Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Stack Canaries

• Idea
  – place special value ("canary") on stack just beyond buffer
  – check for corruption before exiting function
• gcc implementation
  – -fstack-protector
  – -fstack-protector-all

```
unix>./echo-protected
Type a string:1234
1234

unix>./echo-protected
Type a string:12345
*** stack smashing detected ***
```

Supplied by CMU.

The -fstack-protector flag causes gcc to emit stack-canary code for functions that use buffers larger than 8 bytes. The -fstack-protector-all flag causes gcc to emit stack-canary code for all functions.
The operand “%fs:0x28” requires some explanation, as it uses features we haven’t previously discussed. fs is one of a few “segment registers,” which refer to other areas of memory. They are generally not used, being a relic of the early days of the x86 architecture before virtual-memory support was added. You can think of fs as pointing to an area where global variables (accessible from anywhere) may be stored and made read-only. It’s used here to hold the “canary” values. The area is set up by the operating system when the system is booted; the canary is set to a random value so that attackers cannot predict what it is. It’s also in memory that’s read-only so that the attacker cannot modify it.

Note that objdump’s assembler syntax is slightly different from what we normally use in gcc: there are no “q” or “l” suffices on most of the instructions, but the call instruction, strangely, has a q suffix.
Setting Up Canary

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Before call to gets

Stack frame for main

Return address

Canary

buf

[3][2][1][0]

%rsp

echo:

```
...  
movq  %fs:40, %rax  # Get canary
movq  %rax, 8(%rsp) # Put on stack
xorl  %eax, %eax   # Erase canary
...  
```

Supplied by CMU.
Checking Canary

After call to gets

`void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}`

```assembly
echo:
    ...
    movq 8(%rsp), %rax  # Retrieve from stack
    xorq %fs:40, %rax   # Compare with Canary
    je .L2              # Same: skip ahead
    call __stack_chk_fail # ERROR
    .L2:
    ...
```

Supplied by CMU.
Computer Architecture and Optimization (1)

What You Need to Know to Write Better Code
Simplistic View of Processor

```c
while (true) {
    instruction = mem[rip];
    execute(instruction);
}
```
Some Details ...

```c
void execute(instruction_t instruction) {
    decode(instruction, &opcode, &operands);
    fetch(operands, &in_operands);
    perform(opcode, in_operands, &out_operands);
    store(out_operands);
}
```
Pipelines

Decode  Fetch  Perform  Store  Decode  Fetch  Perform  Store

Decode  Fetch  Perform  Store

Decode  Fetch  Perform  Store

Decode  Fetch  Perform  Store

Decode  Fetch  Perform  Store
Analysis

- **Not pipelined**
  - each instruction takes, say, 3.2 nanoseconds
    - 3.2 ns latency
  - 312.5 million instructions/second (MIPS)
- **Pipelined**
  - each instruction still takes 3.2 ns
    - latency still 3.2 ns
  - an instruction completes every .8 ns
    - 1.25 billion instructions/second (GIPS) throughput
Hazards ...
Data Hazards

\[
\begin{align*}
\text{addq } &12(\%\text{rbx}), \%\text{rax} \\
\text{addq } &20, \%\text{rax} \\
\text{movq } &40(\%\text{rax}), \%\text{rsp}
\end{align*}
\]
Coping

Decode 12(%rax), %rax
Decode addq %rax

$20, %rax addq %rax

40(%rax) movq
Control Hazards

    movl $0, %ecx
    .L2:
    movl %edx, %eax
    andl $1, %eax
    addl %eax, %ecx
    shrl $1, %edx
    jne .L2 # what goes in the pipeline?
    movl %ecx, %eax
    ...

...
Coping: Guess ...

- Branch prediction
  - assume, for example, that conditional branches are always taken
  - but don’t do anything to registers or memory until you know for sure
Modern CPU Design

Adapted from slide supplied by CMU.
Performance Realities

There’s more to performance than asymptotic complexity

• Constant factors matter too!
  – easily see 10:1 performance range depending on how code is written
  – must optimize at multiple levels:
    » algorithm, data representations, procedures, and loops
• Must understand system to optimize performance
  – how programs are compiled and executed
  – how to measure program performance and identify bottlenecks
  – how to improve performance without destroying code modularity and generality
Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don’t (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
  » but constant factors also matter
- Have difficulty overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

Supplied by CMU.
Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - must not cause any change in program behavior
  - often prevents it from making optimizations that would only affect behavior under pathological conditions
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - compiler has difficulty anticipating run-time inputs

- When in doubt, the compiler must be conservative
Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - reduce frequency with which computation performed
    - if it will always produce same result
    - especially moving code out of loop

```c
void set_row(long *a, long *b, 
  long I, long n){
  long j;
  for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
}
```

```c
long j;
long ni = n*i;
for (j = 0; j < ni; j++)
  a[ni+j] = b[j];
```
Supplied by CMU, updated for current gcc. This happens when gcc is given the –O1 flag, specifying that it perform a small amount of code optimization.
Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  \[16 \times x \quad \rightarrow \quad x \ll 4\]
  - utility is machine-dependent
  - depends on cost of multiply or divide instruction
    » on Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int ni = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
```

Supplied by CMU.
Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;

long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i'n, (i-1)'n, (i+1)'n

```asm
leaq %rsi, %rax # i+1
leaq -1(%rsi), %rbp # i-1
imulq %rcx, %rsi # i*n
imulq %rcx, %rax # (i+1)*n
imulq %rcx, %rbp # (i-1)*n
addq %rdx, %rsi # i*n+j
addq %rdx, %rax # (i+1)*n+j
addq %rdx, %rbp # (i-1)*n+j
```

1 multiplication: i'n

```asm
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq (%rsi,%rcx), %rax # i*n+j+n
```

Supplied by CMU.

The code in the lower-left box is what gcc produced for the code in the upper left box. On the right is a much better version that was done by hand.
Quiz 1

The fastest means for evaluating
\( n^2 + 2n + 1 \)
requires exactly:
  a) 2 multiplies and 2 additions
  b) one multiply and two additions
  c) one multiply and one addition
  d) three additions
Optimization Blocker #1: Procedure Calls

- Procedure to convert string to lower case

```c
void lower(char *s){
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] = ('A' - 'a');
}
```
Lower Case Conversion Performance

- Time quadruples when string length doubles
- Quadratic performance

Supplied by CMU.
Convert Loop To Goto Form

```c
void lower(char *s){
    int i = 0;
    if (i >= strlen(s))
        goto done;
    loop:
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
        i++;
        if (i < strlen(s))
            goto loop;
    done:
}
```

- `strlen` executed every iteration
Calling Strlen

```c
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **strlen performance**
  - only way to determine length of string is to scan its entire length, looking for null character
- **Overall performance, string of length N**
  - N calls to strlen
  - overall $O(N^2)$ performance

Supplied by CMU.
Improving Performance

```c
void lower2(char *s){
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
  - since result does not change from one iteration to another
  - form of code motion
Lower-Case Conversion Performance

- Time doubles when string-length doubles
  - linear performance of lower2

Supplied by CMU.
Optimization Blocker: Procedure Calls

- Why couldn't compiler move strlen out of inner loop?
  - procedure may have side effects
    » alters global state each time called
  - function may not return same value for given arguments
    » depends on other parts of global state
    » procedure lower could interact with strlen

- Warning:
  - compiler treats procedure call as a black box
  - weak optimizations near them

- Remedies:
  - use of inline functions
    » gcc does this with -O2
  - do your own code motion

```c
int lencnt = 0;
size_t strlen(const char *s) {
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Supplied by CMU.
Based on a slide supplied by CMU.
Based on a slide supplied by CMU.
Based on a slide supplied by CMU.
Optimization Blocker: Memory Aliasing

- Aliasing
  - two different memory references specify single location
  - easy to have happen in C
    » since allowed to do address arithmetic
    » direct access to storage structures
  - get in habit of introducing local variables
    » accumulating within loops
    » your way of telling compiler not to check for aliasing

Supplied by CMU.
C99 to the Rescue

- New attribute
  - `restrict`
    - applied to a pointer, tells the compiler that the object pointed to will be accessed only via this pointer
    - compiler thus doesn’t have to worry about aliasing
    - but the programmer does ...
    - syntax
      ```c
      int *restrict pointer;
      ```
Pointers and Arrays

- `long a[][n]`
  - `a` is a 2-D array of longs, the size of each row is `n`
- `long (*b)[n]`
  - `b` is a pointer to a 1-D array of size `n`

- `a` and `b` are of the same type
Note: we must give gcc the flag “-std=gnu99” for this to be compiled.

Observe that

long (*a)[n]

declares a to be a pointer to an array of n longs.

Thus

long (*restrict a)[n]

declares a to be a restricted pointer to an array of n longs
Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can have dramatic performance improvement
  - compilers often cannot make these transformations
  - lack of associativity and distributivity in floating-point arithmetic

Supplied by CMU.
Benchmark Example: Datatype for Vectors

```c
/* data structure for vectors */
typedef struct{
    int len;
    data_t *data;
} vec_t, *vec_ptr_t;

len

0 1 len-1

/* retrieve vector element and store at val */
int get_vec_element(vec_ptr_t v, int idx, data_t *val){
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

/* return length of vector */
int vec_length(vec_ptr_t v) {
    return v->len;
}
```

Supplied by CMU.
Benchmark Computation

```c
void combine(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

- **Data Types**
  - use different declarations for `data_t`
    - `int`
    - `float`
    - `double`

- **Operations**
  - use different definitions of `OP` and `IDENT`
    - `+`, `0`
    - `*`, `1`

Supplied by CMU.
Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- T = CPE*n + Overhead
  - CPE is slope of line

![Graph showing cycles per element vs. number of elements.](image)

Supplied by CMU.
Benchmark Performance

```c
void combine1(vec_ptr_t v, data_t *dest){
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1 –O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Move vec_length

```c
void combine2(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine1 unoptimized</td>
<td>29.0</td>
<td>29.2</td>
</tr>
<tr>
<td>Combine1-O1</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine2</td>
<td>8.03</td>
<td>8.09</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Eliminate Procedure Calls

```c
void combine3(vec_ptr_t v, data_t *dest){
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine2</td>
<td>8.03</td>
<td>8.09</td>
</tr>
<tr>
<td></td>
<td>10.09</td>
<td>12.08</td>
</tr>
<tr>
<td>Combine3</td>
<td>6.01</td>
<td>8.01</td>
</tr>
<tr>
<td></td>
<td>10.01</td>
<td>12.02</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Eliminate Unneeded Memory References

```c
void combine4(vec_ptr_t v, data_t *dest){
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
    t = t OP d[i];
    *dest = t;
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine1 –01</td>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Quiz 2

Combine4 is pretty fast; we’ve done all the “obvious” optimizations. How much faster will we be able to make it? (Hint: it involves taking advantage of pipelining and multiple functional units on the chip.)

a) 1× (it’s already as fast as possible)
b) 2× – 4×
c) 16× – 64×