td>C4</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

**Table:** $x = 15213$

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>-0</td>
<td>-1</td>
<td>FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

**Table:** $x = 0$
Instruction Decoding: One-to-one mapping?

(a) Linux Kernel

0xc1087c86: e8 25 2c 00 00 call 0xc108a8b0
0xc108a8b0: 55 push ebp
0xc108a8b1: ...

(b) Windows Kernel

0x806eee0a: e8 3d 69 00 00 call 0x806f574c
0x806f574c: 8b ff mov edi, edi
0x806f574e: 55 push ebp
0x806f574f: 8b ec mov ebp, esp

(c) Solaris Kernel

0xfe801a3c: e8 9c 01 00 00 call 0xfe801bdd
0xfe801bdd: 55 push ebp
0xfe801bde: 8b ec mov ebp, esp

(d) FreeBSD/OpenBSD Kernel

0xc04d675f: e8 0c cf 00 00 call 0xc04e3670
0xc04e3670: 55 push ebp
0xc04e3671: 89 e5 mov ebp, esp

Encoding Real x86 Instructions
x86 Instruction Format Reference

Legacy prefixes (optional)
- Lock prefix (1 byte)
- Repeat prefix (1 byte)
- Segment override prefix (1 byte)
- Operand-size override prefix (1 byte)
- Address-size override prefix (1 byte)
- REX prefix (1 byte, 64-bit only)

Encoding
- Opcode (1, 2 bytes, required)
- ModR/M (1 byte, if required)
- SIB (1 byte, if required)
- Displacement (1, 2 or 4 bytes, if required)
- Immediate (1, 2 or 4 bytes, if required)

Decoding Statement Machine

x86 Instruction Operands, MOD-REG-R/M

Byte

<table>
<thead>
<tr>
<th>REG</th>
<th>Value</th>
<th>ModR/M</th>
<th>REG</th>
<th>RM</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>REG</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>REG</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>REG</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>REG</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>REG</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>REG</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>REG</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>REG</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Register addressing modes:
- R = Register
- D = Direct
- I = Immediate
- M = Literal

Immediate addressing modes:
- 0 = Direct
- 1 = Indirect
- 2 = Base
- 3 = Indirect
- 4 = Register
- 5 = Literal
- 6 = Register

ModRM field

<table>
<thead>
<tr>
<th>ModRM</th>
<th>REG</th>
<th>RM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

R = Register
D = Direct
I = Immediate
M = Literal

Reset

Prefix

Opcode

ModRM

SIB

Disp

Imm
**Encodings**

### x86 Architecture

- **Data Representation**
  - [Diagram](image)
- **Code Representation**
  - [Diagram](image)
- **Summary**
  - [Diagram](image)

#### Encoding ADD CL, AL Instruction
- **add cl, al**
- 00 C1 (if \(d=0\)), or 02 C8, if \(d\) bit is set to 1.

#### Encoding ADD ECX, EAX Instruction
- **add ecx, eax**
- Could also encode ADD ECX, EAX using the bytes 03 C8

#### Encoding ADD EDX, DISPLACEMENT Instruction
- **add edx, disp**

#### Encoding ADD EDI, [EBX] Instruction
- **add edi, [ebx]**

---

**Note:**
- The diagrams and images are not directly translatable into text due to their complexity and visual nature. They provide additional visual aids to understand the encoding processes of the specific instructions.

---

**Additional Information:**
- The instructions are encoded using specific opcodes as described.
- The diagrams help in visualizing the encoding process and operand addressing.
- The use of bytes 03 C8 for encoding ADD ECX, EAX is noteworthy for its flexibility in implementation.
- The encoding of ADD instructions involves understanding the role of the destination register (RDX or EDX) and the source register (ECX or EDX, respectively) in the x86 architecture.

---

**Questions:**
- What is the significance of the 00 C1 and 02 C8 opcodes in encoding ADD CL, AL instruction?
- How does the opcode 03 C8 facilitate the encoding of ADD ECX, EAX instruction?
- What does the diagram for ADD EDI, [EBX] Instruction illustrate?
- Can you explain why the encoding process of ADD EDX, DISPLACEMENT Instruction involves the use of the displacement field?
**ADD EAX, [ESI + disp8]**

- **add eax, [esi + disp8]**

**Encoding ADD EBX, [EBP + disp32] Instruction**

- **add ebx, [ebp + disp32]**

**ADD EBP, [disp32 + EAX*1] Instruction**

- **add ebp, [disp32 + eax*1]**

**Encoding ADD ECX, [EBX + EDI*4] Instruction**

- **add ecx, [ebx + edi*4]**
Encoding ADD EBP, [ disp32 + EAX*1 ] Instruction

Examples

402688: 83 fa 06 cmp $0x6,%edx
40268b: 0f 94 c1 sete %cl
40268e: 84 c9 test %cl,%cl
402690: b8 40 00 00 00 mov $0x40,%eax
402695: 75 41 jne 4026d8 <wcstombs@plt+0x258>
402697: 40 84 ff test %dil,%dil
40269a: 0f 84 a0 00 00 00 je 402740 <wcstombs@plt+0x2c0>
40269d: 89 00 mov %esi,%eax
40269e: 3d 00 10 00 00 cmp $0xa000,%eax
4026a0: b8 40 00 00 00 mov $0x40,%eax
4026a5: 75 41 jne 4026d8 <wcstombs@plt+0x258>
4026a8: 0f 84 8f 00 00 00 je 402750 <wcstombs@plt+0x2d0>
4026ac: 81 e6 00 f0 00 00 and $0xf000,%esi
4026ae: 81 fe 00 c0 00 00 cmp $0xc000,%esi
4026b0: 0f 94 c0 sete %al
4026b3: f7 d8 neg %eax
4026b5: 83 e0 3d and $0x3d,%eax
4026b8: 0f 1f 00 nopl (%rax)
4026ba: f3 c3 repz retq
4026da: 66 0f 1f 44 00 00 nopw 0x0(%rax,%rax,1)
...
403594: b8 ff ff ff ff mov $0xffffffff,%eax
403599: c3 retq
References

- Computer Systems: A Programmer's Perspective, 2nd Edition (chapter 2)
- http://www.m5sim.org/X86_Instruction_decoding