



# 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?

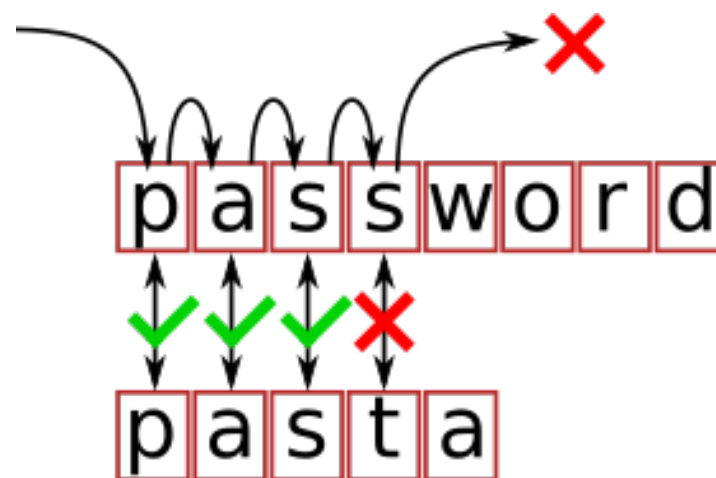
```
char pwd[] = "z2n34uzbnqhw4i";  
  
//...  
  
int check_password(char *buf) {  
    return strcmp(buf, pwd);  
}
```

# 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.

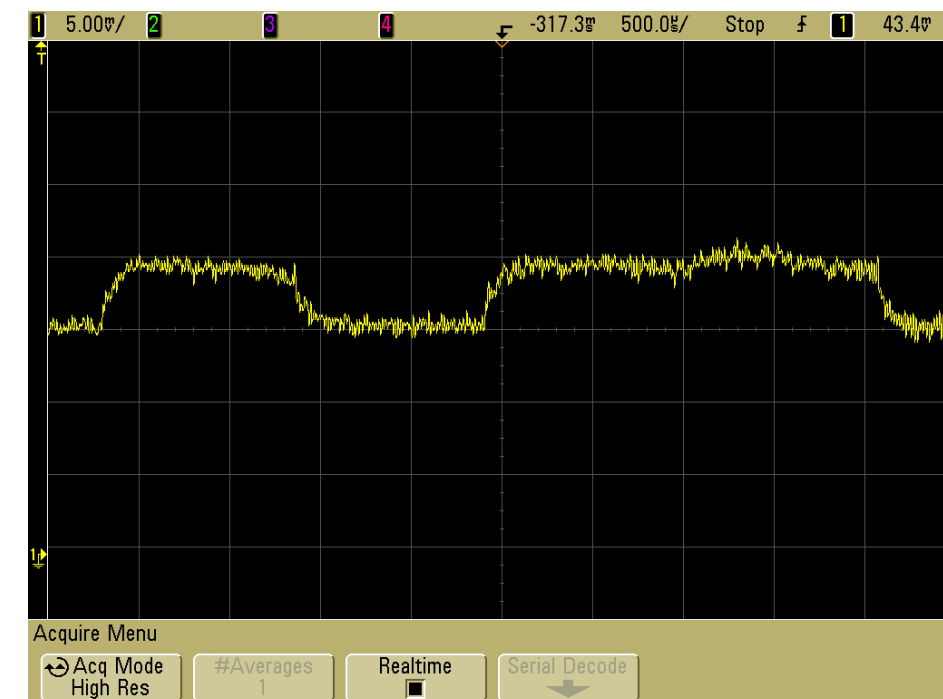
# Side channel examples

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



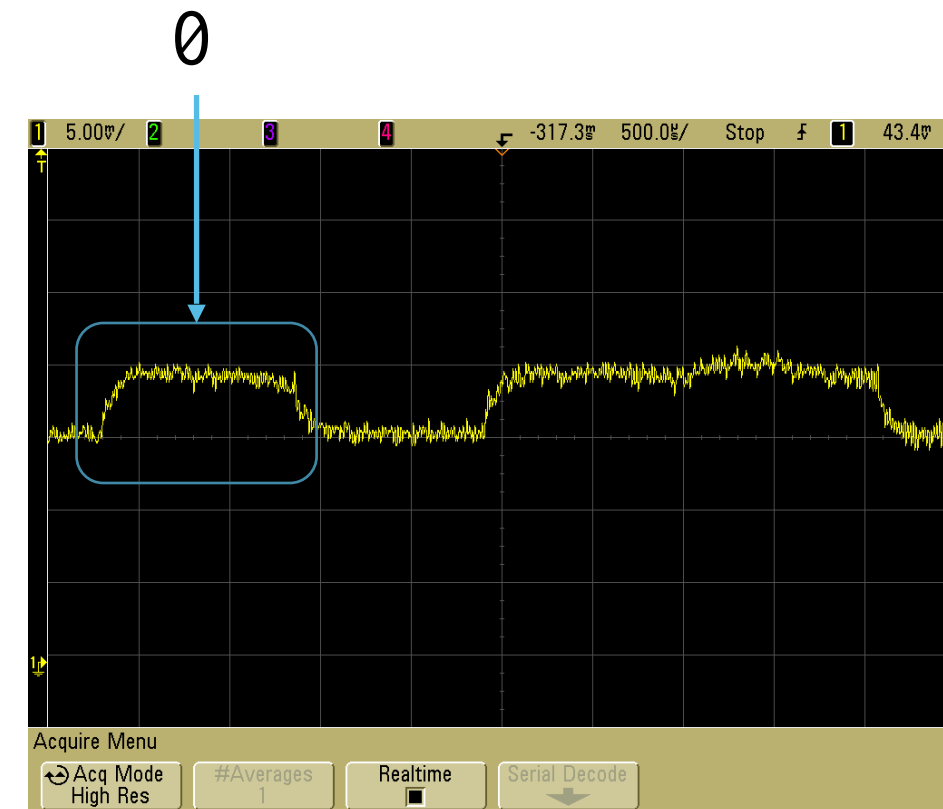
# Side channel examples

- 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.



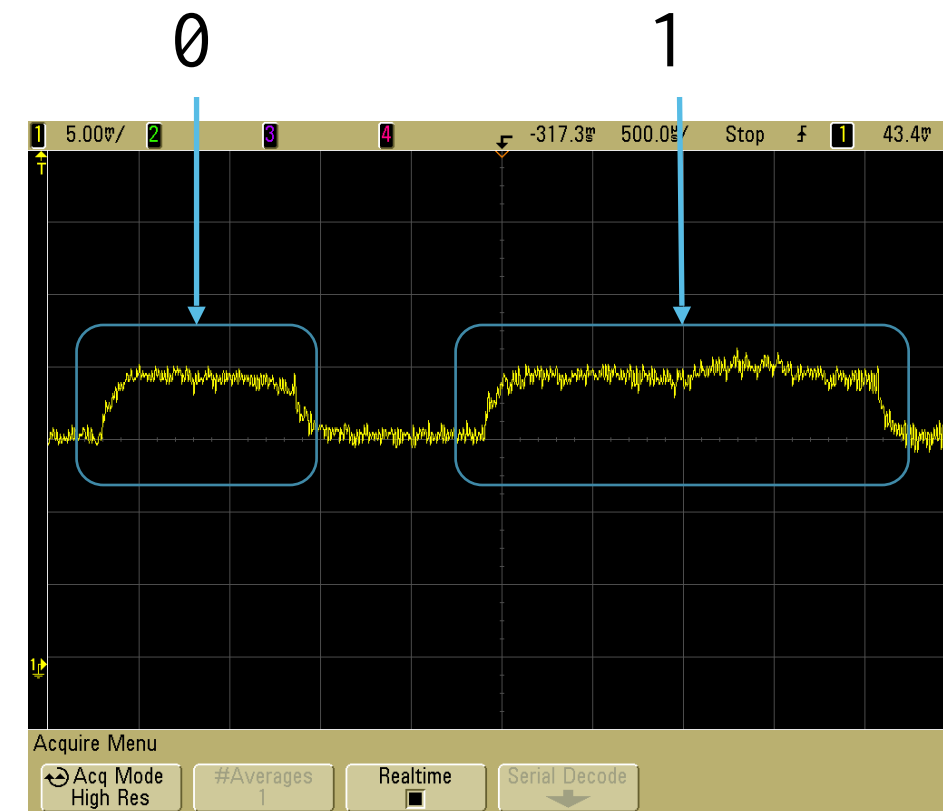
# Side channel examples

- 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.



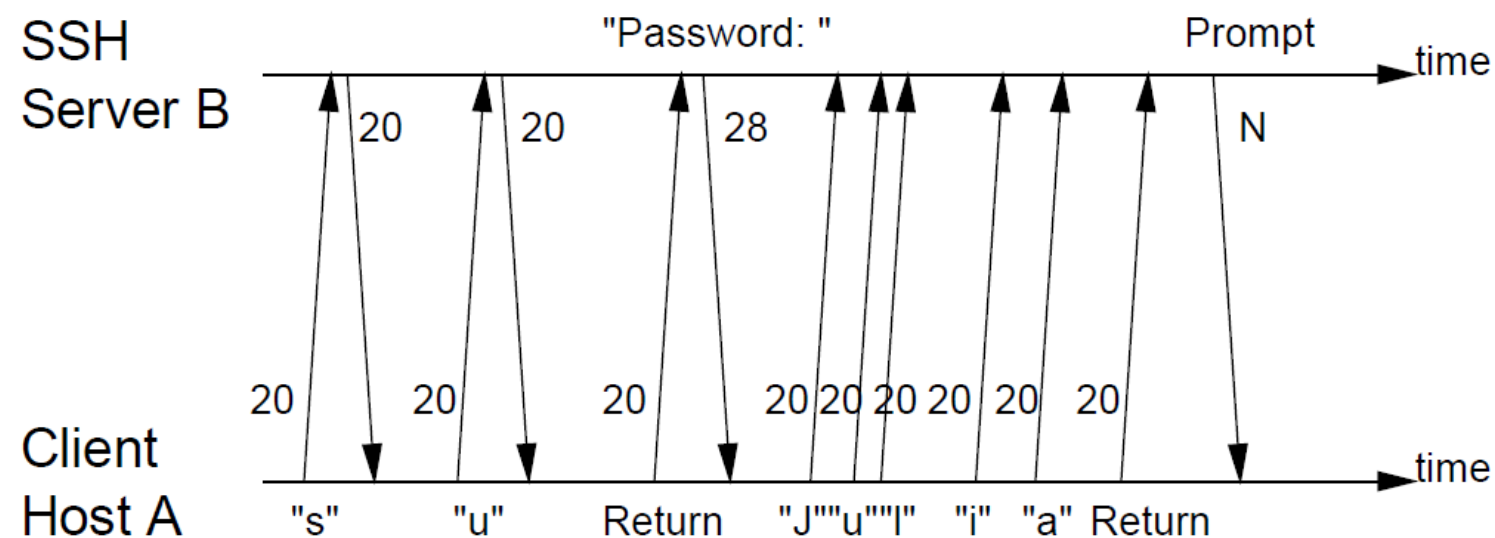
# Side channel examples

- 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.



# Side channel examples

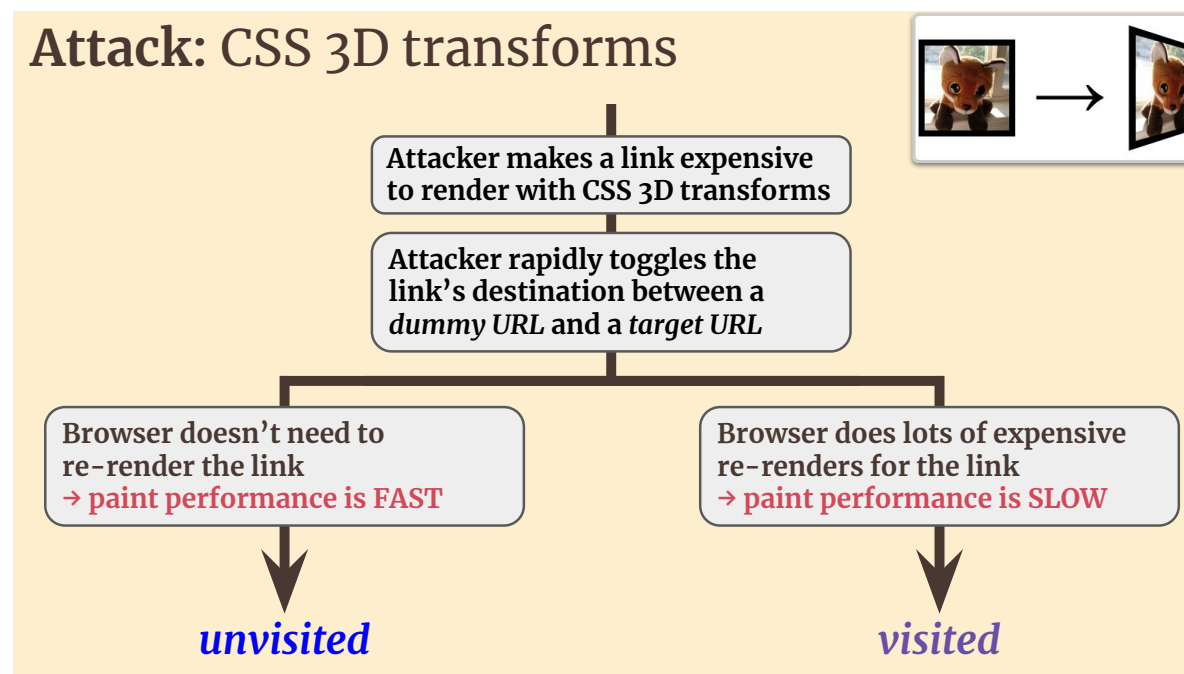
- Timing Analysis of Keystrokes and Timing Attacks on SSH
  - D. Song, D. Wagner, X. Tian, 2001
  - Recover characters typed over SSH by observing packet timing





# 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

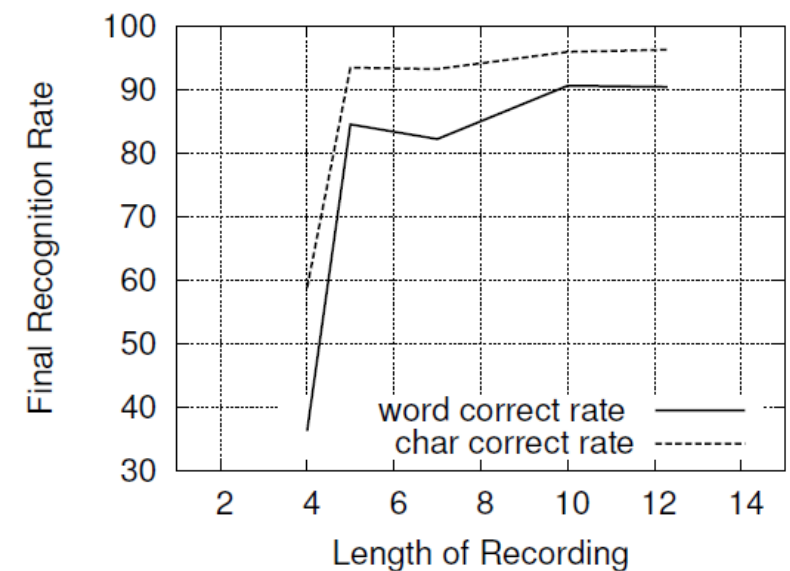


Fig. 7. Length of recording vs. recognition rate.

# 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

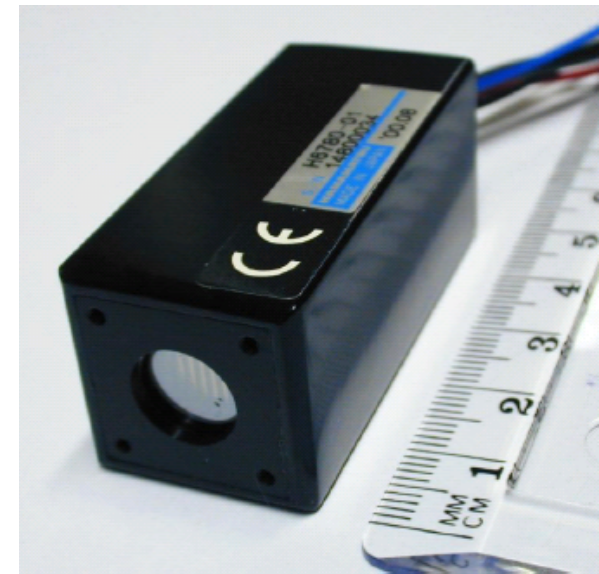
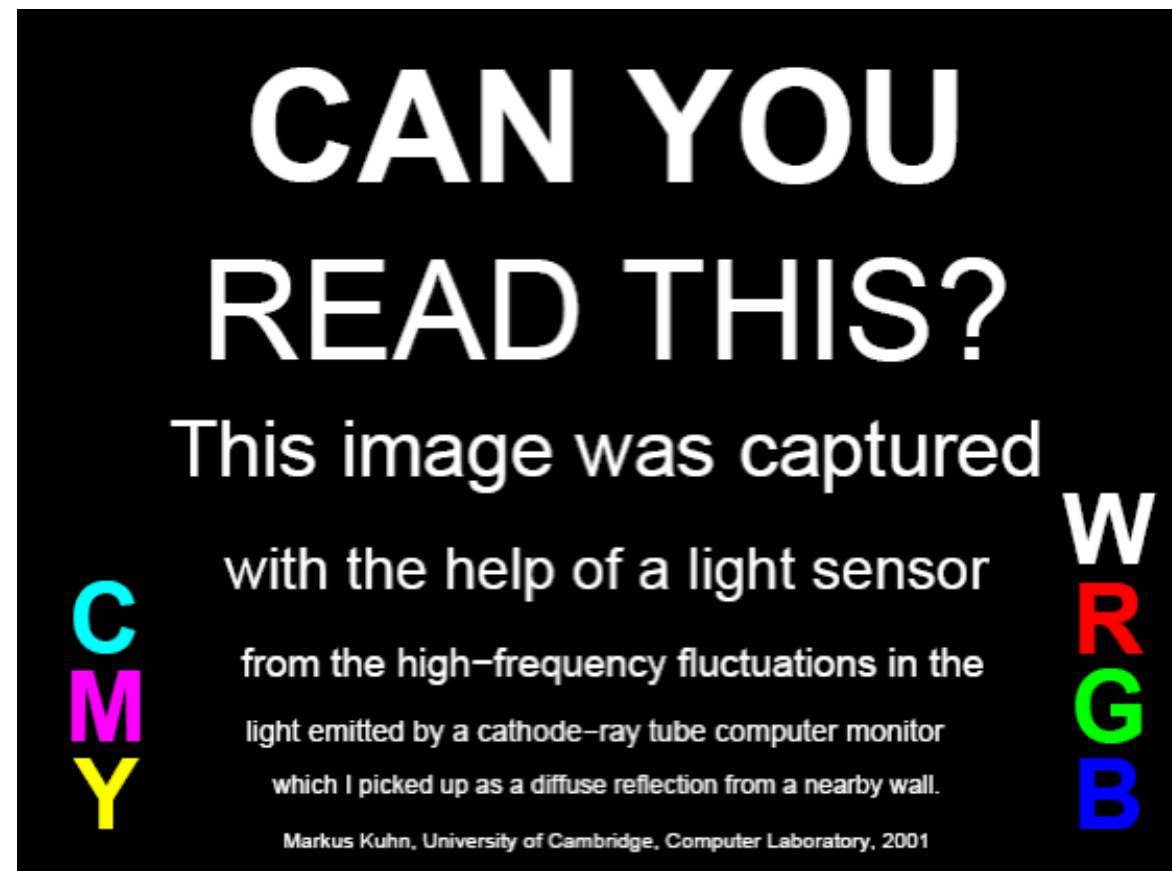


Figure 1. Photomultiplier tube module.

# Source signal



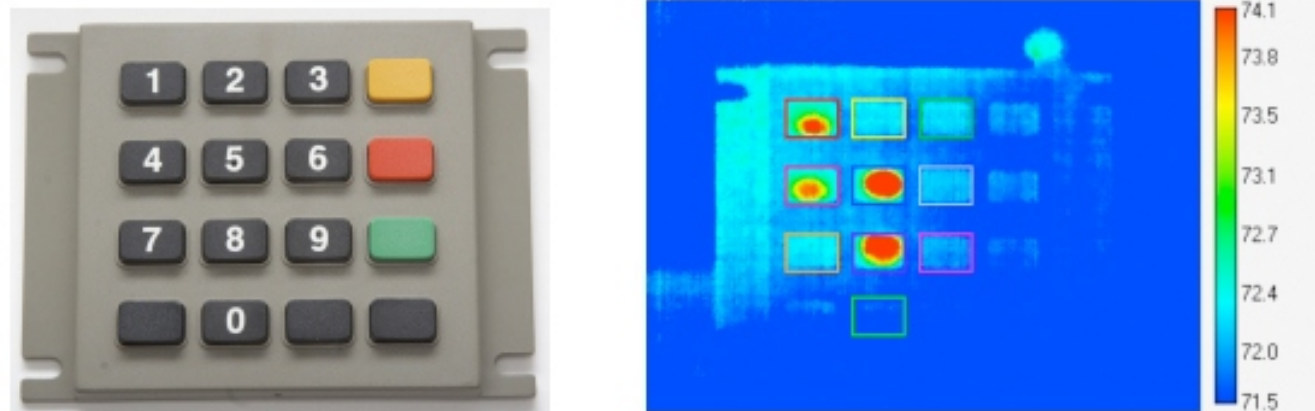


# Bounced off a wall



# 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

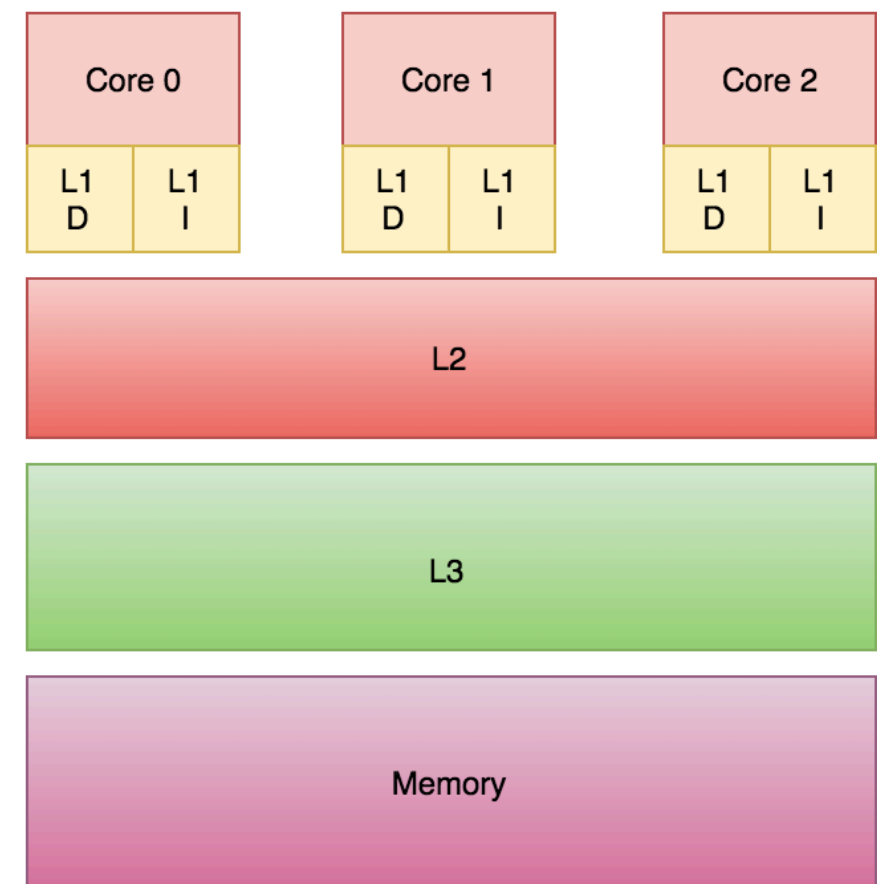
- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

# 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

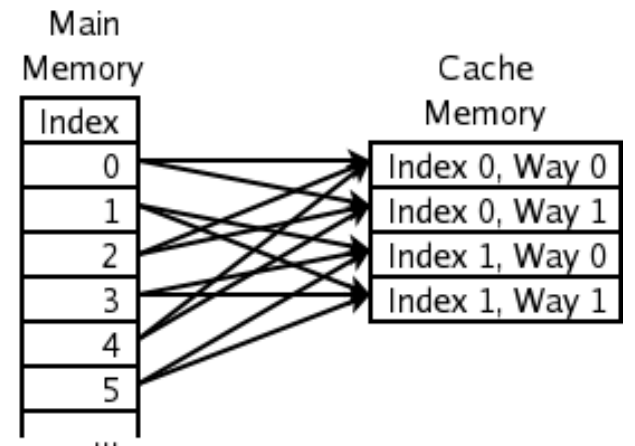
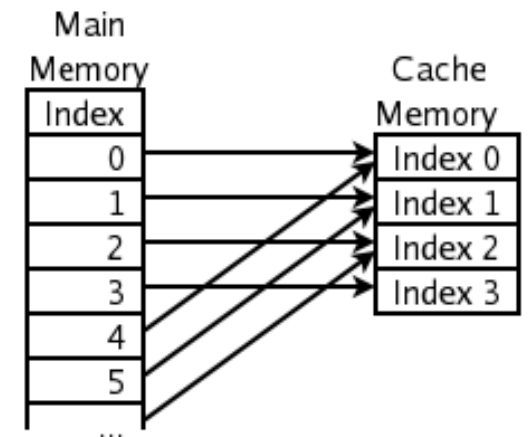
# Cache hierarchy

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



# 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!

# 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 (cf1ush on x86)
- **Measure**: precisely (down to the cycle) how long it takes to do something (rdtsc on x86)
- Attack form: 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 addresses
  - 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 addresses

# 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 addresses

# 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 < 1000000000; i++) {  
        z = y*x;  
    }  
}
```

A: `foo(1.0);`

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

C: They take the same amount of time!

# Which runs faster?

```
void foo(double x) {  
    double z, y = 1.0;  
    for (uint32_t i = 0; i < 1000000000; i++) {  
        z = y*x;  
    }  
}
```

A: foo(1.0); ←

B: foo(1.0e-323);

C: They take the same amount of time!



# 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.

DIT	Meaning
0b0	The architecture makes no statement about the timing properties of any instructions.
0b1	<p>The architecture requires that:</p> <ul style="list-style-type: none"><li>• The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.</li><li>• For certain data processing instructions, the instruction takes a time which is independent of:<ul style="list-style-type: none"><li>◦ The values of the data supplied in any of its registers.</li><li>◦ The values of the NZCV flags.</li></ul></li><li>• For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:<ul style="list-style-type: none"><li>◦ The values of the data supplied in any of its registers.</li><li>◦ The values of the NZCV flags.</li></ul></li></ul>

The data processing instructions affected by this bit are:

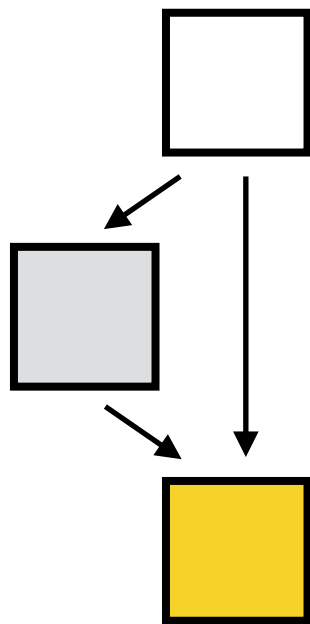
- All cryptographic instructions. These instructions are:
  - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT 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...

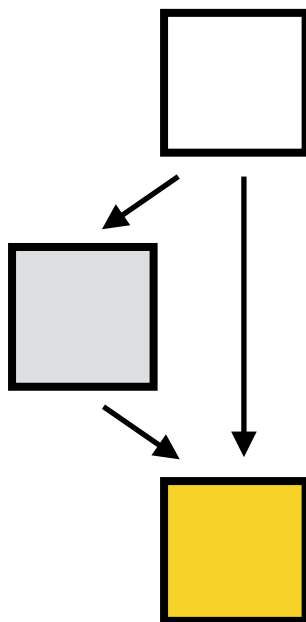
# Fold control flow into data flow

(assumption secret = 1 or 0)

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



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



# Fold control flow into data flow

(assumption  $\text{secret} = 1 \text{ or } 0$ )

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



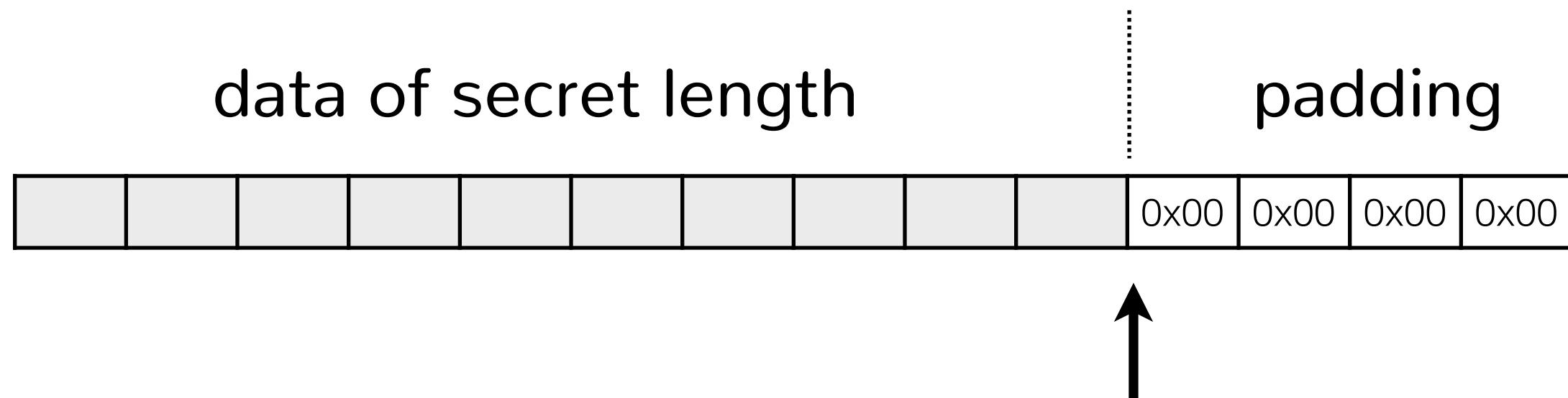
```
x = secret * a  
  + (1-secret) * x;  
  
x = (1-secret) * b  
  + secret * x;
```

# 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

```
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

```
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

## OpenSSL padding oracle attack

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

```
383 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 + int decryption_failed_or_bad_record_mac = 0;
386 387
387 388
388 389     rr = &(s->s3->rrec);
389 389
390 390     /* -417,13 +418,10 @@ dtls1_process_record(SSL *s)
417 418     enc_err = s->method->ssl3_enc->enc(s,0);
418 419     if (enc_err <= 0)
419 420     {
420 420         /* decryption failed, silently discard message */
421 421         if (enc_err < 0)
422 422         {
423 423             rr->length = 0;
424 424             s->packet_length = 0;
425 425         }
426 426         goto err;
427 427
428 428         /* To minimize information leaked via timing, we will always
429 429         * perform all computations before discarding the message.
430 430         */
431 431         decryption_failed_or_bad_record_mac = 1;
432 432     }
433 433
434 434     #ifdef TLS_DEBUG
435 435     /* -453,7 +451,7 @@ printf("\n");
453 451     SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_PRE_MAC_LENGTH_TOO_LONG);
454 452     goto f_err;
455 453
456 453     #else
457 453     goto err;
458 453
459 453     decryption_failed_or_bad_record_mac = 1;
460 453     #endif
461 453
462 453     /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
463 453     /* -464,17 +462,25 @@ printf("\n");
464 462     SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
465 463     goto f_err;
466 463
467 463     #else
468 463     goto err;
469 463
470 463     decryption_failed_or_bad_record_mac = 1;
471 463     #endif
472 463
473 463     rr->length -= mac_size;
474 463     i = s->method->ssl3_enc->mac(s,md,0);
475 463     if (1 < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
476 463     {
477 463         goto err;
478 463         decryption_failed_or_bad_record_mac = 1;
479 463     }
480 463
481 463     }
482 463
483 463     if (decryption_failed_or_bad_record_mac)
484 463     {
485 463         /* decryption failed, silently discard message */
486 463         rr->length = 0;
487 463         s->packet_length = 0;
488 463         goto err;
489 463     }
490 463
491 463     /* r->length is now just compressed */
492 463     if (s->expand != NULL)
493 463     {
```

# Aside: Writing CT code is unholy

## OpenSSL padding oracle attack

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

```
383 383 SSL3_RECORD *rr;
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 386 + int decryption_failed_or_bad_record_mac = 0;
387 387
388 388
389 389
390 390
391 391
392 392
393 393
394 394
395 395
396 396
397 397
398 398
399 399
400 400
401 401
402 402
403 403
404 404
405 405
406 406
407 407
408 408
409 409
410 410
411 411
412 412
413 413
414 414
415 415
416 416
417 417
418 418
419 419
420 420
421 421
422 422
423 423
424 424
425 425
426 426
427 427
428 428
429 429
430 430
431 431
432 432
433 433
434 434
435 435
436 436
437 437
438 438
439 439
440 440
441 441
442 442
443 443
444 444
445 445
446 446
447 447
448 448
449 449
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000
```

# 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.

384	384	unsigned int mac_size;
385	385	unsigned char md[EVP_MAX_MD_SIZE];
386	386	+ int decryption_failed_or_bad_record_mac = 0;
387	387	EVP_DigestUpdate(&md_ctx, md, 2);
388	388	EVP_DigestUpdate(&md_ctx, rec->input, rec->length);
389	389	EVP_DigestFinal_ex(&md_ctx, md, NULL);
417	418	EVP_MD_CTX_copy_ex(&md_ctx, hash);
418	419	EVP_DigestUpdate(&md_ctx, mac_sec, md_size);
419	420	EVP_DigestUpdate(&md_ctx, ssl3_pad_2, npad);
420	421	EVP_DigestUpdate(&md_ctx, md, md_size);
421	422	EVP_DigestFinal_ex(&md_ctx, md, &md_size);
422	423	EVP_MD_CTX_cleanup(&md_ctx);
423	424	if (!send &&
424	425	EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPHER_MODE_CBC &&
425	426	ssl3_cbc_record_digest_supported(hash))
426	427	{
427	428	/* This is a CBC-encrypted record. We must avoid leaking any
428	429	* timing-side channel information about how many blocks of
429	430	* data we are hashing because that gives an attacker a
430	431	* timing-oracle. */
431	432	/* npad is, at most, 48 bytes and that's with MD5:
432	433	* 16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
433	434	* With SHA-1 (the largest hash speed for SSLv3) the hash size
434	435	* goes up 4, but npad goes down by 8, resulting in a smaller
435	436	* total size. */
436	437	unsigned char header[75];
437	438	unsigned j = 0;
438	439	memcpy(header+j, mac_sec, md_size);
439	440	j += md_size;
440	441	memcpy(header+j, ssl3_pad_1, npad);
441	442	j += npad;
442	443	memcpy(header+j, seq, 8);
443	444	j += 8;
444	445	header[j++] = rec->type;
445	446	header[j++] = rec->length >> 8;
446	447	header[j++] = rec->length & 0xff;
447	448	ssl3_cbc_digest_record{
448	449	hash,
449	450	md, &md_size,
450	451	header, rec->input,
451	452	rec->length + md_size, rec->orig_len,
452	453	mac_sec, md_size,
453	454	1 /* is SSLv3 */};
454	455	}
455	456	else
456	457	{
457	458	unsigned int md_size_u;
458	459	/* Chop the digest off the end :-) */
459	460	EVP_MD_CTX_init(&md_ctx);
460	461	EVP_MD_CTX_copy_ex(&md_ctx, hash);
461	462	EVP_DigestUpdate(&md_ctx, mac_sec, md_size);
462	463	EVP_DigestUpdate(&md_ctx, ssl3_pad_1, npad);
463	464	EVP_DigestUpdate(&md_ctx, seq, 8);
464	465	rec_char=rec->type;
465	466	EVP_DigestUpdate(&md_ctx, &rec_char, 1);
466	467	p=md;
467	468	s2n(rec->length, p);
468	469	EVP_DigestUpdate(&md_ctx, md, 2);
469	470	EVP_DigestUpdate(&md_ctx, rec->input, rec->length);

# 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.

```
383 383 SSL3_RECORD *rr;
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 386 int decryption_failed_or_bad_record_mac = 0;
387 387
388 388
389 389
390 390
391 391
392 392
393 393
394 394
395 395
396 396
397 397
398 398
399 399
400 400
401 401
402 402
403 403
404 404
405 405
406 406
407 407
408 408
409 409
410 410
411 411
412 412
413 413
414 414
415 415
416 416
417 417
418 418
419 419
420 420
421 421
422 422
423 423
424 424
425 425
426 426
427 427
428 428
429 429
430 430
431 431
432 432
433 433
434 434
435 435
436 436
437 437
438 438
439 439
440 440
441 441
442 442
443 443
444 444
445 445
446 446
447 447
448 448
449 449
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000
```

# 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.

```
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 386 int decryption_failed_or_bad_record_mac = 0;

755 - EVP_DigestUpdate(&md_ctx,md,2);
756 - EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
757 - EVP_DigestFinal_ex(&md_ctx,md,NULL);
758 -

417 418
418 418
419 419
420 420
421 421
422 422
423 423
424 424
425 425
426 426

421 421
422 422
423 423
424 424

427 425
428 426
429 427

453 451
454 452
455 453
456 454
457 455
458 456
459 457

464 462
465 463
466 464

583 584 maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
584 585 maxpad &= 255;
585 586
587 + ret &= constant_time_ge(maxpad, pad);
588 +
586 589 inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
587 590 mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
588 591 inp_len &= mask;

279 + mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
280 + inp_len &= mask;
281 + ret &= (int)mask;

258 282
259 - /* calculate HMAC and verify it */
283 + key->aux.tls_aad[plen-2] = inp_len>>8;
284 + key->aux.tls_aad[plen-1] = inp_len;
285 +
286 + /* calculate HMAC */
287 key->md = key->head;
288 SHA1_Update(&key->md, key->aux.tls_aad, plen);
289 SHA1_Update(&key->md, out+iv, len);
290 SHA1_Final(mac, &key->md);

290 + #if 1
291 + len -= SHA_DIGEST_LENGTH; /* amend mac */
292 + if (len >= (256+SHA_CBLOCK)) {
293 + j = (len - (256+SHA_CBLOCK)) & (0-SHA_CBLOCK);
294 + j += SHA_CBLOCK - key->md.num;
295 +
```



# 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."

```
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 386 int decryption_failed_or_bad_record_mac = 0;

755 - EVP_DigestUpdate(&md_ctx,md,2);
756 - EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
757 - EVP_DigestFinal_ex(&md_ctx,md,NULL);

237 - unsigned char mac[SHA_DIGEST_LENGTH];
246 + union { unsigned int u[SHA_DIGEST_LENGTH/sizeof(unsigned int)];
247 + unsigned char c[SHA_DIGEST_LENGTH]; } mac;

238 248 /* decrypt HMAC|padding at once */
239 249 aesni_cbc_encrypt(in,out,len,
240 250 &key->ks,ctx->iv,0);
241 251
242 252 if (plen) { /* "TLS" mode of operation */
243 253 /* figure out payload length */
244 254 if (len<(size_t)(out[len-1]+1+SHA_DIGEST_LENGTH))
245 255 return 0;
246 256
247 257 len -= (out[len-1]+1+SHA_DIGEST_LENGTH);
248 258 size_t inp_len, mask, j, i;
249 259 unsigned int res, maxpad, pad, bitlen;
250 260 int ret = 1;
251 261 union { unsigned int u[SHA_LBLOCK];
252 262 unsigned char c[SHA_CBLOCK]; }
253 263 *data = (void *)key->md.data;

249 260 if ((key->aux.tls_aad[plen-4]<8|key->aux.tls_aad[plen-3])
250 261 >= TLS1_1_VERSION) {
251 262 len -= AES_BLOCK_SIZE;
252 263 >= TLS1_1_VERSION
253 264 iv = AES_BLOCK_SIZE;
254 265 }
255 266
256 267 key->aux.tls_aad[plen-2] = len>>8;
257 268 key->aux.tls_aad[plen-1] = len;

583 584 maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
584 585 maxpad &= 255;

587 + ret &= constant_time_ge(maxpad, pad);
588 +

586 589 inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
587 590 mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
588 591 inp_len &= mask;

279 + mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
280 + inp_len &= mask;
281 + ret &= (int)mask;

258 262 /* calculate HMAC and verify it */
263 + key->aux.tls_aad[plen-2] = inp_len>>8;
264 + key->aux.tls_aad[plen-1] = inp_len;
265 +
266 + /* calculate HMAC */
267 key->md = key->head;
268 SHA1_Update(&key->md,key->aux.tls_aad,plen);
269 SHA1_Update(&key->md,out+iv,len);
270 SHA1_Final(mac,&key->md);

290 +if 1
291 + len -= SHA_DIGEST_LENGTH; /* amend mac */
292 + if (len>=(256+SHA_CBLOCK)) {
293 + j = (len-(256+SHA_CBLOCK))&(0-SHA_CBLOCK);
294 + j += SHA_CBLOCK-key->md.num;
295 +
```



# 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

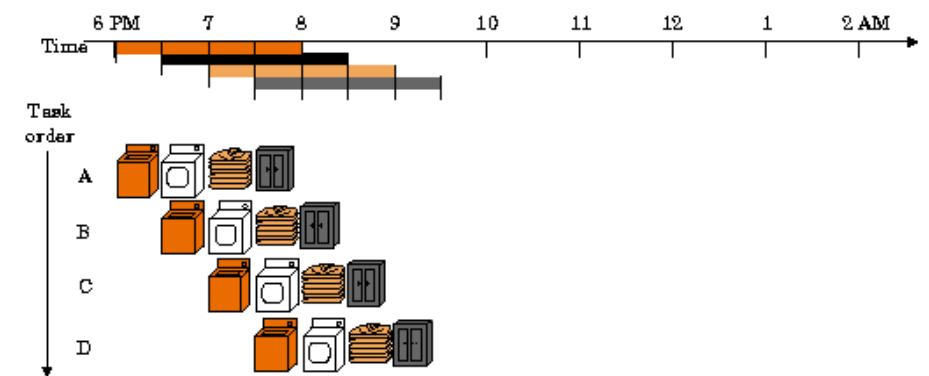
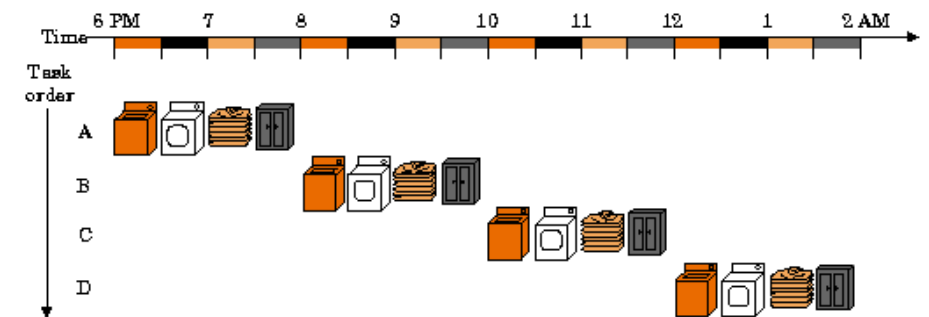
- Overview of side channels in general
- Cache side channels
- Constant-time programming
- Spectre attacks

# Quick review: ISA and $\mu$ Architecture

- Instruction set architecture
  - Defined interface between HW and SW
- $\mu$ Architecture
  - Implementation of the ISA
  - “Behind the curtain” details
    - E.g. cache specifics
- **Key issue:**  $\mu$ Architectural details can sometimes become “architecturally visible”

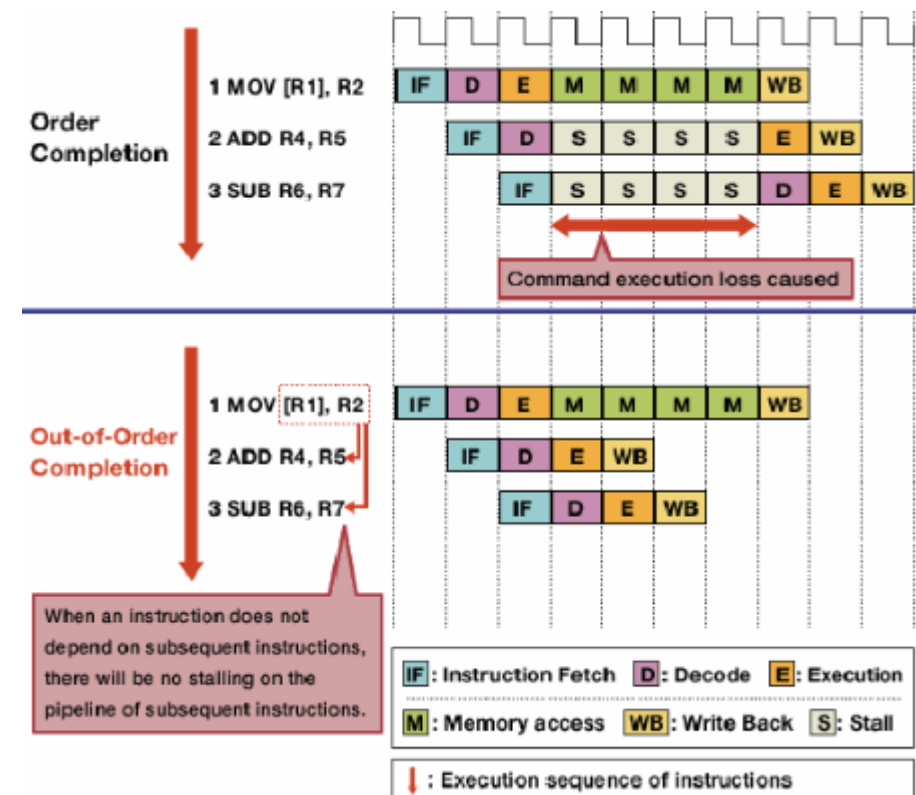
# Review: Instruction pipelining

- 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



# Review: Out-of-order execution

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

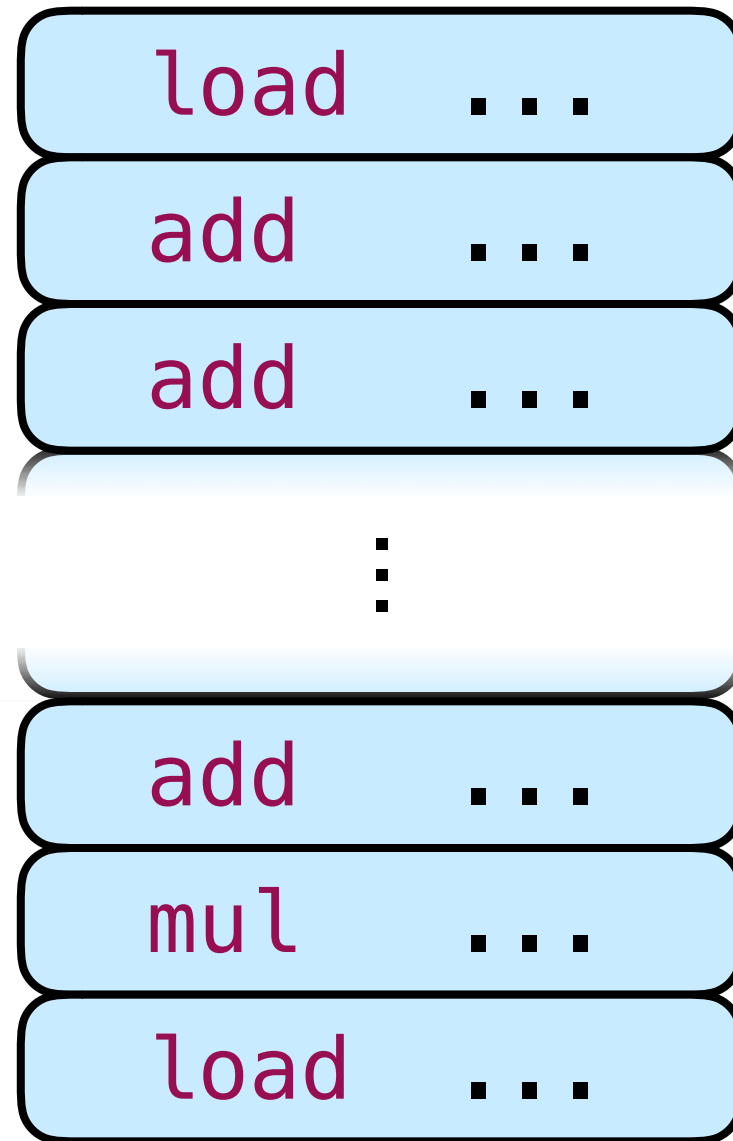


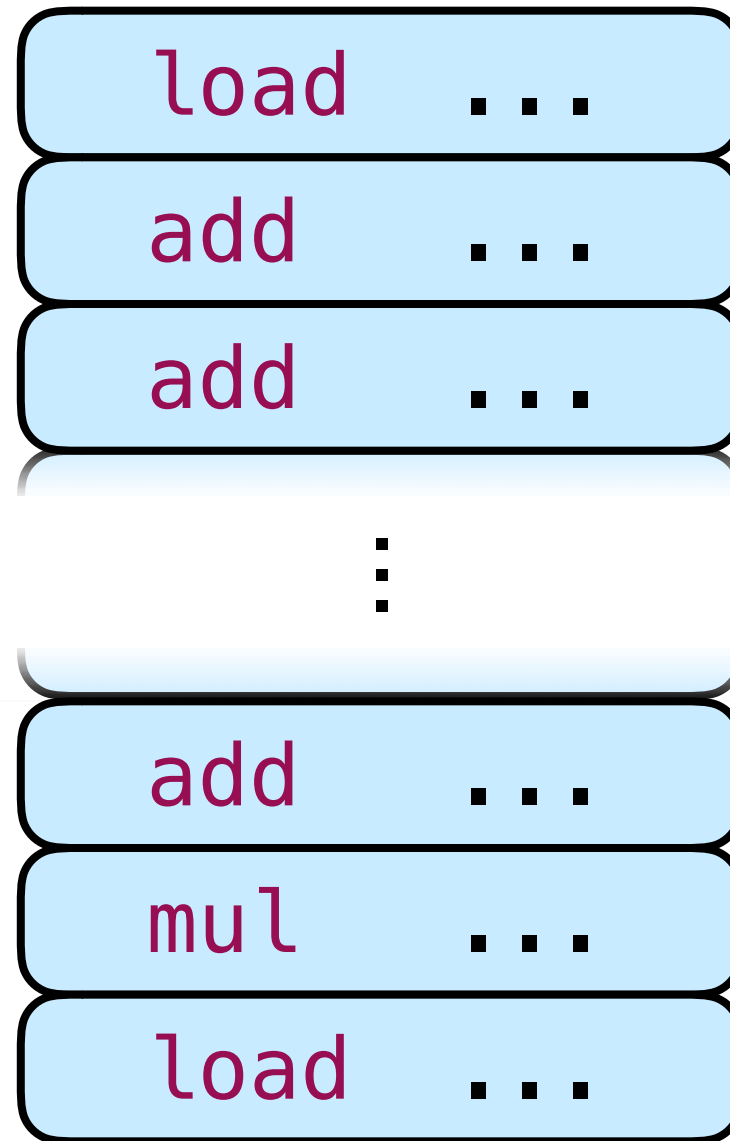
# Review: Speculative execution

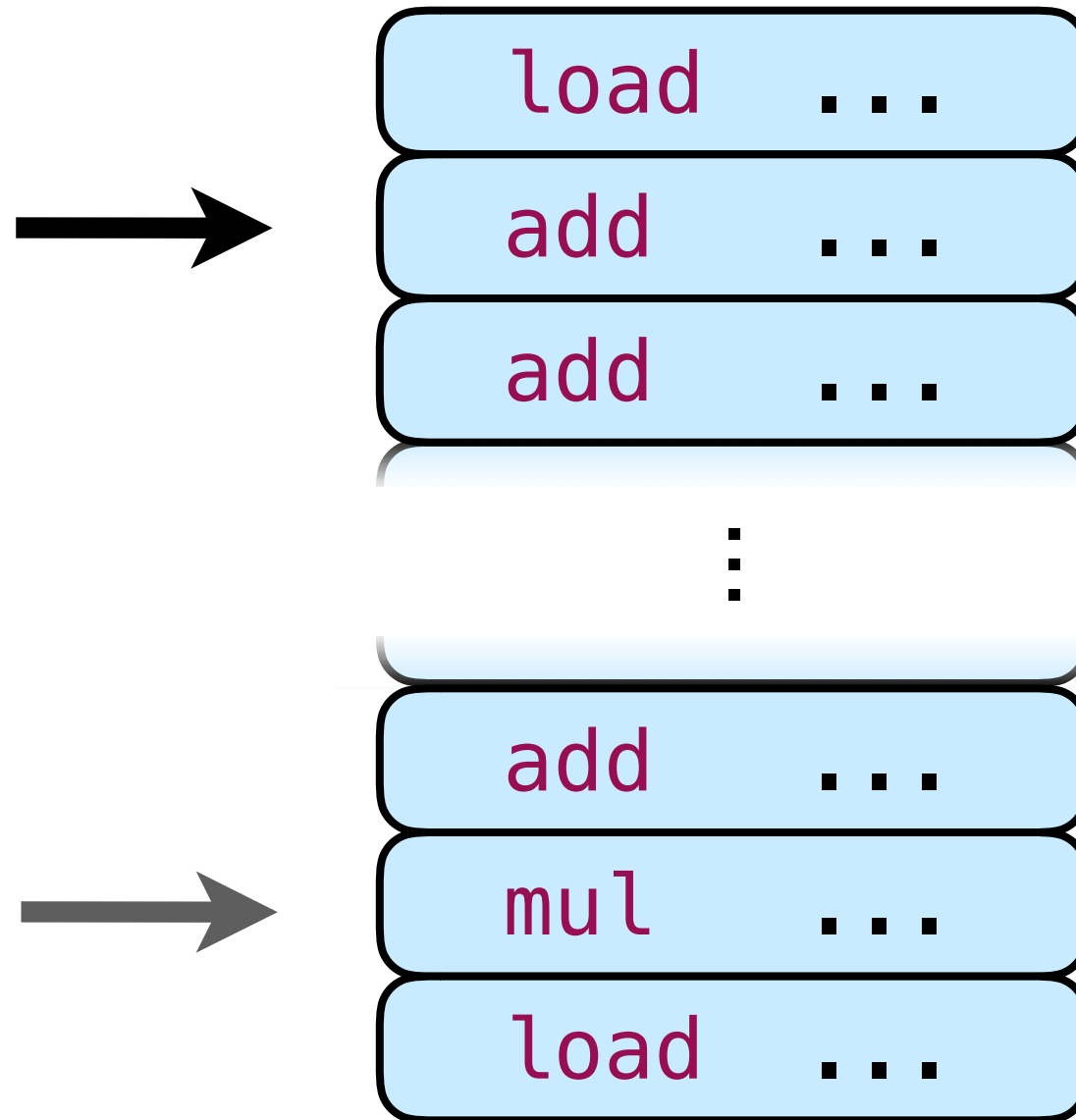
- Control flow could depend on output of earlier instruction
  - E.g. conditional branch, function pointer
- Rather than wait to know which way to go, the processor may “speculate” about the direction/target of a branch
  - Guess based on the past
  - If the guess is correct, performance is improved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

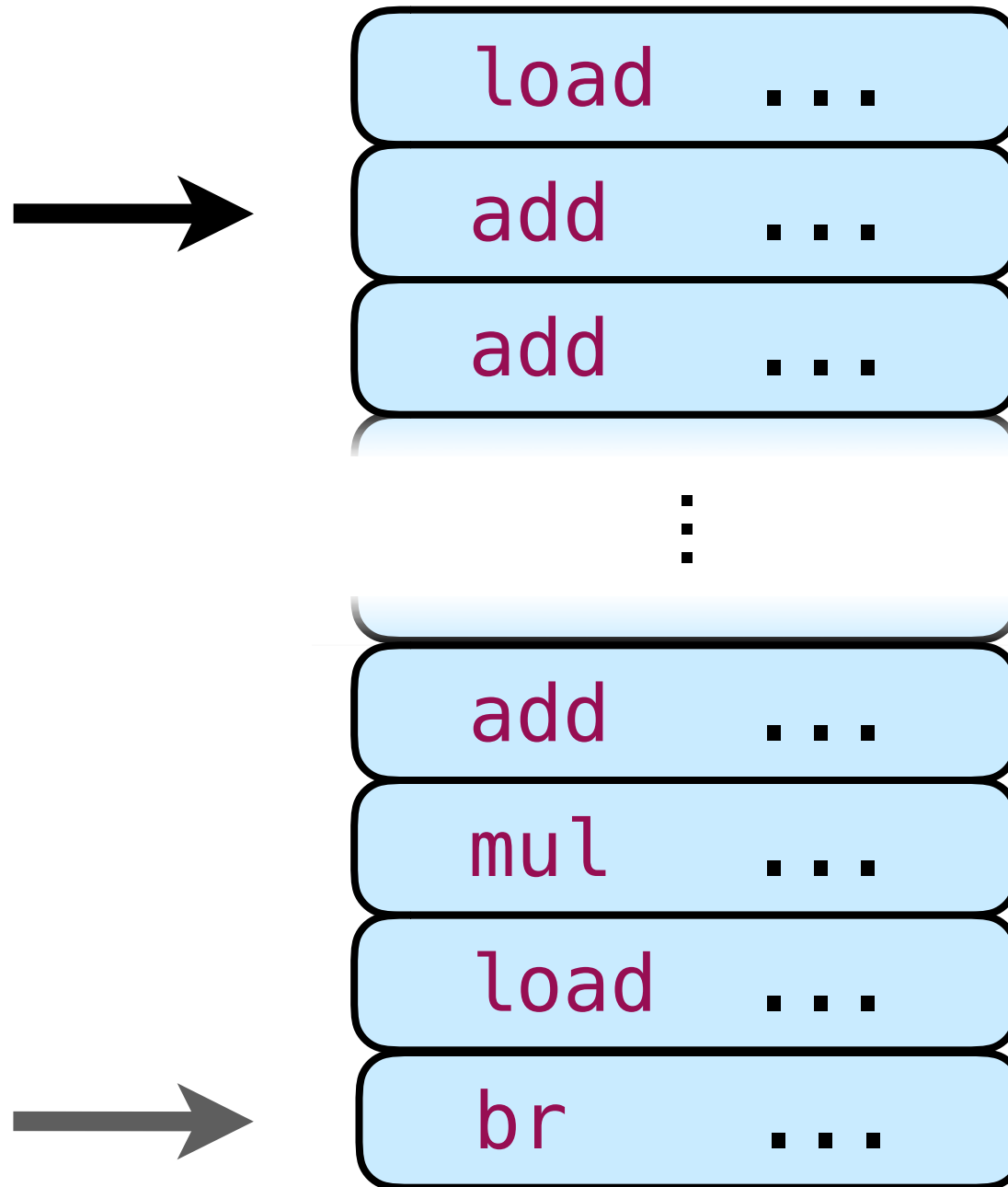


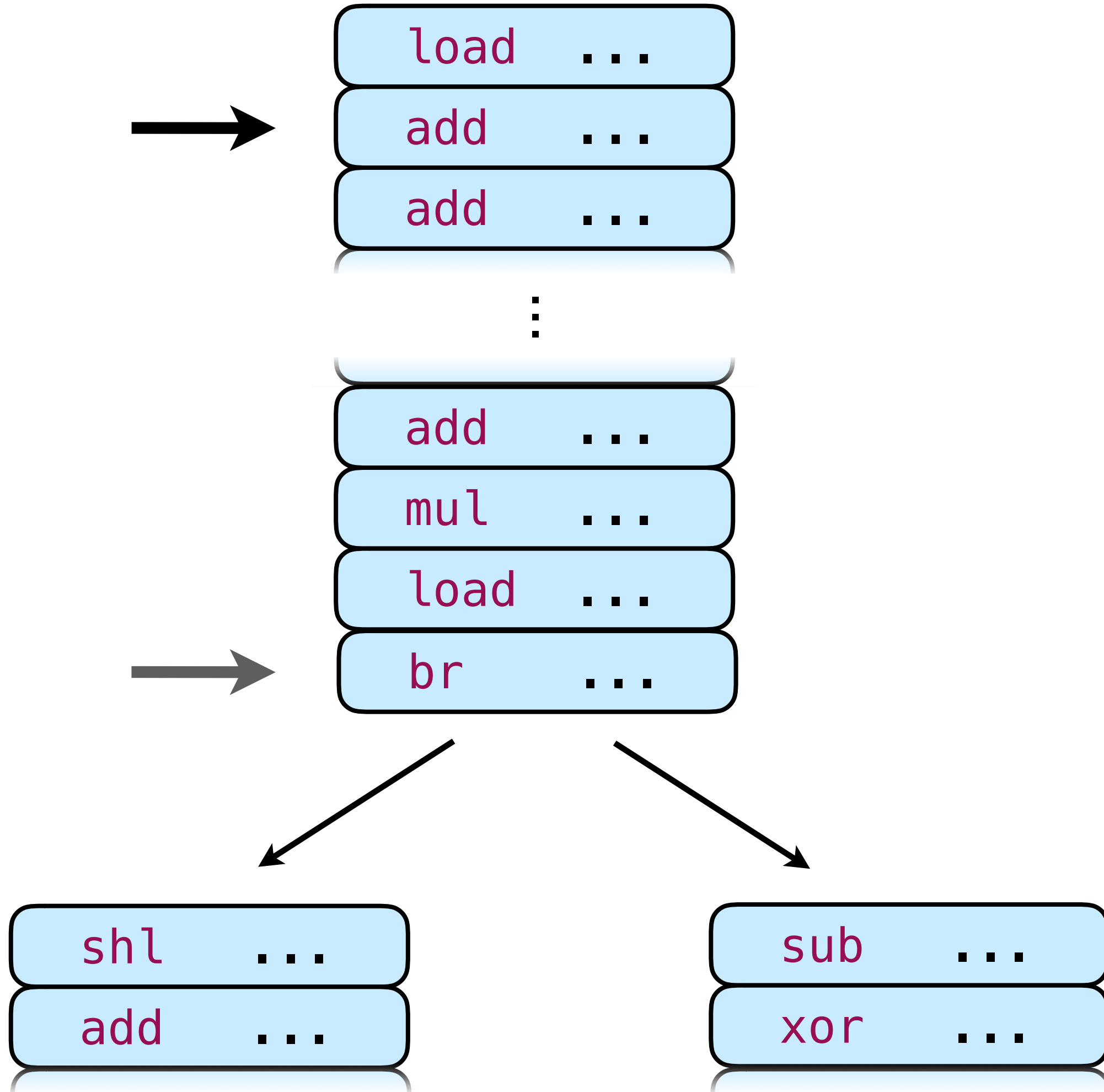


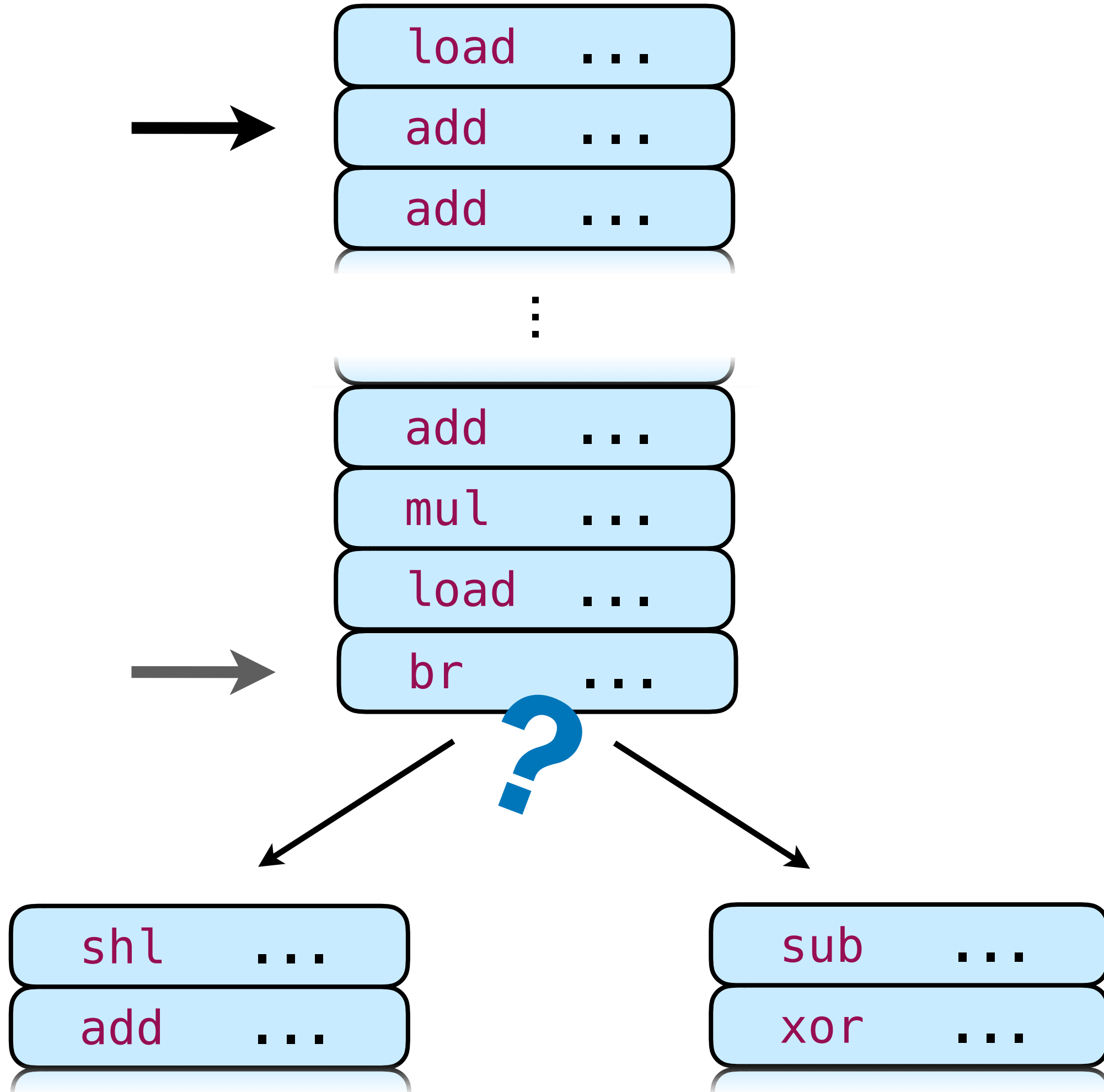


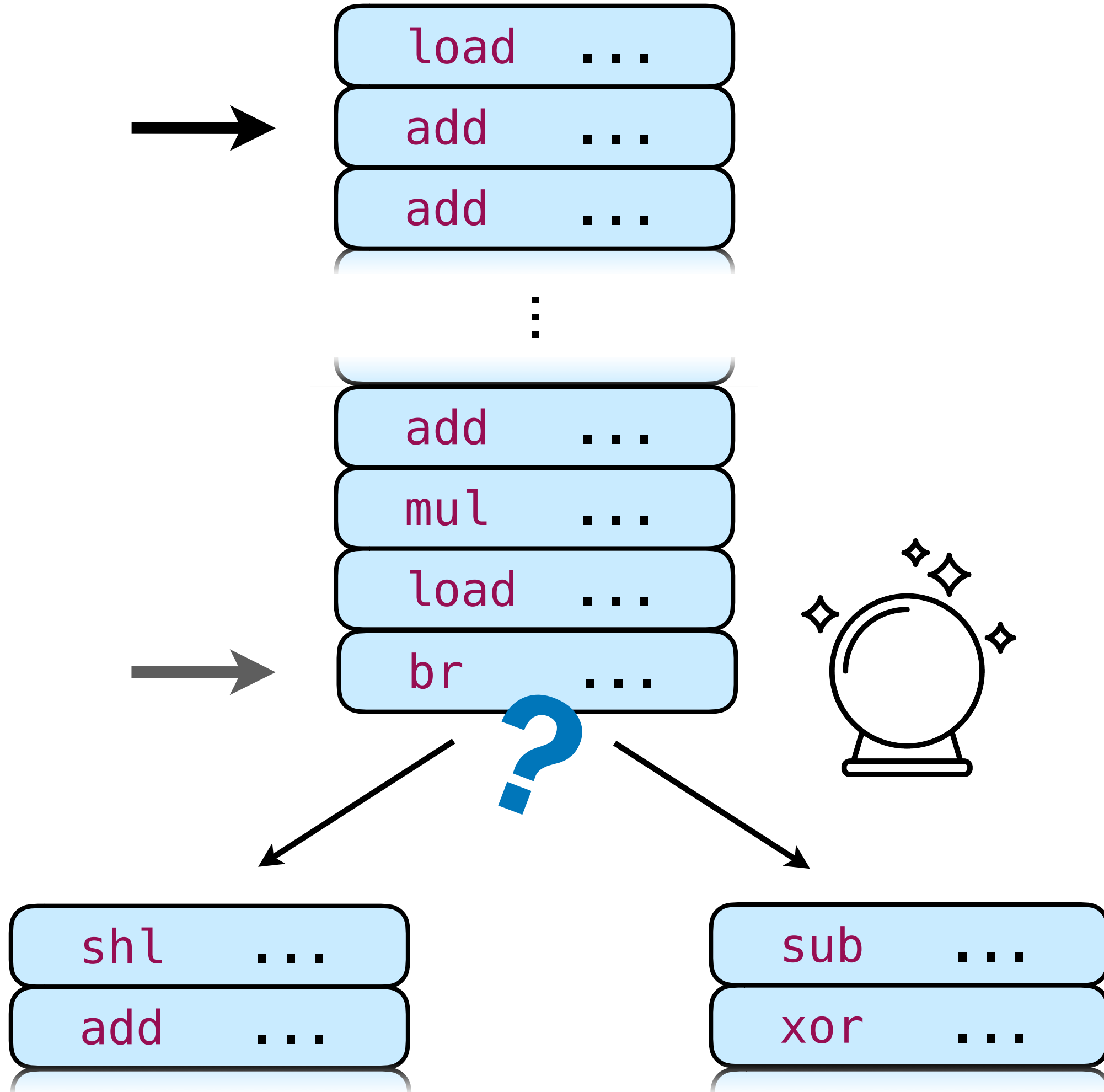




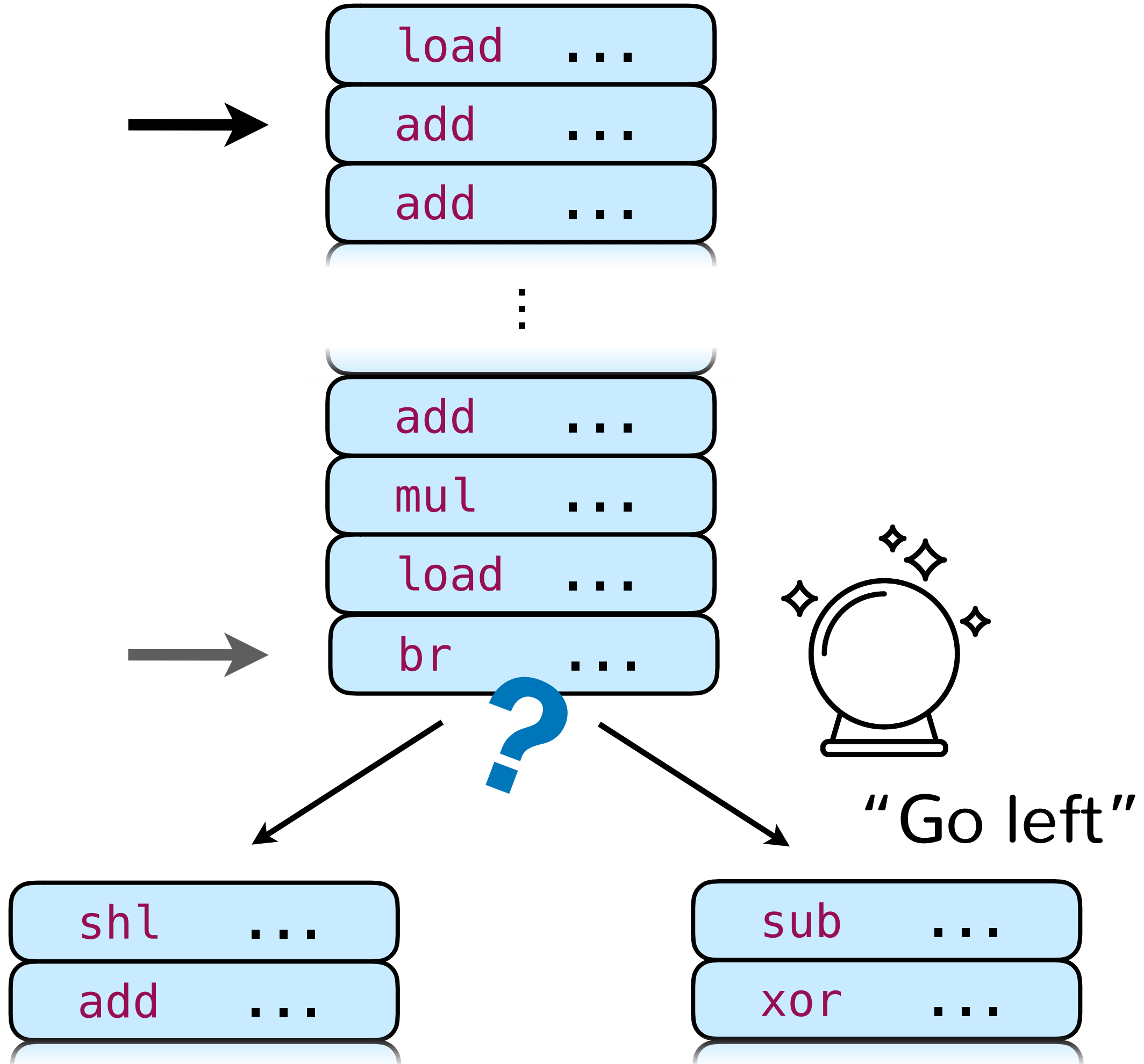




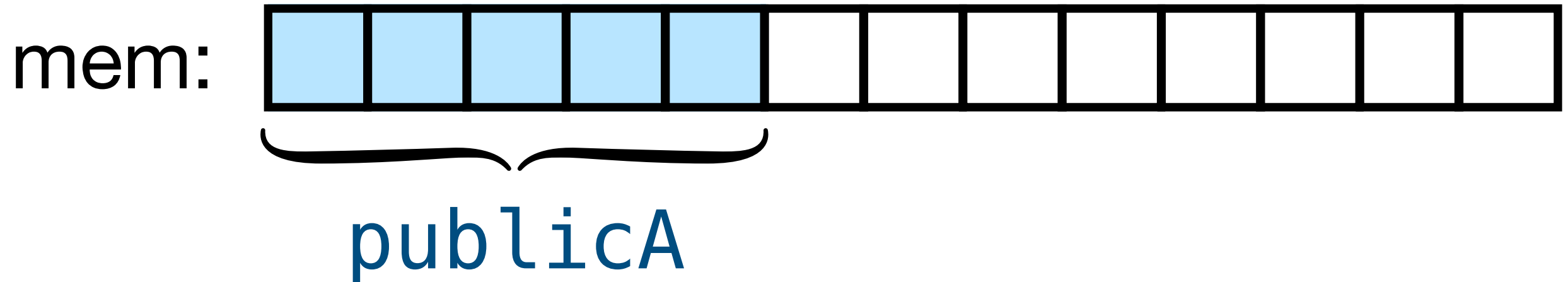








```
if (n < publicLen) {  
    x = publicA[n];  
    y = publicB[x];  
} else {  
    ...  
}
```



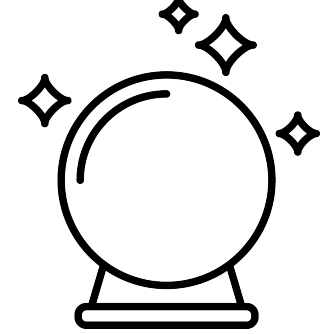
```
if (n < publicLen) {
```

```
    x = publicA[n];
```

```
    y = publicB[x];
```

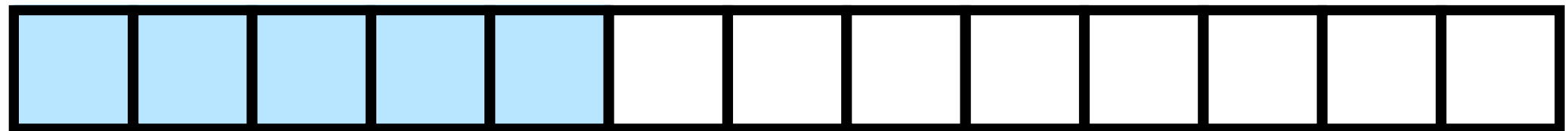
```
} else {
```

```
    ...
```



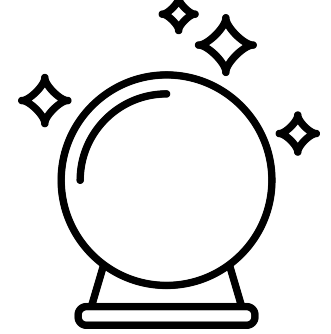
“Condition is true”

mem:



publicA

```
if (n < publicLen) {
```



```
→ x = publicA[n];
```

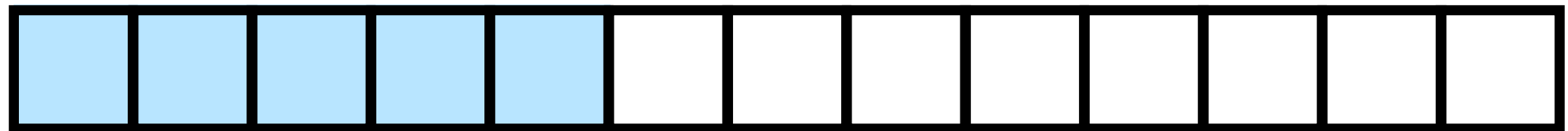
“Condition is true”

```
    y = publicB[x];
```

```
} else {
```

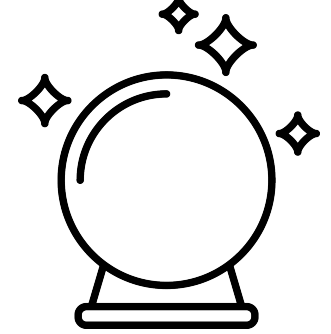
```
    ...
```

mem:



publicA

```
if (n < publicLen) {
```



```
→ x = publicA[n];
```

“Condition is true”

```
    y = publicB[x];
```

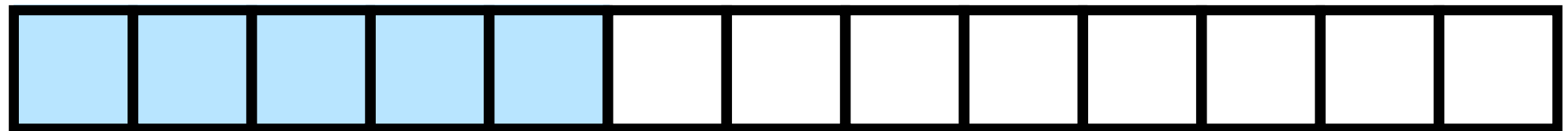
```
} else {
```

```
    ...
```

publicA + n

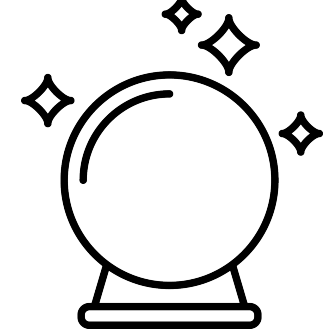


mem:



publicA

```
if (n < publicLen) {
```



```
→ x = publicA[n];
```

“Condition is true”

```
y = publicB[x];
```

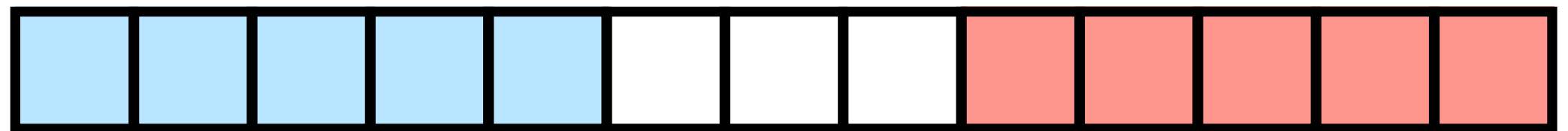
```
} else {
```

```
...
```

publicA + n



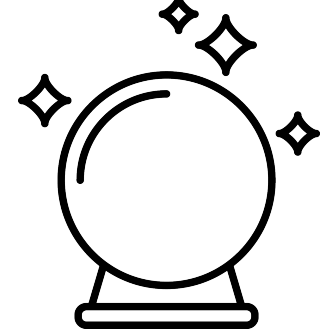
mem:



publicA

secretKey

```
if (n < publicLen) {
```



```
    x = publicA[n];
```

“Condition is true”

```
    → y = publicB[x];
```

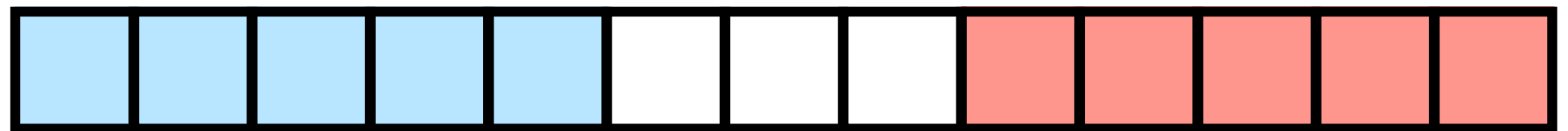
```
} else {
```

```
    ...
```

publicA + n



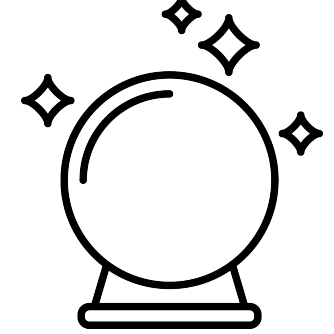
mem:



publicA

secretKey

```
if (n < publicLen) {
```



```
    x = publicA[n];
```

“Condition is true”

```
    → y = publicB[x];
```

```
    } else {
```

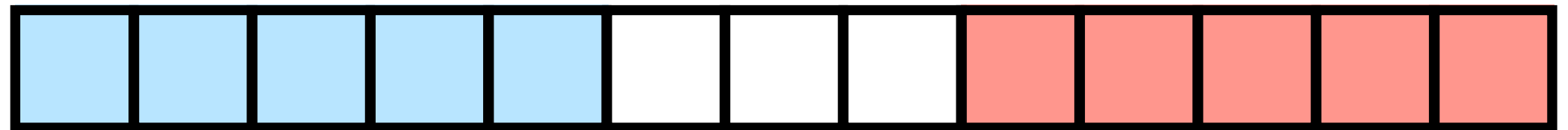
↓ Secret  
memory  
access!

...

publicA + n



mem:



publicA

secretKey



```
if (n < publicLen) {
```

```
    x = publicA[n];
```

```
    → y = publicB[x];
```

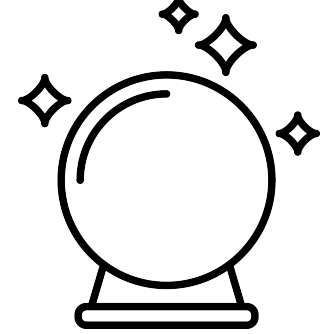
```
} else {
```

```
    ...
```

publicA + n

↓ Secret  
memory  
access!

“Condition is true”



mem:



publicA

secretKey

```
if (n < publicLen) {
```

```
    x = publicA[n];
```

```
    → y = publicB[x];
```

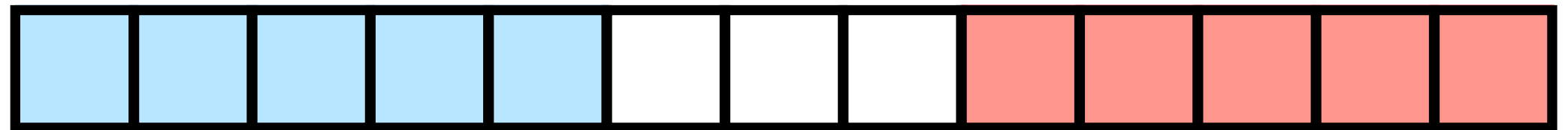
```
} else {
```

```
    ...
```

publicA + n

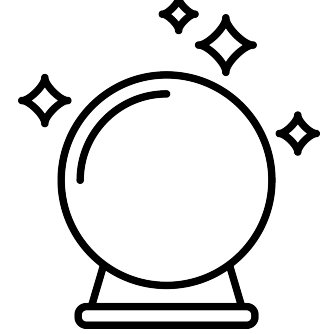


mem:



publicA

secretKey



"Condition is true"

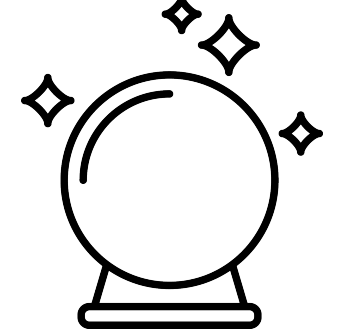
```
if (n < publicLen) {
```

```
    x = publicA[n];
```

```
    y = publicB[x];
```

```
} else {
```

↓ Secret  
memory  
access!



“Condition is true”

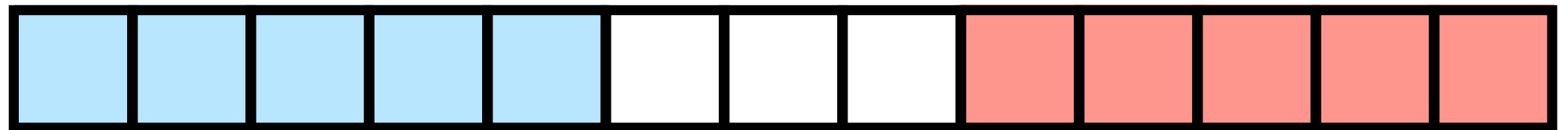


...

publicA + n



mem:



publicA

secretKey

# How do you use this as attacker?

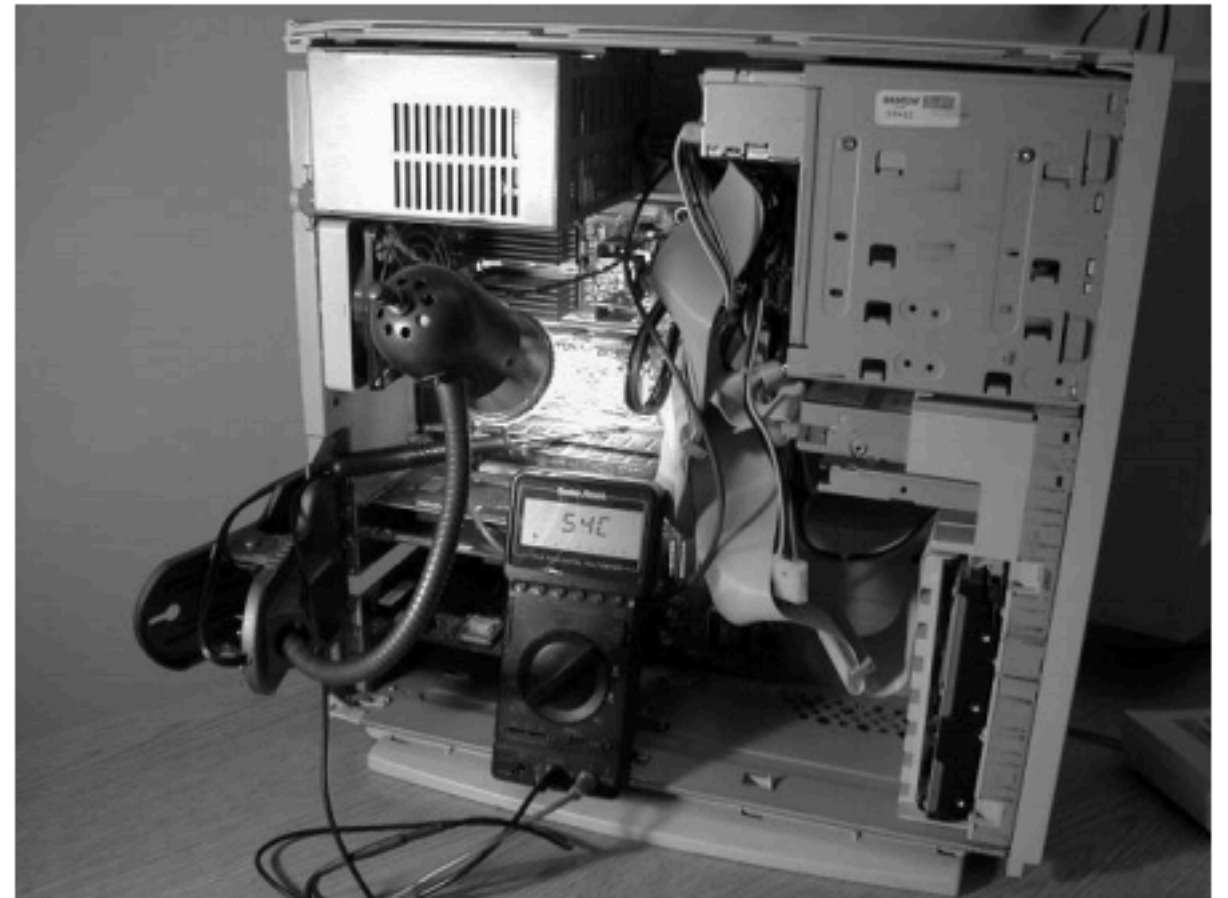
- Train the branch to predict true
- Execute branch w/ victim address
  - CPU will misspeculate and read secret data
  - Secret data not visible at the ISA level, visible in the cache
- Exfiltrate secret with cache attack

```
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