15-213
“The Class That Gives CMU Its Zip!”

Bits, Bytes, and Integers
August 28, 2008

Topics
- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C
- Representations of Integers
  - Basic properties and operations
  - Implications for C

Binary Representations

Base 2 Number Representation
- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
- Represent $1.5213 \times 10^4$ as $1.110110110111 \times 2^{13}$

Electronic Implementation
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

Encoding Byte Values

Byte = 8 bits
- Binary: $00000000_2$ to $11111111_2$
- Decimal: $0_{10}$ to $255_{10}$
  - First digit must not be 0 in C
- Hexadecimal: $00_{16}$ to $FF_{16}$
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write $FA1D37B_{16}$ in C as $0xFA1D37B$
    » Or $0xfalD37b$

Byte-Oriented Memory Organization

Programs Refer to Virtual Addresses
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

Compiler + Run-Time System Control Allocation
- Where different program objects should be stored
- All allocation within single virtual address space
Machine Words

Machine Has “Word Size”
- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space \( \approx 1.8 \times 10^{19} \) bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

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)

Data Representations

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

» Or any other pointer

Byte Ordering

How should bytes within multi-byte word be ordered in memory?

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

Big Endian
- Least significant byte has highest address

Little Endian
- Least significant byte has lowest address

Example
- Variable \( x \) has 4-byte representation \( 0x01234567 \)
- Address given by &\( x \) is \( 0x100 \)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01 23 45 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>

Reading Byte-Reversed Listings

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

Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365</td>
<td>5b</td>
<td>pop $ebx</td>
</tr>
<tr>
<td>8048366</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl 0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers
- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Examining Data Representations

Code to Print Byte Representation of Data
- Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p	0x%6X
", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal

show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):
```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcbb9 0x3b
0x11ffffcbb8 0x00
0x11ffffcbb8 0x00
```
Representing Integers

int A = 15213;
int B = -15213;
long int C = 15213;

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

IA32, x86-64 A Sun A
6D
3B
00
00

IA32 C x86-64 C Sun C
6D
3B
00
00

IA32, x86-64 B Sun B
93
C4
FF
FF

Two's complement representation (Covered later)

Representing Pointers

int B = -15213;
int *P = &B;

Sun P IA32 P x86-64 P
EF D4 0C
FF FB 89
FB FF EC
2C BF FF

Different compilers & machines assign different locations to objects

Representing Strings

char S[6] = "15213";

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

Compatibility
- Byte ordering not an issue

Linux/Alpha S Sun S
31 31
35 35
32 32
31 31
33 33
00 00

Boolean Algebra

Developed by George Boole in 19th Century
- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And
- A&B = 1 when both A=1 and B=1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Or
- A|B = 1 when either A=1 or B=1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not
- ~A = 1 when A=0

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)
- A^B = 1 when either A=1 or B=1, but not both

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Application of Boolean Algebra

Applied to Digital Systems by Claude Shannon
- 1937 MIT Master’s Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

\[
\begin{align*}
A \& \sim B & \quad \text{Connection when} \\
\sim A \& B & = A^B
\end{align*}
\]

General Boolean Algebras

Operate on Bit Vectors
- Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
& \quad 01010101 & \quad 01010101 & \quad 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

Representation
- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) if \( j \in A \)
  - \( 01101001 \) \( \{0, 3, 5, 6\} \)
  - \( 01010101 \) \( \{0, 2, 4, 6\} \)

Operations
- \& Intersection \quad 01000001 \( \{0, 6\} \)
- | Union \quad 01111101 \( \{0, 2, 3, 4, 5, 6\} \)
- ^ Symmetric difference \quad 00111100 \( \{2, 3, 4, 5\} \)
- ~ Complement \quad 10101010 \( \{1, 3, 5, 7\} \)

Bit-Level Operations in C

Operations \&, |, ~, ^ Available in C
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)
- \(~0x41 --> 0xBE\)
- \(~01000001_2 --> 10111110_2\)
- \(~0x00 --> 0xFF\)
- \(~0x0000000_2 --> 11111111_2\)
- \(0x69 \& 0x55 --> 0x41\)
- \(01101001_2 \& 01010101_2 --> 01000001_2\)
- \(0x69 | 0x55 --> 0x7D\)
- \(01101001_2 | 01010101_2 --> 01111101_2\)
Contrast: Logic Operations in C

Contrast to Logical Operators
- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

Examples (char data type)
- !0x41 --> 0x00
- !0x00 --> 0x01
- !!0x41 --> 0x01
- 0x69 && 0x55 --> 0x01
- 0x69 || 0x55 --> 0x01
- p && *p (avoids null pointer access)

Shift Operations
Left Shift: x << y
- Shift bit-vector x left y positions
  - Throw away extra bits on left
  - Fill with 0's on right
- Log. >> 2
  - 00010000
- Arith. >> 2
  - 00101000

Right Shift: x >> y
- Shift bit-vector x right y positions
  - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on right
- Log. >> 2
  - 00010000
- Arith. >> 2
  - 11101000

Undefined Behavior
- Shift amount < 0 or ≥ word size

Integer C Puzzles
- Assume 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

Examples:
- x < 0
- x >= 0
- x & 7 == 7
- x >= -1
- x > y
d, x * x >= 0
- x > 0 && y > 0
- x >= 0
- x <= 0
- (x|-x)>>31 == -1
- ux >> 3 == ux/8
- x >> 3 == x/8
- x & (x-1) != 0

Encoding Integers
Unsigned
\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

Two's Complement
\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

Sign Bit
- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Encodings Example (Cont.)

\[ x = \begin{bmatrix} 15213 & : & 00111011 & 01101101 \\ y = -15213 & : & 11000100 & 10010011 \end{bmatrix} \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>

Encoding Example (Cont.)

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>

Numeric Ranges

Unsigned Values
- \( U_{\text{Min}} = 0 \)
- \( U_{\text{Max}} = 2^w - 1 \)

Two's Complement Values
- \( T_{\text{Min}} = -2^{w-1} \)
- \( T_{\text{Max}} = 2^{w-1} - 1 \)

Other Values
- Minus 1
- \( 111...1 \)

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>UMax</th>
<th>TMax</th>
<th>TMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>255</td>
<td>127</td>
<td>-128</td>
</tr>
<tr>
<td>16</td>
<td>65,535</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td>32</td>
<td>4,294,967</td>
<td>2,147,483</td>
<td>-2,147,483</td>
</tr>
<tr>
<td>64</td>
<td>18,446,744</td>
<td>9,223,732</td>
<td>-9,223,732</td>
</tr>
</tbody>
</table>

C Programming

- \#include <limits.h>
- K&R App. B11
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific

Observations
- \( |T_{\text{Min}}| = T_{\text{Max}} + 1 \)
  - Asymmetric range
- \( U_{\text{Max}} = 2^w T_{\text{Max}} + 1 \)

Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>X</th>
<th>B2U(X)</th>
<th>B2T(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

Equivalence
- Same encodings for nonnegative values

Uniqueness
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

\( \Rightarrow \) Can Invert Mappings
- \( U2B(x) = B2U^{-1}(x) \)
  - Bit pattern for unsigned integer
- \( T2B(x) = B2T^{-1}(x) \)
  - Bit pattern for two's comp integer
Mapping Between Signed & Unsigned

- **Two's Complement**: Maintain Same Bit Pattern
  - T2U: \( x \) → \( u_x \)
  - U2T: \( u_x \) → \( x \)

Define mappings between unsigned and two's complement numbers based on their bit-level representations.

Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0111</td>
<td>-8</td>
<td>-7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>-6</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>-5</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>-4</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>-2</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relation between Signed & Unsigned

- **Two's Complement**: Maintain Same Bit Pattern
  - T2U: \( x \) → \( u_x \)
  - U2T: \( u_x \) → \( x \)

\[ u_x = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]
Signed vs. Unsigned in C

Constants
- By default are considered to be signed integers
- Unsigned if have “U” as suffix 0U, 4294967259U

Casting
- Explicit casting between signed & unsigned same as U2T and T2U
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;
- Implicit casting also occurs via assignments and procedure calls
  tx = ux;
  uy = ty;

Casting Surprises
Expression Evaluation
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

<table>
<thead>
<tr>
<th>Constant₁</th>
<th>Constant₂</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>

Explanation of Casting Surprises
2's Comp. → Unsigned
- Ordering Inversion
- Negative → Big Positive

Code Security Example #1

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    // Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```
- Similar to code found in FreeBSD's implementation of getpeername.
- There are legions of smart people trying to find vulnerabilities in programs
  - Think of it as a very stringent testing environment
**Typical Usage**

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
   /* Byte count len is minimum of buffer size and maxlen */
   int len = KSIZE < maxlen ? KSIZE : maxlen;
   memcpy(user_dest, kbuf, len);
   return len;
}
```

```c
#define MSIZE 528
void getstuff() {
   char mybuf[MSIZE];
   copy_from_kernel(mybuf, MSIZE);
   printf("%s\n", mybuf);
}
```

**Malicious Usage**

```c
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
   /* Byte count len is minimum of buffer size and maxlen */
   int len = KSIZE < maxlen ? KSIZE : maxlen;
   memcpy(user_dest, kbuf, len);
   return len;
}
```

```c
#define MSIZE 528
void getstuff() {
   char mybuf[MSIZE];
   copy_from_kernel(mybuf, -MSIZE);
   . . .
}
```

**Sign Extension**

**Task:**
- Given w-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer with same value

**Rule:**
- Make \( k \) copies of sign bit:
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)
- \( k \) copies of MSB

**Sign Extension Example**

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>001111011 01101101</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Why Should I Use Unsigned?

Don’t Use Just Because Number Nonnegative

- Easy to make mistakes
  ```c
  unsigned i;
  for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
  ```
- Can be very subtle
  ```c
  #define DELTA sizeof(int)
  int i;
  for (i = CNT; i-DELTA >= 0; i-= DELTA)
    ...
  ```

Do Use When Performing Modular Arithmetic

- Multiprecision arithmetic

Do Use When Using Bits to Represent Sets

- Logical right shift, no sign extension

Complement & Increment

Claim: Following Holds for 2’s Complement

### Complement

- Observation:
  ```c
  ~x + x == 1111…111
  ```

### Increment

- ~x + x == -1
- ~x + x + (-x + 1) == -x + (-x + 1)
- ~x + 1 == -x

Warning: Be cautious treating `int’s as integers`

Comp. & Incr. Examples

<table>
<thead>
<tr>
<th>x</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
<td></td>
</tr>
<tr>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
<td></td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
</tbody>
</table>

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

<table>
<thead>
<tr>
<th>u</th>
<th>v</th>
<th>u + v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard Addition Function

- Ignores carry output

**Implements Modular Arithmetic**

\[ s = UAdd_w(u, v) = u + v \mod 2^w \]

\[
UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
Visualizing Integer Addition

**Integer Addition**
- 4-bit integers \( u, v \)
- Compute true sum \( \text{Add}_4(u, v) \)
- Values increase linearly with \( u \) and \( v \)
- Forms planar surface

**Visualizing Unsigned Addition**

**Wraps Around**
- If true sum \( \geq 2^w \)
- At most once

Mathematical Properties

**Modular Addition Forms an Abelian Group**
- Closed under addition
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]
- Commutative
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]
- Associative
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]
- 0 is additive identity
  \[ \text{UAdd}_w(0, u) = u \]
- Every element has additive inverse
  \[ \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]

Two’s Complement Addition

**TAdd and UAdd have Identical Bit-Level Behavior**
- Signed vs. unsigned addition in C:
  \[ \text{int } s, t, u, v; \]
  \[ s = \text{(int) } ((\text{unsigned}) u + (\text{unsigned}) v); \]
  \[ t = u + v \]
- Will give \( s == t \)
Characterizing TAdd

**Functionality**
- True sum requires \( w + 1 \) bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

\[
\text{TAdd}(u, v) = \begin{cases} 
    u + v + 2^{-w-1} & \text{if sum } < \text{TMin}_w \\
    u + v & \text{if } \text{TMin}_w \leq u + v \leq \text{TMax}_w \\
    u + v - 2^{-w-1} & \text{if sum } > \text{TMax}_w
\end{cases}
\]

Visualizing 2’s Comp. Addition

**Values**
- 4-bit two’s comp.
- Range from -8 to +7

**Wraps Around**
- If sum \( \geq 2^{w-1} \)
  - Becomes negative
  - At most once
- If sum \( \leq -2^{w-1} \)
  - Becomes positive
  - At most once

Mathematical Properties of TAdd

**Isomorphic Algebra to UAdd**
- \( \text{TAdd}_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \)
  - Since both have identical bit patterns

Two’s Complement Under TAdd Forms a Group
- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse

\[
T\text{Comp}_w(u) = \begin{cases} 
    -u & u \neq \text{TMin}_w \\
    \text{TMin}_w & u = \text{TMin}_w
\end{cases}
\]

Multiplication

**Computing Exact Product of \( w \)-bit numbers \( x, y \)**
- Either signed or unsigned

**Ranges**
- **Unsigned**: \( 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  - Up to \( 2w \) bits
- **Two’s complement min**: \( x \times y \geq (-2^{w-1})^2(2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \)
  - Up to \( 2w-1 \) bits
- **Two’s complement max**: \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)
  - Up to \( 2w \) bits, but only for \( \text{TMin}_w^2 \)

**Maintaining Exact Results**
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages
**Unsigned Multiplication in C**

Operands: $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th>$v$</th>
</tr>
</thead>
</table>

True Product: $2^w$ bits

| $u \cdot v$ |

Discard $w$ bits: $w$ bits

| UMult$(u, v)$ |

---

**Standard Multiplication Function**
- Ignores high order $w$ bits

**Implements Modular Arithmetic**

$$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$

---

**Code Security Example #2**

- SUN XDR library
  - Widely used library for transferring data between machines

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```

XDR Code

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /* Allocate buffer for ele_cnt objects, each of ele_size bytes *
    * and copy from locations designated by ele_src */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL) /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object $i$ to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

XDR Vulnerability

```c
malloc(ele_cnt * ele_size)
```

What if:
- $\text{ele_cnt} = 2^{20} + 1$
- $\text{ele_size} = 4096 = 2^{12}$
- Allocation = ??

How can I make this function secure?
**Signed Multiplication in C**

Operands: $w$ bits

<table>
<thead>
<tr>
<th>$u$</th>
<th>*</th>
<th>$v$</th>
</tr>
</thead>
</table>

True Product: $2^w$ bits

<table>
<thead>
<tr>
<th>$u \cdot v$</th>
</tr>
</thead>
</table>

Discard $w$ bits: $w$ bits

<table>
<thead>
<tr>
<th>$\text{TMult}_w(u, v)$</th>
</tr>
</thead>
</table>

**Standard Multiplication Function**

- Ignores high order $w$ bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

**Power-of-2 Multiply with Shift**

**Operation**

- $u \ll k$ gives $u \times 2^k$
- Both signed and unsigned

**Examples**

- $u \ll 3 == u \times 8$
- $u \ll 5 - u \ll 3 == u \times 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

**Compiled Multiplication Code**

**C Function**

```c
int mul12(int x)
{
    return x*12;
}
```

**Compiled Arithmetic Operations**

```c
leal (%eax,%eax,2), %eax
sall $2, %eax
```

- C compiler automatically generates shift/add code when multiplying by constant

**Unsigned Power-of-2 Divide with Shift**

**Quotient of Unsigned by Power of 2**

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift

<table>
<thead>
<tr>
<th>$u$</th>
</tr>
</thead>
</table>

| $u / 2^k$ |

Result:

| $\lfloor u / 2^k \rfloor$ |

**Table**

<table>
<thead>
<tr>
<th>Operands: $u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u / 2^k$</td>
<td></td>
</tr>
</tbody>
</table>

**Binary Point**

<table>
<thead>
<tr>
<th>$\text{Hex}$</th>
<th>$\text{Binary}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>00111011 101101101</td>
</tr>
<tr>
<td>7606.5</td>
<td>10 D 66 000011101101</td>
</tr>
<tr>
<td>950.8125</td>
<td>03 86 00000011 10110110</td>
</tr>
<tr>
<td>59.4257813</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

```c
unsigned udiv8(unsigned x) {
    return x/8;
}
```

Compiled Arithmetic Operations

```c
shrl $3, %eax
```

Explanation

- Uses logical shift for unsigned

For Java Users

- Logical shift written as >>>

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- $x >> k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$

<table>
<thead>
<tr>
<th>Operands:</th>
<th>$x$</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>$/ 2^k$</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Division:

$\lfloor x / 2^k \rfloor$

Result: RoundDown($x / 2^k$)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y &gt;&gt; 1</td>
<td>-15213</td>
<td>C4 93</td>
<td>1100010010010011</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>1100010010010011</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>1111110010010011</td>
</tr>
<tr>
<td>y &gt;&gt; 16</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>1111111110010011</td>
</tr>
</tbody>
</table>

Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want $\lfloor x / 2^k \rfloor$ (Round Toward 0)
- Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
  - In C: $(x + (1<<k)-1) >> k$
  - Biases dividend toward 0

Case 1: No rounding

Dividend:

```
  \[ \underbrace{\ldots 1}_{k} \underbrace{0 0 0 0}_{+2^k-1} \]
```

Divisor:

```
  \[ \underbrace{\ldots 1}_{k} \underbrace{0 0 0 0}_{/ 2^k} \]
```

Biases has no effect

Case 2: Rounding

Dividend:

```
  \[ \underbrace{\ldots 0}_{k} \underbrace{1 1 1 1}_{+2^k-1} \]
```

Incremented by 1

Binomial Point

Biasing adds 1 to final result

Divisor:

```
  \[ \underbrace{\ldots 1}_{k} \underbrace{1 1 1 1}_{/ 2^k} \]
```

Incremented by 1

Biased has effect
Compiled Signed Division Code

C Function

```c
int idiv8(int x) {
    return x/8;
}
```

Compiled Arithmetic Operations

```c
testl %eax, %eax
js L4
L3:
sarl $3, %eax
# Arithmetic shift
return x >> 3;
```

For Java Users

- Arithmetic shift written as `>>`

---

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms

- Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
    
    \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
  - Multiplication Commutative
    
    \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
  - Multiplication is Associative
    
    \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
  - 1 is multiplicative identity
    
    \[ \text{UMult}_w(u, 1) = u \]
  - Multiplication distributes over addition
    
    \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]

---

Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to \( w \) bits
- Two’s complement multiplication and addition
  - Truncating to \( w \) bits

Both Form Rings

- Isomorphic to ring of integers mod \( 2^w \)

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,
  
  \[ u > 0 \Rightarrow u + v > v \]
  \[ u > 0, v > 0 \Rightarrow u \cdot v > 0 \]

These properties are not obeyed by two’s comp. arithmetic

<table>
<thead>
<tr>
<th>( T_{Max} + 1 )</th>
<th>( T_{Min} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 15213 \times 30426 )</td>
<td>( -10030 ) (16-bit words)</td>
</tr>
</tbody>
</table>

---

Integer C Puzzles Revisited

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt; 0 )</td>
<td>((x \times 2) &lt; 0)</td>
</tr>
<tr>
<td>( ux &gt;= 0 )</td>
<td>(x \ll 7 \Rightarrow x &lt; 30 &lt; 0)</td>
</tr>
<tr>
<td>( ux &lt; -1 )</td>
<td>(x &gt; 7 \Rightarrow -x &lt; -y)</td>
</tr>
<tr>
<td>( ux &gt; 7 )</td>
<td>(x \gg 0 \Rightarrow x + y &gt; 0)</td>
</tr>
<tr>
<td>( x &gt; 0 )</td>
<td>(-x &lt; 0)</td>
</tr>
<tr>
<td>( x &lt; 0 )</td>
<td>(-x &gt;= 0)</td>
</tr>
<tr>
<td>( x &lt; 0 )</td>
<td>((x</td>
</tr>
<tr>
<td>( ux &lt;&lt; 3 )</td>
<td>(ux/8)</td>
</tr>
<tr>
<td>( x &gt; 3 )</td>
<td>(x/8)</td>
</tr>
<tr>
<td>( x &lt; 0 )</td>
<td>(x &amp; (x-1) != 0)</td>
</tr>
</tbody>
</table>

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```