CSE 127: Computer Security

Side-channels

Deian Stefan

Slides adopted from Stefan Savage, Nadia Heninger, Sunjay Cauligi

Context

- Isolation is key to building secure systems
  - Used to implement privilege separation, least privilege and complete mediation
  - Basic idea: **protect the secret or sensitive stuff so it can’t be accessed across a trust boundary**
- **Assumption:** we know what the trust boundaries are and that access to something is easy to identify

How can we get at protected data?

How can we get at protected data?

- Find a bug in the kernel, VMM, runtime system!
  - ➤ Huge and have a huge attack surface: syscalls
  - ➤ Hard to get right (e.g., confused deputy attacks)

How can we get at protected data?

- Find a bug in the kernel, VMM, runtime system!
  - Huge and have a huge attack surface: syscalls
  - Hard to get right (e.g., confused deputy attacks)
- Find a hardware bug that let’s you bypass isolation

Side channels

- We often think of systems as black boxes:
  - As abstractions that consume input and produce output
  - We assume that all side effects are about output (e.g., values in memory or I/O)
- Sometimes information is revealed in **how** it is produced

Side channels

- We often think of systems as black boxes:
  - As abstractions that consume input and produce output
  - We assume that all side effects are about output (e.g., values in memory or I/O)
- Sometimes information is revealed in **how** it is produced
  - How long, how fast, how loud, how hot… artifacts of the **implementation** not the abstraction

Side channels

- We often think of systems as black boxes:
  - As abstractions that consume input and produce output
  - We assume that all side effects are about output (e.g., values in memory or I/O)
- Sometimes information is revealed in **how** it is produced
  - How long, how fast, how loud, how hot… artifacts of the **implementation** not the abstraction
  - This can produce a **side channel**: a source of information beyond the output specified by the abstraction

Today

- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

Consumption side channels

- How long does this password-check take?

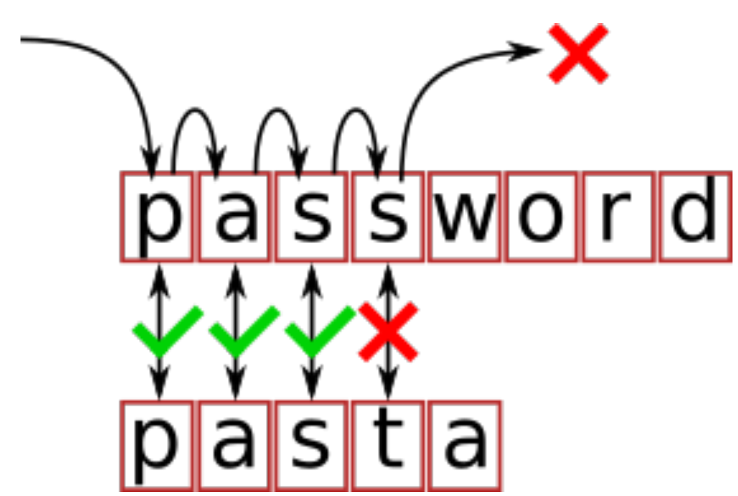
```
char pwd[] = “z2n34uzbnqhw4i”;

//...

int check_password(char *buf) {
    return strcmp(buf, pwd);
}
```

Consumption side channels

- **Consumption**: how much of a resource is being used to perform the operation?
  - Eg.g., time, power, memory, network, etc.
- **Emission**: what out-of-band signal is generated in the course of performing the operation?
  - E.g., electro-magnetic radiation, sound, movement, error messages, etc.

Side channel examples

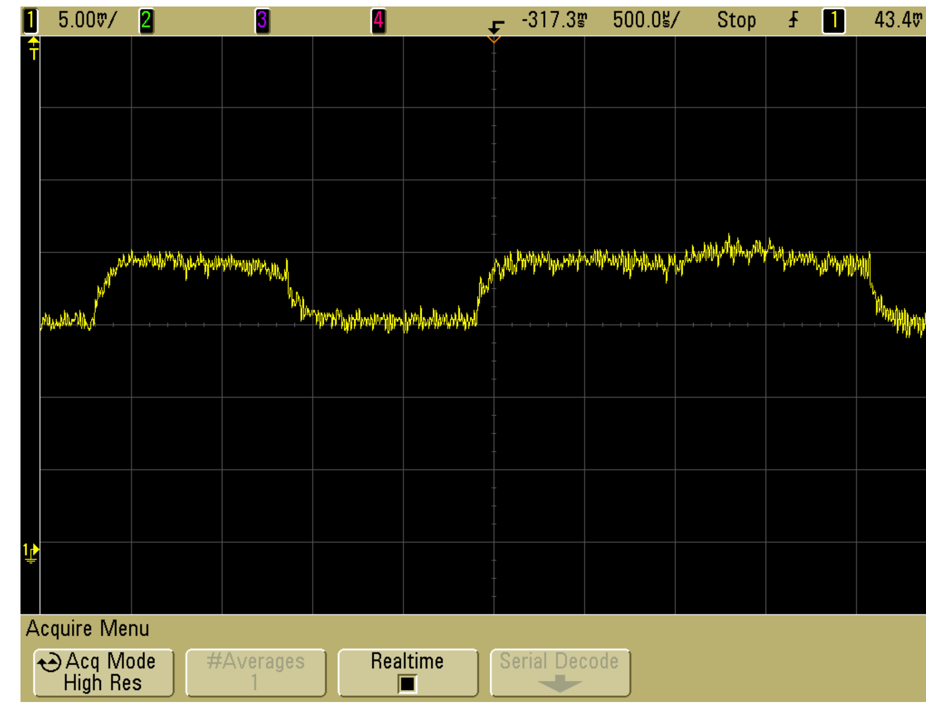
- Tenex password verification
  - Alan Bell, 1974
  - Character-at-a-time comparison + virtual memory
  - Recover the full password in linear time



https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/

Side channel examples

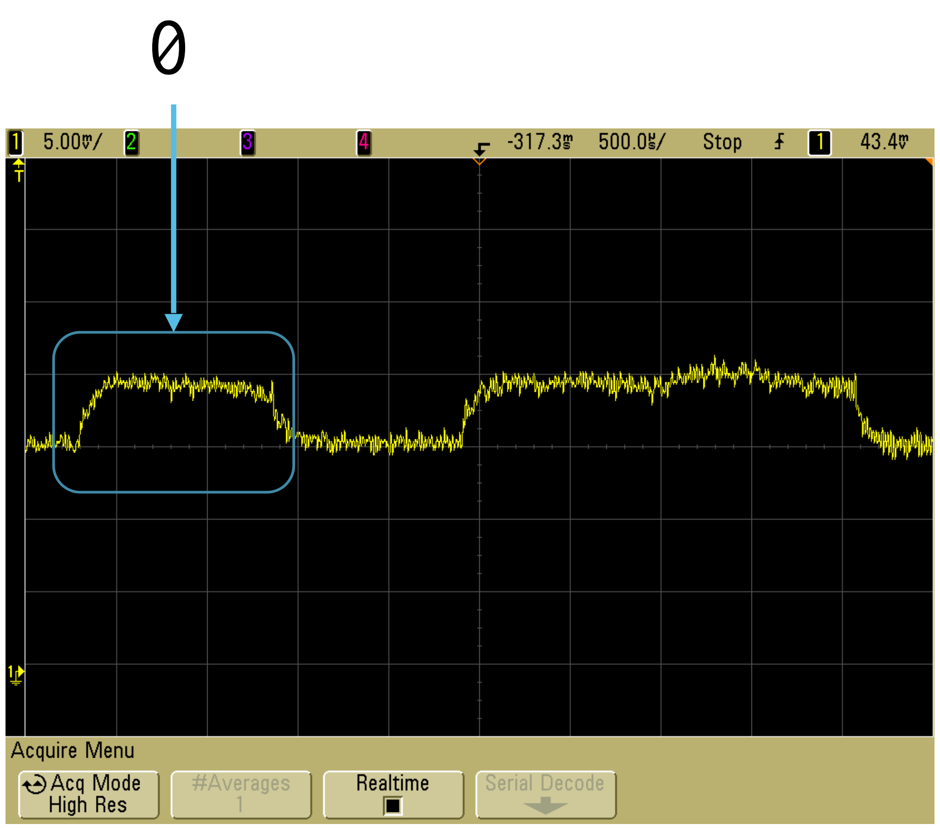
- Secret cryptographic key value maintained in hardware
  - Can never be read, only used
- Simple Power Analysis (SPA)
- Differential Power Analysis (DPA)
  - Paul Kocher, 1999
  - Using signal processing techniques on a very large number of samples, iteratively test hypothesis about secret key bit values.



https://en.wikipedia.org/wiki/Power_analysis#/media/File:Power_attack_full.png

Side channel examples

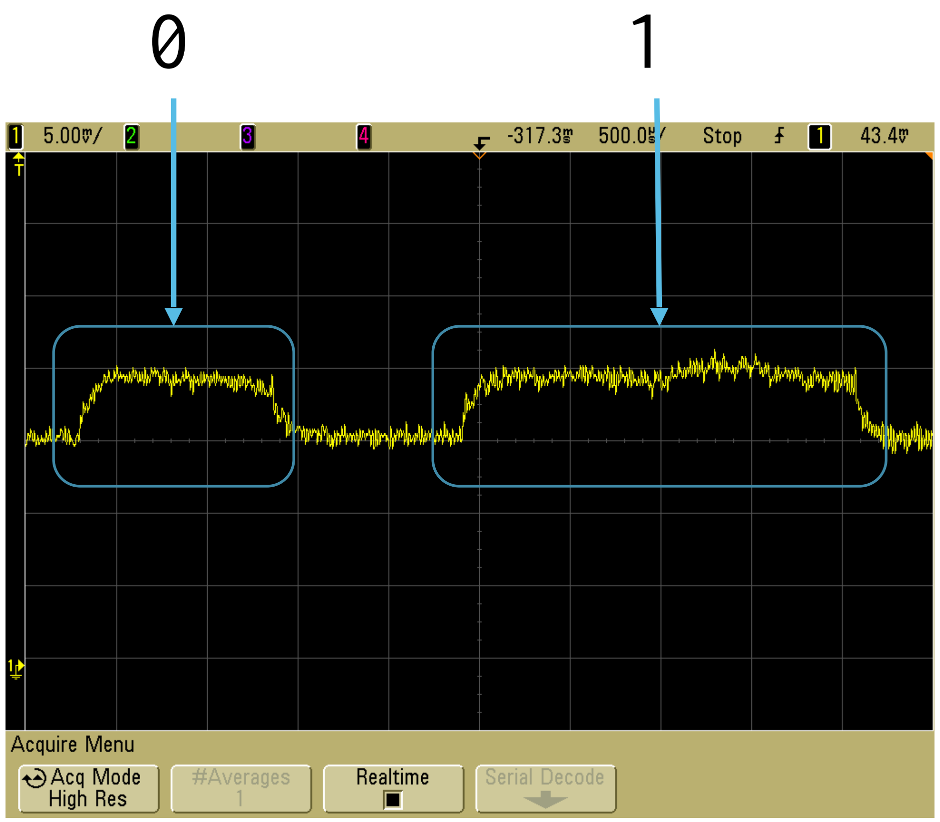
- Secret cryptographic key value maintained in hardware
  - Can never be read, only used
- Simple Power Analysis (SPA)
- Differential Power Analysis (DPA)
  - Paul Kocher, 1999
  - Using signal processing techniques on a very large number of samples, iteratively test hypothesis about secret key bit values.



https://en.wikipedia.org/wiki/Power_analysis#/media/File:Power_attack_full.png

Side channel examples

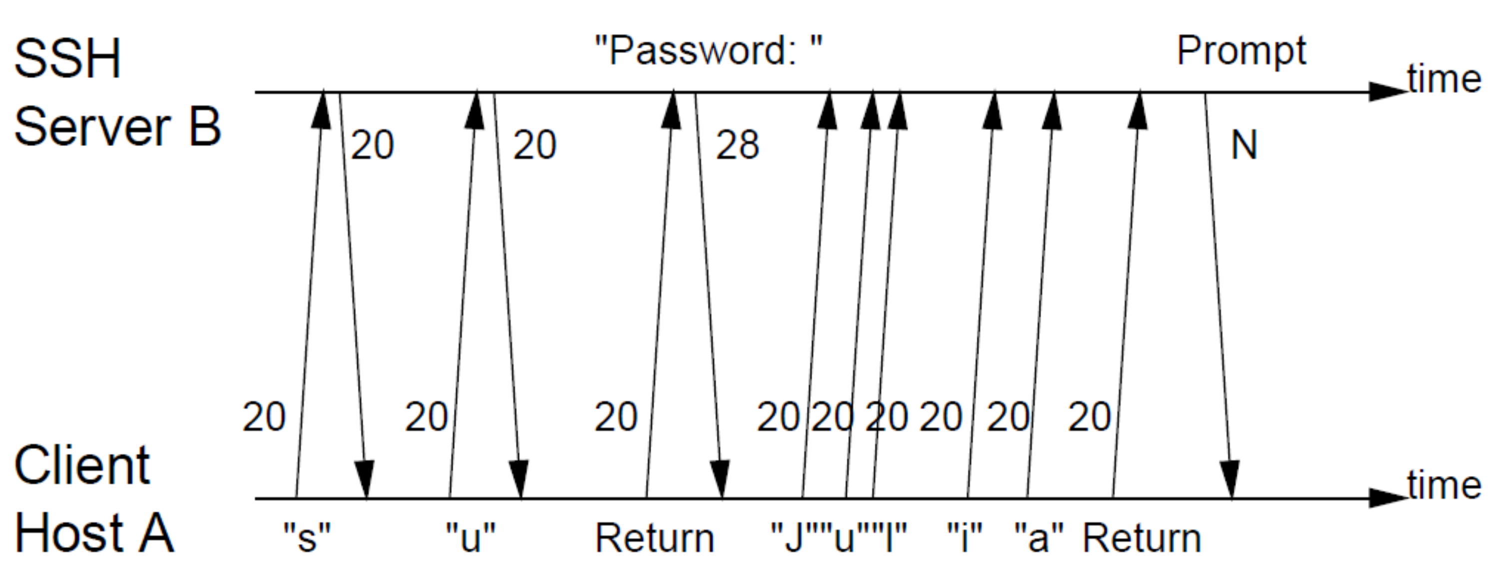
- Secret cryptographic key value maintained in hardware
  - ➤ Can never be read, only used
- Simple Power Analysis (SPA)
- Differential Power Analysis (DPA)
  - ➤ Paul Kocher, 1999
  - ➤ Using signal processing techniques on a very large number of samples, iteratively test hypothesis about secret key bit values.



https://en.wikipedia.org/wiki/Power_analysis#/media/File:Power_attack_full.png

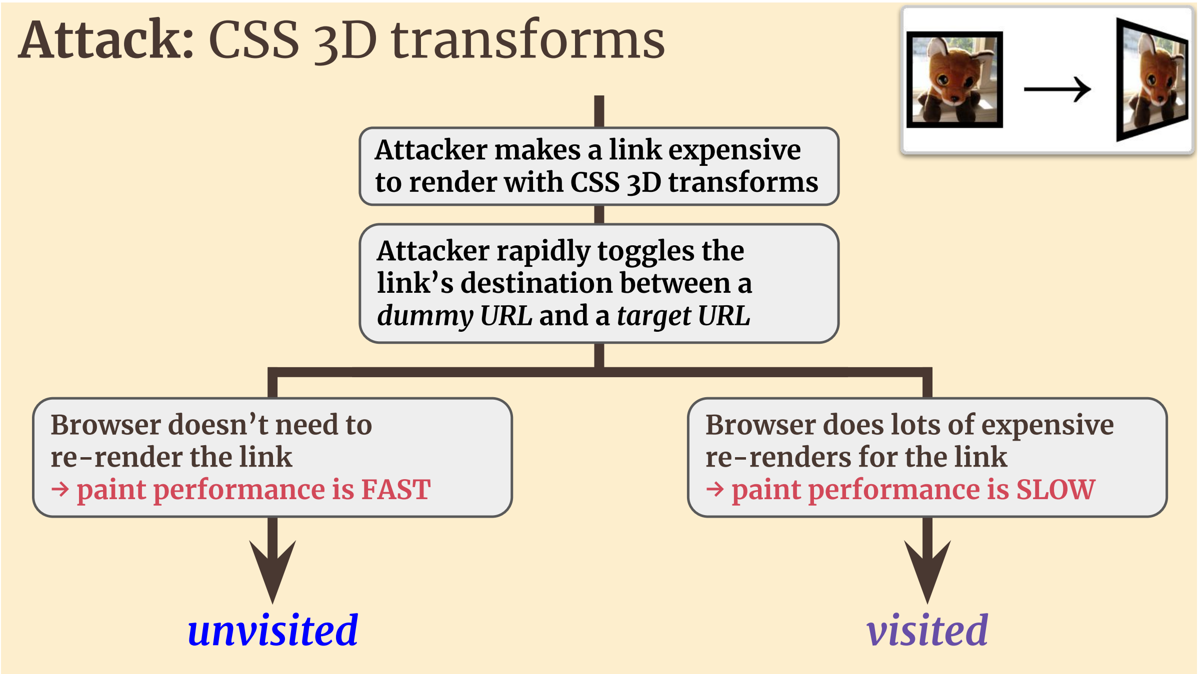
Side channel examples

- Timing Analysis of Keystrokes and Timing Attacks on SSH
  - D. Song, D. Wagner, X. Tian, 2001
  - Recover characters typed over SSH by observing packet timing



Side channel examples

- An empirical study of privacy-violating information flows in JavaScript web applications
  - D. Jang, R. Jhala, S. Lerner, H. Shacham, 2010
- Browser history re:visited
  - M. Smith, C. Disselkoen, S. Narayan, F. Brown, D. Stefan, 2018



Side channel examples

- Keyboard Acoustic Emanations
  - ➤ D. Asonov, R. Agrawal, 2004
  - ➤ Recover keys typed by their sound
- Keyboard Acoustic Emanations Revisited
  - ➤ Li Zhuang, Feng Zhou, J. D. Tygar, 2009

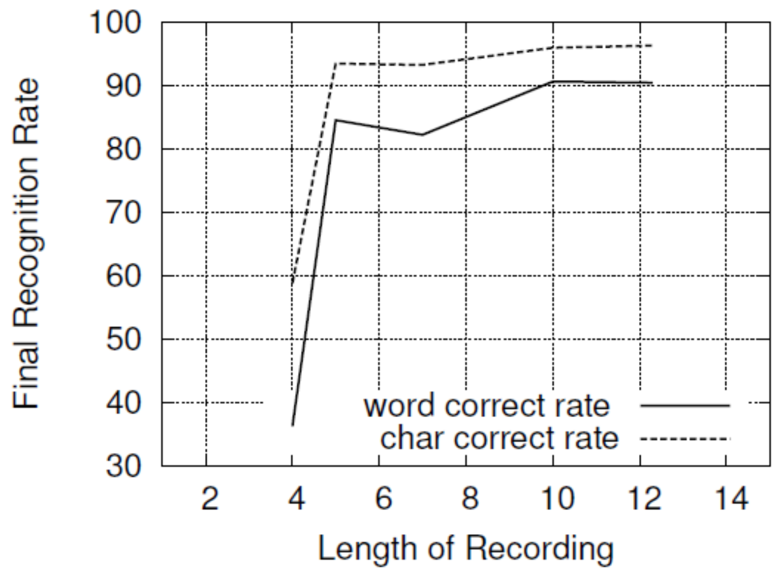
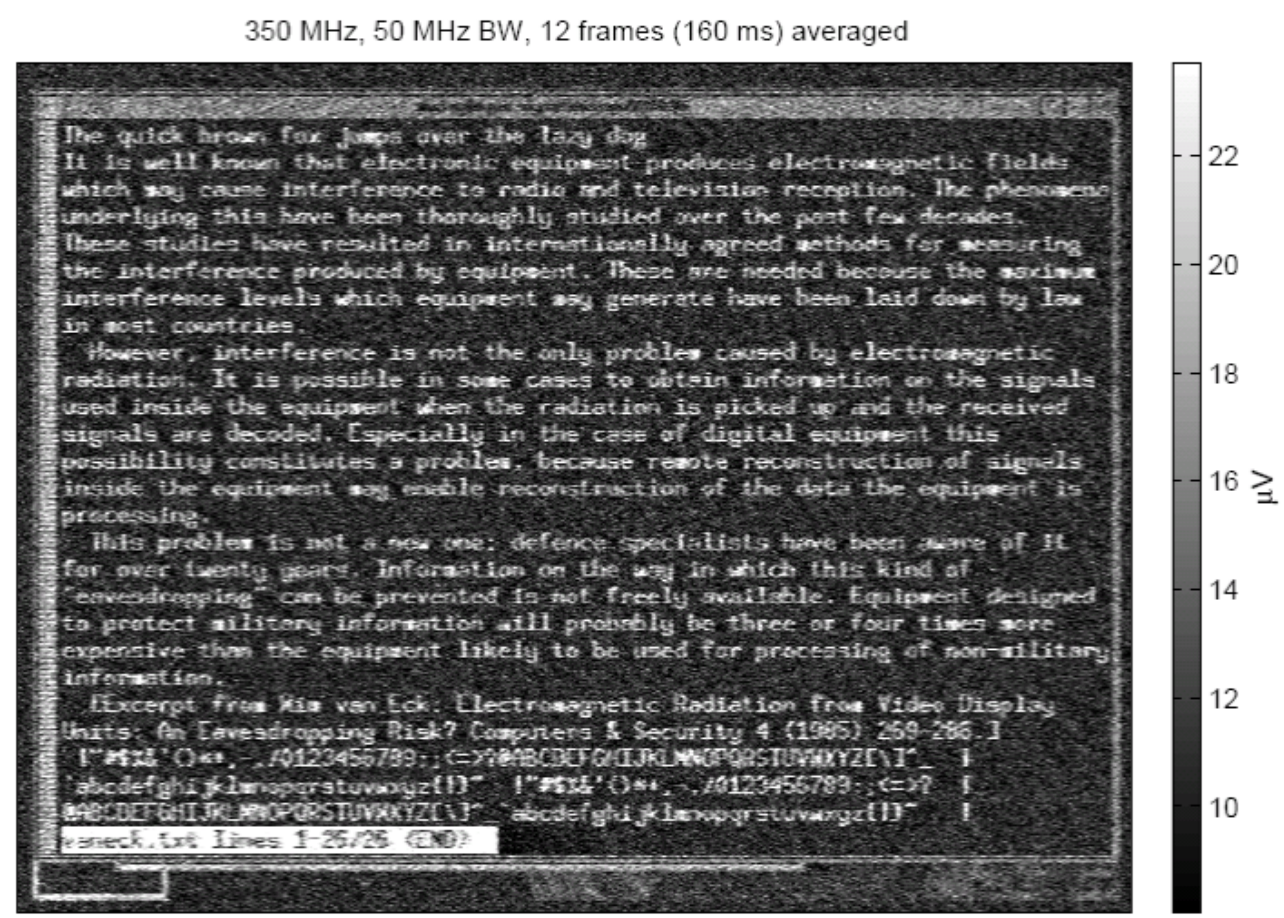


Fig. 7. Length of recording vs. recognition rate.

https://www.microsoft.com/en-us/research/publication/side-channel-leaks-in-web-applications-a-reality-today-a-challenge-tomorrow/

Remote reading of LCD screens via RF (Kuhn, 2004)

- Image displays simultaneously along line
- Pick up radiation from screen connection cable



Target and antenna in a modern office building 10 m apart, with two other offices and three plasterboard walls (−2.7 dB each) in between. Single-shot recording of 8 megasamples with storage oscilloscope at 50 Msamples/s, then offline correlation and averaging of 12 frames.

Optical domain emanations (Kuhn, 2002)

- Light emitted by CRT is
  - Video signal combined with phosphor response
- Can use fast photosensor to separate signal from HF components of light
- Even if reflected off diffuse surface (i.e., a white wall) from across the street

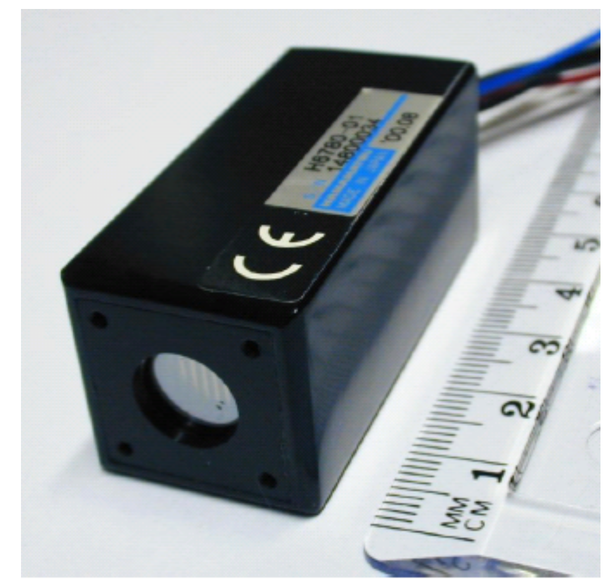
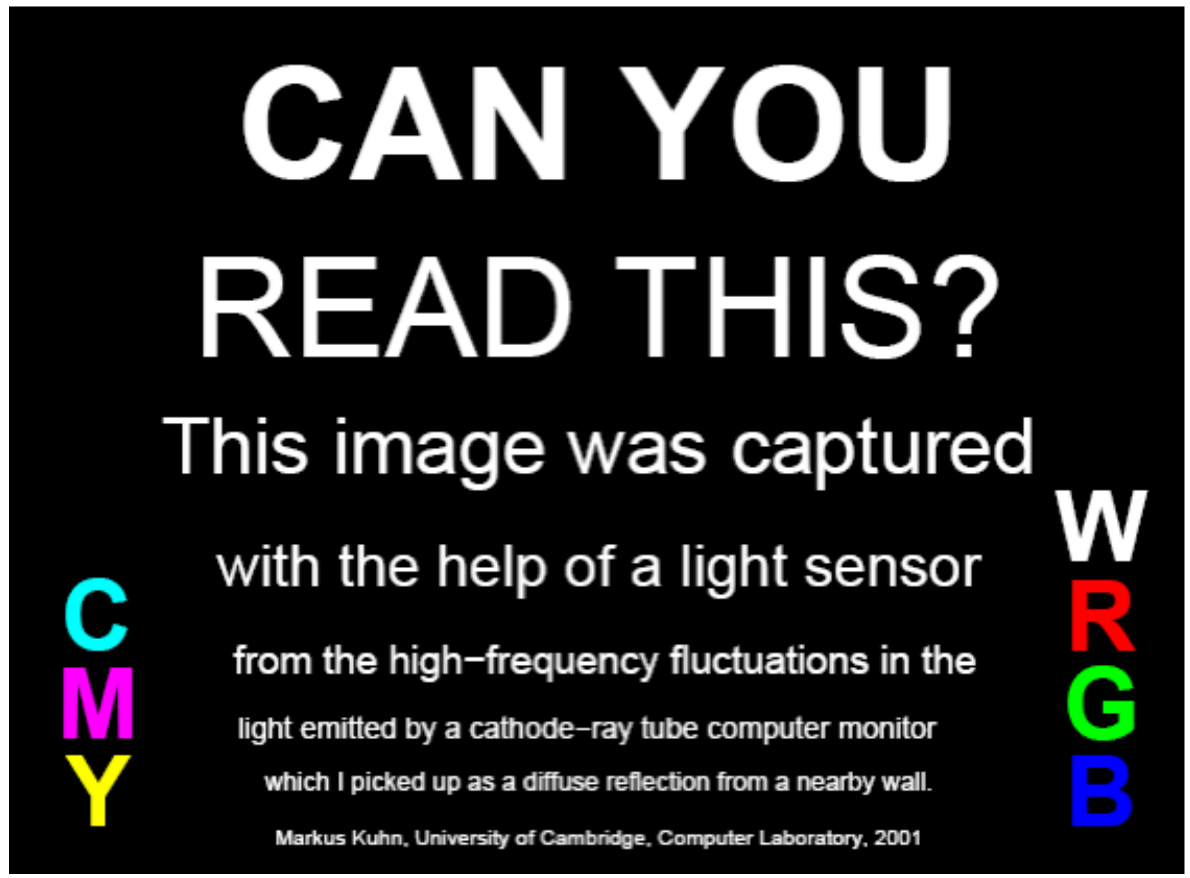
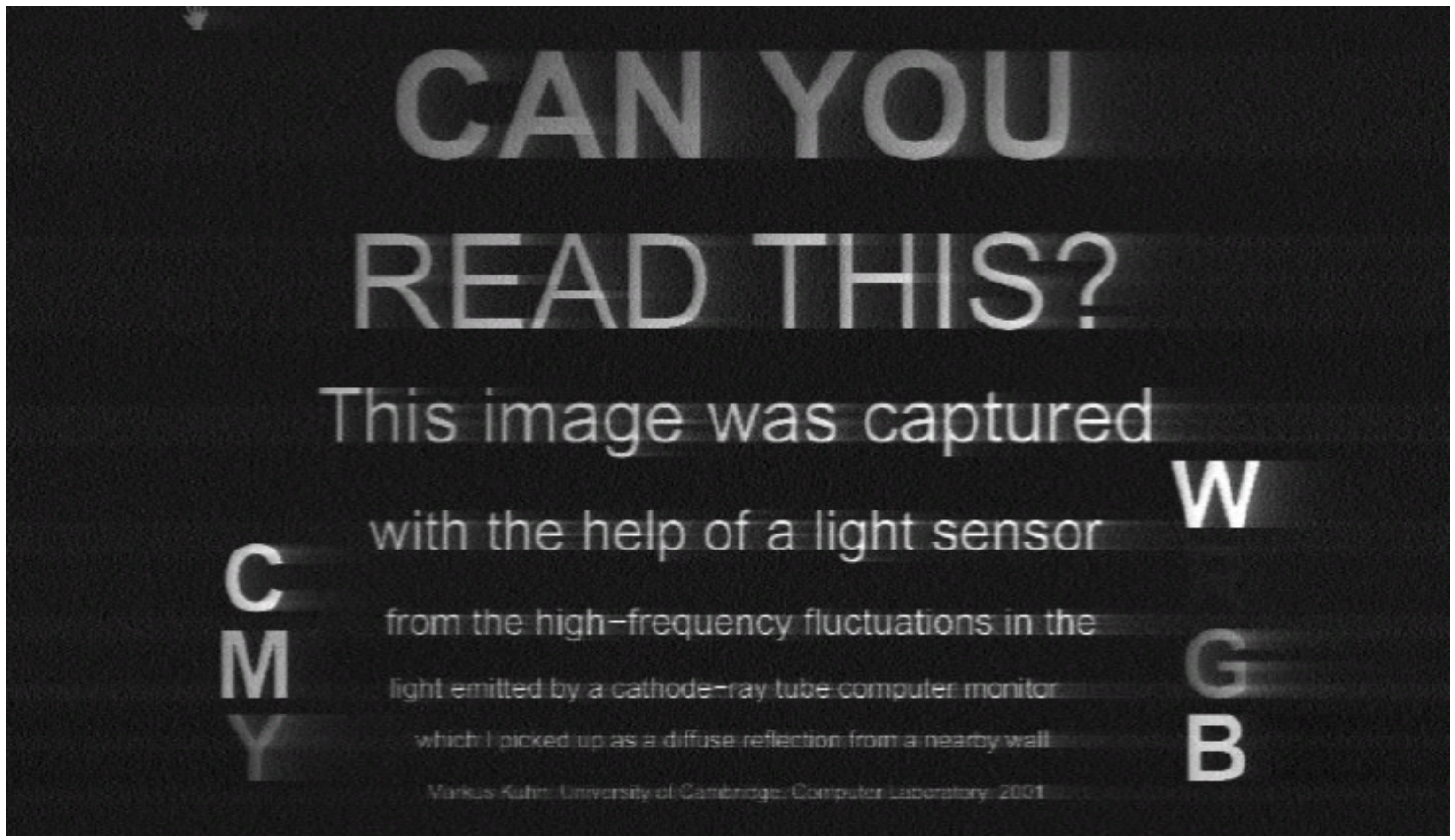


Figure 1. Photomultiplier tube module.

Source signal



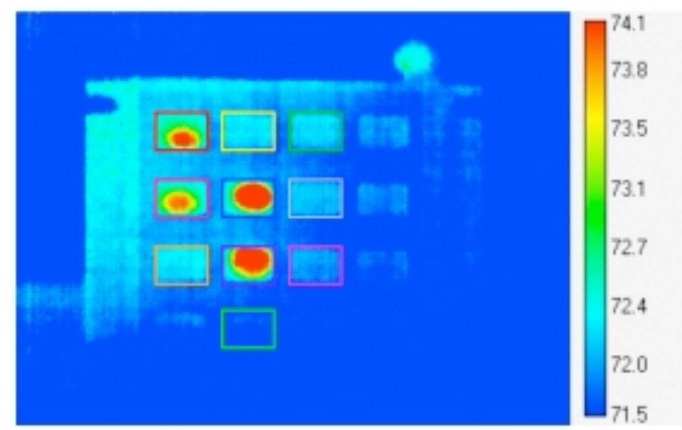
Bounced off a wall



Heat of the Moment

Meiklejohn et al. 2011

Figure 2: The Diebold plastic ATM keypad with rubber keys, model 19-019062-001M REV1.



Active side channels

- Faults can create additional side channels or amplify existing ones
  - Erroneous bit flips during secret operations may make it easier to recover secret internal state
- Attackers can induce faults—**fault injection attacks**
  - Glitch power, voltage, clock
  - Vary temperature
  - Subject to light, EM radiation

Aside: covert channels

- Side channels are inadvertent artifacts of the implementation that can be analyzed to extract information across a trust boundary
- **Covert channels:** same idea, but put on purpose
  - One party is trying to leak information in a way that it won't be obvious
  - By encoding that information into some side channel
    - E.g., variation in time, memory usage, etc.
  - Incredibly difficult to protect against

Mitigating side channels

- Eliminate dependency on secret data
- Make everything the same
  - Use the same of amount of resources every time
  - Hard (many optimizations in hardware, compilers, etc.)
  - Expensive (everything runs at worst-case performance)
- Hide
  - “Blinding” can be applied to input for some algorithms
- Adding random noise?
  - Attacker just needs more measurements to extract signal

Today

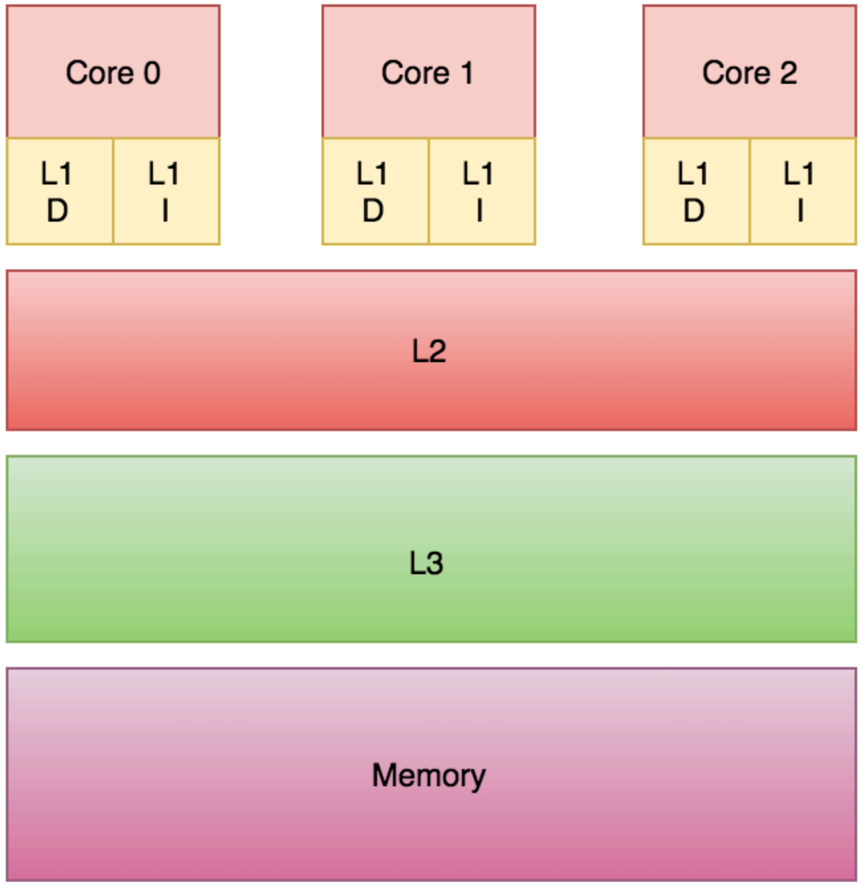
- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

What is the cache?

- Main memory is huge… but slow
- Processors try to “cache” recently used memory in faster, but smaller capacity, memory cells closer to the actual processing core

Cache hierarchy

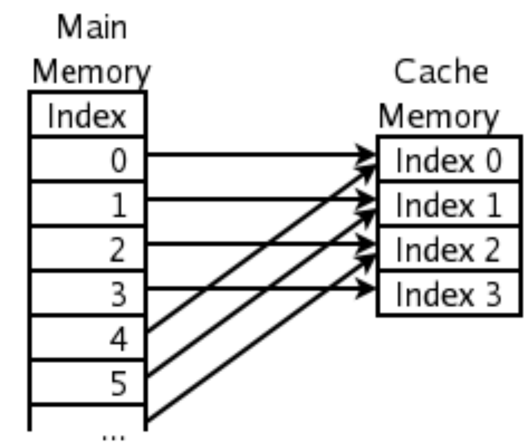
- Caches are such a great idea, let's have caches for caches!
- The close to the core, the:
  - Faster
  - Smaller



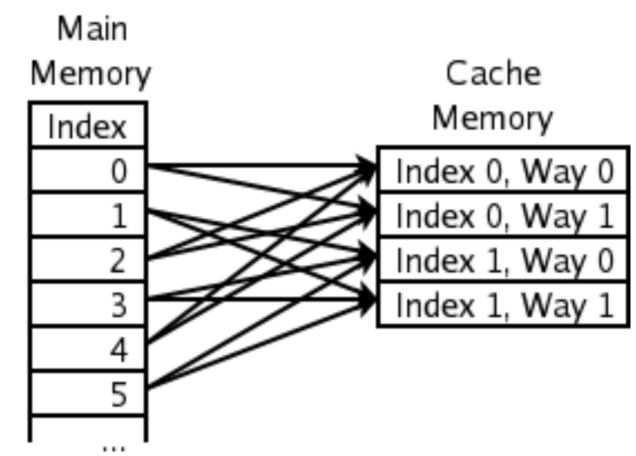
https://en.wikipedia.org/wiki/Cache_hierarchy

How is the cache organized?

- Cache line: unit of granularity
  - E.g., 64 bytes



- Cache lines grouped into sets
  - Each memory address is mapped to a set of cache lines



- What happens when we have collisions?
  - Evict!

https://en.wikipedia.org/wiki/CPU_cache

Cache side channel attacks

- Cache is a shared system resource
  - “Just a performance optimization”
  - Not isolated by process, VM, or privilege level
- We abuse this shared resource to learn information about another process, VM, etc.

Threat model

- Attacker and victim are isolated (e.g., in separate processes) but on the same physical system
- Attacker is able to invoke (directly or indirectly) functionality exposed by the victim
  - What's an example of this?
- Attacker should not be able to infer anything about the contents of victim memory

How is this an attack vector?

- Many algorithms have memory access patterns that are dependent on sensitive memory contents
  - What are some examples of this?
- So? If attacker can observe access patterns they can learn secrets

What can the attacker do?

- **Prime**: place a known address in the cache (by reading it)
- **Evict**: access mem until address is no longer cached (force capacity misses)
- **Flush**: remove address from the cache (`cflush` on x86)
- **Measure**: precisely (down the the cycle) how long it takes to do something (`rdtsc` on x86)
- <u>Attack form</u>: manipulate cache into known state, make victim run, try to infer what changed in the change

Three basic techniques

- Evict and time
  - Kick stuff out of the cache and see if victim slows down as a result
- Prime and probe
  - Put stuff in the cache, run the victim and see if you slow down as a result
- Flush and reload
  - Flush a particular line from the cache, run the victim and see if your accesses are still fast as a result

Evict & Time

- Baseline
  - Run the victim code several times and time it
- Evict (portions of) the cache
- Run the victim code again and retime it
- If it is slower than before, cache lines evicted by the attacker must've been used by the victim
  - We now know something about victim <u>addresses</u>
  - In some cases addresses are secret (e.g., AES)

Prime & Probe

- Prime the cache
  - ➢ Access many memory locations (covering all cache lines of interest) so previous cache contents are replaced with attacker addresses
  - ➢ Time access to each cache line ("in cache" reference)
- Run victim code
- Attacker retimes access to own memory locations
  - ➢ If any are slower then it means the corresponding cache line was used by the victim
  - ➢ We now know something about the victim <u>addresses</u>

Flush & Reload

- Time memory access to (potentially) shared regions
- Flush (specific lines from) the cache
- Invoke victim code
- Retime access to flushed addresses, if still fast was used by victim
  - Because we flushed it it should be slow, victim must have reloaded it
  - We now know something about the victim <u>addresses</u>

Today

- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

Timing (+ cache) side channels

- Good for the attacker:
  - Remote attackers can exploit timing channels
  - Co-located attacker (on same physical machine) can abuse cache side channel
- Good for defense
  - Can eliminate timing channels
  - Performance overhead of doing so is reasonable

To understand how to eliminate the channels we need to understand 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

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

Why? Floating-point time variability

Processor	+ subnormal	+ special	× subnormal	× special	÷ subnormal	÷ special	÷ x^2	÷ 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	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗

Some instructions introduce time variability

- **Problem:** Certain instructions take different amounts of time depending on the operands
  - ➢ If input data is secret: might leak some of it!
- **Solution?**
  - ➢ In general, don't use variable-time instructions

When ARMv8.4-DIT is implemented:

Data Independent Timing.

<table>
<tr><th>DIT</th><th>Meaning</th></tr>
<tr><td>0b0</td><td>The architecture makes no statement about the timing properties of any instructions.</td></tr>
<tr><td>0b1</td><td>The architecture requires that:
The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.
For certain data processing instructions, the instruction takes a time which is independent of:
The values of the data supplied in any of its registers.
The values of the NZCV flags.
For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its registers.
The values of the NZCV flags.</td></tr>
</table>

The data processing instructions affected by this bit are:

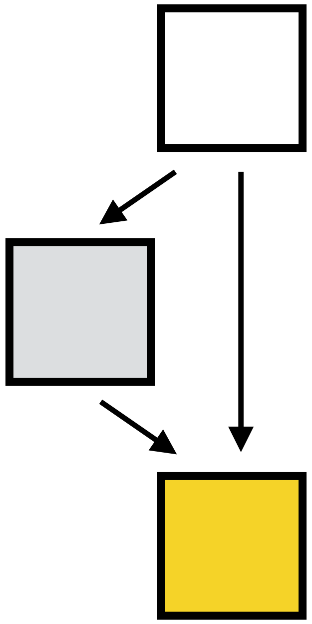
- All cryptographic instructions. These instructions are:
  - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination and pass their condition execution check. The instructions are:
  - BFI, BFC, CLZ, CMN, CMP, MLA, MLAS, MLS, MOVT, MUL, MULS, NOP, PKHBT, PKHTB, RBIT, REV, REV16, REVSH, RRX, SADD16, SADD8, SASX, SBFX, SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLAL**, SMLAW*, SMLSD*, SMMLA*, SMMLS*, SMMUL*, SMUAD*, SMUL*, SSAX, SSUB16, SSUB8, SXTAB*, SXTAH, SXTB*, SXTH, TEQ, TST, UADD*, UASX, UBFX, UHADD*, UHASX, UHSAX, UHSUB*, UMAAL, UMLAL, UMLALS, UMULL, UMULLS, USADA8, USAX, USUB*, UXTAB*, UXTAH, UXTB*, UXTH, ADC (register-shifted register), ADCS (register-shifted register), ADD (register-shifted register), ADDS (register-shifted register), AND (register-shifted register), ANDS (register-shifted register), ASR (register-shifted register), ASRS (register-shifted register), BIC (register-shifted register), BICS (register-shifted register), EOR (register-shifted register), EORS (register-shifted register), LSL (register-shifted register), LSLS (register-shifted register), LSR (register-shifted register), LSRS (register-shifted register), MOV (register-shifted register), MOVS (register-shifted register), MVN (register-shifted register), MVNS (register-shifted register), ORR (register-shifted register), ORRS (register-shifted register), ROR (register-shifted register), RORS (register-shifted register), RSB (register-shifted register), RSBS (register-shifted register), RSC (register-shifted register), RSCS (register-shifted register), SBC (register-shifted register), SBCS (register-shifted register), SUB (register-shifted register), and SUBS (register-shifted register).

Control flow introduces time variability

```
m=1
for i = 0 ... len(d):
      if d[i] = 1:
        m = c * m mod N
      m = square(m) mod N
return m
```

if-statements on secrets are 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

Why padding branches doesn't work

- **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
- **What can we do?**

Don’t branch on secrets!

Real code needs to branch…

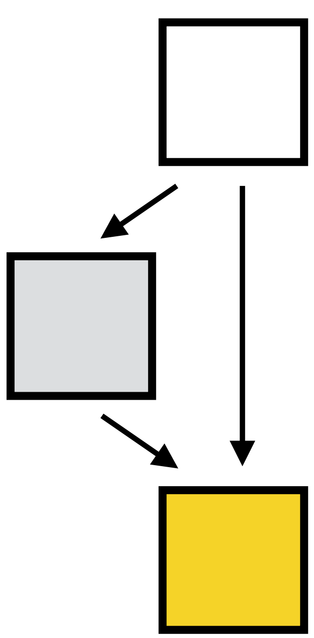
Fold control flow into data flow

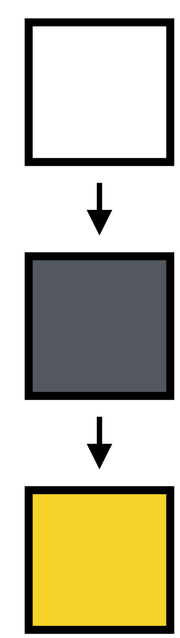
(assumption secret = 1 or 0)

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

➡

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





Fold control flow into data flow

(assumption secret = 1 or 0)

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

➡

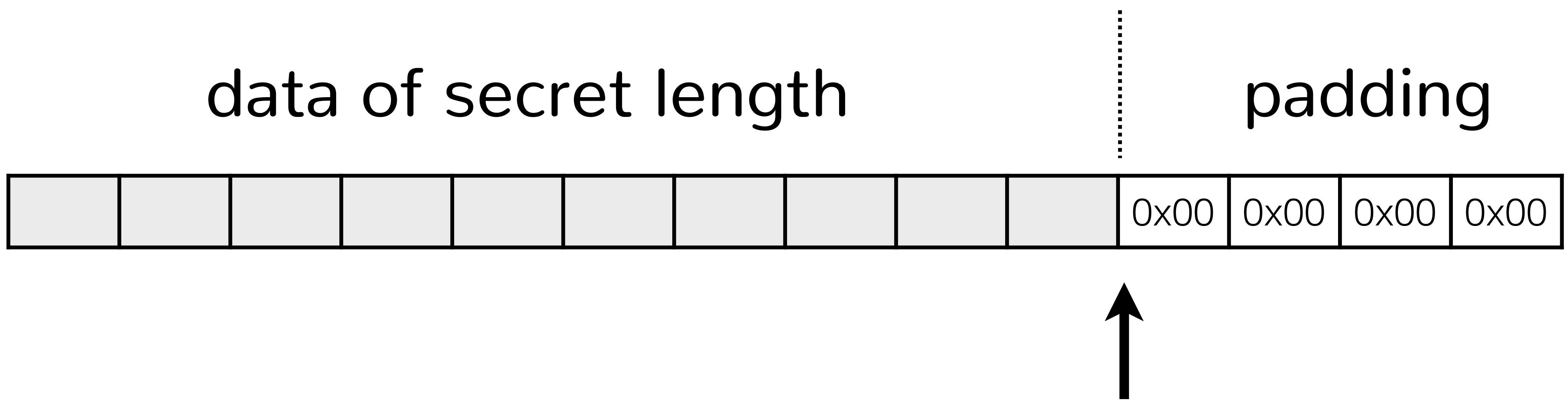
```
x = secret * a
  + (1-secret) * x;

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

Fold control flow into data flow

- Multiple ways to fold control flow into data flow
  - Previous example: takes advantage of arithmetic
  - What’s another way?

```
/* Constant-time helper macro that selects l or r depending on all-1 or all-0
 * mask m */
#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
```

An example from mbedTLS



Goal: get the length of the padding so we can remove it

An example from mbedTLS

```
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;
}
```

An example from mbedTLS

```
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?

An example from mbedTLS

```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?

An example from mbedTLS

```
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?

An example from mbedTLS

```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?

An example from mbedTLS

```
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 = CT_SEL(done & !prev_done, i, *data_len);
    }

    return 0;
}
```

Is this safe?

Control flow introduces time variability

- **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!

Memory access patterns introduce time variability

```
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]];
...
```

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 = CT_SEL(EQ(secret, i), arr[i], x)
```

```
/* Constant-time helper macro that selects l or r depending on all-1 or all-0
 * mask m */
#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
```

Summary: what introduces time variability?

- 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

Solution: constant-time programming

- 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!

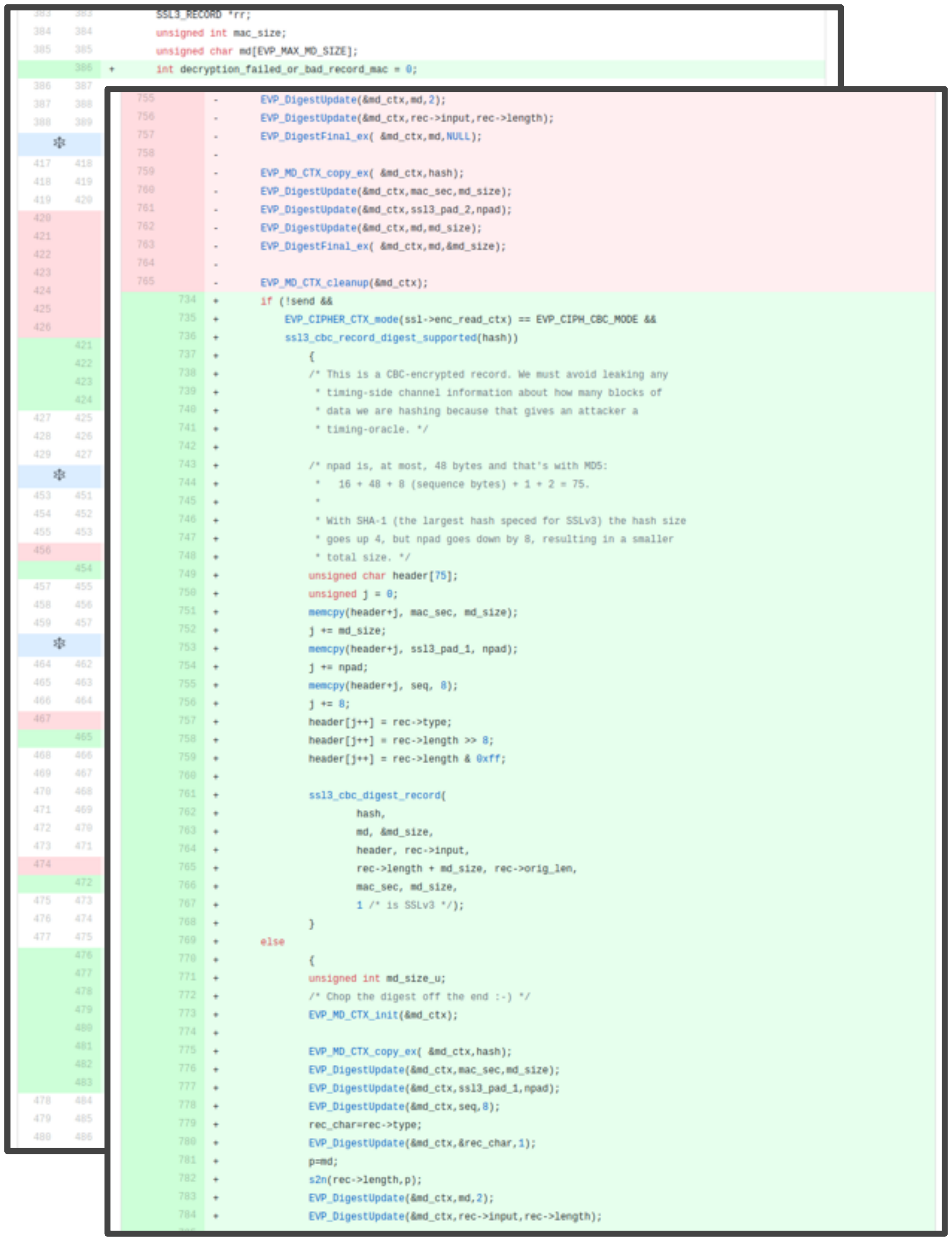
Aside: Writing CT code is unholy

```diff
         SSL3_RECORD *rr;
         unsigned int mac_size;
         unsigned char md[EVP_MAX_MD_SIZE];
+        int decryption_failed_or_bad_record_mac = 0;


         rr= &(s->s3->rrec);
@@ -417,13 +418,10 @@ dtls1_process_record(SSL *s)
         enc_err = s->method->ssl3_enc->enc(s,0);
         if (enc_err <= 0)
                 {
-                /* decryption failed, silently discard message */
-                if (enc_err < 0)
-                        {
-                        rr->length = 0;
-                        s->packet_length = 0;
-                        }
-                goto err;
+                /* To minimize information leaked via timing, we will always
+                 * perform all computations before discarding the message.
+                 */
+                decryption_failed_or_bad_record_mac = 1;
                 }

 #ifdef TLS_DEBUG
@@ -453,7 +451,7 @@ printf("\n");
                         SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_PRE_MAC_LENGTH_TOO_LONG);
                         goto f_err;
 #else
-                        goto err;
+                        decryption_failed_or_bad_record_mac = 1;
 #endif
                         }
                 /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
@@ -464,17 +462,25 @@ printf("\n");
                         SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
                         goto f_err;
 #else
-                        goto err;
+                        decryption_failed_or_bad_record_mac = 1;
 #endif
                         }
                 rr->length-=mac_size;
                 i=s->method->ssl3_enc->mac(s,md,0);
                 if (i < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
                         {
-                        goto err;
+                        decryption_failed_or_bad_record_mac = 1;
                         }
                 }

+        if (decryption_failed_or_bad_record_mac)
+                {
+                /* decryption failed, silently discard message */
+                rr->length = 0;
+                s->packet_length = 0;
+                goto err;
+                }
+
         /* r->length is now just compressed */
         if (s->expand != NULL)
                 {
```

OpenSSL padding oracle attack

Canvel, et al. “Password Interception in a SSL/TLS Channel.” Crypto, Vol. 2729. 2003.

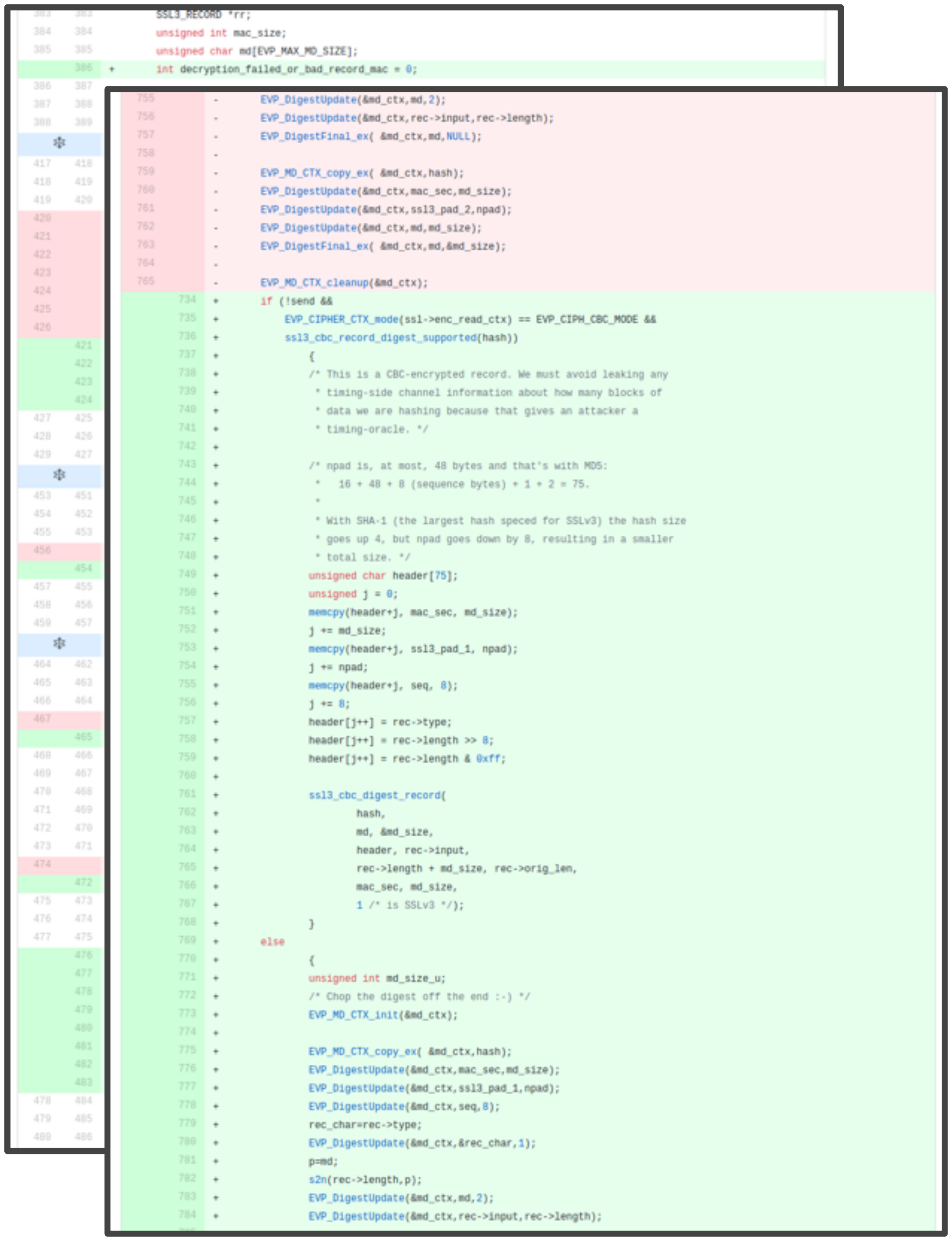
Aside: Writing CT code is unholy



OpenSSL padding oracle attack

Canvel, et al. “Password Interception in a SSL/TLS Channel.” Crypto, Vol. 2729. 2003.

Aside: Writing CT code is unholy



OpenSSL padding oracle attack

Canvel, et al. "Password Interception in a SSL/TLS Channel." Crypto, Vol. 2729. 2003.

Lucky 13 timing attack

Al Fardan and Paterson. "Lucky thirteen: Breaking the TLS and DTLS record protocols." Oakland 2013.

Aside: Writing CT code is unholy

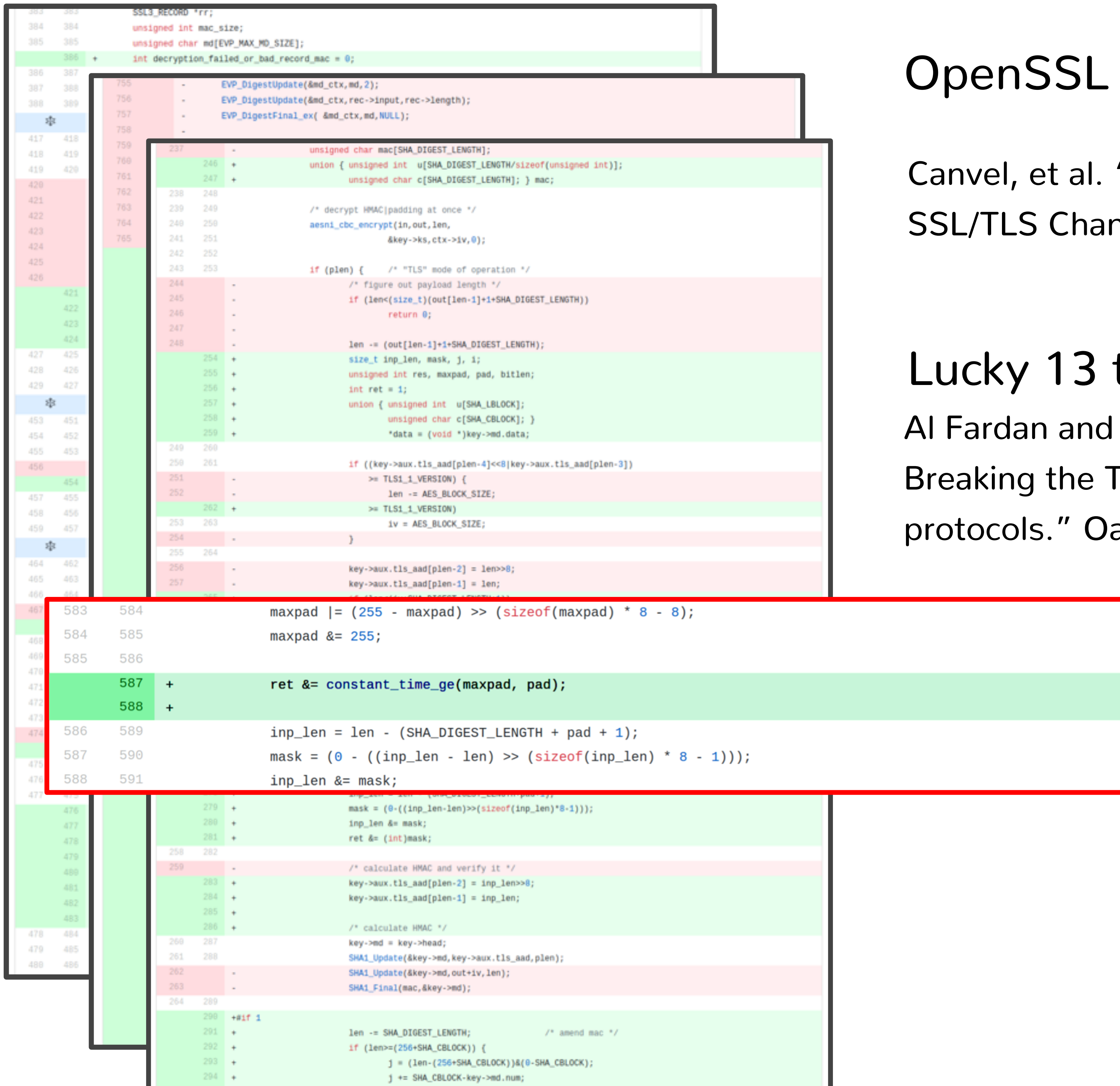
OpenSSL padding oracle attack

Canvel, et al. “Password Interception in a SSL/TLS Channel.” Crypto, Vol. 2729. 2003.

Lucky 13 timing attack

Al Fardan and Paterson. “Lucky thirteen: Breaking the TLS and DTLS record protocols.” Oakland 2013.

Aside: Writing CT code is unholy



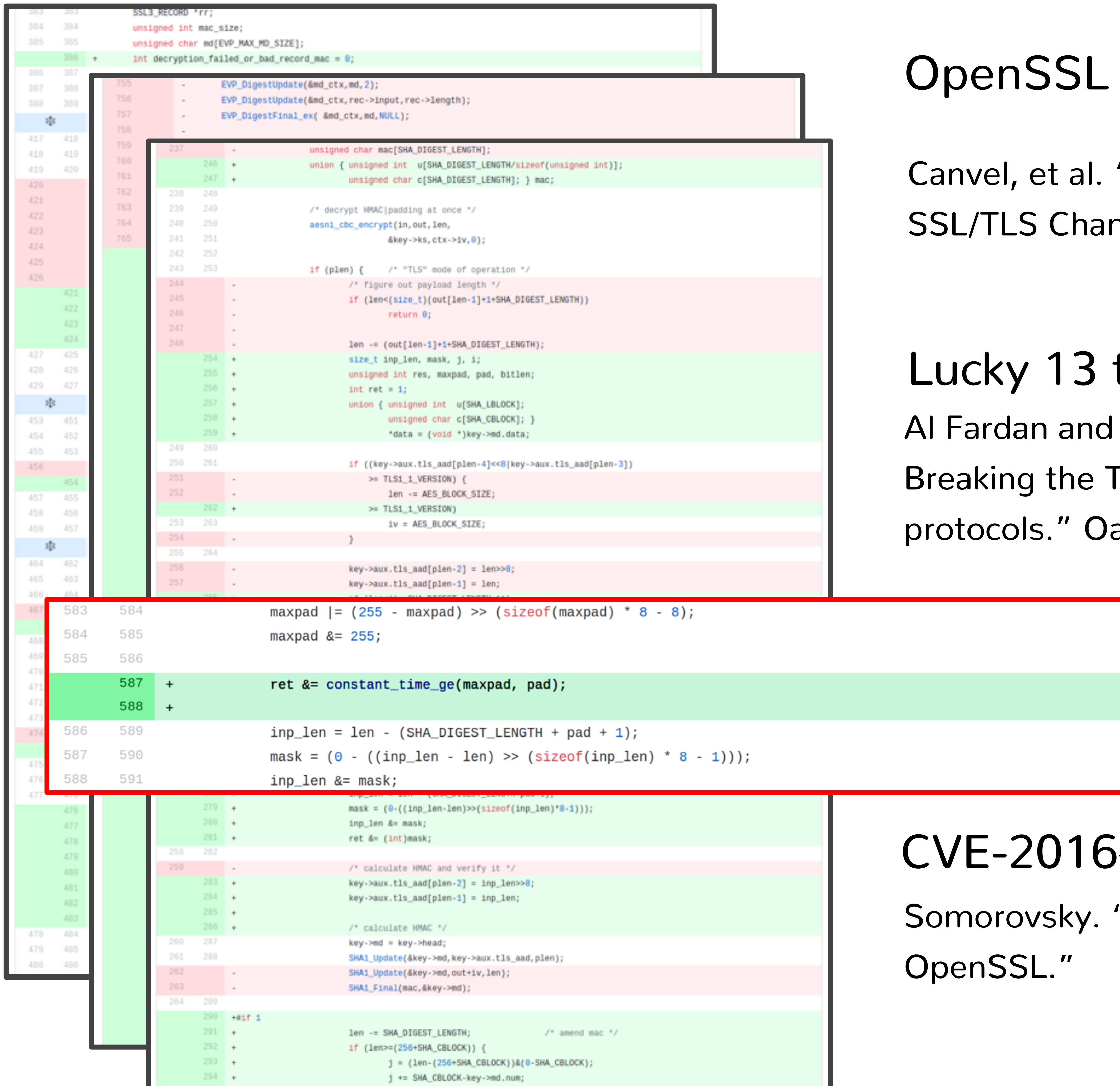
OpenSSL padding oracle attack

Canvel, et al. "Password Interception in a SSL/TLS Channel." Crypto, Vol. 2729. 2003.

Lucky 13 timing attack

Al Fardan and Paterson. "Lucky thirteen: Breaking the TLS and DTLS record protocols." Oakland 2013.

Aside: Writing CT code is unholy



OpenSSL padding oracle attack

Canvel, et al. “Password Interception in a SSL/TLS Channel.” Crypto, Vol. 2729. 2003.

Lucky 13 timing attack

Al Fardan and Paterson. “Lucky thirteen: Breaking the TLS and DTLS record protocols.” Oakland 2013.

CVE-2016-2107

Somorovsky. “Curious padding oracle in OpenSSL.”

What can we do about this?

- Design new programming languages!
  - E.g., FaCT language lets you write code that is guaranteed to be constant time

```
export
void get_zeros_padding( secret uint8 input[], secret mut uint32 data_len)
{

    data_len = 0;
    for( uint32 i = len input; i > 0; i-=1 ) {
        if (input[i-1] != 0) {
            data_len = i;
            return;
        }
    }
}
```

Automatically transform code when possible!

```
export
void conditional_swap(secret mut uint32 x,
                      secret mut uint32  y,
                      secret bool cond) {
  if (cond) {
    secret uint32 tmp = x;
    x = y;
    y = tmp;
  }
}
```

```
export
void conditional_swap(secret mut uint32 x,
                      secret mut uint32  y,
                      secret bool cond) {
  secret mut bool __branch1 = cond;
  { // then part
    secret uint32 tmp = x;
    x = CT_SEL(__branch1, y, x);
    y = CT_SEL(__branch1, tmp, y);
  }
  __branch1 = !__branch1;
  {... else part ...}
}
```

Raise type error otherwise!

- Some transformations not possible
  - E.g., loops bounded by secret data
- Some transformations would produce slow code
  - E.g., accessing array at secret index

Today

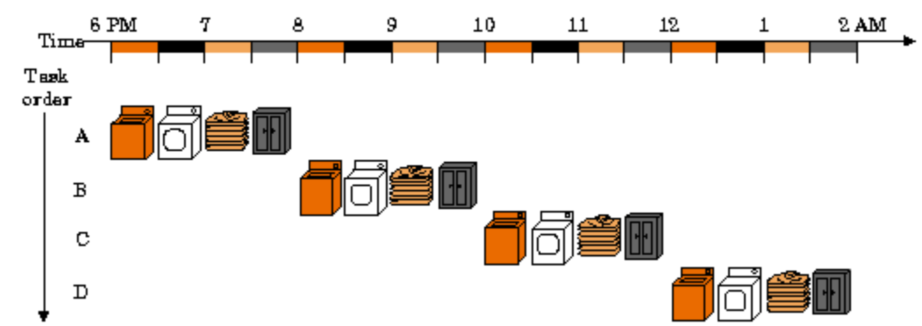
- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

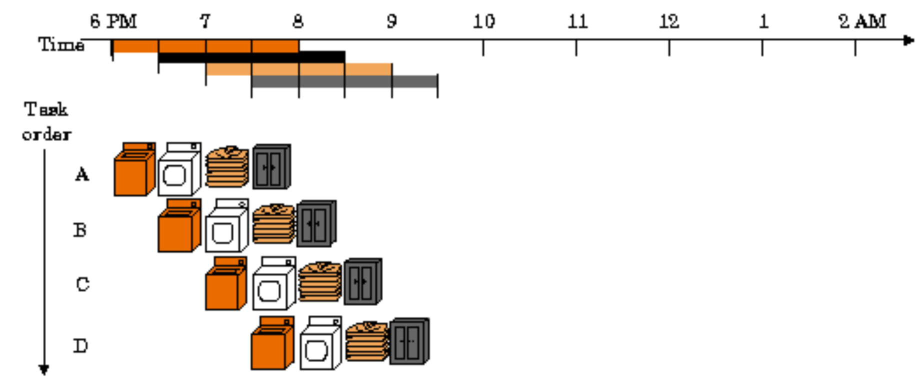
Quick review: ISA and μArchitecture

- Instruction set architecture
  - Defined interface between HW and SW
- μArchitecture
  - Implementation of the ISA
  - “Behind the curtain” details
    - E.g. cache specifics
- **Key issue:** μArchitectural details cam sometimes become “architecturally visible”

Review: Instruction pipelining

- Processors break up instructions into smaller parts so that these parts could be processed in parallel
- µArchitectural optimization
  - ➤ Instructions appear to be executed one at a time, in order
  - ➤ Dependencies are resolved behind the scenes

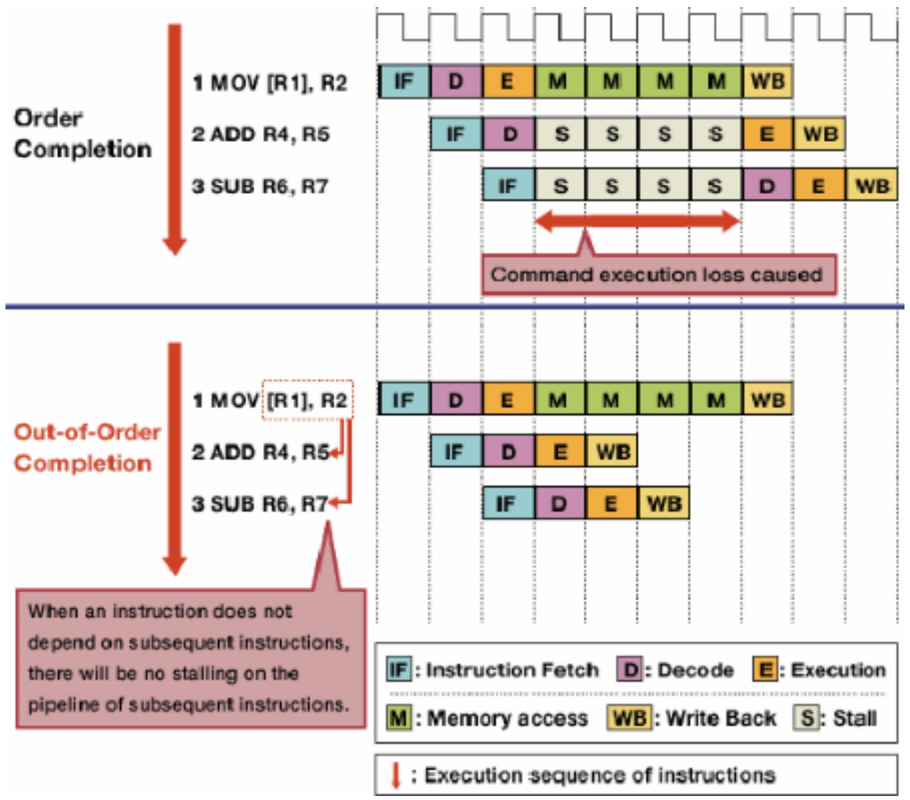




https://www.cs.fsu.edu/~hawkes/cda3101lects/chap6/index.html?$$$F6.1.html$$$

Review: Out-of-order execution

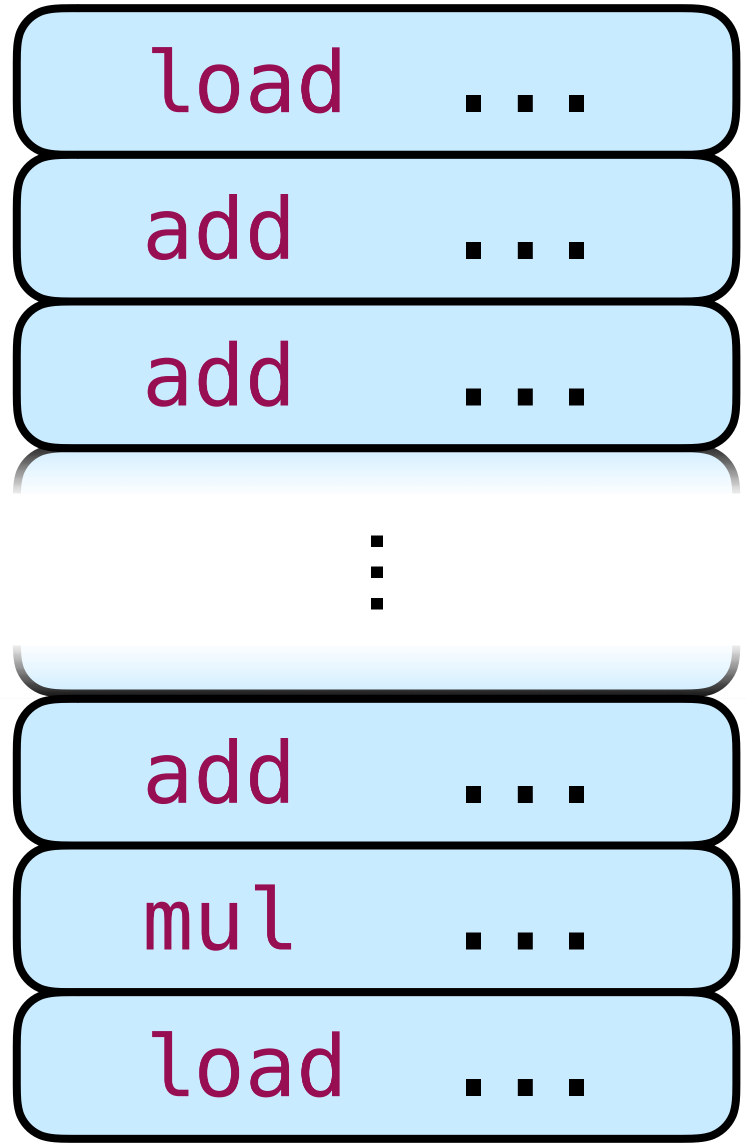
- Some instructions can be safely executed in a different order than they appear
- Avoid unnecessary pipeline stalls
- µArchitectural optimization
  - ➤ Architecturally, it appears that instructions are executed in order
- Can go wrong: Meltdown attacks

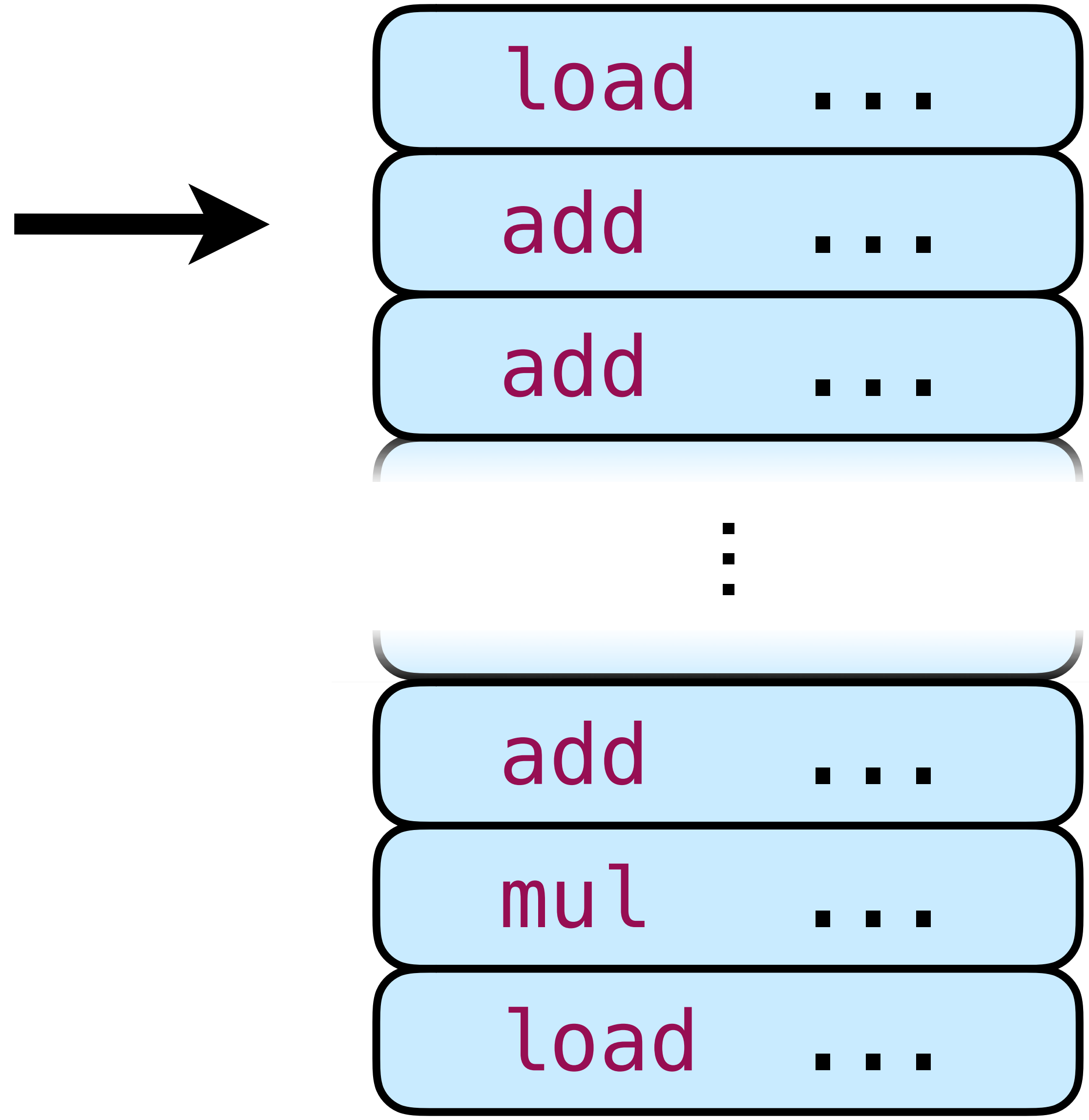


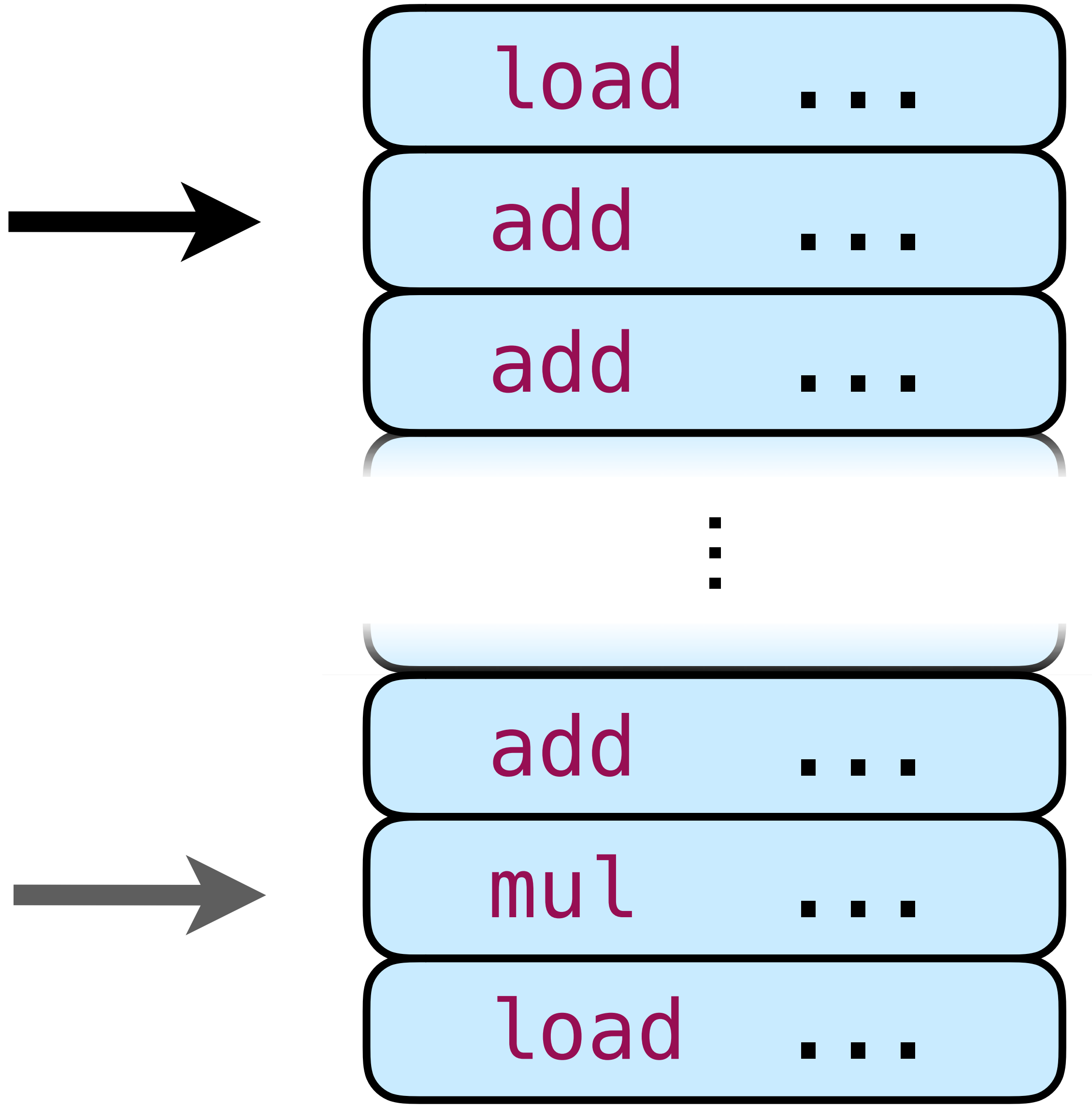
https://renesasrulz.com/doctor_micro/rx_blog/b/weblog/posts/pipeline-and-out-of-order-instruction-execution-optimize-performance

Review: Speculative execution

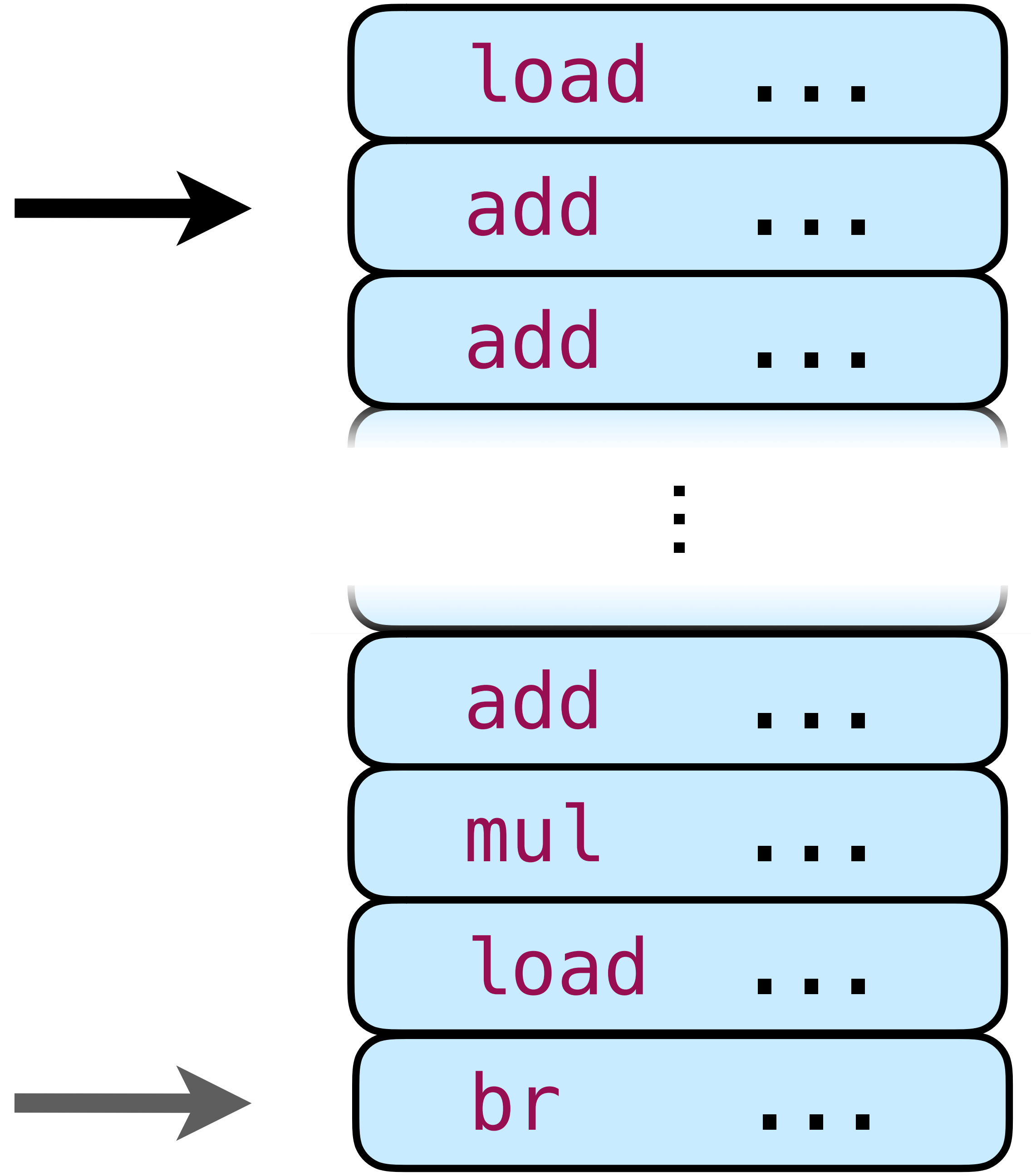
- Control flow could depend on output of earlier instruction
  - E.g. conditional branch, function pointer
- Rather than wait to know which way to go, the processor may "speculate" about the direction/target of a branch
  - Guess based on the past
  - If the guess is correct, performance is improved
  - If the guess is wrong, speculated computation is discarded and everything is re-computed using the correct value.
- µArchitectural optimization
  - At the ISA level, only correct, in-order execution is visible

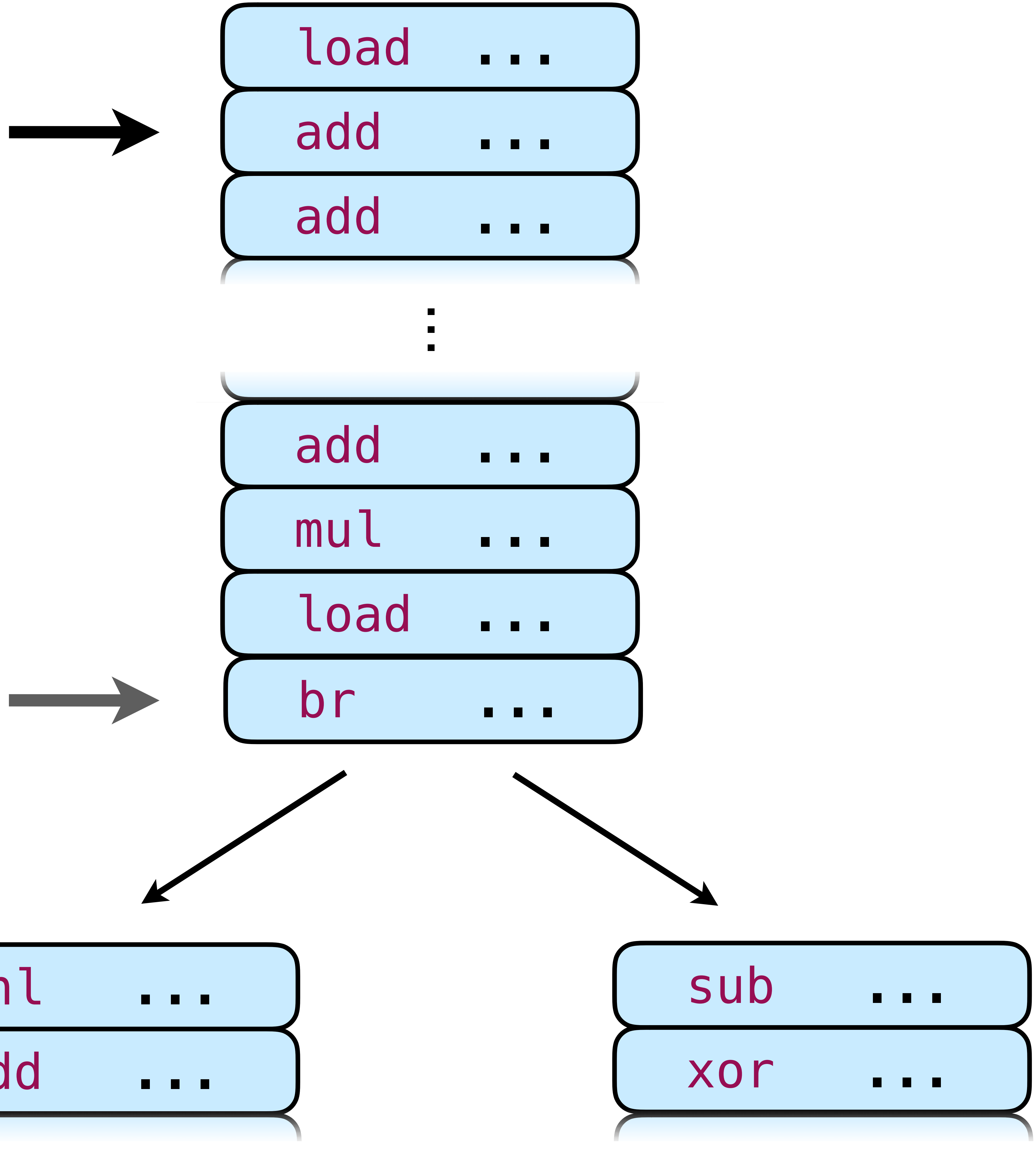

load ...
add ...
add ...
⋮
add ...
mul ...
load ...


load ...
add ...
add ...
add ...
mul ...
load ...

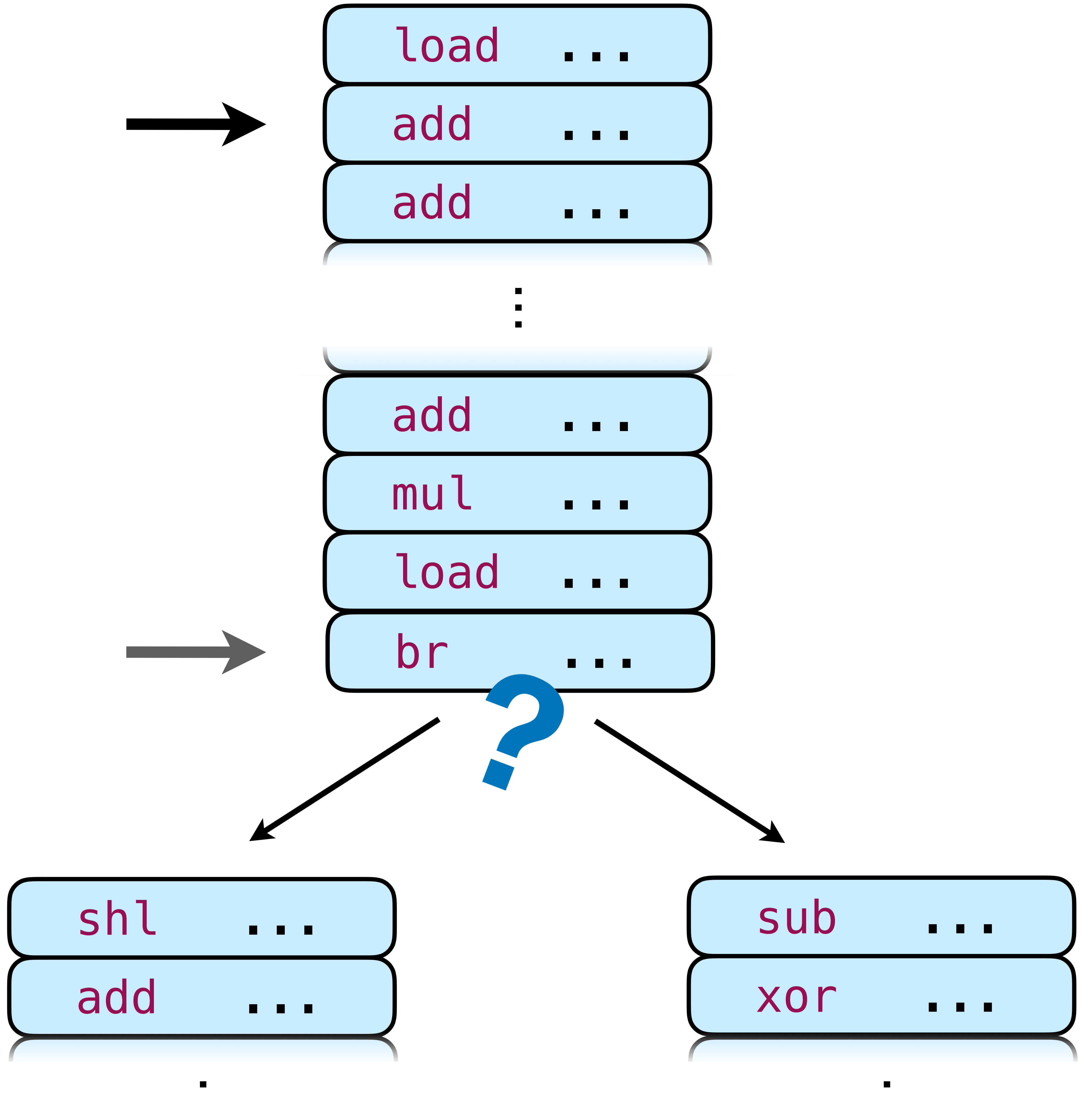


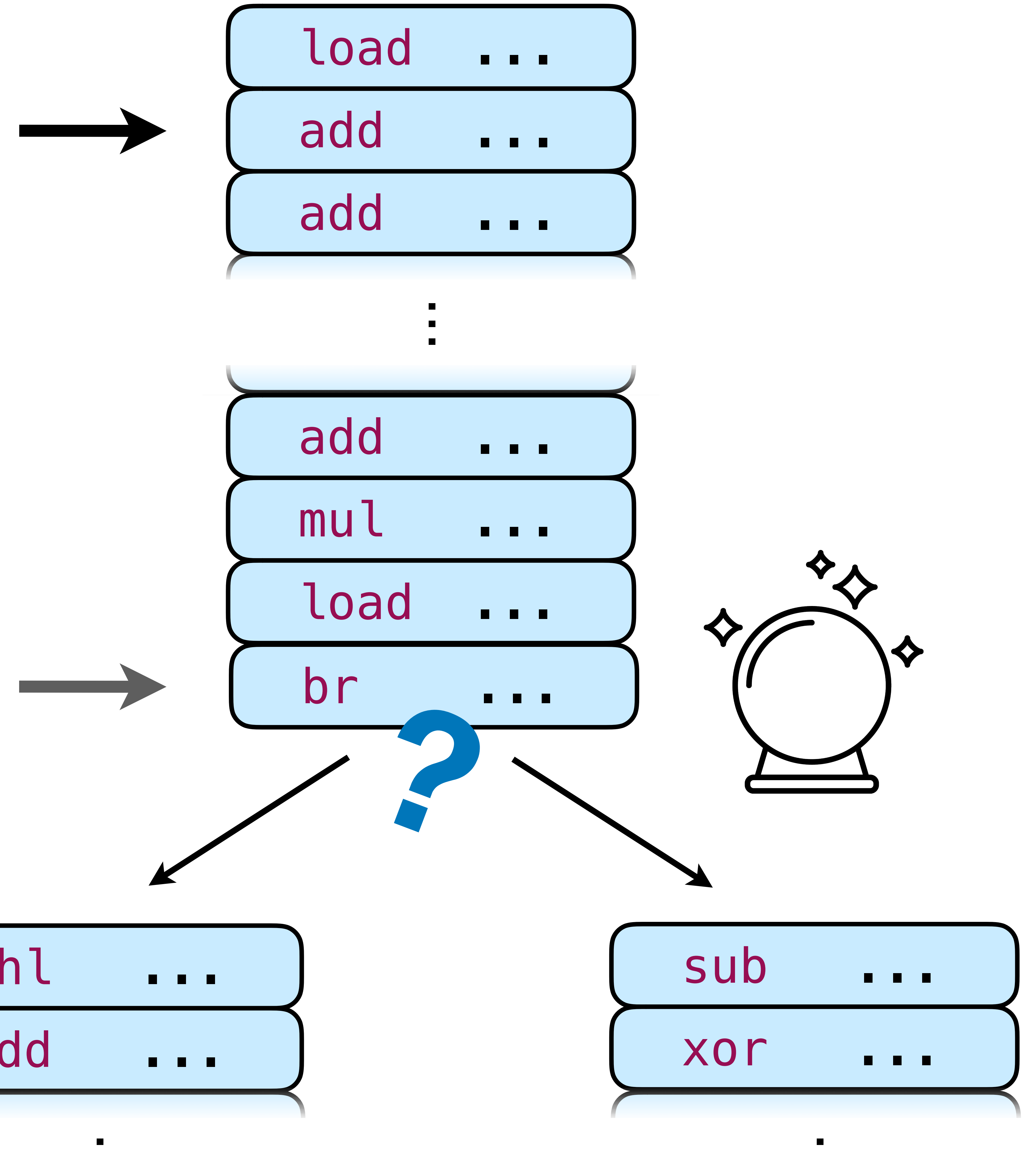
load ...
add ...
add ...
⋮
add ...
mul ...
load ...


load ...
add ...
add ...
⋮
add ...
mul ...
load ...
br ...

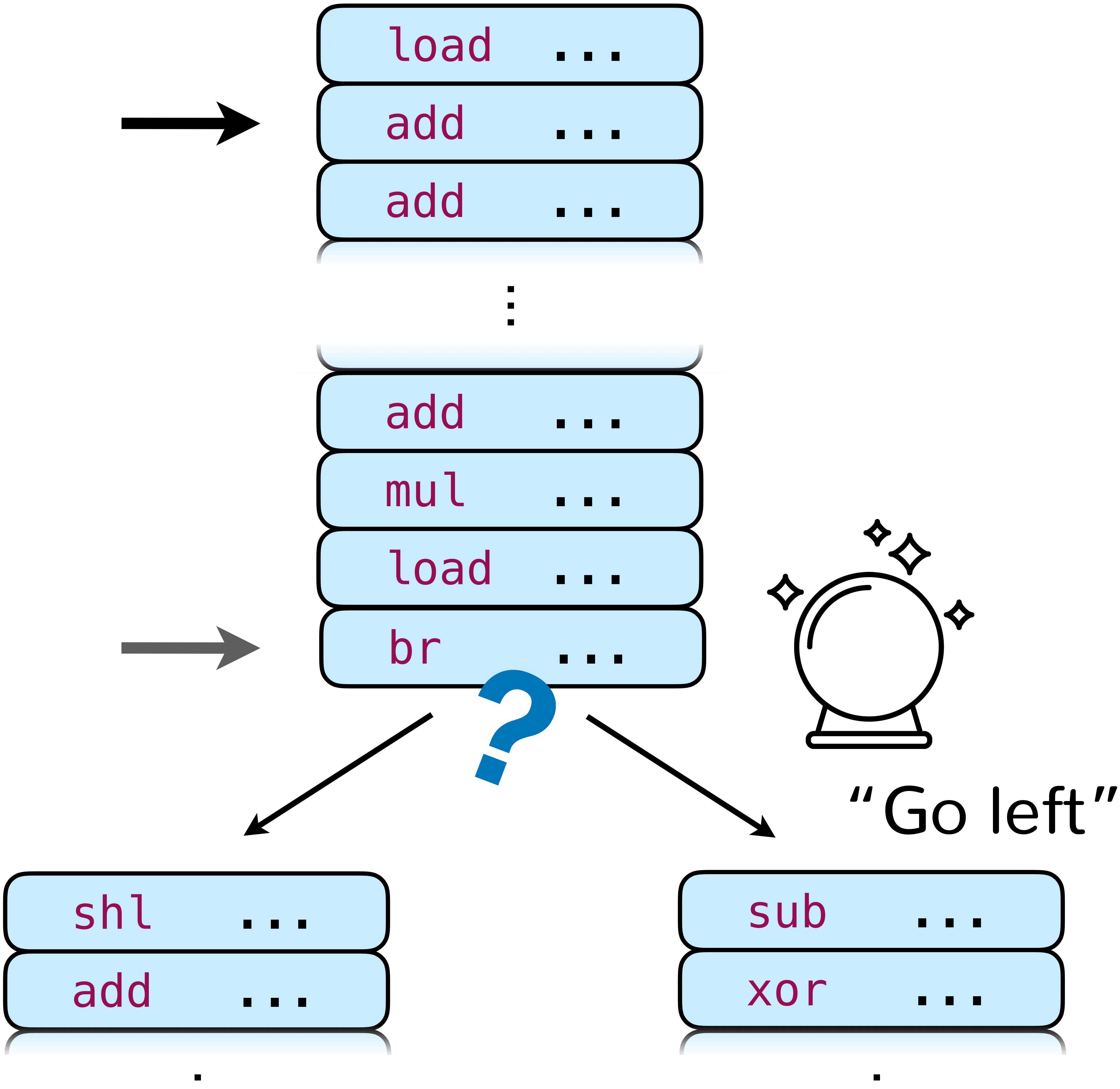


load ...
add ...
add ...
add ...
mul ...
load ...
br ...
shl ...
add ...
sub ...
xor ...


load ...
add ...
add ...
add ...
mul ...
load ...
br ...
?
shl ...
add ...
sub ...
xor ...

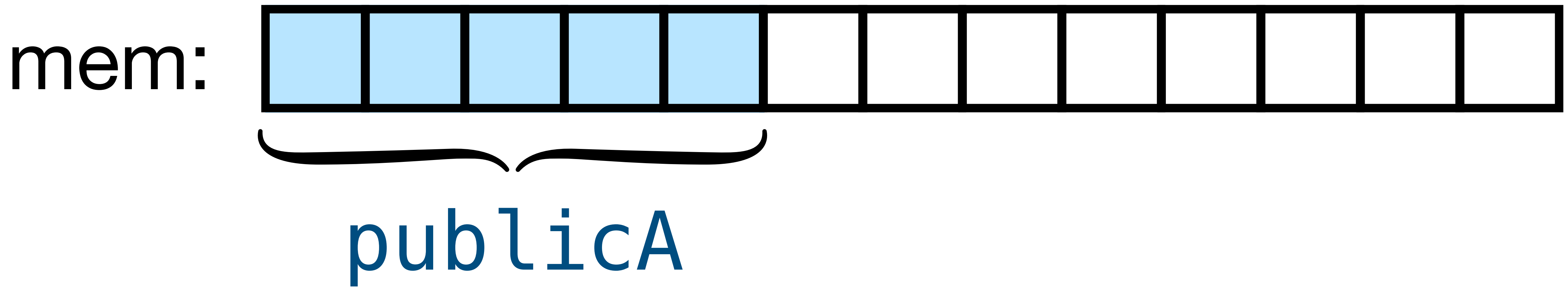


load ...
add ...
add ...
add ...
mul ...
load ...
br ...
?
shl ...
add ...
sub ...
xor ...

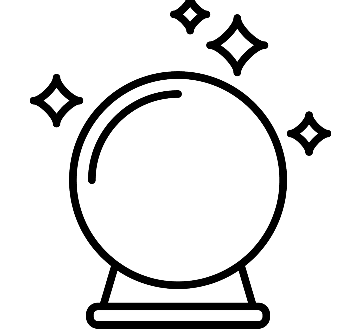


load ...
add ...
add ...
add ...
mul ...
load ...
br ...
"Go left"
shl ...
add ...
sub ...
xor ...

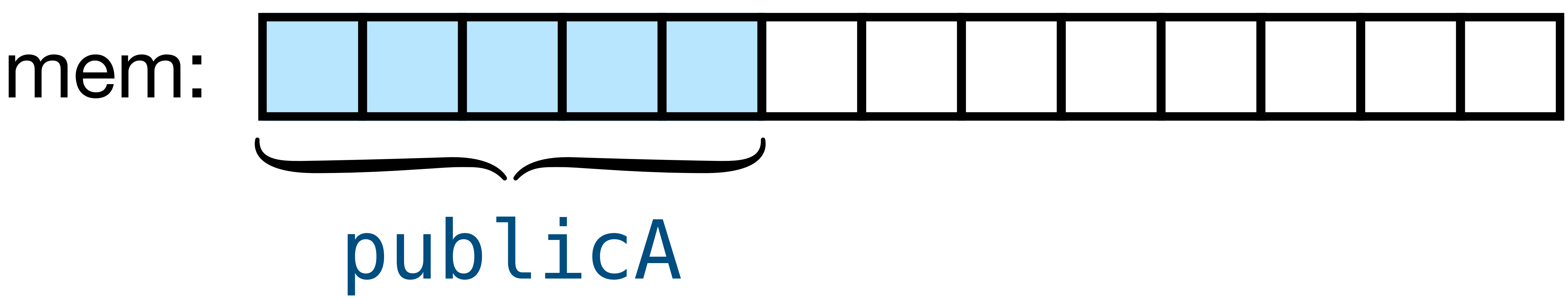
```
if (n < publicLen) {
  x = publicA[n];
  y = publicB[x];
} else {
  ...
```



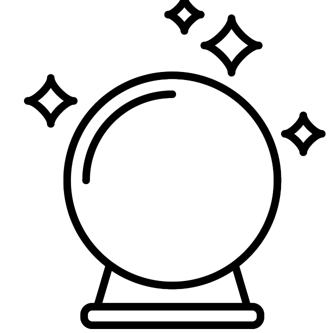
```
if (n < publicLen) {
  x = publicA[n];
  y = publicB[x];
} else {
  ...
```



“Condition is true”

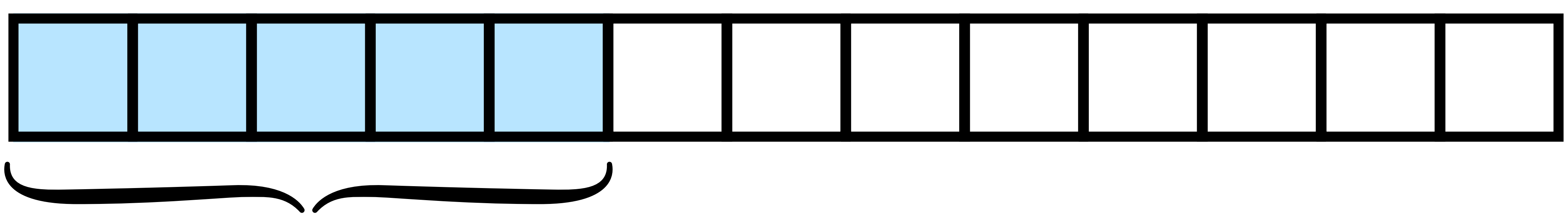


```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```



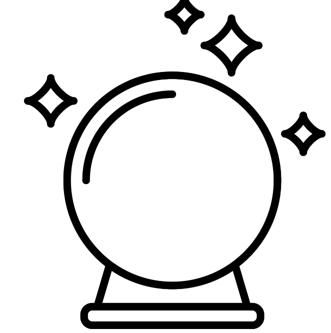
“Condition is true”

mem:

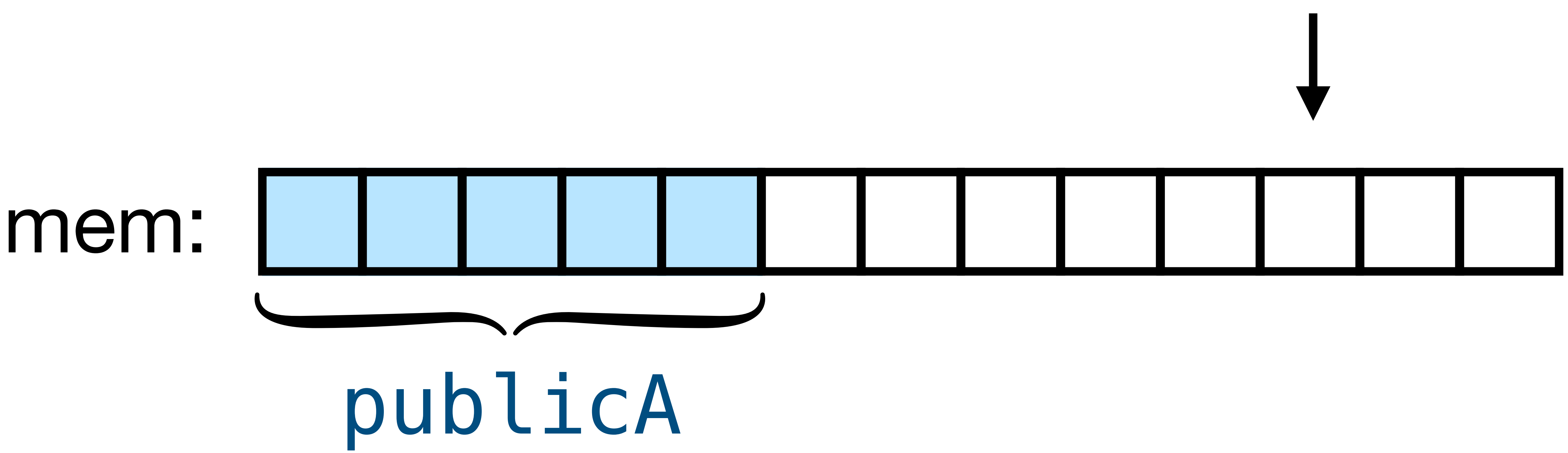


publicA

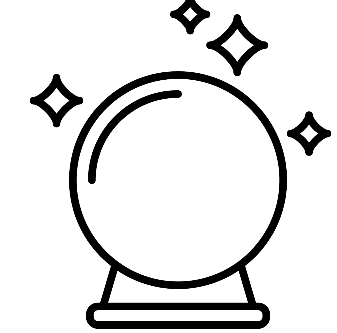
```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {

    ...
```



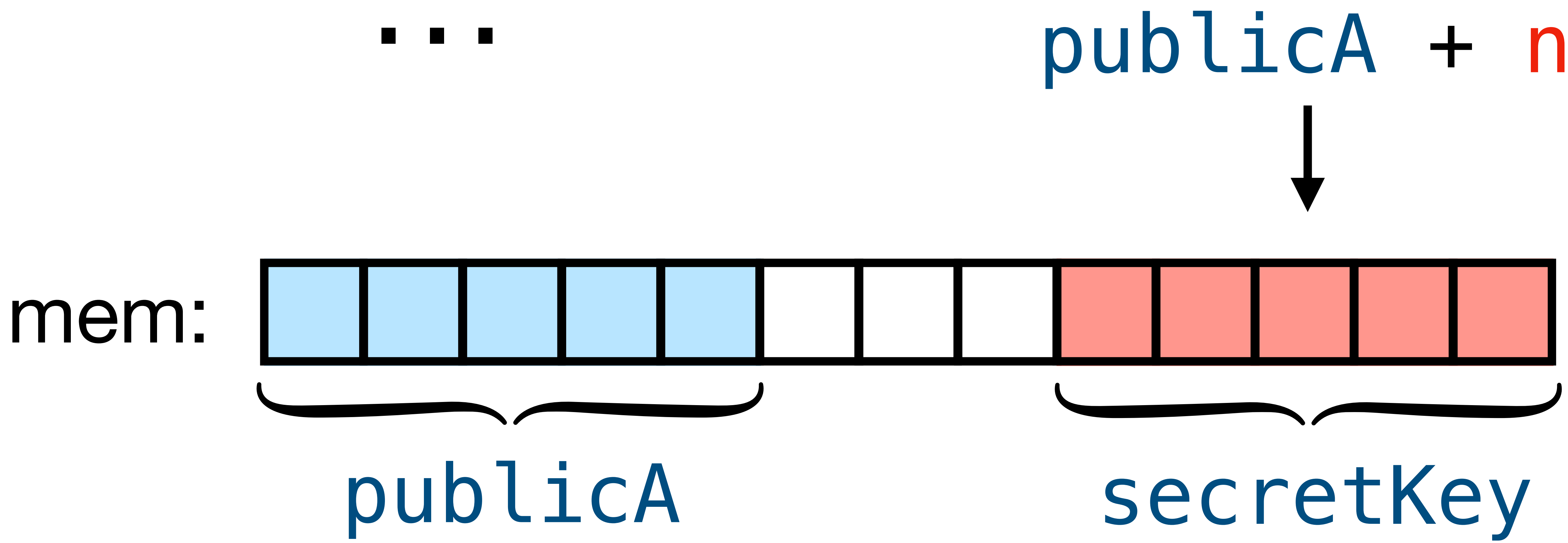
“Condition is true”



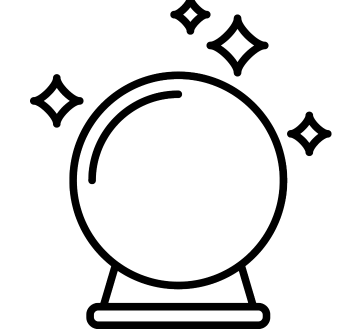
```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```



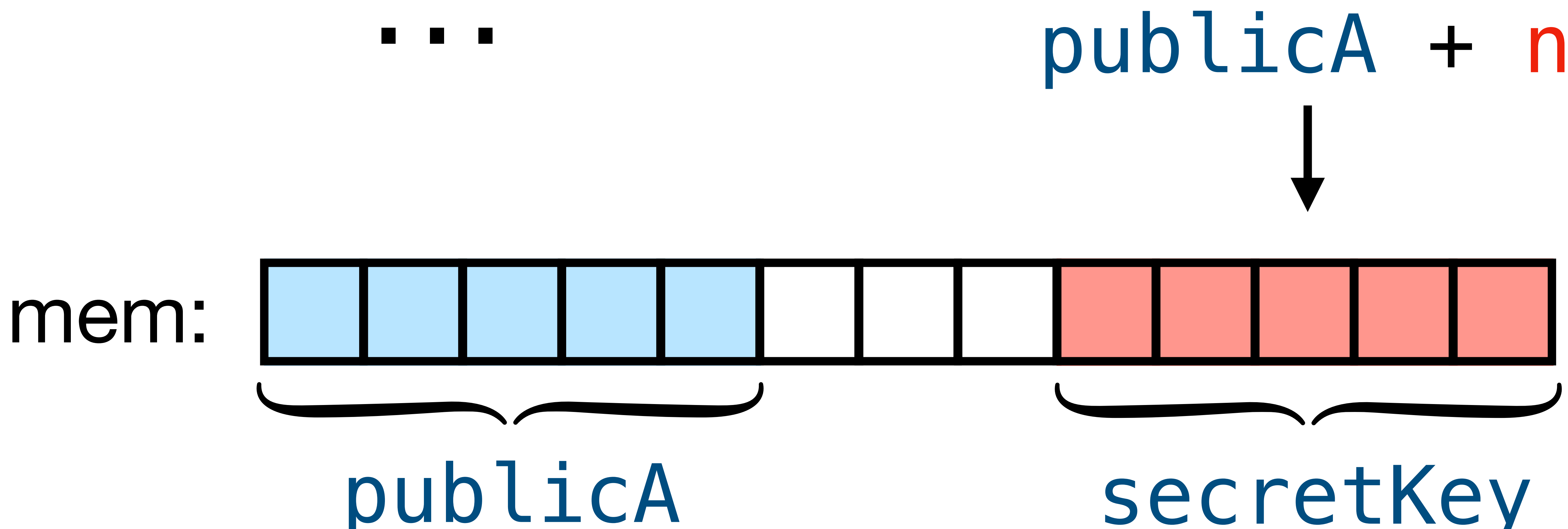
“Condition is true”



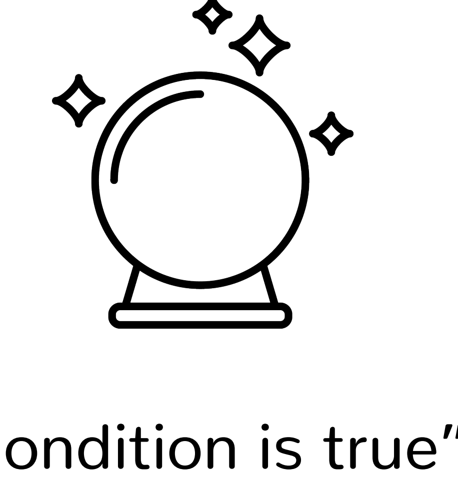
```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {

    ...
```



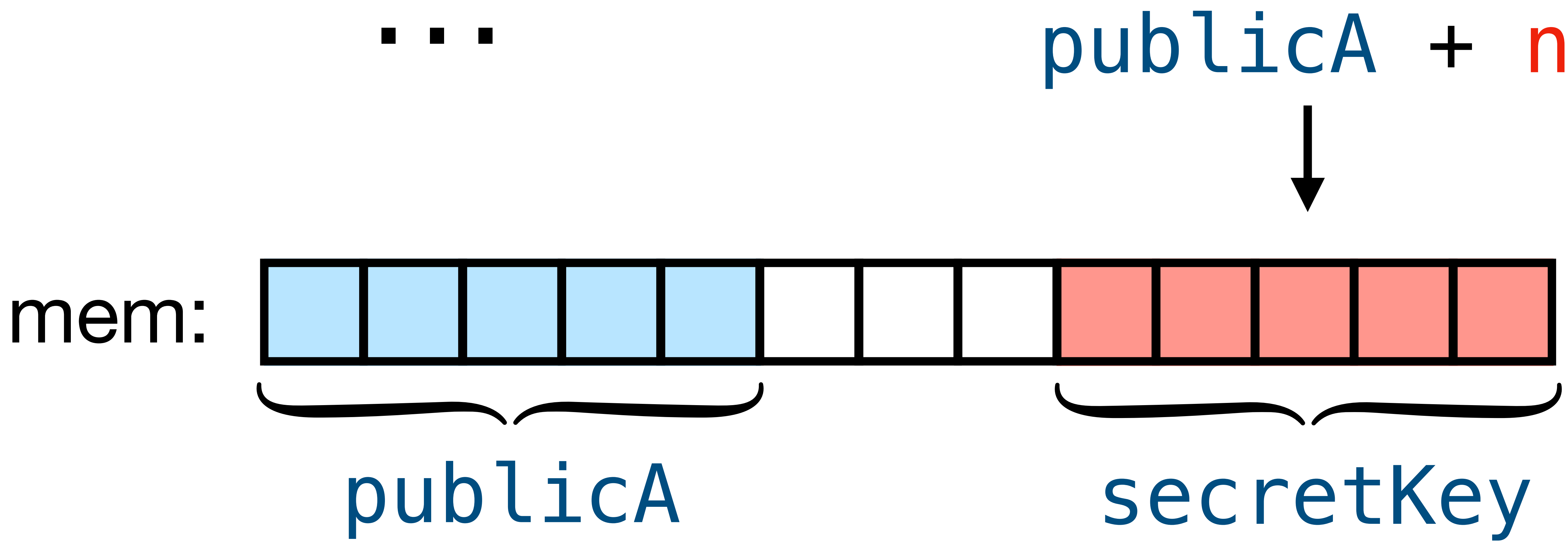
“Condition is true”



```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```

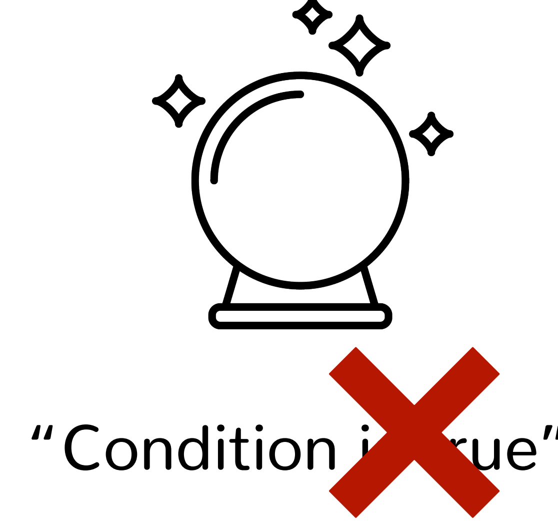


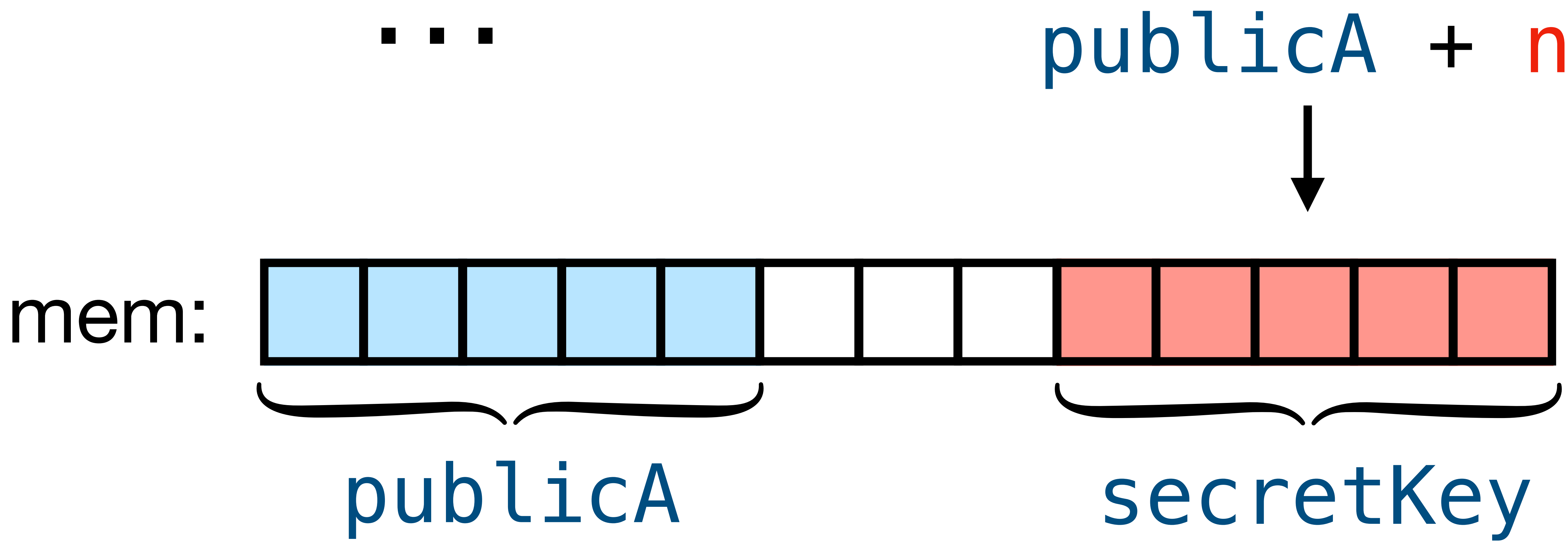
Secret memory access!



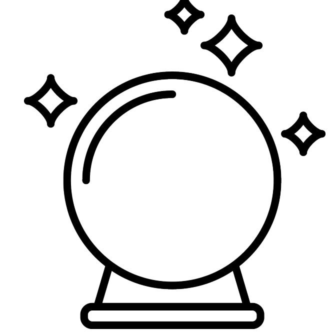
```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```

Secret memory access!





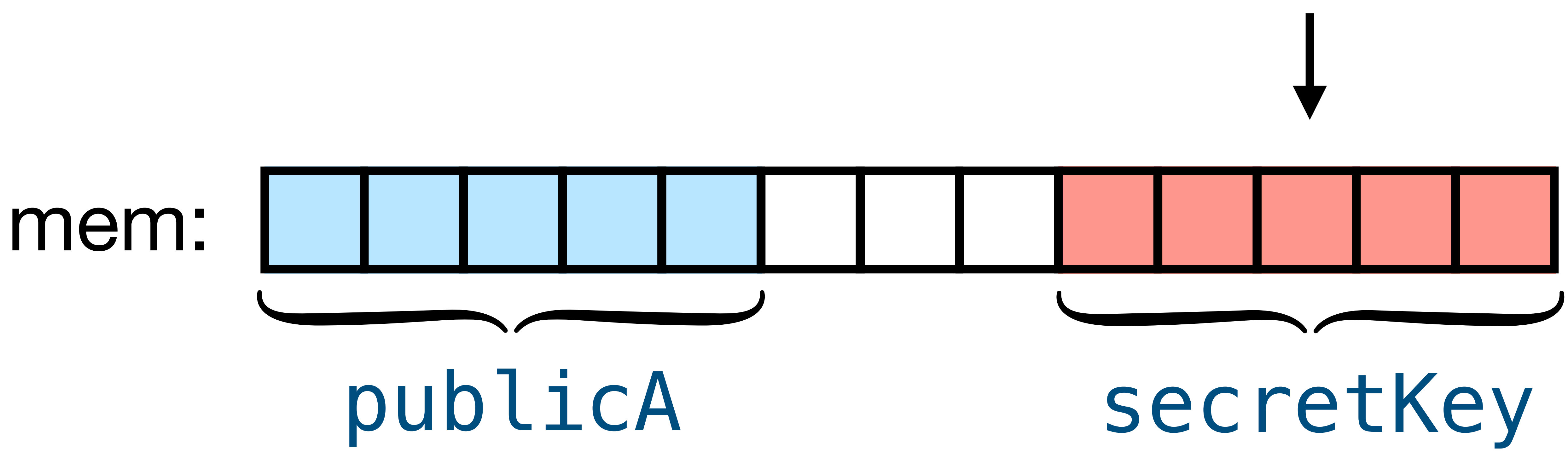
```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```



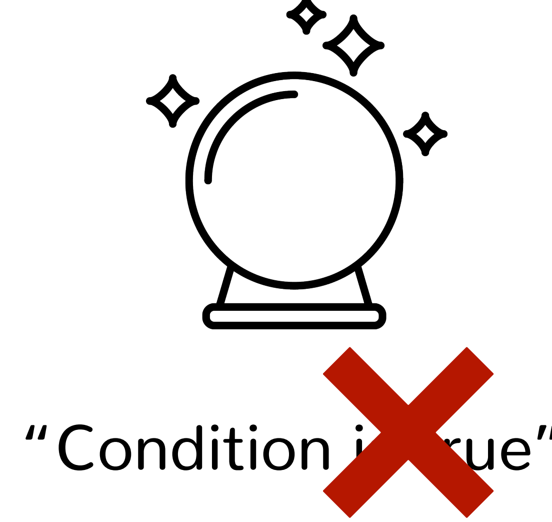
~~“Condition is true”~~

Secret memory access!

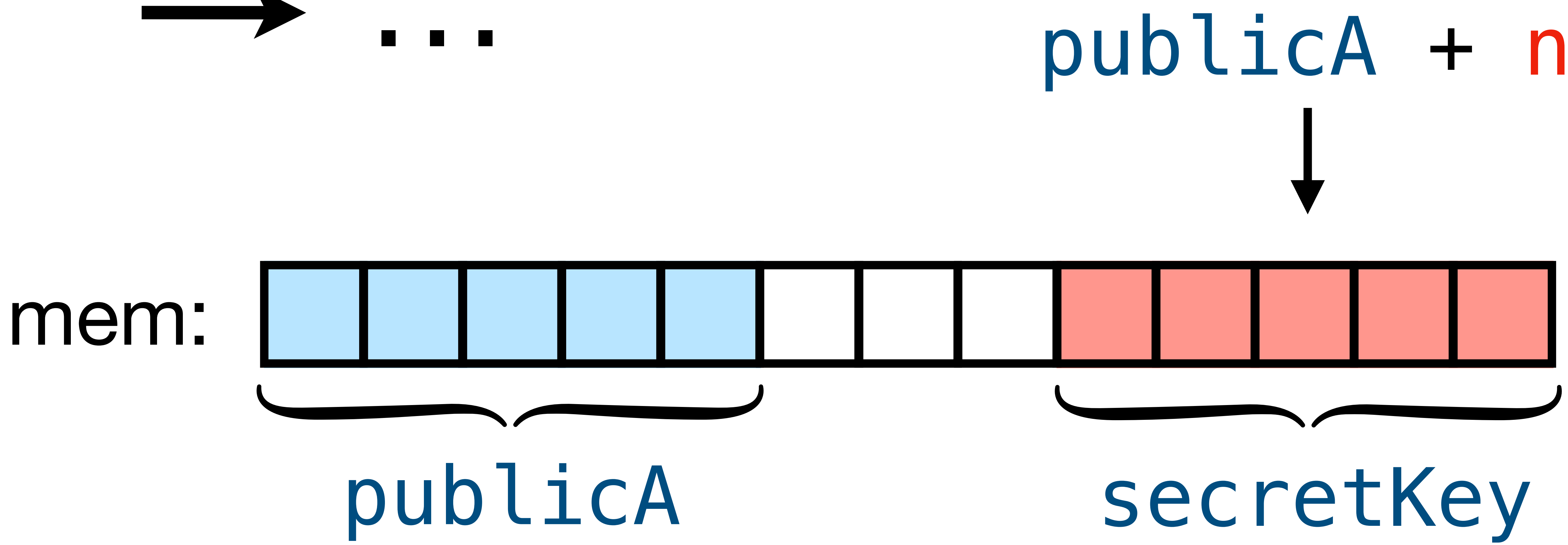
publicA + n



```
if (n < publicLen) {
    x = publicA[n];
    y = publicB[x];
} else {
    ...
```



Secret memory access!



How do you use this as attacker?

- Train the branch to predict true
- Execute branch w/ victim address
  - CPU will misspeculate and read secret data
  - Secret data not visible at the ISA level, visible in the cache
- Exfiltrate secret with cache attack

```
if (n < publicLen) {
  x = publicA[n];
  y = publicB[x];
} else {
  ...
```

Open research question:

How can we mitigate Spectre?

Another scary attack: Rowhammer

- Spectre attacks: read protected memory
- Rowhammer: write to protected memory
  - Fault injection attack

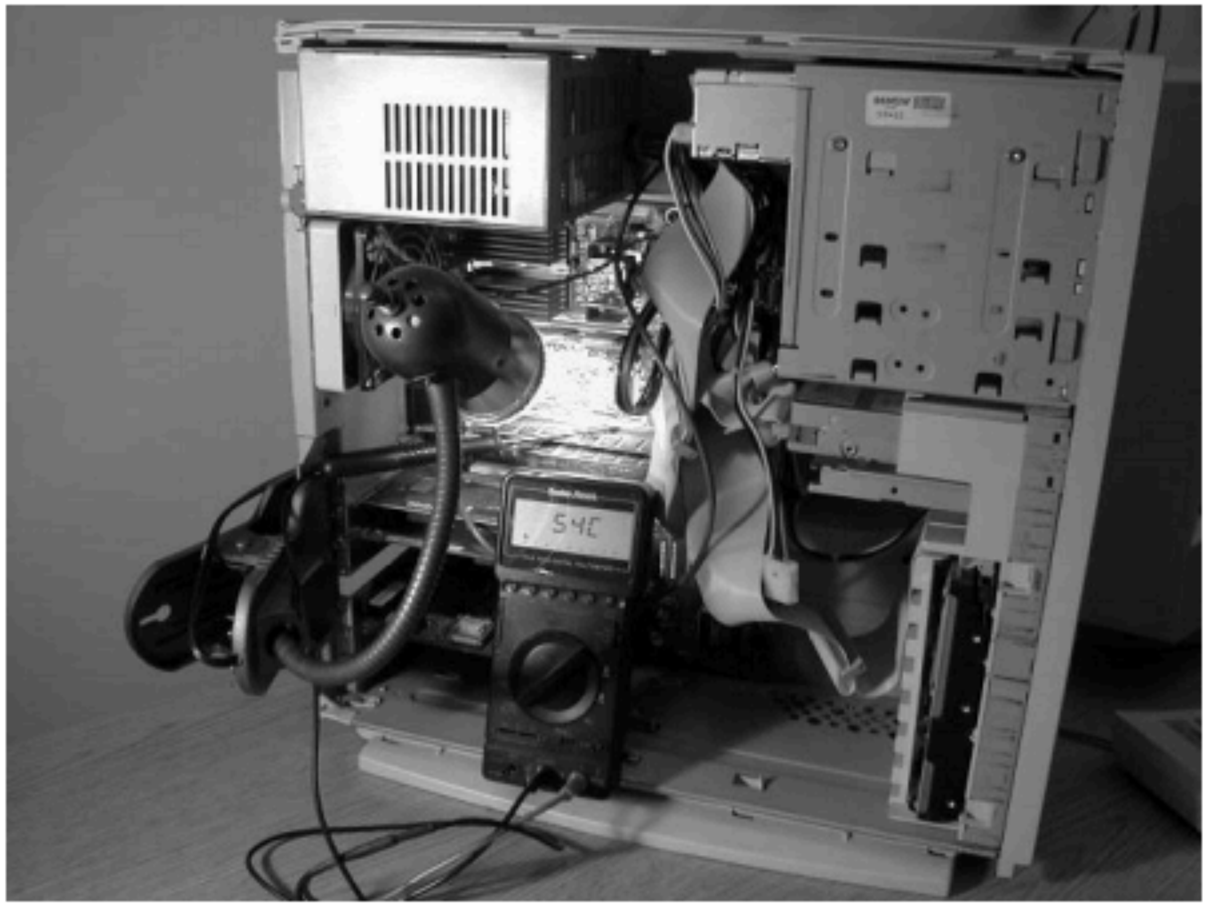


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

Today

- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks