

# Constant-time programming in FaCT

**Sunjay Cauligi**, UC San Diego

Fraser Brown, Ranjit Jhala, Brian Johannesmeyer, John Renner,  
Gary Soeller, Deian Stefan, Riad Wahby, Conrad Watt

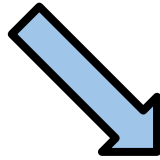
# Timing side channels



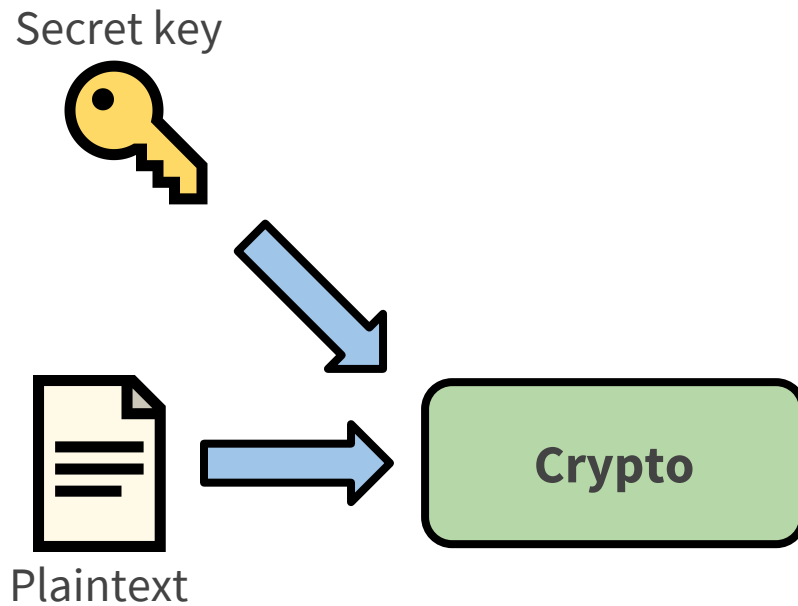
**Crypto**

# Timing side channels

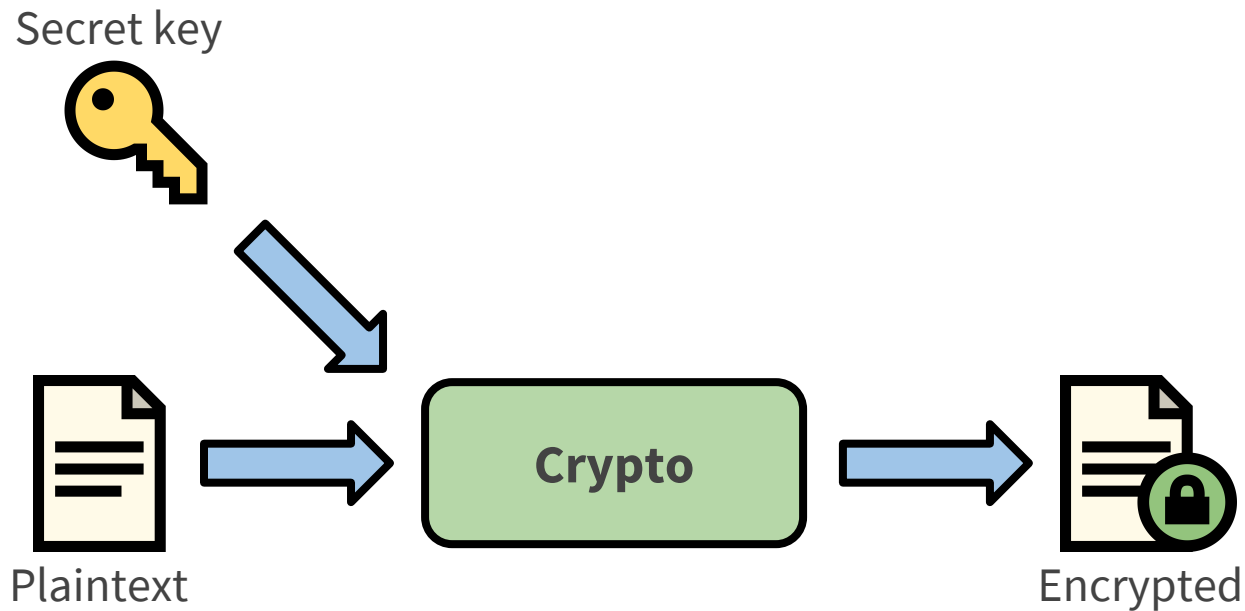
Secret key



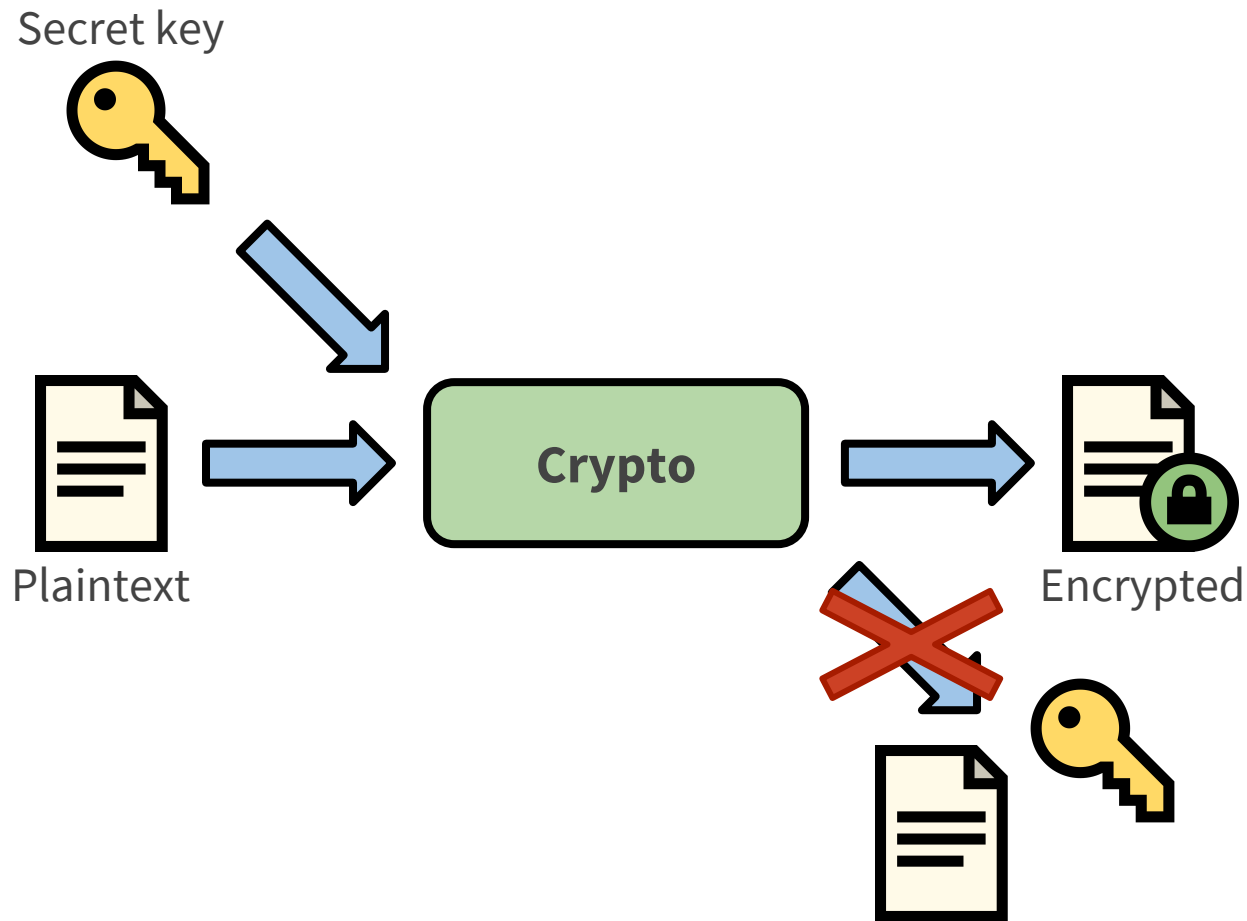
# Timing side channels



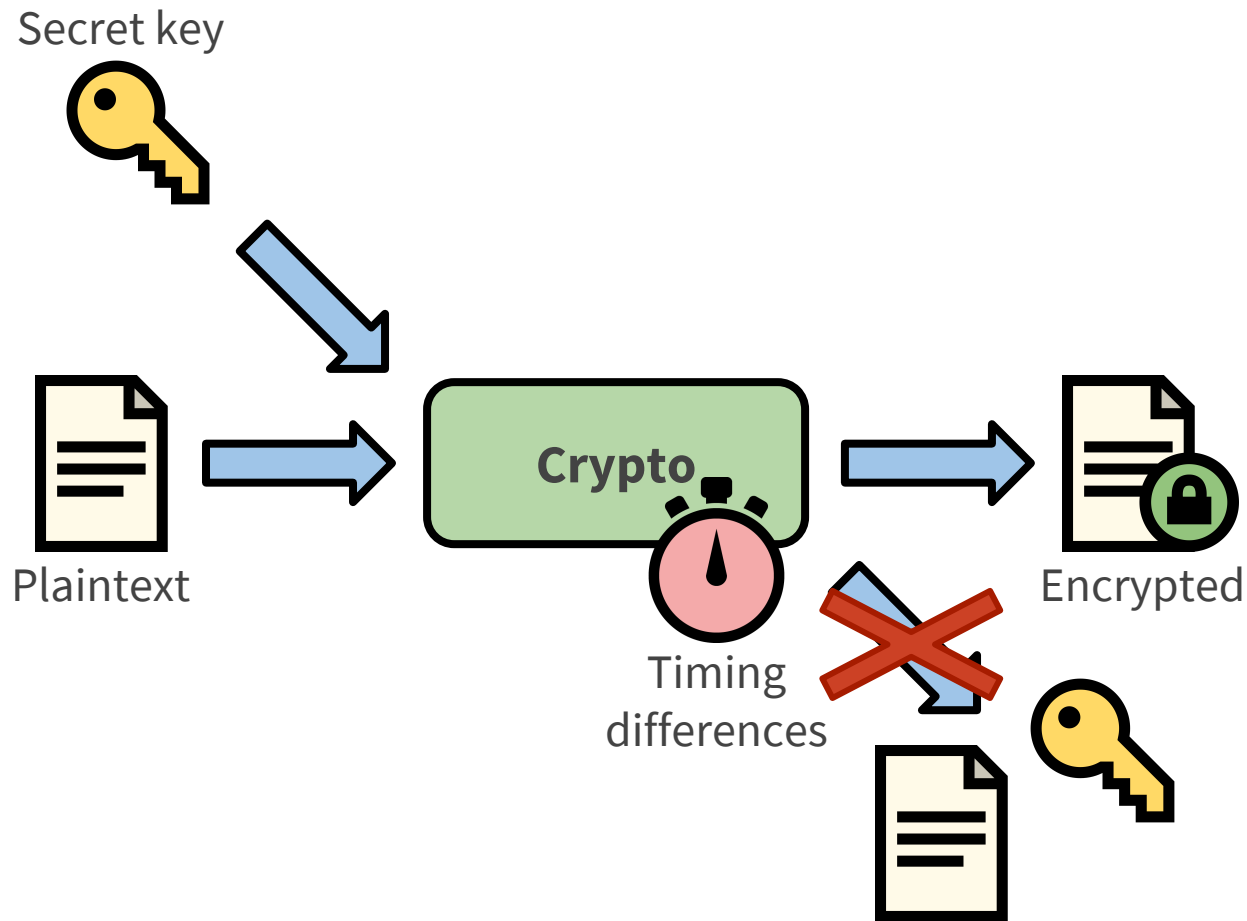
# Timing side channels



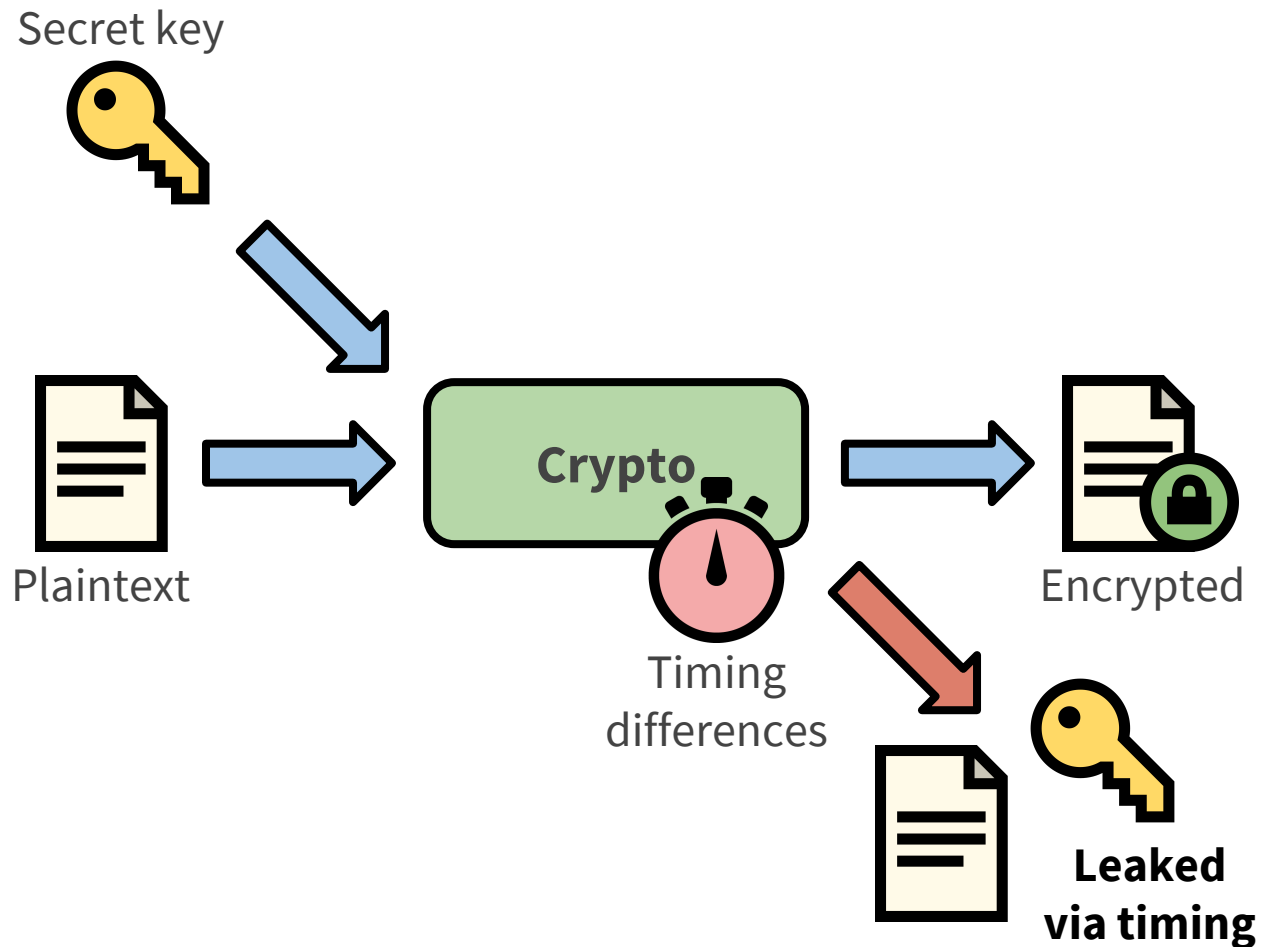
# Timing side channels



# Timing side channels



# Timing side channels



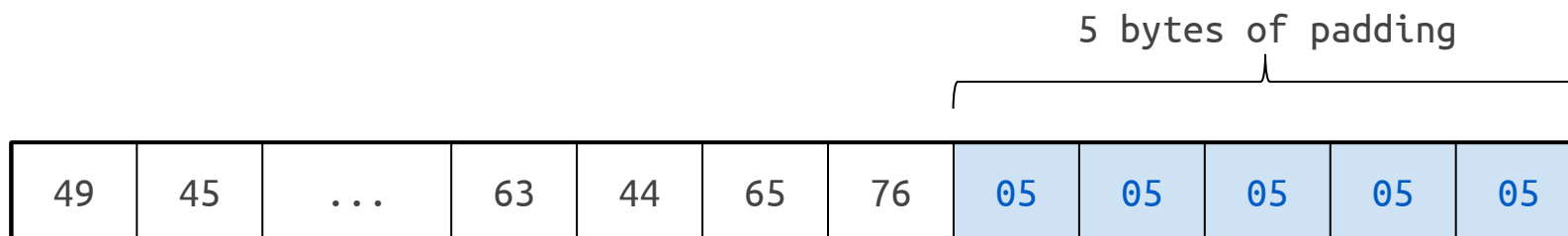


# Constant-time coding example

- Check for valid padding
  - PKCS #7 padding
  - Each padding byte holds length of padding

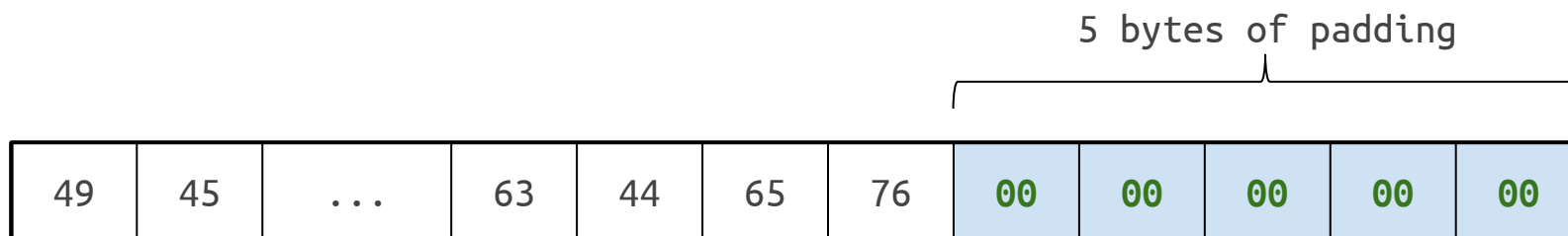
# Constant-time coding example

- Check for valid padding
  - PKCS #7 padding
  - Each padding byte holds length of padding



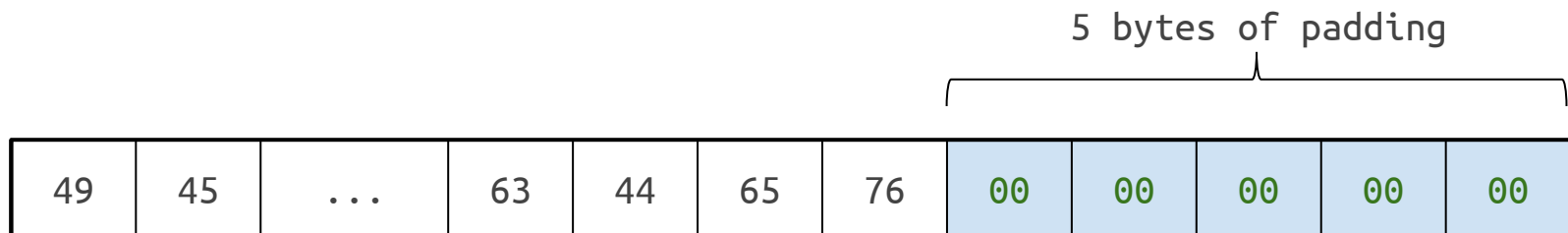
# Constant-time coding example

- Check for valid padding
  - PKCS #7 padding
  - Each padding byte holds length of padding
- Replace padding with null bytes
- Return padding length, or error



# Constant-time coding example

- Check for valid padding
  - PKCS #7 padding
  - Each padding byte holds length of padding
- Replace padding with null bytes
- Return padding length, or error
- Must be careful: buffer contents are secret
  - That includes padding!



```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

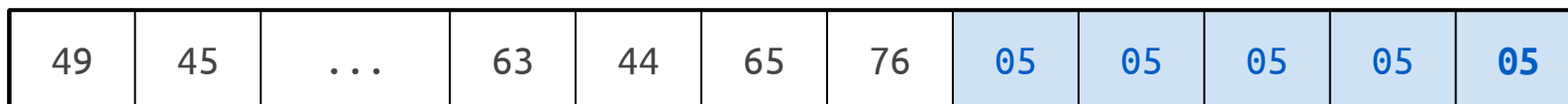
49	45	...	63	44	65	76	05	05	05	05	05
----	----	-----	----	----	----	----	----	----	----	----	----

```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

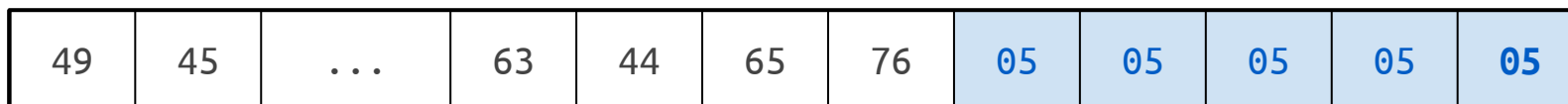


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

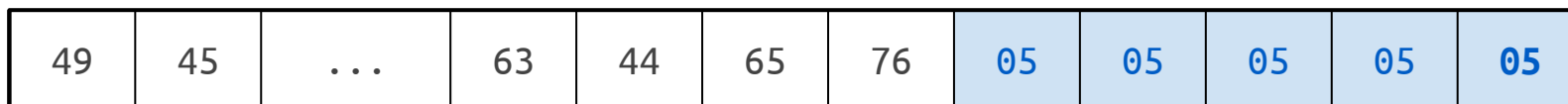


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```



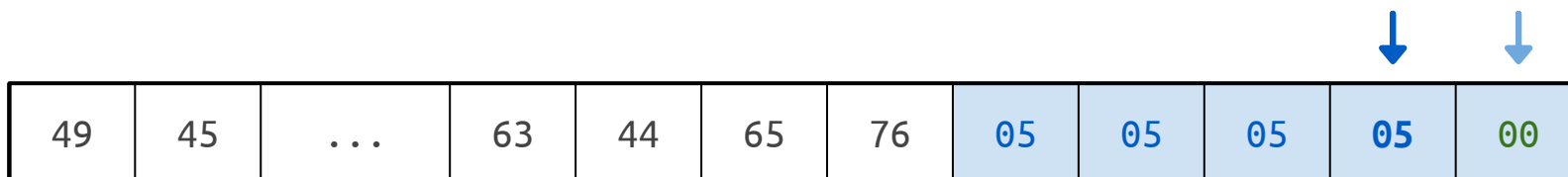


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

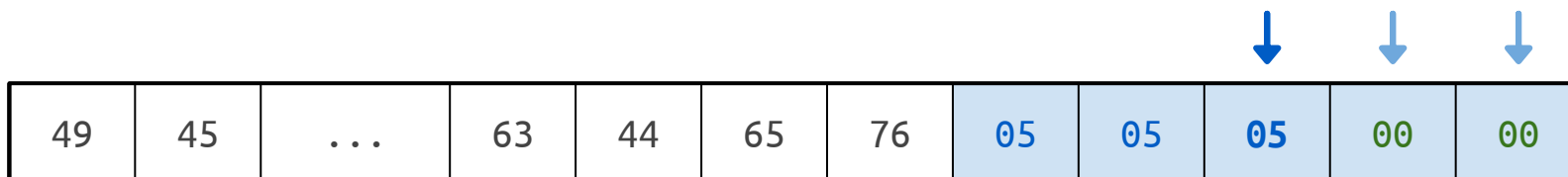


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

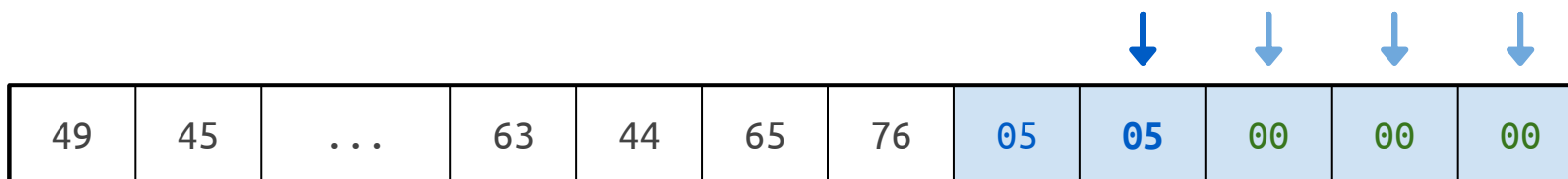


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

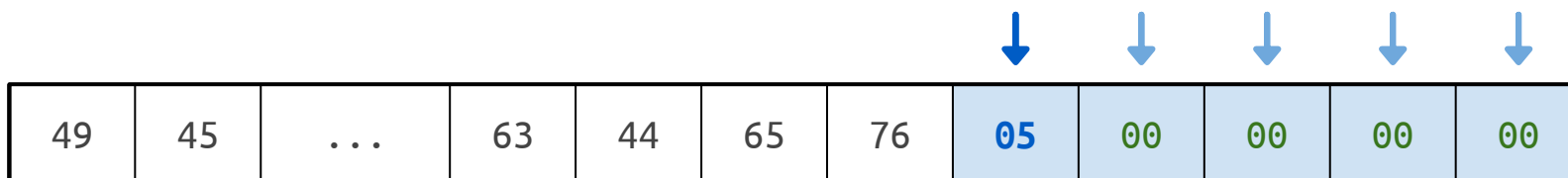


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

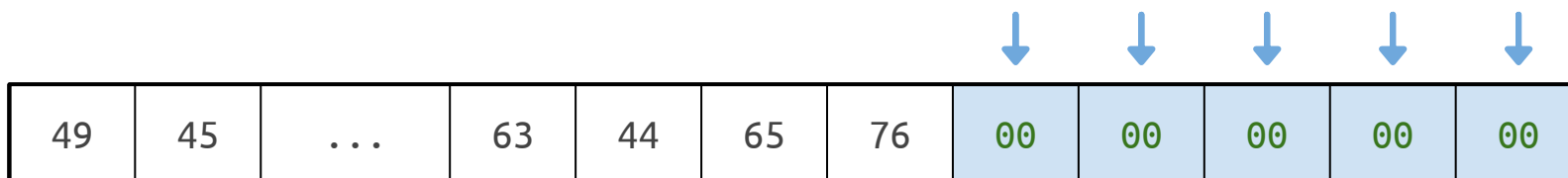


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

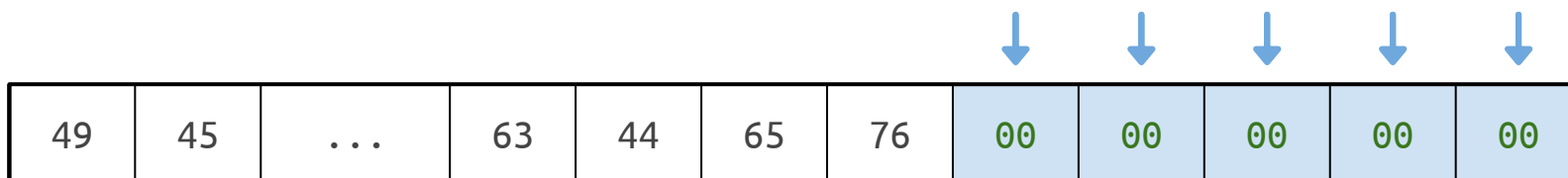


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

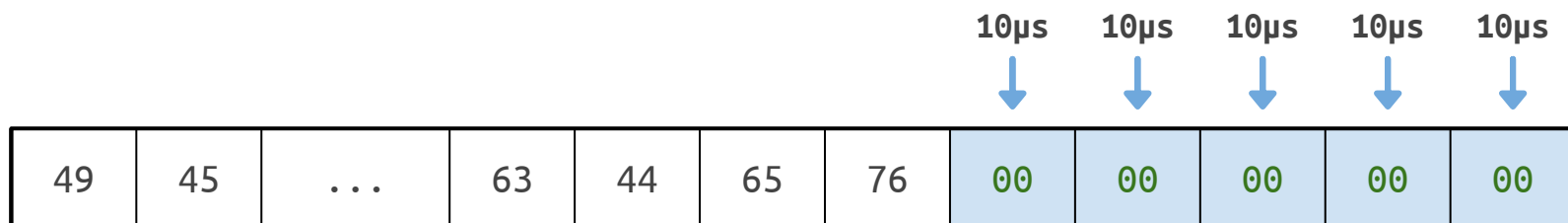


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```



```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

49	45	...	63	44	65	76	05	05	07	05	05
----	----	-----	----	----	----	----	----	----	----	----	----

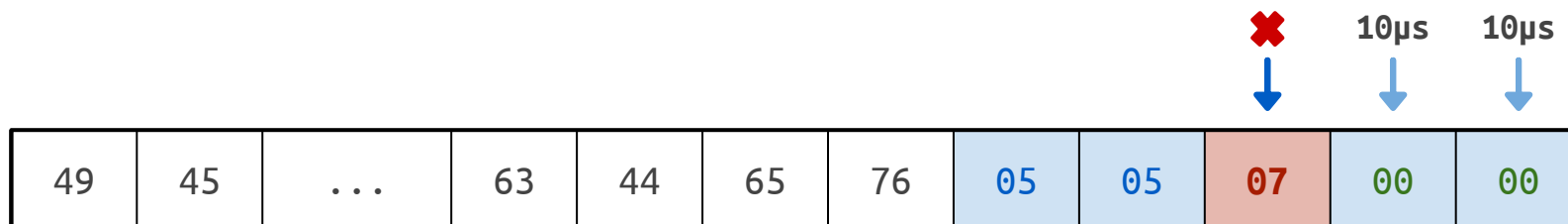


```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```



```

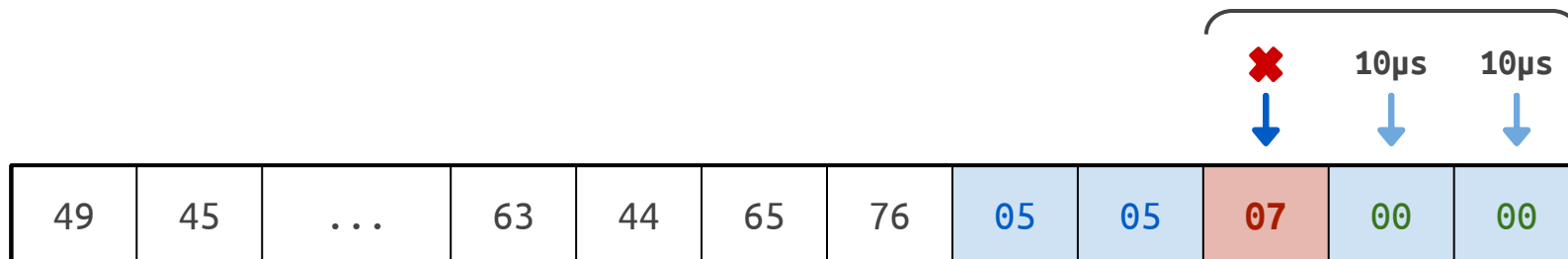
int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```



Padding oracle!



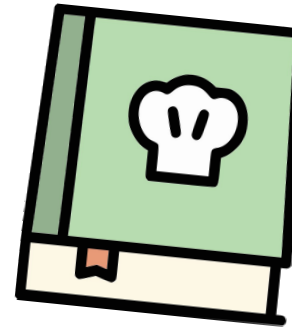
```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

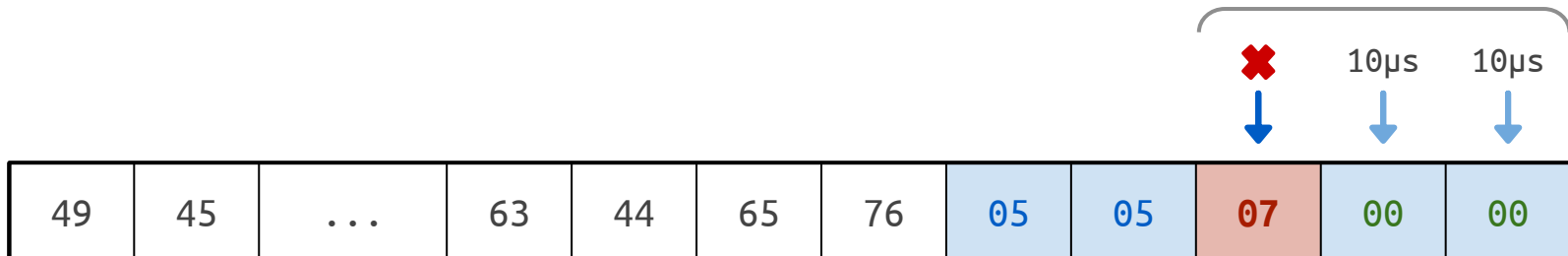
It's dangerous to  
return early!



Use this instead.



Padding oracle!



```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

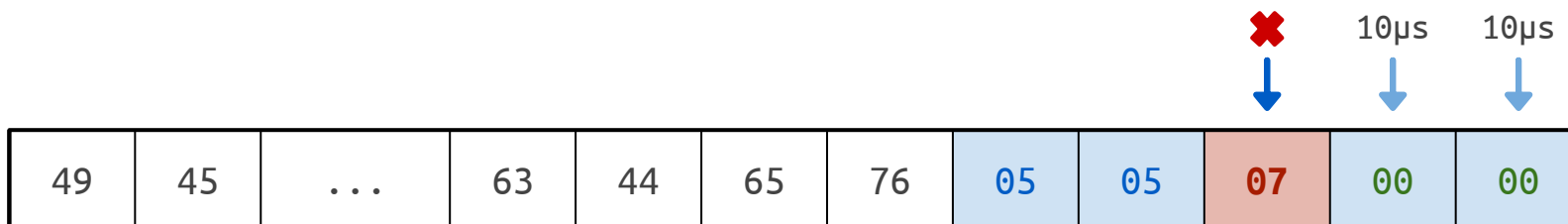
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```



```

int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

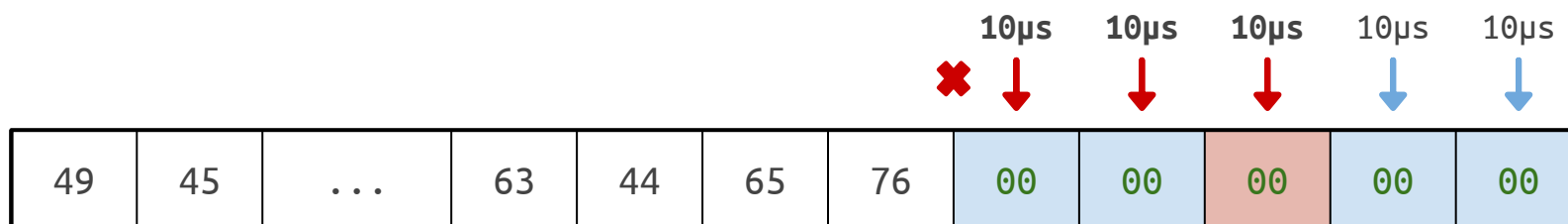
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            return -1;
        buf[buflen-i-1] = 0;
    }
    return padlen;
}

```

```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```



```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

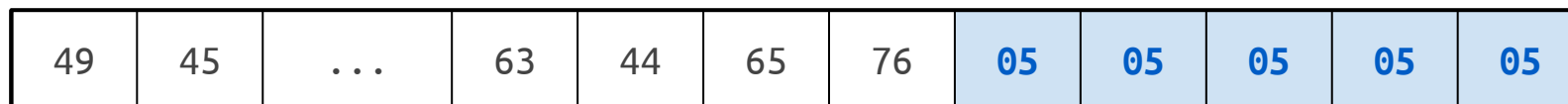
```

49	45	...	63	44	65	76	05	05	05	05	05
----	----	-----	----	----	----	----	----	----	----	----	----

```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

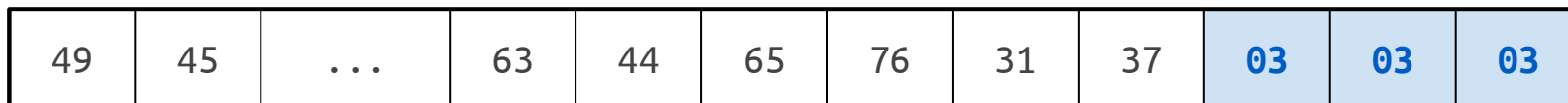
```



```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```

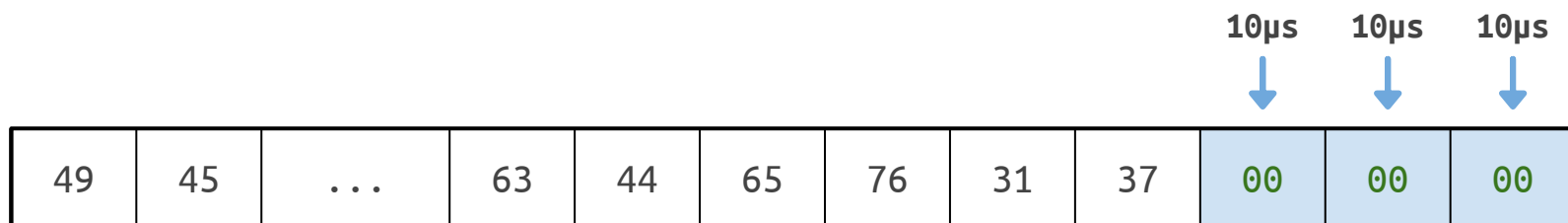




```

int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```

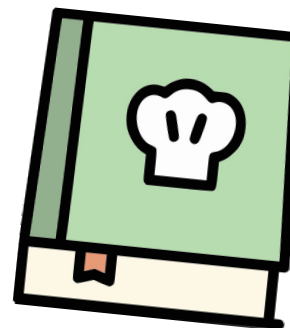


```

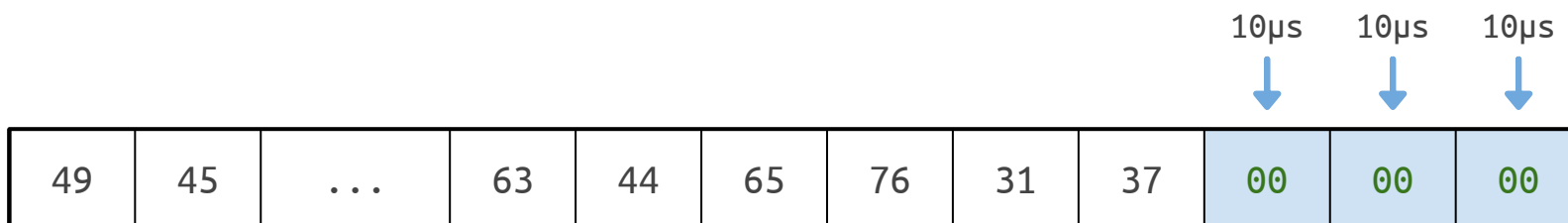
int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```

It's dangerous to  
bound loops with secrets!



Use this instead.



```

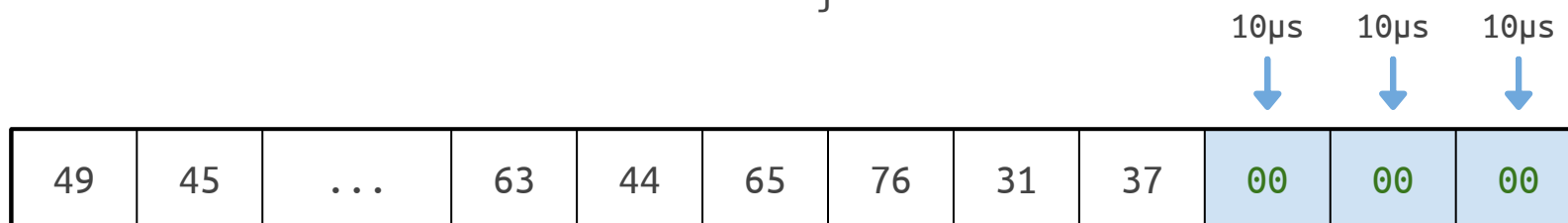
int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```

```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

```



```

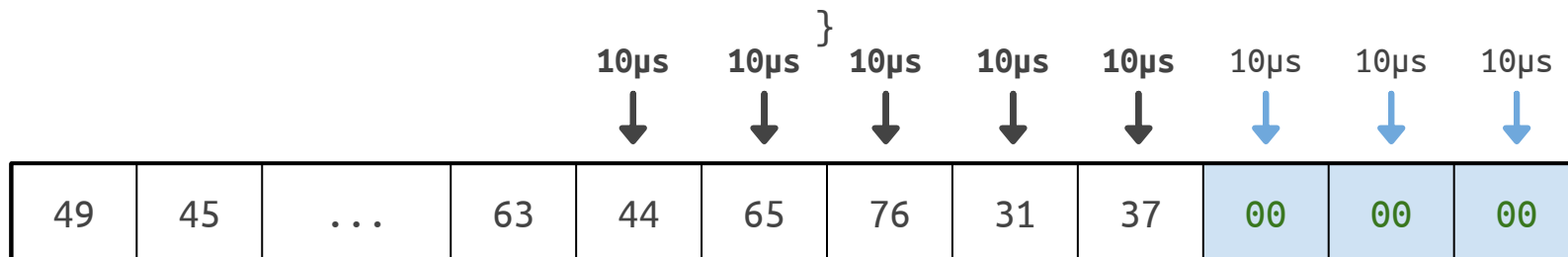
int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = 0; i < padlen; i++) {
        if (buf[buflen-i-1] != padlen)
            ok = 0;
        buf[buflen-i-1] = 0;
    }
    return ok ? padlen : -1;
}

```

```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

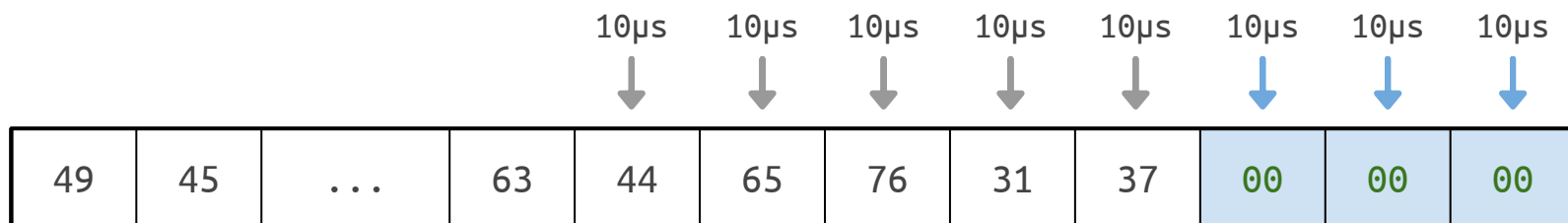
```



```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

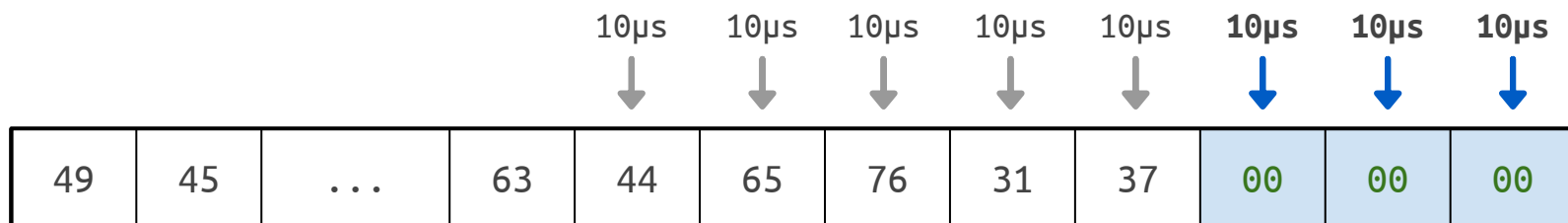
```



```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

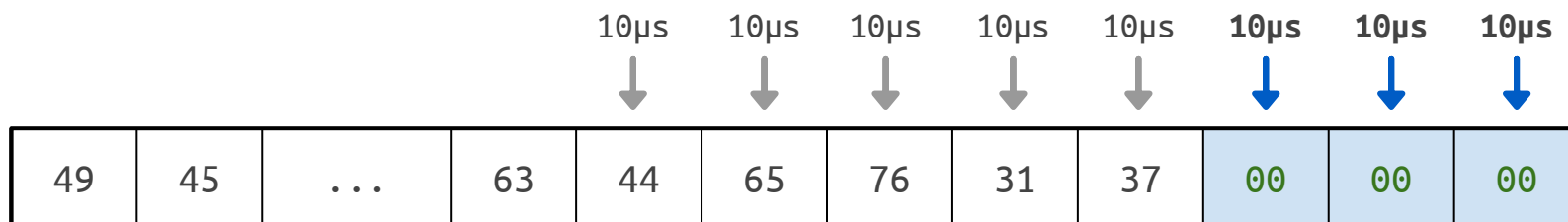
```



```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

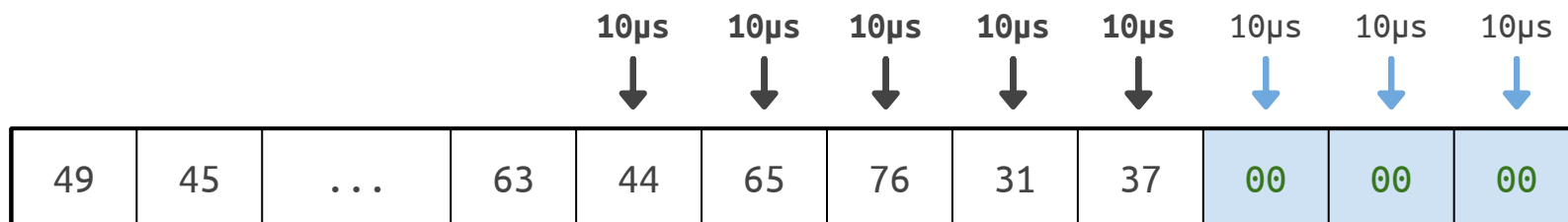
```



```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

```

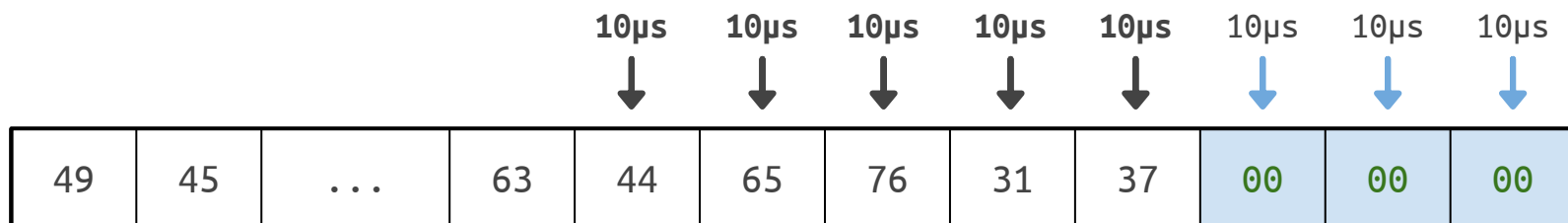




```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

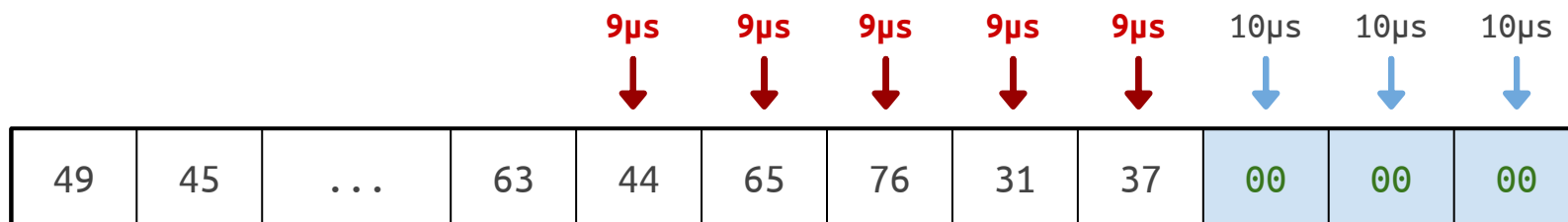
```



```

int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

```

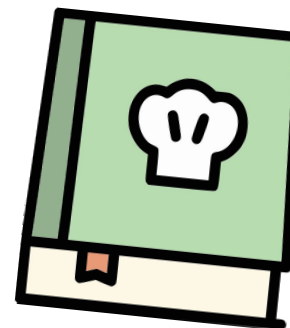


```

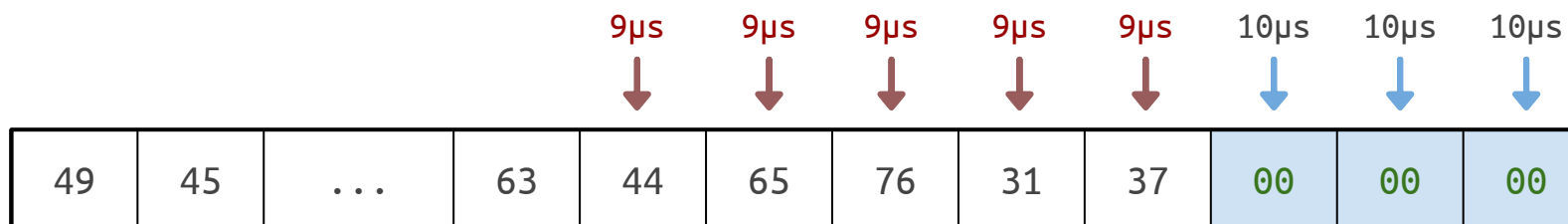
int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}

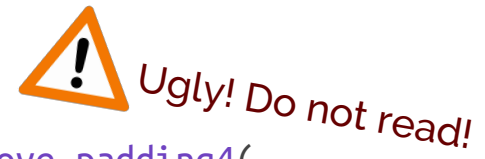
```

It's dangerous to have branching code!



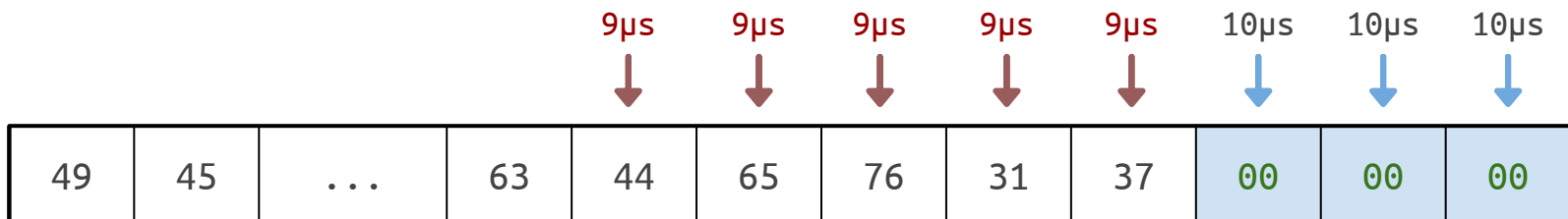
Use this instead.

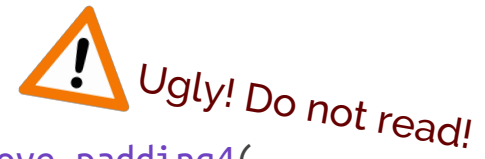




```
int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}
```

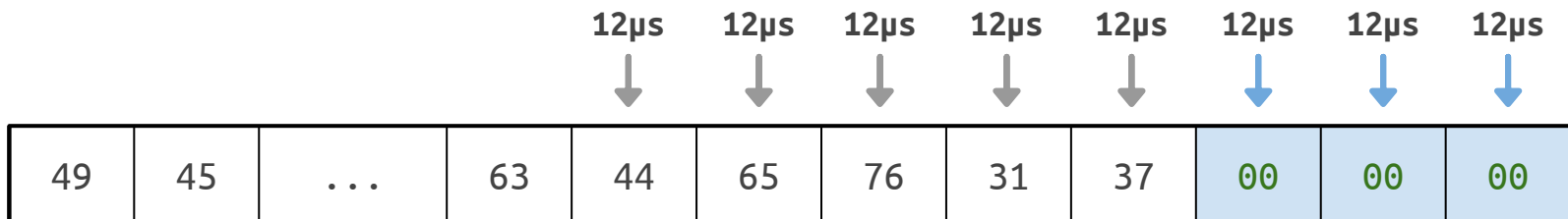
```
int32_t remove_padding4(
    uint8_t* buf,
    uint32_t buflen) {
    uint32_t ok = -1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        uint32_t proper_index =
            ct_ge_u32(i, buflen - padlen);
        uint32_t matches_pad =
            ct_eq_u8(b, padlen);
        ok &= matches_pad & proper_index;
        b = ~proper_index & b;
        buf[i] = b;
    }
    return (ok & padlen) | ~ok;
}
```





```
int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
    uint8_t ok = 1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        if (i >= buflen - padlen) {
            if (b != padlen)
                ok = 0;
            b = 0;
        }
        buf[i] = b;
    }
    return ok ? padlen : -1;
}
```

```
int32_t remove_padding4(
    uint8_t* buf,
    uint32_t buflen) {
    uint32_t ok = -1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        uint32_t proper_index =
            ct_ge_u32(i, buflen - padlen);
        uint32_t matches_pad =
            ct_eq_u8(b, padlen);
        ok &= matches_pad & proper_index;
        b = ~proper_index & b;
        buf[i] = b;
    }
    return (ok & padlen) | ~ok;
}
```



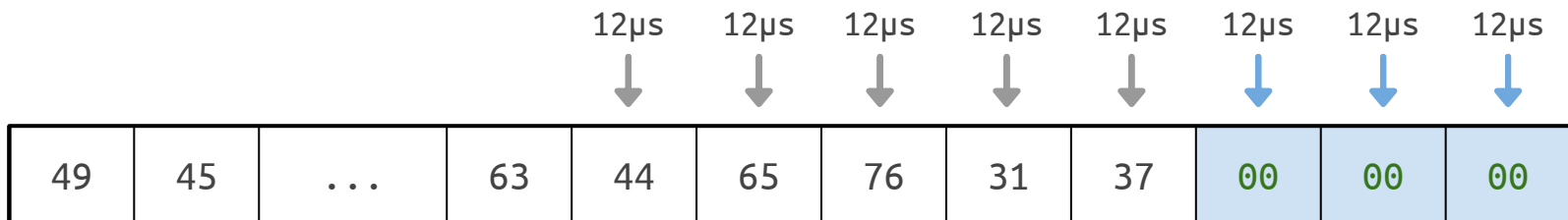
```

int32_t remove_padding4(
    uint8_t* buf,
    uint32_t buflen) {
    uint32_t ok = -1;
    uint8_t padlen = buf[buflen-1];
    uint32_t i;
    for (i = buflen-255; i < buflen; i++) {
        uint8_t b = buf[i];
        uint32_t proper_index = ct_ge_u32(i, buflen - padlen);
        uint32_t matches_pad = ct_eq_u8(b, padlen);
        ok &= matches_pad & proper_index;
        b = ~proper_index & b;
        buf[i] = b;
    }
    return (ok & padlen) | ~ok;
}

```



Ugly! Do not read!



# Error-prone in practice

```
384 384 SSL_RECORD *rr;
385 385 unsigned int mac_size;
386 386 unsigned char md[EVP_MAX_MD_SIZE];
387 387 int decryption_failed_or_bad_record_mac = 0;
388 388
389 389 rrr = &(s->s3->rrec);
390 390
391 391 @@ -417,13 +418,10 @@ dtls1_process_record(SSL *s)
417 418 enc_err = s->method->ssl3_enc->enc(s,0);
418 418 if (enc_err <= 0)
419 419 {
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     /* To minimize information leaked via timing, we will always
428 428     * perform all computations before discarding the message.
429 429     */
430 430     decryption_failed_or_bad_record_mac = 1;
431 431 }
432 432
433 433 #ifdef TLS_DEBUG
434 434 @@ -453,7 +451,7 @@ printf("\n");
453 453 SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_PRE_MAC_LENGTH_TOO_LONG);
454 454 goto f_err;
455 455
456 456 #else
457 457     goto err;
458 458
459 459 #endif
460 460     decryption_failed_or_bad_record_mac = 1;
461 461
462 462 }
463 463
464 464 /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
465 465 @@ -464,17 +462,25 @@ printf("\n");
466 466 SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
467 467 goto f_err;
468 468
469 469 #else
470 470     goto err;
471 471
472 472 #endif
473 473     decryption_failed_or_bad_record_mac = 1;
474 474
475 475 }
476 476
477 477 rr->length=mac_size;
478 478 s->method->ssl3_enc->enc(s,md,0);
479 479 if (1 < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
480 480 {
481 481     goto err;
482 482     decryption_failed_or_bad_record_mac = 1;
483 483 }
484 484
485 485 if (decryption_failed_or_bad_record_mac)
486 486 {
487 487     /* decryption failed, silently discard message */
488 488     rr->length = 0;
489 489     s->packet_length = 0;
490 490     goto err;
491 491 }
492 492
493 493
494 494 /* r->length is now just compressed */
495 495 if (s->expand != NULL)
496 496 {
```

## OpenSSL padding oracle attack

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

# Error-prone in practice

```
384 384 SSL_RECORD *rr;
385 385 unsigned int mac_size;
386 386 unsigned char
387 387 int decryption
388 388
389 389 rrr = &(s->s3->
390 390 00 -417,13 +418,10 00
391 391
392 392 enc_err = s->
393 393 if (enc_err <
394 394 {
395 395
396 396 /* de
397 397 if (e
398 398
399 399
400 399
401 399
402 399
403 399
404 399
405 399
406 399
407 399
408 399
409 399
410 399
411 399
412 399
413 399
414 399
415 399
416 399
417 399
418 399
419 399
420 399
421 399
422 399
423 399
424 399
425 399
426 399
427 399
428 399
429 399
430 399
431 399
432 399
433 399
434 399
435 399
436 399
437 399
438 399
439 399
440 399
441 399
442 399
443 399
444 399
445 399
446 399
447 399
448 399
449 399
450 399
451 399
452 399
453 399
454 399
455 399
456 399
457 399
458 399
459 399
460 399
461 399
462 399
463 399
464 399
465 399
466 399
467 399
468 399
469 399
470 399
471 399
472 399
473 399
474 399
475 399
476 399
477 399
478 399
479 399
480 399
481 399
482 399
483 399
484 399
485 399
486 399
487 399
488 399
489 399
490 399
491 399
492 399
493 399
494 399
495 399
496 399
497 399
498 399
499 399
500 399
501 399
502 399
503 399
504 399
505 399
506 399
507 399
508 399
509 399
510 399
511 399
512 399
513 399
514 399
515 399
516 399
517 399
518 399
519 399
520 399
521 399
522 399
523 399
524 399
525 399
526 399
527 399
528 399
529 399
530 399
531 399
532 399
533 399
534 399
535 399
536 399
537 399
538 399
539 399
540 399
541 399
542 399
543 399
544 399
545 399
546 399
547 399
548 399
549 399
550 399
551 399
552 399
553 399
554 399
555 399
556 399
557 399
558 399
559 399
560 399
561 399
562 399
563 399
564 399
565 399
566 399
567 399
568 399
569 399
570 399
571 399
572 399
573 399
574 399
575 399
576 399
577 399
578 399
579 399
580 399
581 399
582 399
583 399
584 399
585 399
586 399
587 399
588 399
589 399
590 399
591 399
592 399
593 399
594 399
595 399
596 399
597 399
598 399
599 399
600 399
601 399
602 399
603 399
604 399
605 399
606 399
607 399
608 399
609 399
610 399
611 399
612 399
613 399
614 399
615 399
616 399
617 399
618 399
619 399
620 399
621 399
622 399
623 399
624 399
625 399
626 399
627 399
628 399
629 399
630 399
631 399
632 399
633 399
634 399
635 399
636 399
637 399
638 399
639 399
640 399
641 399
642 399
643 399
644 399
645 399
646 399
647 399
648 399
649 399
650 399
651 399
652 399
653 399
654 399
655 399
656 399
657 399
658 399
659 399
660 399
661 399
662 399
663 399
664 399
665 399
666 399
667 399
668 399
669 399
670 399
671 399
672 399
673 399
674 399
675 399
676 399
677 399
678 399
679 399
680 399
681 399
682 399
683 399
684 399
685 399
686 399
687 399
688 399
689 399
690 399
691 399
692 399
693 399
694 399
695 399
696 399
697 399
698 399
699 399
700 399
701 399
702 399
703 399
704 399
705 399
706 399
707 399
708 399
709 399
710 399
711 399
712 399
713 399
714 399
715 399
716 399
717 399
718 399
719 399
720 399
721 399
722 399
723 399
724 399
725 399
726 399
727 399
728 399
729 399
730 399
731 399
732 399
733 399
734 399
735 399
736 399
737 399
738 399
739 399
740 399
741 399
742 399
743 399
744 399
745 399
746 399
747 399
748 399
749 399
750 399
751 399
752 399
753 399
754 399
755 399
756 399
757 399
758 399
759 399
760 399
761 399
762 399
763 399
764 399
765 399
766 399
767 399
768 399
769 399
770 399
771 399
772 399
773 399
774 399
775 399
776 399
777 399
778 399
779 399
780 399
781 399
782 399
783 399
784 399
785 399
786 399
787 399
788 399
789 399
790 399
791 399
792 399
793 399
794 399
795 399
796 399
797 399
798 399
799 399
800 399
801 399
802 399
803 399
804 399
805 399
806 399
807 399
808 399
809 399
810 399
811 399
812 399
813 399
814 399
815 399
816 399
817 399
818 399
819 399
820 399
821 399
822 399
823 399
824 399
825 399
826 399
827 399
828 399
829 399
830 399
831 399
832 399
833 399
834 399
835 399
836 399
837 399
838 399
839 399
840 399
841 399
842 399
843 399
844 399
845 399
846 399
847 399
848 399
849 399
850 399
851 399
852 399
853 399
854 399
855 399
856 399
857 399
858 399
859 399
860 399
861 399
862 399
863 399
864 399
865 399
866 399
867 399
868 399
869 399
870 399
871 399
872 399
873 399
874 399
875 399
876 399
877 399
878 399
879 399
880 399
881 399
882 399
883 399
884 399
885 399
886 399
887 399
888 399
889 399
890 399
891 399
892 399
893 399
894 399
895 399
896 399
897 399
898 399
899 399
900 399
901 399
902 399
903 399
904 399
905 399
906 399
907 399
908 399
909 399
910 399
911 399
912 399
913 399
914 399
915 399
916 399
917 399
918 399
919 399
920 399
921 399
922 399
923 399
924 399
925 399
926 399
927 399
928 399
929 399
930 399
931 399
932 399
933 399
934 399
935 399
936 399
937 399
938 399
939 399
940 399
941 399
942 399
943 399
944 399
945 399
946 399
947 399
948 399
949 399
950 399
951 399
952 399
953 399
954 399
955 399
956 399
957 399
958 399
959 399
960 399
961 399
962 399
963 399
964 399
965 399
966 399
967 399
968 399
969 399
970 399
971 399
972 399
973 399
974 399
975 399
976 399
977 399
978 399
979 399
980 399
981 399
982 399
983 399
984 399
985 399
986 399
987 399
988 399
989 399
990 399
991 399
992 399
993 399
994 399
995 399
996 399
997 399
998 399
999 399
1000 399
```

## Lucky 13 timing attack

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



# Error-prone in practice

```
384 384 SSL_RECORD *rr;
385 385 unsigned int mac_size;
386 386 unsigned char
387 387 int decryption
388 388 +
389 389 rrr = &(s->s3->
390 390 00 -417,13 +418,10 00
417 418 enc_err = s->
419 419 if (enc_err <
420 420 {
421 421 /* de
422 422 if (e
423 423 -
424 424 +
425 425 goto
426 426 +
427 427 /* To
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 +
```

Further refinements

Decryption path has no more measurable timing differences

# Error-prone in practice

```
384 384 SSL_RECORD *rr;
385 385 unsigned int mac_size;
386 386 unsigned char
387 387 int decryption
388 388 +
389 389 rrr = &(s->s3->
390 390 00 -417,13 +418,10 00
417 418 enc_err = s->
419 419 if (enc_err <
420 420 {
421 421 /* de
422 422 if (e
423 423 -
424 424 -
425 425 -
426 426 -
427 427 goto
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 +
```

Further refinements

Decryption path has no more measurable timing differences

# Error-prone in practice

CVE-2016-2107

Somorovsky. “Curious padding oracle in OpenSSL.”

```
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;
```

# Writing constant-time code is hard

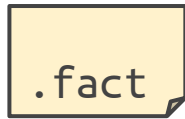
- **Challenge:** manually ensuring code is CT
  - Standard programming constructs introduce timing leaks
  - Manually keep track of secret vs. public
- **Consequence:** vulnerabilities!
  - Difficult to write correct code
  - Hard to understand what CT code is doing
  - Hard to maintain CT code

# This is what DSLs are for!

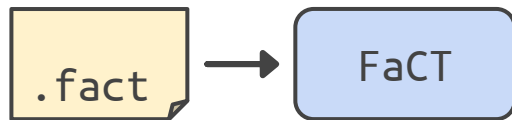
- Explicitly distinguish secret vs. public values
- Type system to prevent writing leaky code
- Compiler to transform high-level constructs to CT

# FaCT

# FaCT

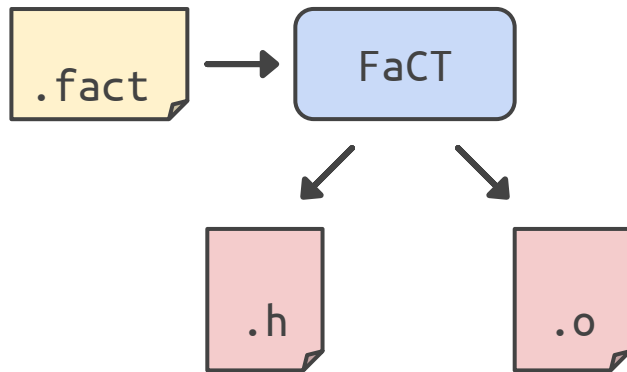


# FaCT

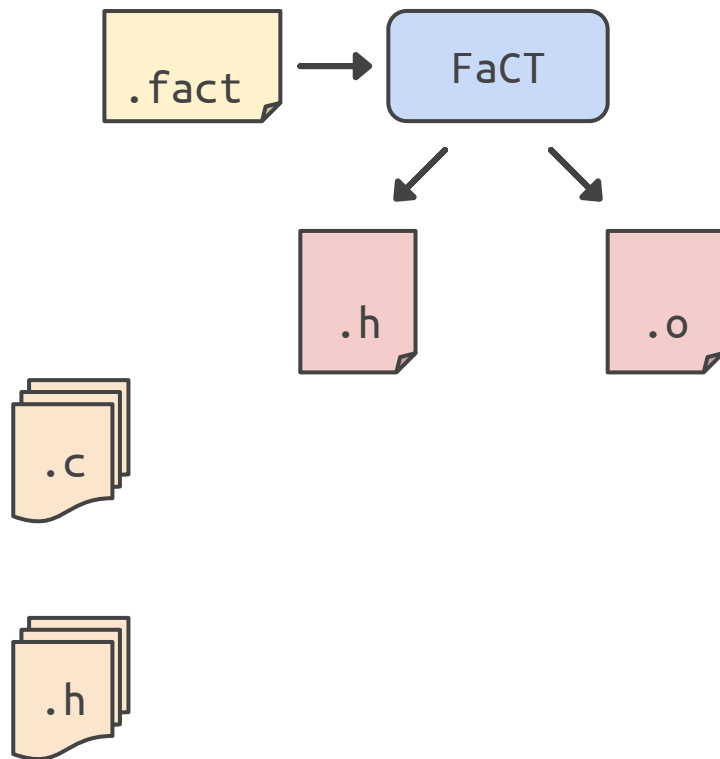




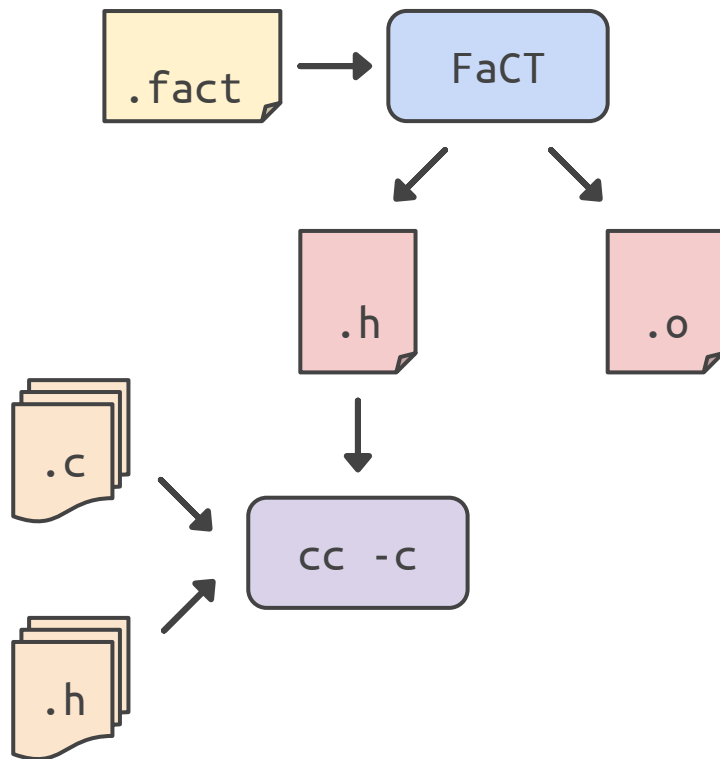
# FaCT



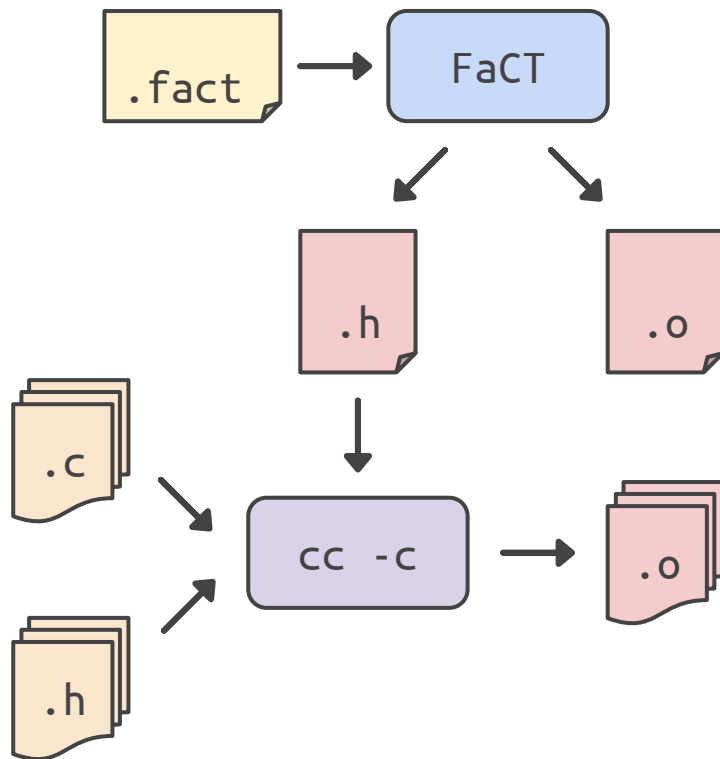
# FaCT



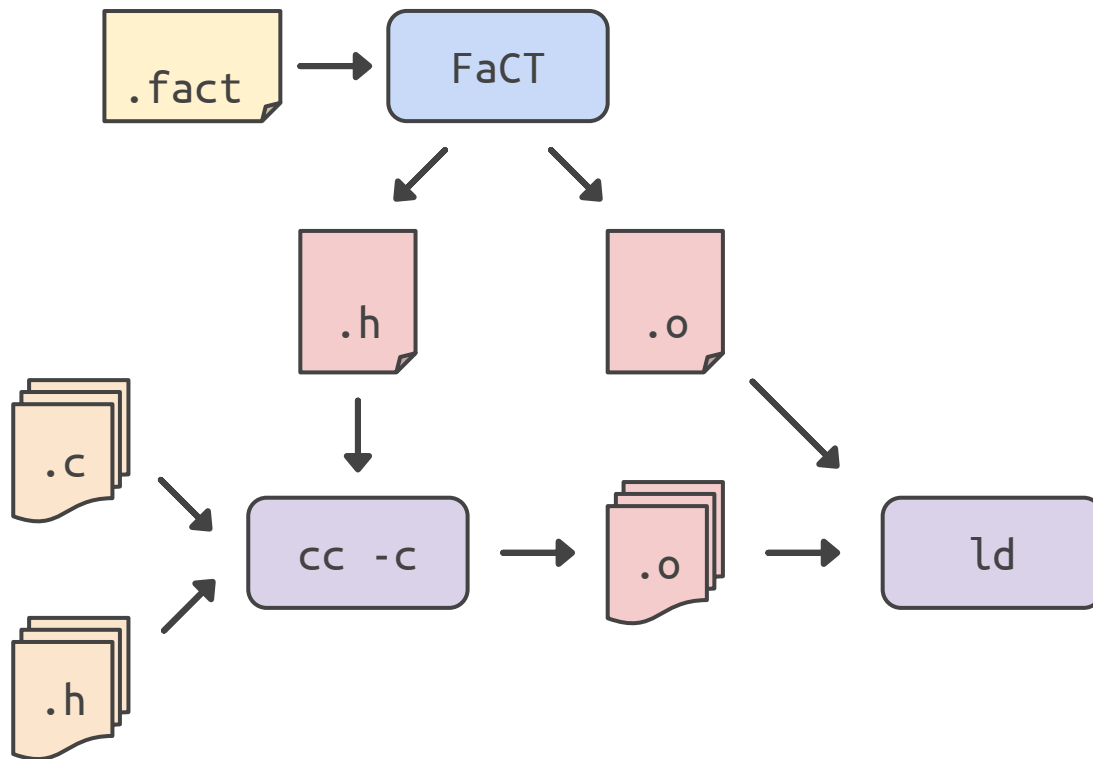
# FaCT



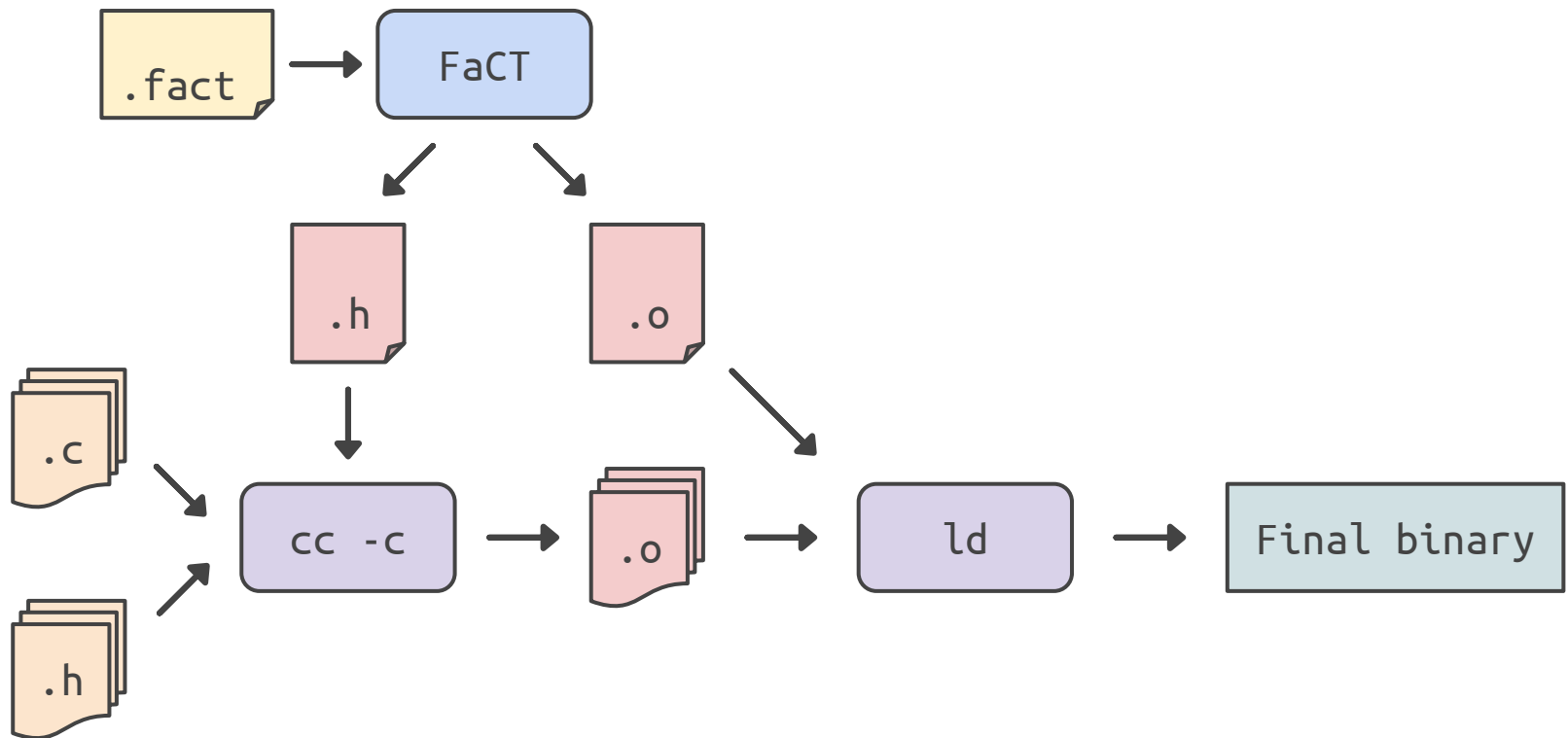
# FaCT



# FaCT



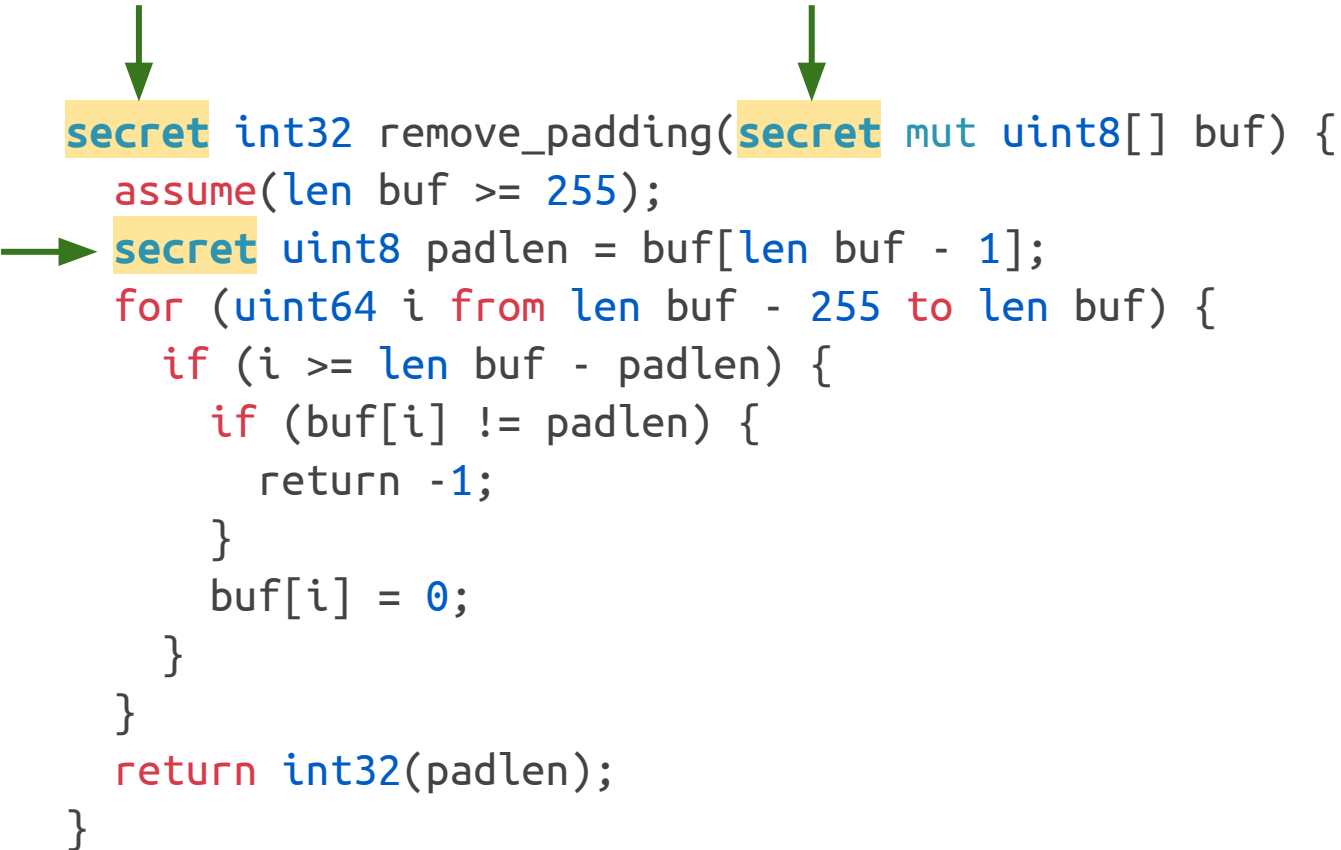
# FaCT



# What does FaCT look like?

```
secret int32 remove_padding(secret mut uint8[] buf) {  
    assume(len buf >= 255);  
    secret uint8 padlen = buf[len buf - 1];  
    for (uint64 i from len buf - 255 to len buf) {  
        if (i >= len buf - padlen) {  
            if (buf[i] != padlen) {  
                return -1;  
            }  
            buf[i] = 0;  
        }  
    }  
    return int32(padlen);  
}
```

# What does FaCT look like?



```
secret int32 remove_padding(secret mut uint8[] buf) {  
    assume(len buf >= 255);  
    secret uint8 padlen = buf[len buf - 1];  
    for (uint64 i from len buf - 255 to len buf) {  
        if (i >= len buf - padlen) {  
            if (buf[i] != padlen) {  
                return -1;  
            }  
            buf[i] = 0;  
        }  
    }  
    return int32(padlen);  
}
```



# What does FaCT look like?



```
secret int32 remove_padding(secret mut uint8[] buf) {  
    assume(len buf >= 255);  
    secret uint8 padlen = buf[len buf - 1];  
    for (uint64 i from len buf - 255 to len buf) {  
        if (i >= len buf - padlen) {  
            if (buf[i] != padlen) {  
                return -1;  
            }  
            buf[i] = 0;  
        }  
    }  
    return int32(padlen);  
}
```

# What does FaCT look like?

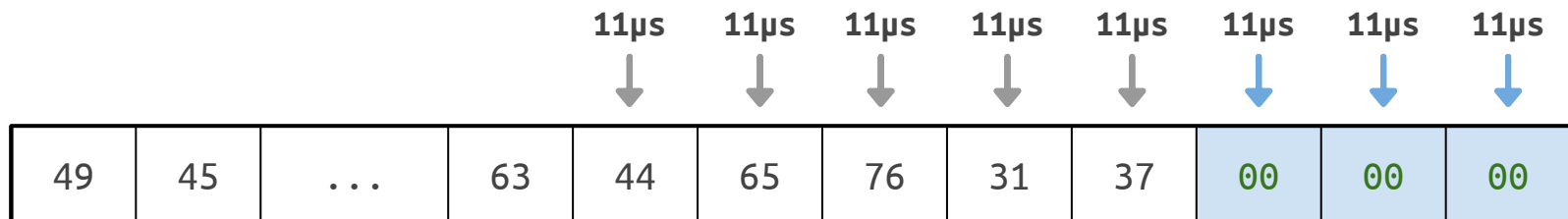
```
secret int32 remove_padding(secret mut uint8[] buf) {  
→ assume(len buf >= 255);  
  secret uint8 padlen = buf[len buf - 1];  
  for (uint64 i from len buf - 255 to len buf) {  
    if (i >= len buf - padlen) {  
      if (buf[i] != padlen) {  
        return -1;  
      }  
      buf[i] = 0;  
    }  
  }  
  return int32(padlen);  
}
```

# What does FaCT look like?

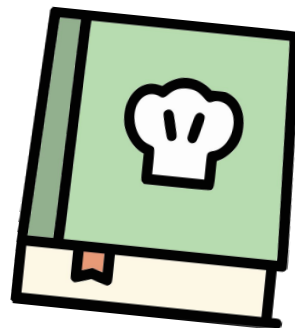
```
secret int32 remove_padding(secret mut uint8[] buf) {  
    assume(len buf >= 255);  
    secret uint8 padlen = buf[len buf - 1];  
    for (uint64 i from len buf - 255 to len buf) {  
→    if (i >= len buf - padlen) {  
→    if (buf[i] != padlen) {  
→    return -1;  
        }  
        buf[i] = 0;  
    }  
}  
return int32(padlen);  
}
```

# What does FaCT look like?

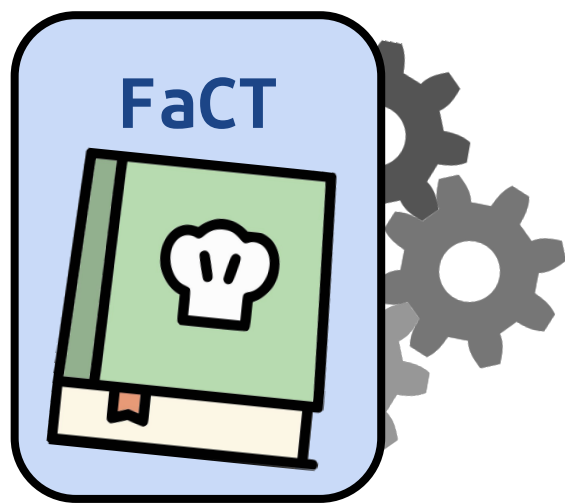
```
secret int32 remove_padding(secret mut uint8[] buf) {  
    assume(len buf >= 255);  
    secret uint8 padlen = buf[len buf - 1];  
    for (uint64 i from len buf - 255 to len buf) {  
        if (i >= len buf - padlen) {  
            if (buf[i] != padlen) {  
                return -1;  
            }  
            buf[i] = 0;  
        }  
    }  
    return int32(padlen);  
}
```



# Automatically transform code



# Automatically transform code



# Automatically transform code

- Transform secret branches into straight-line code
- Keep track of static control flow

```
if (s) {  
    if (s2) {  
        x = 42;  
    } else {  
        x = 17;  
    }  
    y = x + 2;  
}
```

# Automatically transform code

- Transform secret branches into straight-line code
- Keep track of static control flow

```
if (s) {  
    if (s2) {  
        x = 42;  
    } else {  
        x = 17;  
    }  
    y = x + 2;  
}
```



```
x = ct_select(s & s2, 42, x);  
x = ct_select(s & ~s2, 17, x);  
y = ct_select(s, x + 2, y);
```



# Automatically transform code

- Transform away early returns
- Keep track of current return state

```
if (s) {  
    return 42;  
}  
return 17;
```

# Automatically transform code

- Transform away early returns
- Keep track of current return state

```
if (s) {  
    return 42;  
}  
return 17;
```



```
rval = ct_select(s & ~returned, 42, rval);  
returned |= s;  
  
rval = ct_select(~returned, 17, rval);  
returned |= true;  
  
:  
:  
  
return rval;
```

# Automatically transform code

- Transform function side effects
  - Depends on control flow state of caller
- Pass the current control flow as an extra parameter

```
void fn(mut x) {  
    x = 42;  
}
```

```
if (s) {  
    fn(ref x);  
}
```

# Automatically transform code

- Transform function side effects
  - Depends on control flow state of caller
- Pass the current control flow as an extra parameter

```
void fn(mut x) {  
    x = 42;  
}
```



```
void fn(mut x, bool state) {  
    x = ct_select(state, 42, x);  
}
```

```
if (s) {  
    fn(ref x);  
}
```




```
fn(ref x, s);
```

# Labels ensure no leakage

- No assignment from `secret` to `public`
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:

# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds



```
for (uint32 i from 0 to secret_value) {  
    do_operation();  
}
```

# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds

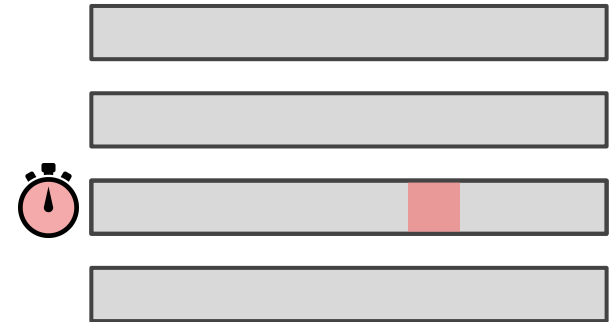
```
for (uint32 i from 0 to public_value) {  
    if (i < secret_value) {  
        do_operation();  
    }  
}
```

# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices

```
x = sensitive_buffer[secret_value];
```

Cache lines



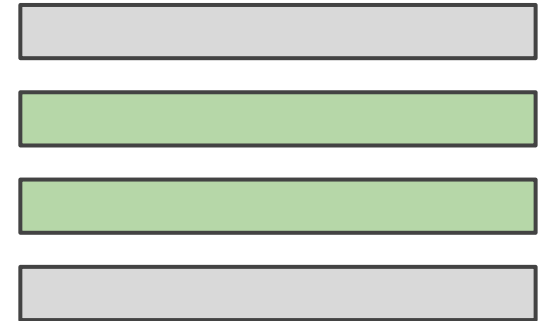


# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices

```
for (uint32 i from public_lo to public_hi) {  
  if (i == secret_value) {  
    x = sensitive_buffer[i];  
  }  
}
```

Cache lines



# Labels ensure no leakage

- No assignment from `secret` to `public`
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices
  - Variable-time instructions



```
x = public_value / secret_value2;
```

# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices
  - Variable-time instructions

```
x = public_value / public_value2;
```


OR

```
x = ct_div(public_value, secret_value2);
```

# Labels ensure no leakage

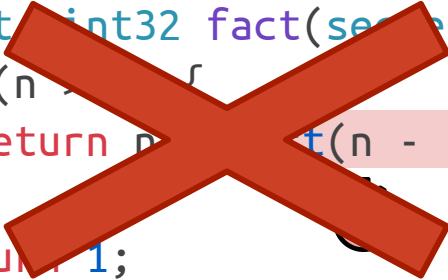
- No assignment from `secret` to `public`
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices
  - Variable-time instructions
  - Recursive calls

```
secret uint32 fact(secret uint32 n) {  
    if (n > 1) {  
        return n * fact(n - 1);  
    }  
    return 1;  
}
```




# Labels ensure no leakage

- No assignment from **secret** to **public**
- Type system tracks control flow label
  - Only transform secret control flow
- Prevent secret expressions we can't transform:
  - Loop bounds
  - Array indices
  - Variable-time instructions
  - Recursive calls



```
secret_int32 fact(secret_uint32 n) {  
  if (n > 1) {  
    return n * fact(n - 1);  
  }  
  return 1;  
}
```


# Transformations are tricky



```
if (secret_modeval == HAS_32_BYTES) {  
    buf[31] = 0;  
}
```

# Transformations are tricky


```
if (secret_modeval == HAS_32_BYTES) {  
    buf[31] = 0;  
}
```



```
m = (secret_modeval == HAS_32_BYTES);  
buf[31] = ct_select(m, 0, buf[31]);
```

# Transformations are tricky

```
if (secret_modeval == HAS_32_BYTES) {  
    buf[31] = 0;  
}
```




```
m = (secret_modeval == HAS_32_BYTES);  
buf[31] = ct_select(m, 0, buf[31]);
```

- **Problem:** secret if-statements always perform branches
  - Does not guard execution
  - Similar problem for secret early return



# Transformations are tricky

```
if (secret_modeval == HAS_32_BYTES) {  
    buf[31] = 0;  
}
```

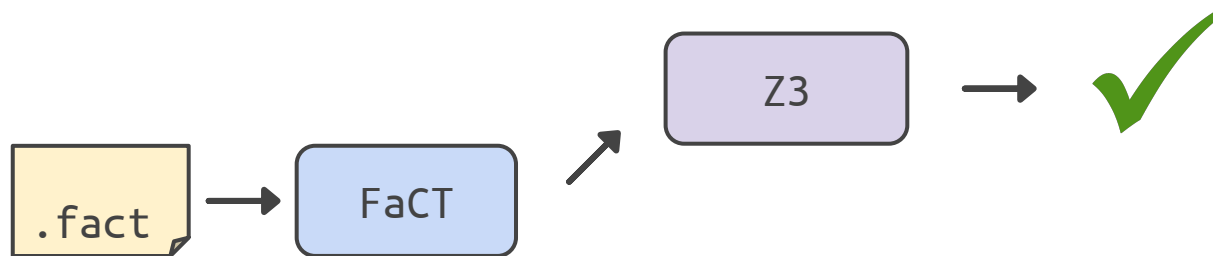


```
m = (secret_modeval == HAS_32_BYTES);  
buf[31] = ct_select(m, 0, buf[31]);
```

- **Problem:** secret if-statements always perform branches
  - Does not guard execution
  - Similar problem for secret early return
- **Solution:** disallow these programs!

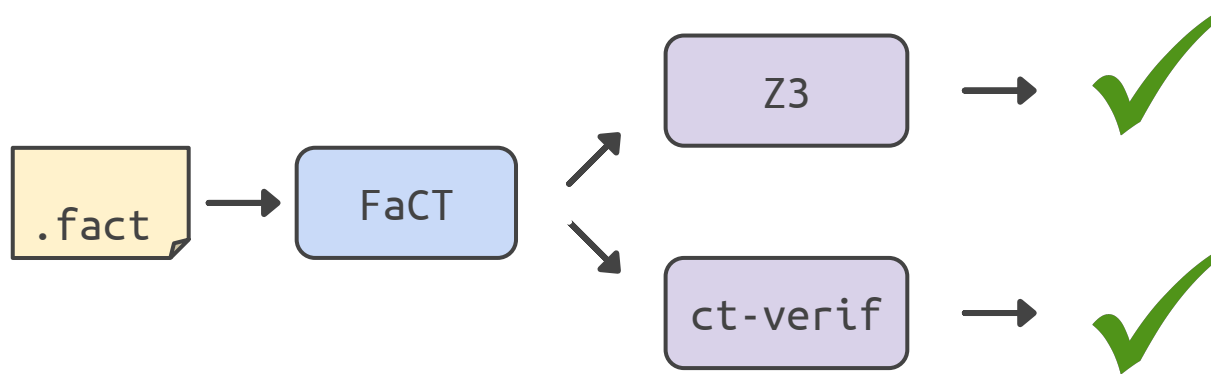
# Verifying constant-time code

- FaCT is memory safe and has no undefined behavior
  - Generate constraints while type checking
  - Aware of secret-if semantics



# Verifying constant-time code

- FaCT is memory safe and has no undefined behavior
  - Generate constraints while type checking
  - Aware of secret-if semantics
- FaCT generates constant-time code
  - Verified with external tool



# Ok, but how good is it really?

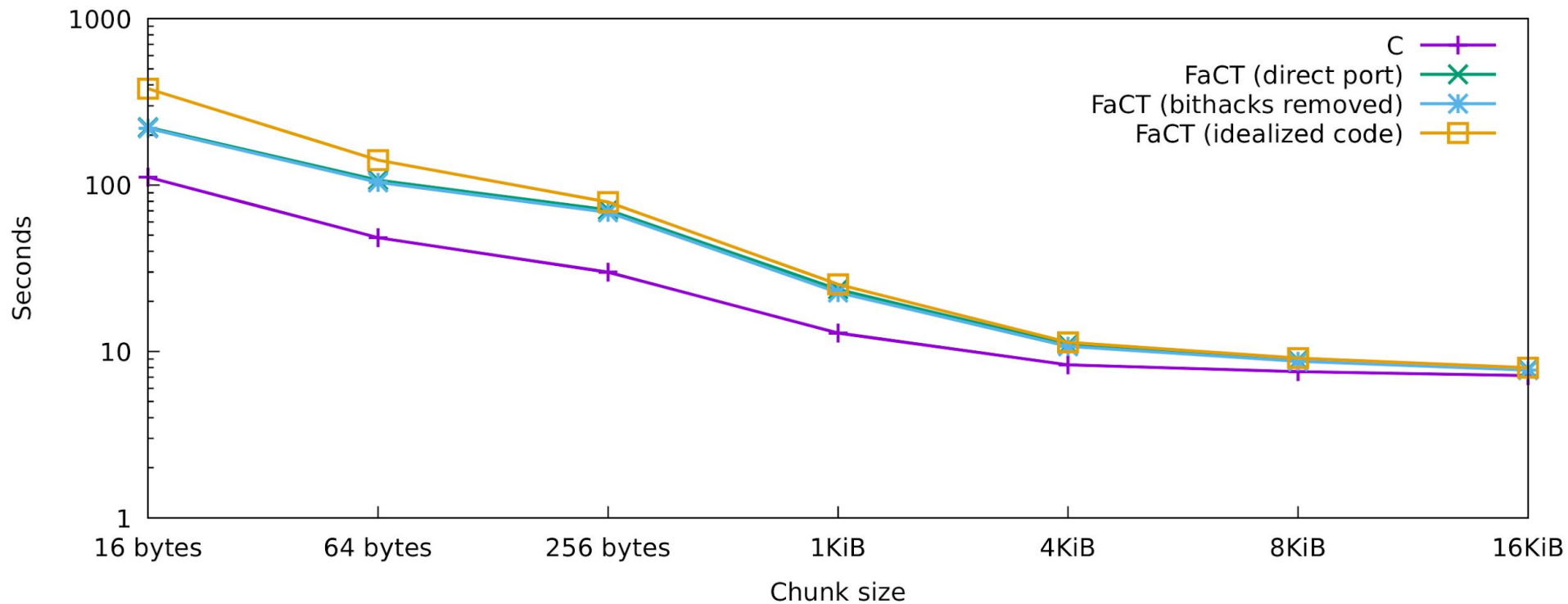
- How fast is it?
- How usable is it?

# Ok, but how good is it really?

- How fast is it?
- How usable is it?
- Case studies:
  - libsodium's `crypto_secretbox`
  - OpenSSL's AES-CBC-HMAC-SHA1
  - mbedTLS's Montgomery multiplication & sliding window exponentiation
  - DJB/Langley's curve-25519

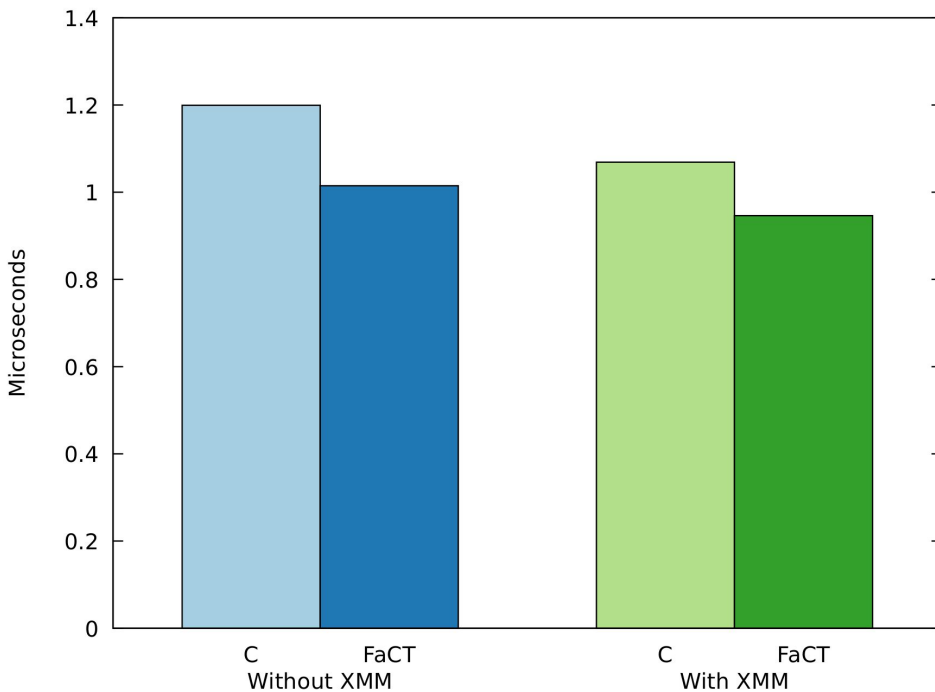
# Performance numbers

## OpenSSL AES-CBC-HMAC-SHA1 in TLS mode

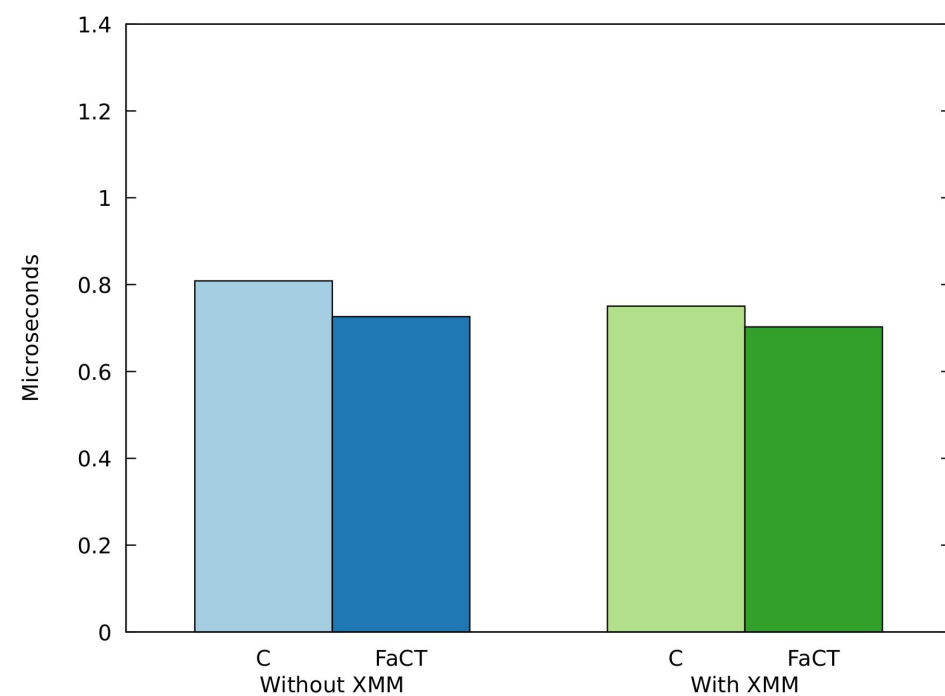


# Performance numbers

## secretbox encryption



## secretbox decryption



# Future directions

- Optimize transformations
- Add other backends (ARM, CT-WASM, ...)
- Verify the FaCT compiler





# FaCT

<https://github.com/PLSysSec/FaCT>

- DSL for cryptographic code
- Automatic transformation to constant-time
- Easily fits into your existing toolchain
- Usable and fast

