Constant-time programming in C
What’s our goal?

• **Goal:** Write C programs that don’t leak sensitive data

• **Assumption:** no explicit leaks
  ➤ E.g., writing secret data to public location

• **Approach:** constant-time programming
  ➤ More robust approach than random fuzzing/padding
  ➤ Why?
What’s our goal?

- **Goal**: Write C programs that don’t leak sensitive data

- **Assumption**: no explicit leaks
  - E.g., writing secret data to public location

- **Approach**: constant-time programming
  - More robust approach than random fuzzing/padding
    - Why? Completely eliminates time-variability!
What introduces time-variability?
Which runs faster?

```c
void foo(double x) {
    double z, y = 1.0;
    for (uint32_t i = 0; i < 100000000; i++) {
        z = y * x;
    }
}
```

A: foo(1.0);

B: foo(1.0e-323);

C: They take the same amount of time!
Example: floating-point operations

<table>
<thead>
<tr>
<th>Processor</th>
<th>+ subnormal</th>
<th>+ special</th>
<th>× subnormal</th>
<th>× special</th>
<th>÷ subnormal</th>
<th>÷ special</th>
<th>$\sqrt{\text{subnormal}}$</th>
<th>$\sqrt{\text{special}}$</th>
<th>$\sqrt{x^2}$</th>
<th>$\sqrt{x^4}$</th>
<th>$\sqrt{-x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7-7700 (Kaby Lake)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Core i7-6700K (Skylake)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Core i7-3667U (Ivy Bridge)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Xeon X5660 (Westmere)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Atom D2550 (Cedarview)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>AMD Phenom II X6 1100T</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>AMD Ryzen 7 1800x</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

**Single-precision operations**

<table>
<thead>
<tr>
<th>Processor</th>
<th>+ subnormal</th>
<th>+ special</th>
<th>× subnormal</th>
<th>× special</th>
<th>÷ subnormal</th>
<th>÷ special</th>
<th>$\sqrt{\text{subnormal}}$</th>
<th>$\sqrt{\text{special}}$</th>
<th>$\sqrt{x^2}$</th>
<th>$\sqrt{x^4}$</th>
<th>$\sqrt{-x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7-7700 (Kaby Lake)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Core i7-6700K (Skylake)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Core i7-3667U (Ivy Bridge)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Xeon X5660 (Westmere)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Intel Atom D2550 (Cedarview)</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>AMD Phenom II X6 1100T</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>AMD Ryzen 7 1800x</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

**Double-precision operations**
Leaks due to variable-time instructions

• **Problem:** Certain instructions take different amounts of time depending on the operands
  ➤ What’s another example?

• **Solution?**
Unsafe language-level operators

• Operators that lead to variable-time instructions
  ➤ E.g., /, %

• Operators that lead to conditional branches
  ➤ E.g., ||, &&, ?:
  ➤ Why? (We’ll see in a bit!)
What’s the problem with this code?

```c
s0;
for (uint32_t i = 0; i < secret; i++) {
    s1;
    s2;
}
s3;
s4;
```
How do we fix this?

```c
uint32_t done = 0;
for (uint32_t i = 0; i < pub_max; i++) {
    done |= (max == secret);
    if (!done) {
        s1;
        s2;
    }
}
```

Is this right? A: yes, B: no
Why are if-statements on secrets unsafe?

s0;
if (secret) {
  s1;
  s2;
}
s3;
Why are if-statements on secrets unsafe?

```plaintext
s0;
if (secret) {
    s1;
    s2;
}
s3;
```
Why are if-statements on secrets unsafe?

```plaintext
s0;
if (secret) {
    s1;
    s2;
}
s3;
```

<table>
<thead>
<tr>
<th>secret</th>
<th>run</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>s0; s1; s2; s3;</td>
<td>4</td>
</tr>
<tr>
<td>false</td>
<td>s0; s3;</td>
<td>2</td>
</tr>
</tbody>
</table>
Can we pad else branch?

```c
if (secret) {
    s1;
    s2;
} else {
    s1';
    s2';
}
```

Is this safe? A: yes, B: no

where s1 and s1’ take same amount of time
Issue with conditional branching

• **Problem:** Instructions are loaded from cache
  ➤ Which instructions were loaded (or not) observable

• **Problem:** Hardware tried to predict where branch goes
  ➤ Success (or failure) of prediction is observable

• **Solution?**
Solution: don’t branch on secrets!
Solution: fold control flow into data flow

(assumption secret = 1 or 0)

```java
if (secret) {
    x = a;
}

x = secret * a + (1-secret) * x;
```
Solution: fold control flow into data flow

if (secret) {
    x = a;
}

➡

x = secret * a
+ (1-secret) * x;

(assumption secret = 1 or 0)
Solution: fold control flow into data flow

(assumption secret = 1 or 0)

```c
if (secret) {
    x = a;
} else {
    x = b;
}
```

```c
x = secret * a + (1-secret) * x;
```

```c
x = (1-secret) * b + secret * x;
```

Solution: fold control flow into data flow

- Multiple ways to fold control flow in

- Previous example: takes advantage of arithmetic

- What’s another way?

```c
if (secret) {
    x = a;
}
```

```c
x = (-secret & (a^x)) ^ x
```
Solution: fold control flow into data flow

- Useful to create library of primitives
  - E.g., bit ? a : b ➔ select(a, b, bit);

```c
unsigned select (unsigned a, unsigned b, unsigned bit)
{
    /* -0 = 0, -1 = 0xff....ff */
    unsigned mask = - bit;
    unsigned ret = mask & (a^b);
    ret = ret ^ a;
    return ret;
}
```

Code from https://cryptocoding.net
A more complex example

```c
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe? A: yes, B: no
A more complex example

```c
static int get_zeros_padding( unsigned char *input, size_t input_len, size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if (done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe? A: yes, B: no
A more complex example

```c
static int get_zeros_padding( unsigned char *input, size_t input_len,
                               size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        *data_len = select(i, *data_len, done & !prev_done);
    }

    return 0;
}
```

Is this safe? A: yes, B: no
Leaks via control flow

- **Problem:** Control flow that depends on secret data can lead to information leakage
  - Loops
  - If-statements (switch, etc.)
  - Early returns, goto, break, continue
  - Function calls

- **Solution:** Control flow should not depend on secrets, fold secret control flow into data!
void cond_assign( uint8_t *X, const uint8_t *Y, size_t len, unsigned char assign ) {
    /* make sure assign is 0 or 1 */
    assign = ( assign != 0 );

    for (size_t i = 0; i < len; i++) {
        X[i] = X[i] * ( 1 - assign ) + Y[i] * assign;
    }
}

A: yes, B: no
How do we fix this?

Make it hard for compiler to optimize some code, but really... look at the generated assembly!
Non-example: `strcmp(A, B)` from last lecture

- Why is this not a problem due to memory access?

- What would be an example of a leak via memory access?
static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
...
   // All other round keys are found from the previous round keys.
   for (i = Nk; i < Nb * (Nr + 1); ++i) {
      
      k = (i - 1) * 4;
      tempa[0] = RoundKey[k + 0];
      tempa[1] = RoundKey[k + 1];
      tempa[2] = RoundKey[k + 2];
      tempa[3] = RoundKey[k + 3];

      tempa[0] = sbox[tempa[0]];
      tempa[1] = sbox[tempa[1]];
   }
   
...
Why is this a problem?

• **Problem**: Accessing memory based on secret
  
  ➤ arr[secret]

• Why is this a problem?
  
  ➤ duration(arr[secret]) depends on whether or not arr[secret] is in the cache!

  ➤ What happens if attacker can influence cache?
How do we fix this?

- Only access memory at public index
- How do we express `arr[secret]`?

```c
x = arr[secret]  ➡  for(size_t i = 0; i < arr_len; i++)
                        x = select(arr[i], x, secret == i)
```
Summary

• Duration of certain operations depends on data
  ➤ Do not use operators that are variable time

• Control flow
  ➤ Do not branch based on a secret

• Memory access
  ➤ Do not access memory based on a secret
Challenges with writing constant-time code

• Duration of certain operations depends on data
  ➤ Transform to safe, known CT operations

• Control flow
  ➤ Turn control flow into data flow problem: select!

• Memory access
  ➤ Loop over public bounds of array!