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?

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

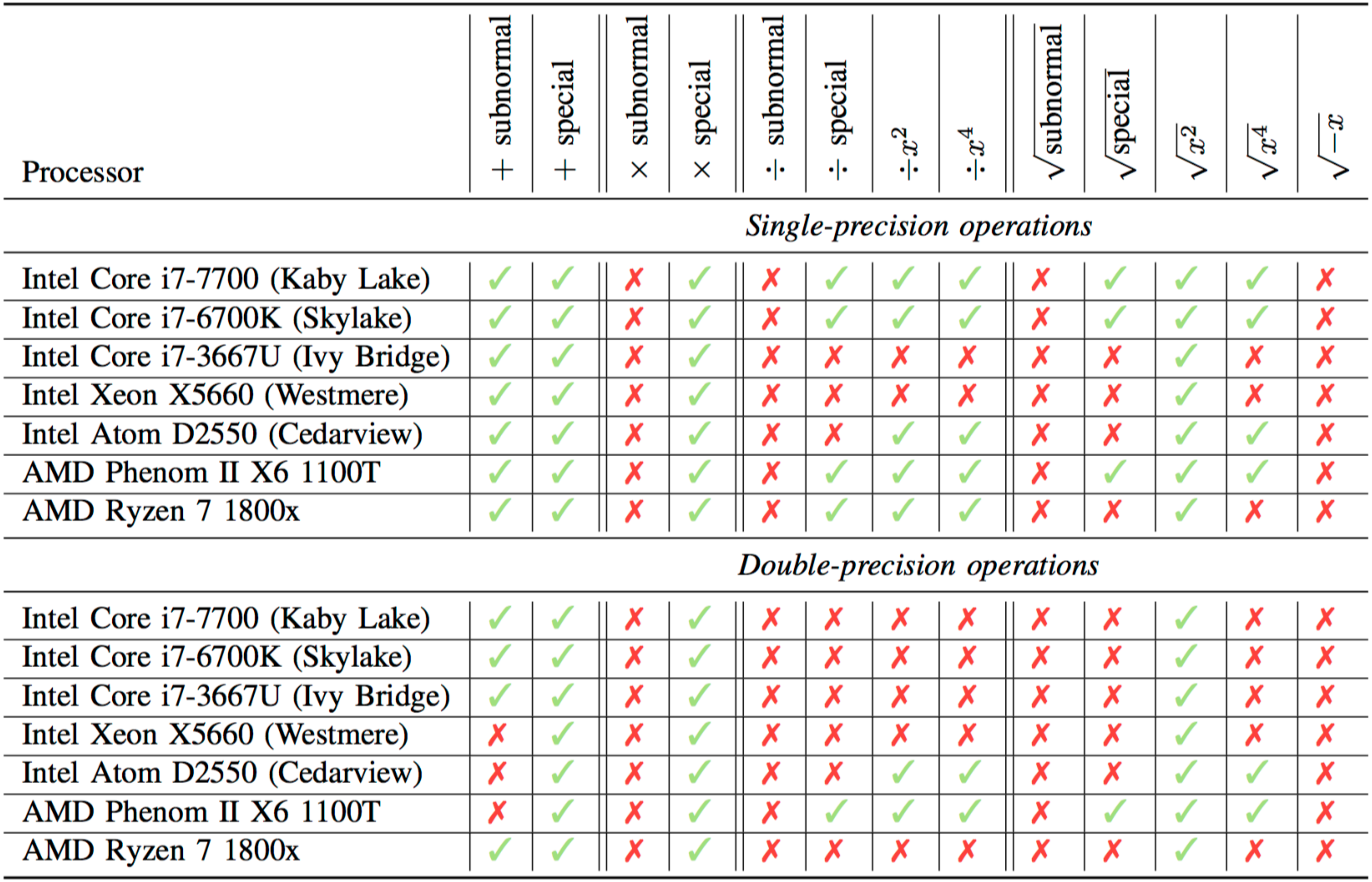
A: `foo(1.0);`

B: `foo(1.0e-323);`

C: They take the same amount of time!

Code from D. Kohlbrenner

Example: floating-point operations



Processor	+ subnormal	+ special	× subnormal	× special	÷ subnormal	÷ special	$\div x^2$	$\div x^4$	$\sqrt{\text{subnormal}}$	$\sqrt{\text{special}}$	$\sqrt{x^2}$	$\sqrt{x^4}$	$\sqrt{-x}$
Single-precision operations													
Intel Core i7-7700 (Kaby Lake)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-6700K (Skylake)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-3667U (Ivy Bridge)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Xeon X5660 (Westmere)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Atom D2550 (Cedarview)	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗
Double-precision operations													
Intel Core i7-7700 (Kaby Lake)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-6700K (Skylake)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-3667U (Ivy Bridge)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Xeon X5660 (Westmere)	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Atom D2550 (Cedarview)	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✗	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗

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?

```
s0;
for (uint32_t i = 0; i < secret; i++) {
 s1;
 s2;
}
s3;
s4;
```

How do we fix this?

```
s0;
uint32_t done = 0;
for (uint32_t i = 0; i < pub_max; i++) {
  done |= (max == secret);
  if (!done) {
    s1;
    s2;
 }
}
s3;
s4;
```

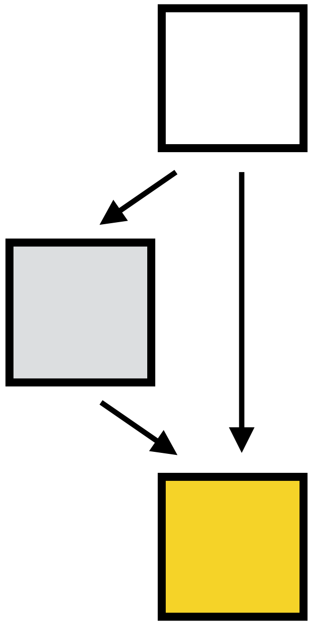
Is this right? A: yes, **B: no**

Why are if-statements on secrets unsafe?

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

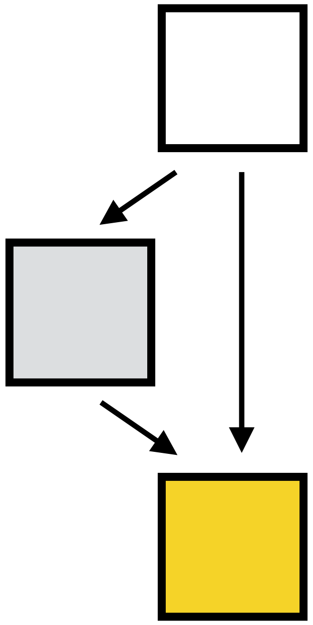
Why are if-statements on secrets unsafe?

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



Why are if-statements on secrets unsafe?

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



secret	run	⏱
true	s0;s1;s2;s3;	**4**
false	s0;s3;	**2**

Can we pad else branch?

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

where `s1` and `s1'` take same amount of time

Is this safe? A: yes, **B: no**

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)

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

➡

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

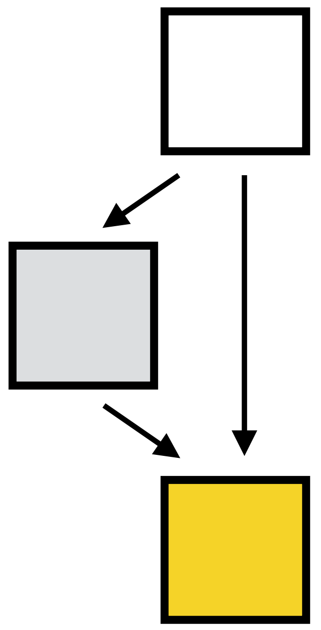
Solution: fold control flow into data flow

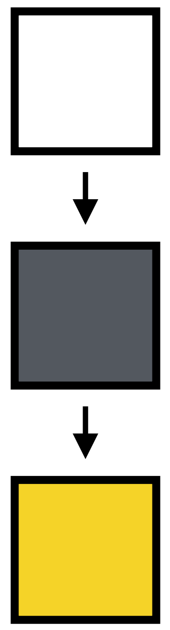
(assumption secret = 1 or 0)

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

➡

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





Solution: fold control flow into data flow

(assumption secret = 1 or 0)

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

➡

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

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?

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

➡

```
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);

```
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

```
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

```
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!

Is this code safe?

```c
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;
  }
}
```

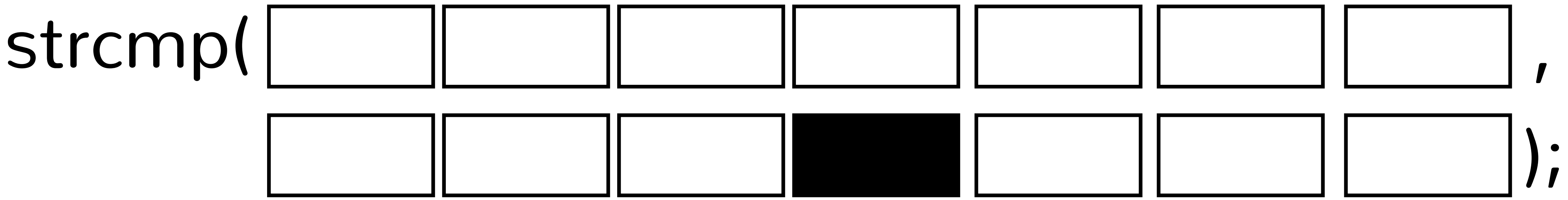
A: yes, **B: no**

How do we fix this?

Make it hard for compiler to optimize some code, but really… look at the generated assembly!

Accessing memory can leak too

- **Non-example:** strcmp(A, B) from last lecture



strcmp(

,

);

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

- What would be an example of a leak via memory access?

What’s the problem with this code?

```
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]];
     tempa[2] = sbox[tempa[2]];
     tempa[3] = sbox[tempa[3]];
...
```

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]?

```
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!