Lecture 2: Integers
Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last Time: Bits & Bytes

- Bits, Bytes, Words
- Decimal, binary, hexadecimal representation
- Virtual memory space, addressing, byte ordering
- Boolean algebra
- Bit versus logical operations in C
Today: Integers

• Representation: unsigned and signed
• Conversion, casting
• Expanding, truncating
• Addition, negation, multiplication, shifting
• Summary
Encoding integers

Unsigned

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

Two’s complement

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

- A C `short` is 2 bytes long

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

- Sign bit
  - For 2’s complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative
Encoding example (cont.)

\[ x = 15213: \quad 00111011 \ 01101101 \]
\[ y = -15213: \quad 11000100 \ 10010011 \]

<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>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</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
  – UMin = 0
    • 000...0
  – UMax = $2^w - 1$
    • 111...1

• Two’s complement values
  – Tmin = $-2^{w-1}$
    • 100...0
  – Tmax = $2^{w-1} - 1$
    • 011...1

<table>
<thead>
<tr>
<th>Values for w=16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decimal</strong></td>
</tr>
<tr>
<td>UMax</td>
</tr>
<tr>
<td>Tmax</td>
</tr>
<tr>
<td>Tmin</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Values for different word sizes

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

- **Observations**
  - $|T_{\text{Min}}| = T_{\text{Max}} + 1$
  - Asymmetric range
  - $U_{\text{Max}} = 2 \times T_{\text{Max}} + 1$

- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific
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

- ⇒ **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
Today: Integers

• Representation: unsigned and signed
• Conversion, casting
• Expanding, truncating
• Addition, negation, multiplication, shifting
• Summary
Mapping between signed & unsigned

- Mappings between unsigned and two’s complement numbers:
  keep bit representations and reinterpret
# 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>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>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>
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>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>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Signed values range from -8 to -1, and unsigned values range from 0 to 15. The mapping is achieved by adding 16 to the signed values before mapping to the unsigned range.
Relation between signed & unsigned

Two's complement

Unsigned

Maintain same bit pattern

\[ ux = \begin{cases} 
  x & x \geq 0 \\
  x + 2^w & x < 0 
\end{cases} \]

Large negative weight becomes

Large positive weight
Conversion visualized

• 2’s Comp. → Unsigned
  – Ordering inversion
  – Negative → big positive

2’s complement range

\[
\begin{align*}
T_{\text{Max}} & \rightarrow U_{\text{Max}} - 1 \\
T_{\text{Max} + 1} & \rightarrow 0 \\
T_{\text{Max}} & \rightarrow \text{unsigned range}
\end{align*}
\]
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: TMIN = -2,147,483,648, TMAX = 2,147,483,647

<table>
<thead>
<tr>
<th>Constant 1</th>
<th>Constant 2</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>Signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Code security example

```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;
}
```

• Similar to code found in FreeBSD’s implementation of `getpeername`
• There are legions of smart people trying to find vulnerabilities in programs
Typical usage

/* 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;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
Malicious usage

/* 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;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
Summary: casting signed ↔ unsigned

• Bit pattern is maintained
• But reinterpreted
• Can have unexpected effects: adding or subtracting $2^w$

• Expression containing signed and unsigned int
  – int is cast to unsigned!!
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
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:
  – $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

\[X\] $\Rightarrow$ $X'$ with $k$ copies of MSB

\[\text{w} \quad \text{w+k}\]
Sign extension example

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

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Summary:
Expanding, truncating: basic rules

• Expanding (e.g., short int to int)
  – Unsigned: zeros added
  – Signed: sign extension
  – Both yield expected result

• Truncating (e.g., unsigned to unsigned short)
  – Unsigned/signed: bits are truncated
  – Result reinterpreted
  – Unsigned: mod operation
  – Signed: similar to mod
  – For small numbers yields expected behaviour
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
Negation: complement & increment

- Claim: the following holds for 2’s complement:
  \[ \sim x + 1 = -x \]

- Complement
  - Observation: \( \sim x + x = 1111\ldots111 = -1 \)

<table>
<thead>
<tr>
<th>x</th>
<th>1 0 0 1 1 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ ( \sim x )</td>
<td>0 1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>( \sim x )</td>
<td>0 1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>( -1 )</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

- Complete proof?
Complement & increment examples

### $x = 15213$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$\sim x$</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\sim x + 1$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

### $x = 0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\sim 0 + 1$</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned addition

Operands: \( w \) bits

True sum: \( w+1 \) bits

Discard carry: \( w \) bits

- Standard addition function
  - Ignores carry output
- Implements modular arithmetic

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

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

True Sum
$2^{w+1}$
$2^w$
0
Modular Sum

Overflow

$U\text{Add}_4(u, v)$

Overflow
Mathematical properties

• Modular addition forms an Abelian group
  – Closed under addition
    \[ 0 \leq U\text{Add}_w(u, v) \leq 2^w - 1 \]
  – Commutative
    \[ U\text{Add}_w(u, v) = U\text{Add}_w(v, u) \]
  – Associative
    \[ U\text{Add}_w(t, U\text{Add}_w(u, v)) = U\text{Add}_w(U\text{Add}_w(t, u), v) \]
  – 0 is additive identity
    \[ U\text{Add}_w(u, 0) = u \]
  – Every element has additive inverse
    Let: \[ U\text{Comp}_w(u) = 2^w - u \]
    Then: \[ U\text{Add}_w(u, U\text{Comp}_w(u)) = 0 \]
Two’s complement addition

Operands: \( w \) bits

\[ u \]
\[ + \]
\[ v \]

True sum: \( w+1 \) bits

\[ u + v \]

Discard carry: \( w \) bits

\[ TAdd_w(u, v) \]

- \( TAdd \) and \( UAdd \) have identical bit-level behavior
  - Signed vs. unsigned addition in C:

```c
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
```

- Will give: \( s == t \)
TAdd overflow

- **Functionality**
  - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer

![Diagram showing true sum and TAdd result with overflow handling.](image-url)
Visualizing 2’s compl. 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 $< -2^{w-1}$
    - Becomes positive
    - At most once
Characterizing TAdd

- **Functionality**
  - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer

**TAdd**($u, v$)

\[
TAdd(u, v) = \begin{cases} 
  u + v + 2^w & \text{if } u + v < TMin_w \\
  u + v & \text{if } TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^w & \text{if } TMax_w < u + v 
\end{cases}
\]

(Neg. overflow)
(Pos. overflow)
Mathematical properties of TAdd

• Group isomorphic to unsigneds with UAdd
  • Since both have identical bit patterns
    \[ TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v))) \]

• 2’s complement under TAdd forms a group
  • Closed, commutative, associative,
  • 0 is additive identity
  • Every element has additive inverse

\[ TComp_w(u) = \begin{cases} 
- u & u \neq TMin_w \\
TMin_w & u = TMin_w 
\end{cases} \]
Multiplication

• Computing exact product of w-bit numbers x, y
  – Either signed or unsigned

• Ranges
  – Unsigned (up to $2^w$ bits):
    \[ 0 \leq x \times y \leq (2^w - 1)2 = 2^{2w} - 2^{w+1} + 1 \]
  – Two’s complement min (up to $2^{w-1}$ bits):
    \[ x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \]
  – Two’s complement max (up to $2^w$ bits, but only for $(TMin_w)^2$):
    \[ x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \]

• 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

True product: $2 \cdot w$ bits

Discard $w$ bits: $w$ bits

- Standard multiplication function
  - Ignores high order $w$ bits
- Implements modular arithmetic

$$UMult_w(u, v) = u \cdot v \mod 2^w$$
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);
```

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

\texttt{malloc(ele\_cnt * ele\_size)}

• What if:
  \begin{itemize}
    \item \texttt{ele\_cnt} = 2^{20} + 1
    \item \texttt{ele\_size} = 4096 = 2^{12}
    \item Allocation = ??
  \end{itemize}

• How can I make this function secure?
Signed multiplication in C

Operands: $w$ bits

True product: $2w$ bits

Discard $w$ bits: $w$ bits

- 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

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True product: } w+k \text{ bits} \\
\text{Discard } k \text{ bits: } w \text{ bits}
\end{array}
\]

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

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

Explanation

```
t <- x+x*2
return t <<< 2;
```

- C compiler automatically generates shift/add code when multiplying by constant
Unsigned power-of-2 divide w/ shift

- Quotient of unsigned by power of 2
  - \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
</tr>
</tbody>
</table>
Compiled unsigned division code

C Function

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

Compiled Arithmetic Operations

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

Explanation

```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users: logical shift written as >>>

46
Signed power-of-2 divide w/ shift

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

$$
\begin{array}{c}
x
\end{array}
\begin{array}{c}
/2^k
\end{array}
\begin{array}{c}
x/2^k
\end{array}
\begin{array}{c}
{\text{Binary Point}}
\end{array}
$$

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y &gt;&gt; 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 8$</td>
<td>-59.4257813</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct power-of-2 divide

- Quotient of negative number by power of 2
  - We want \( \lfloor x/2^k \rfloor \) (round toward 0)
  - We compute it as \( \lfloor (x + 2^k - 1)/2^k \rfloor \)
    - In C: \((x + (1<<k)-1) >> k\)
    - Biases the dividend toward 0

- Case 1: No rounding

\[
\begin{array}{c}
\text{Dividend:} \\
+2^k - 1
\end{array}
\begin{array}{c}
\text{Divisor:} \\
/2^k
\end{array}
\begin{array}{c}
\left[ u/2^k \right]
\end{array}
\begin{array}{c}
\text{Binary Point}
\end{array}
\]

\[
\begin{array}{c}
\text{u} \quad k
\end{array}
\begin{array}{c}
1 \cdots 0 \cdots 0 0
0 \cdots 0 0 1 \cdots 1 1
0 \cdots 0 1 0 \cdots 0 0
1 \cdots 1 1 1 \cdots 1 1
\end{array}
\]

Biasing has no effect
Correct power-of-2 divide (Cont.)

• Case 2: Rounding:

Dividend: \[ u \]
\[ + 2^k - 1 \]

Divisor: \[ \frac{u}{2^k} \]
\[ \left\lfloor \frac{u}{2^k} \right\rfloor \]

Biasing adds 1 to final result

\[ \frac{u}{2^k} \]

\[ \left\lfloor \frac{u}{2^k} \right\rfloor \]
Compiled signed division code

C function

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

Compiled arithmetic operations

```asm
testl %eax, %eax
js   L4
L3:
sarl $3, %eax
ret
L4:
addl $7, %eax
jmp  L3
```

Explanation

```c
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- For Java users
  - Arith. shift written as >>
Arithmetic: basic rules

• Addition:
  – Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  – Unsigned: addition mod $2^w$
    • Mathematical addition + possible subtraction of $2^w$
  – Signed: modified addition mod $2^w$ (result in proper range)
    • Mathematical addition + possible addition or subtraction of $2^w$

• Multiplication:
  – Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  – Unsigned: multiplication mod $2^w$
  – Signed: modified multiplication mod $2^w$ (result in proper range)
Arithmetic: basic rules

• Unsigned ints, 2’s complement ints are isomorphic rings:
  \[ \Rightarrow \text{isomorphism} = \text{casting} \]

• Left shift
  – Unsigned/signed: multiplication by \(2^k\)
  – Always logical shift

• Right shift
  – Unsigned: logical shift, \(\text{div} \ (\text{division} + \text{round to zero}) \text{ by } 2^k\)
  – Signed: arithmetic shift
  • Positive numbers: \(\text{div} \ (\text{division} + \text{round to zero}) \text{ by } 2^k\)
  • Negative numbers: \(\text{div} \ (\text{division} + \text{round away from zero}) \text{ by } 2^k\)
    Use biasing to fix
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
Unsigned arithmetic

Unsigned multiplication with addition forms a commutative ring:

- Addition is a commutative group
- Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
- Multiplication is 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
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
Two’s complement 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 (mathematical) integer arithmetic
  - Both are rings
  - True 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 complement arithmetic
    \[
    T_{\text{Max}} + 1 = T_{\text{Min}} \\
    15213 \times 30426 = -10030
    \] (e.g. 16-bit words)
Why should I use unsigned?

- **Don’t** use just because the number is non-negative
  - Easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  - Can be very subtle
    - `#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
Next time: Floating Point

• Background: Fractional binary numbers
• IEEE floating point standard: Definition
• Example and properties
• Rounding, addition, multiplication
• Floating point in C