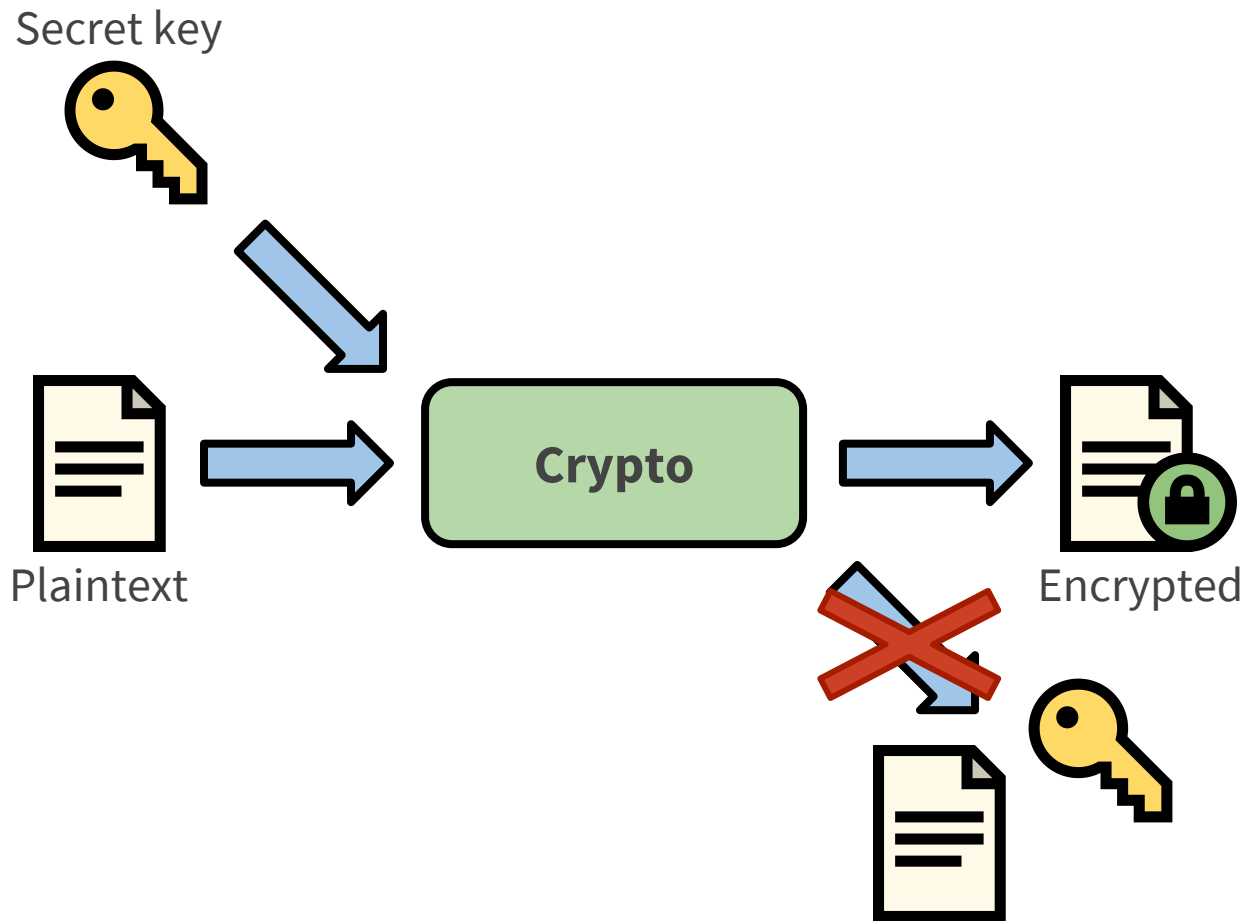


FaCT

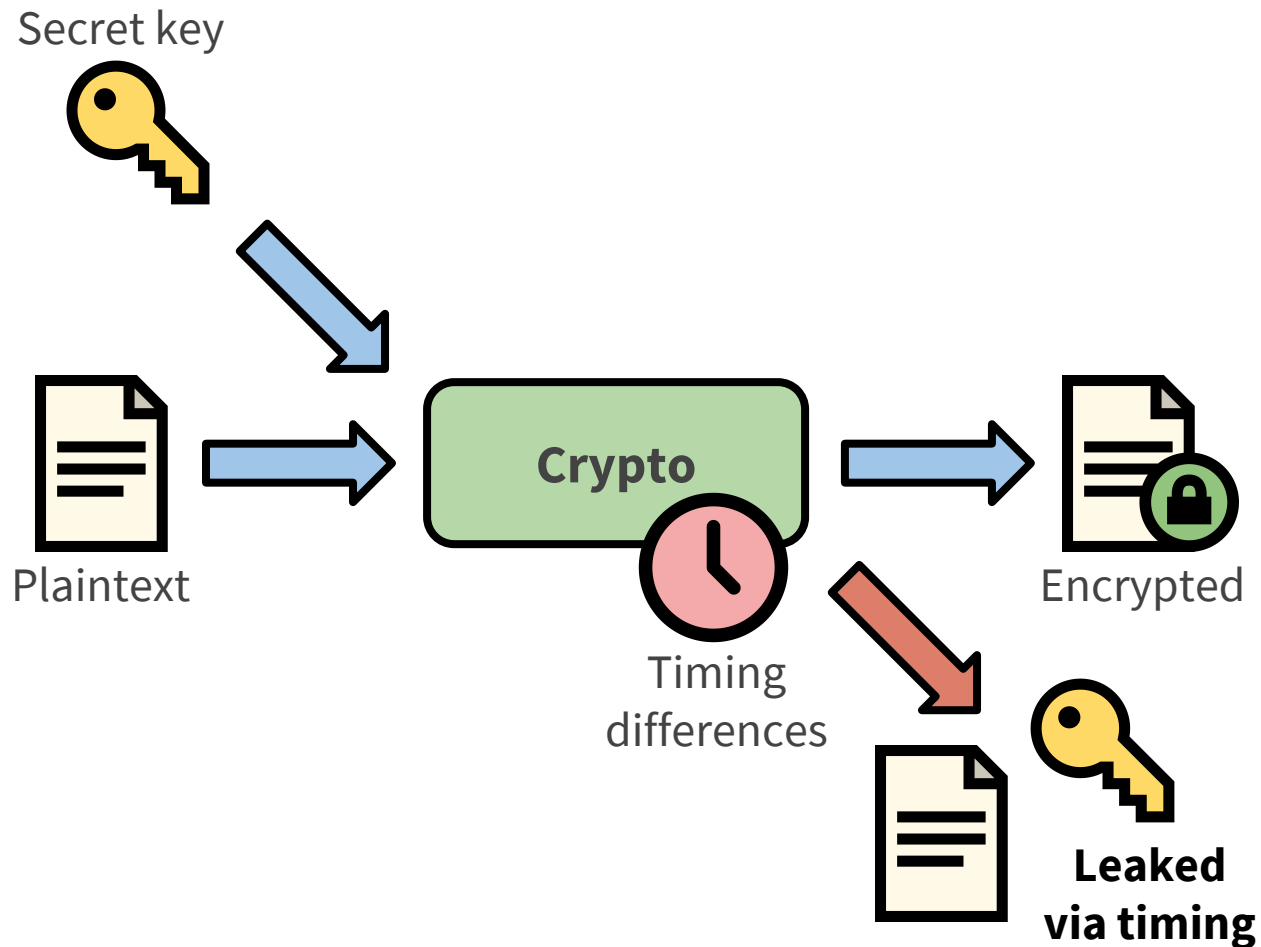
Sunjay Cauligi, Gary Soeller,
Fraser Brown, Brian Johannesmeyer,
Yunlu Huang, Ranjit Jhala, Deian Stefan

A Flexible, Constant-Time
Programming Language

Timing side channels

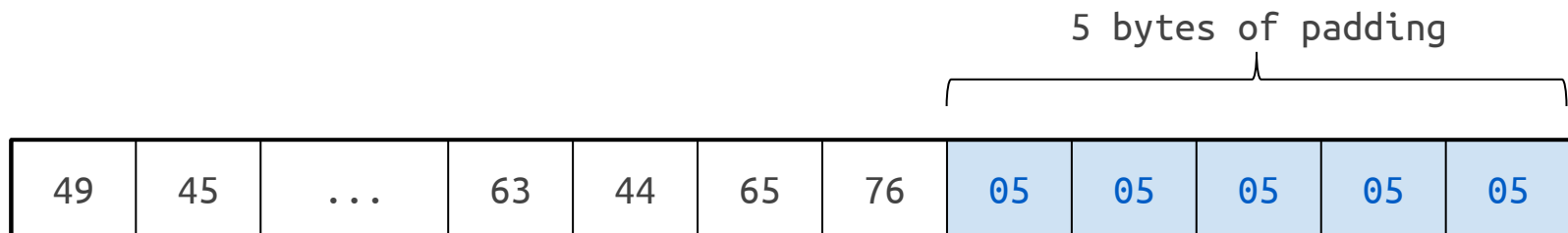


Timing side channels



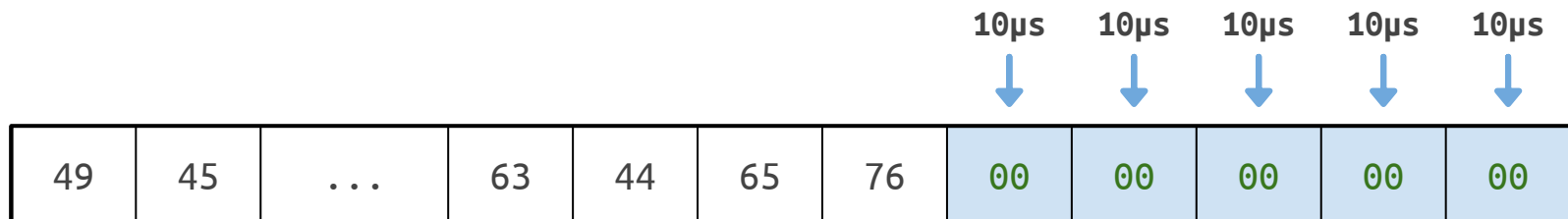
Writing secure code

- Check for valid padding
 - PKCS #7 padding
 - Each padding byte holds length of padding
- Replace padding with null bytes
- Buffer contents should be secret
 - That includes padding!



Writing secure code

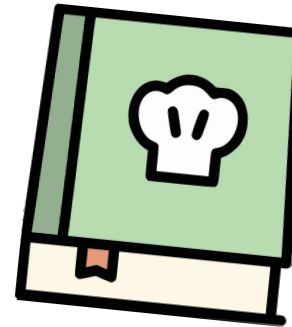
```
int32_t remove_padding(  
    uint8_t* buf,  
    uint32_t buflen) {  
  
    uint8_t padlen = buf[buflen-1];  
    uint32_t i;  
    for (i = 0; i < padlen; i++) {  
        if (buf[buflen-i-1] != padlen)  
            return -1;  
        buf[buflen-i-1] = 0;  
    }  
    return padlen;  
}
```



Writing secure code

```
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;  
}
```

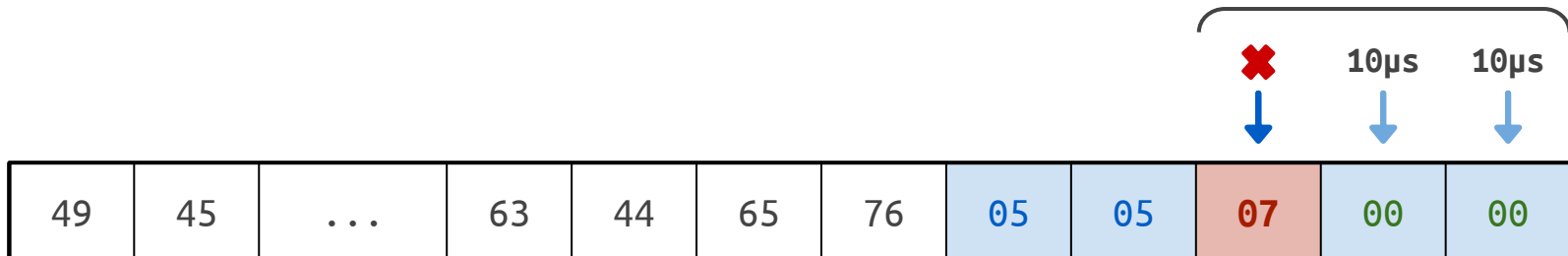
It's dangerous to
return early!



Use this instead.



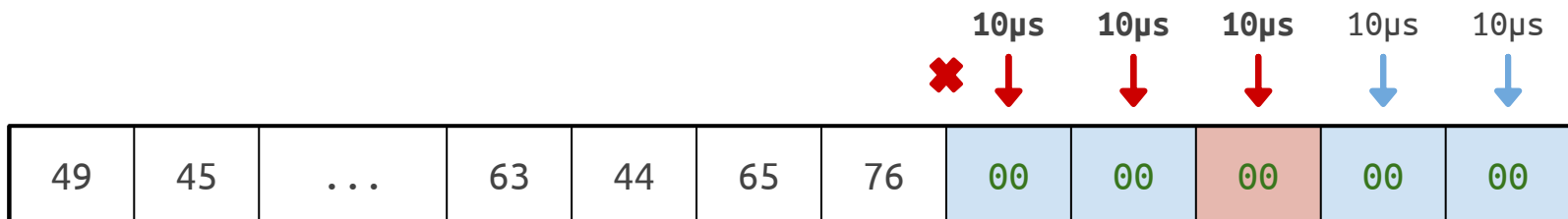
Padding oracle!



Writing secure code

```
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;  
}
```

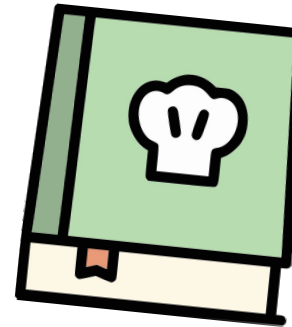
```
int32_t remove_padding2(  
    uint8_t* buf,  
    uint32_t buflen) {  
    uint8_t ok = 1;  
    uint8_t padlen = buf[buflen-1];  
    uint32_t i;  
    for (i = 0; i < padlen; i++) {  
        if (buf[buflen-i-1] != padlen)  
            ok = 0;  
        buf[buflen-i-1] = 0;  
    }  
    return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



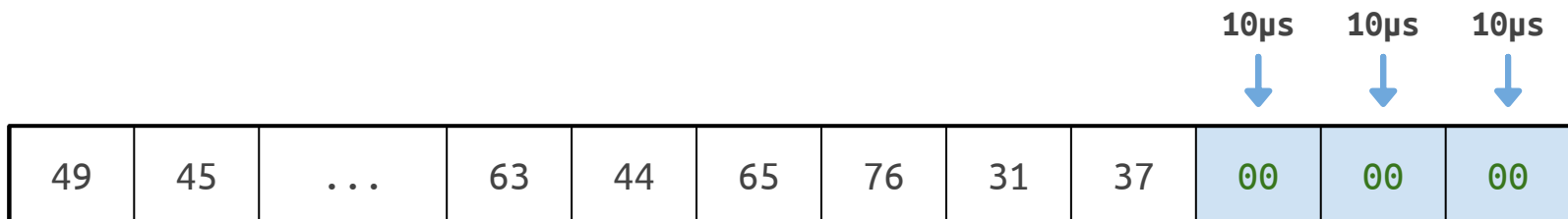
Writing secure code

```
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!



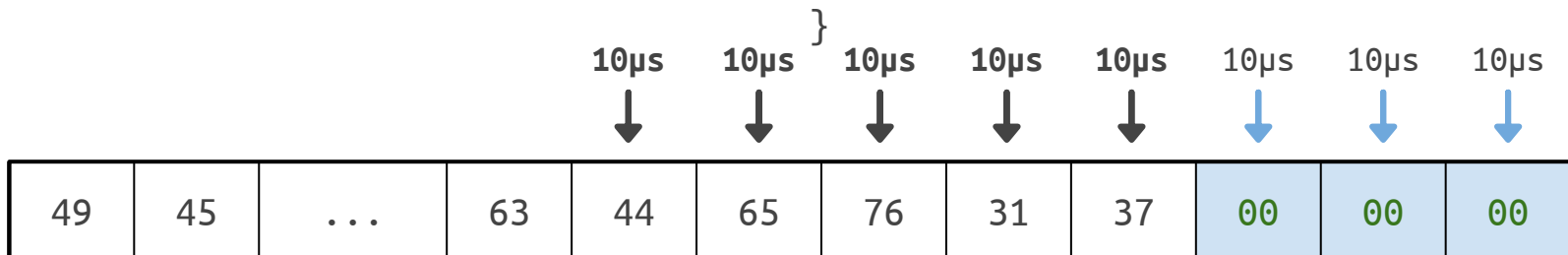
Use this instead.



Writing secure code

```
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;  
}
```

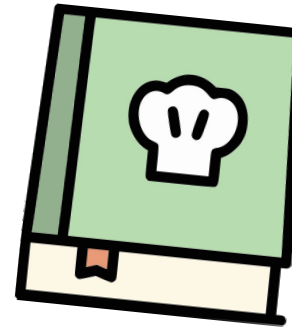
```
int32_t remove_padding3(  
    uint8_t* buf,  
    uint32_t buflen) {  
    uint8_t ok = 1;  
    uint8_t padlen = buf[buflen-1];  
    uint32_t i;  
    for (i = buflen-256; i < buflen; i++) {  
        uint8_t b = buf[i];  
        if (i >= buflen - padlen) {  
            if (b != padlen)  
                ok = 0;  
            b = 0;  
        }  
        buf[i] = b;  
    }  
    return ok ? padlen : -1;  
}
```



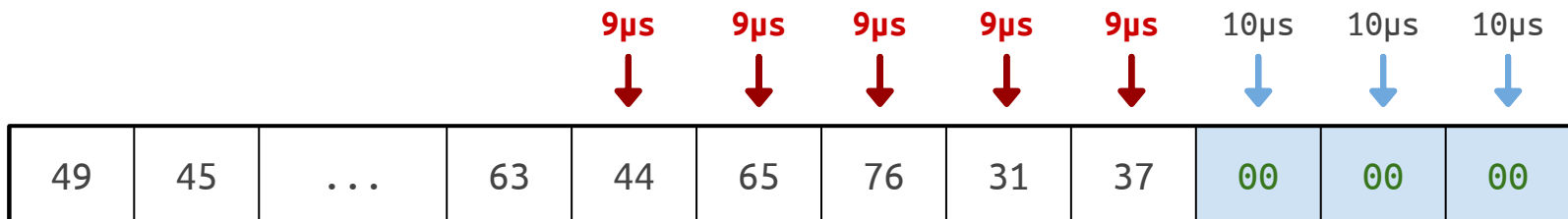
Writing secure code

```
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-256; 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.

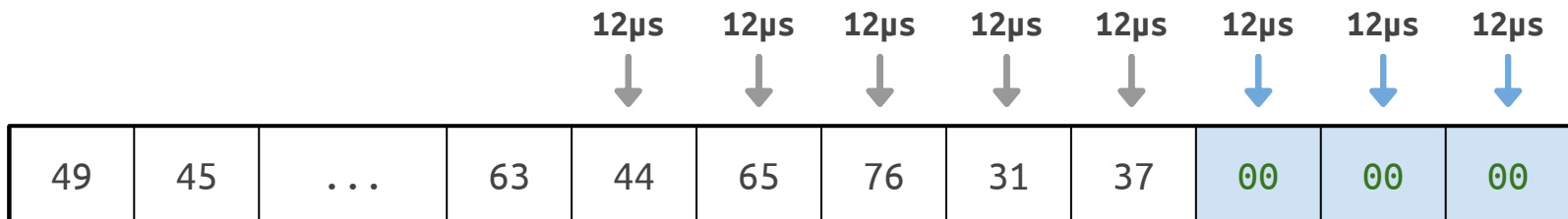


Writing secure code



```
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-256; 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-256; 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;
}
```

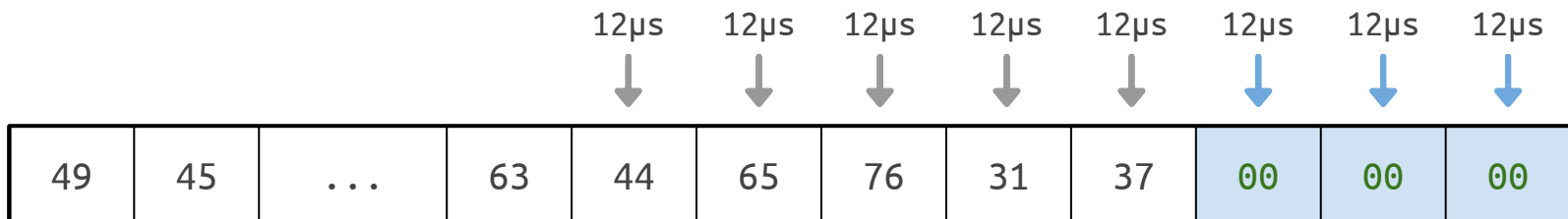


Writing secure code

```
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-256; 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->ssli3_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 /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
464 464 @@ -464,17 +462,25 @@ printf("\n");
465 465 SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
466 466 goto f_err;
467 467
468 468 #else
469 469     goto err;
470 470
471 471 #endif
472 472     decryption_failed_or_bad_record_mac = 1;
473 473 }
474 474
475 475 }
476 476
477 477 rr->length=mac_size;
478 478 s->method->ssli3_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 /* r->length is now just compressed */
494 494 if (s->expand != NULL)
495 495 {
```

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

```
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 -
759 - EVP_MD_CTX_copy_ex(&md_ctx,hash);
760 - EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
761 - EVP_DigestUpdate(&md_ctx,ssl3_pad_2,npad);
762 - EVP_DigestUpdate(&md_ctx,md,md_size);
763 - EVP_DigestFinal_ex(&md_ctx,md,&md_size);
764 -
765 - EVP_MD_CTX_cleanup(&md_ctx);
766 -
767 - if (!isend &&
768 -     EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPHER_CBC_MODE &&
769 -     ssl3_cbc_record_digest_supported(hash))
770 - {
771 -     /* This is a CBC-encrypted record. We must avoid leaking any
772 -      * timing-side channel information about how many blocks of
773 -      * data we are hashing because that gives an attacker a
774 -      * timing-oracle. */
775 -
776 -     /* npad is, at most, 48 bytes and that's with MD5:
777 -      * 16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
778 -      *
779 -      * With SHA-1 (the largest hash speced for SSLv3) the hash size
780 -      * goes up 4, but npad goes down by 8, resulting in a smaller
781 -      * total size. */
782 -     unsigned char header[75];
783 -     unsigned j = 0;
784 -     memcpy(header+j, mac_sec, md_size);
785 -     j += md_size;
786 -     memcpy(header+j, ssl3_pad_1, npad);
787 -     j += npad;
788 -     memcpy(header+j, seq, 8);
789 -     j += 8;
790 -     header[j++] = rec->type;
791 -     header[j++] = rec->length >> 8;
792 -     header[j++] = rec->length & 0xff;
793 -
794 -     ssl3_cbc_digest_record(
795 -         hash,
796 -         md, &md_size,
797 -         header, rec->input,
798 -         rec->length + md_size, rec->orig_len,
799 -         mac_sec, md_size,
800 -         1 /* is SSLv3 */);
801 - }
802 - else
803 - {
804 -     unsigned int md_size_u;
805 -     /* Chop the digest off the end :-> */
806 -     EVP_MD_CTX_init(&md_ctx);
807 -
808 -     EVP_MD_CTX_copy_ex(&md_ctx,hash);
809 -     EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
810 -     EVP_DigestUpdate(&md_ctx,ssl3_pad_1,npad);
811 -     EVP_DigestUpdate(&md_ctx,seq,8);
812 -     rec_char=rec->type;
813 -     EVP_DigestUpdate(&md_ctx,&rec_char,1);
814 -     p=md;
815 -     szn(rec->length,p);
816 -     EVP_DigestUpdate(&md_ctx,md,2);
817 -     EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
```

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 -
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;
```


That's a lot of work, but
even if we get everything right...

Compiler optimizations get in the way

```
/* Return either x or y depending on
   whether bit is set */
uint32_t ct_select_u32(
    uint32_t x,
    uint32_t y,
    uint8_t pred) {
    uint32_t mask = -(!!pred);
    return (mask & x) | (~mask & y);
}
```

```
gcc 5.4: -O2 -m32 -march=i386
xor edx, edx
cmp BYTE PTR [esp+12], 0
setne dl
mov eax, edx
neg eax
and eax, DWORD PTR [esp+4]
dec edx
and edx, DWORD PTR [esp+8]
or eax, edx
ret
```

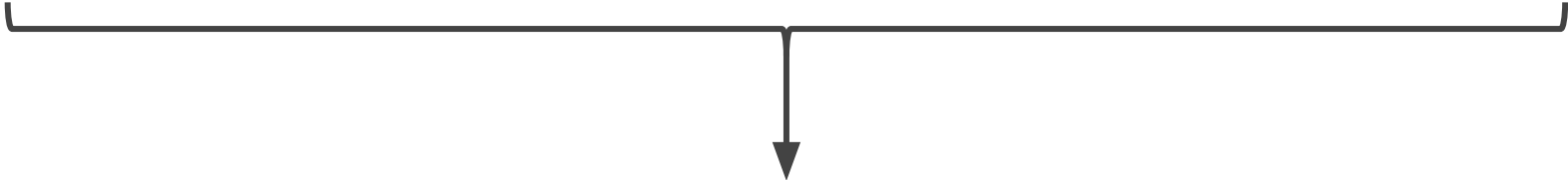
Compiler optimizations get in the way

```
/* Return either x or y depending on
   whether bit is set */
uint32_t ct_select_u32(
    uint32_t x,
    uint32_t y,
    uint8_t pred) {
    uint32_t mask = -(!!pred);
    return (mask & x) | (~mask & y);
}
```

```
clang 3.6: -O2 -m32 -march=i386
    cmp byte ptr [esp + 12], 0
    → jne .LBB0_1
       lea eax, [esp + 8]
       mov eax, dword ptr [eax]
       ret
    .LBB0_1:
       lea eax, [esp + 4]
       mov eax, dword ptr [eax]
       ret
```

Checking up on the compiler

```
word32 u = 0;  
for (i=0; i<1024; i+=cacheLineSize)  
    u &= *(const word32 *)((const void *)(((const byte *)Te)+i));
```

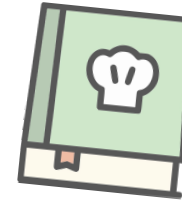


Assembly:

< optimized out >

Checking up on the compiler

```
volatile word32 _u = 0;  
word32 u = _u;  
for (i=0; i<1024; i+=cacheLineSize)  
    u &= *(const word32 *)((const void *)(((const byte *)Te)+i));
```



“...I know **volatile is an abuse under GCC** but
its [sic] usually enough to tame the optimizer

...I don't know [sic] if it's worth the
additional complexity / lack of readability”

We can trick the compiler, but
this semantic gap has a high cost...

Inefficient assembly

`(mask & x) | (~mask & y)`

1.65 cycles

```
and esi, edi
not edi
and r8d, edi
or esi, r8d
```

VS.

0.04 cycles

```
test edi, edi
cmov esi, r8d
```

`lo = lo1 + lo2`

`hi = hi1 + hi2 + (lo >> 31)`

1.01 cycles

```
add edi, esi
mov eax, edi
shr eax, 31
add r8d, r9d
add r8d, eax
```

VS.

0.13 cycles

```
add edi, esi
adc r8d, r9d
```

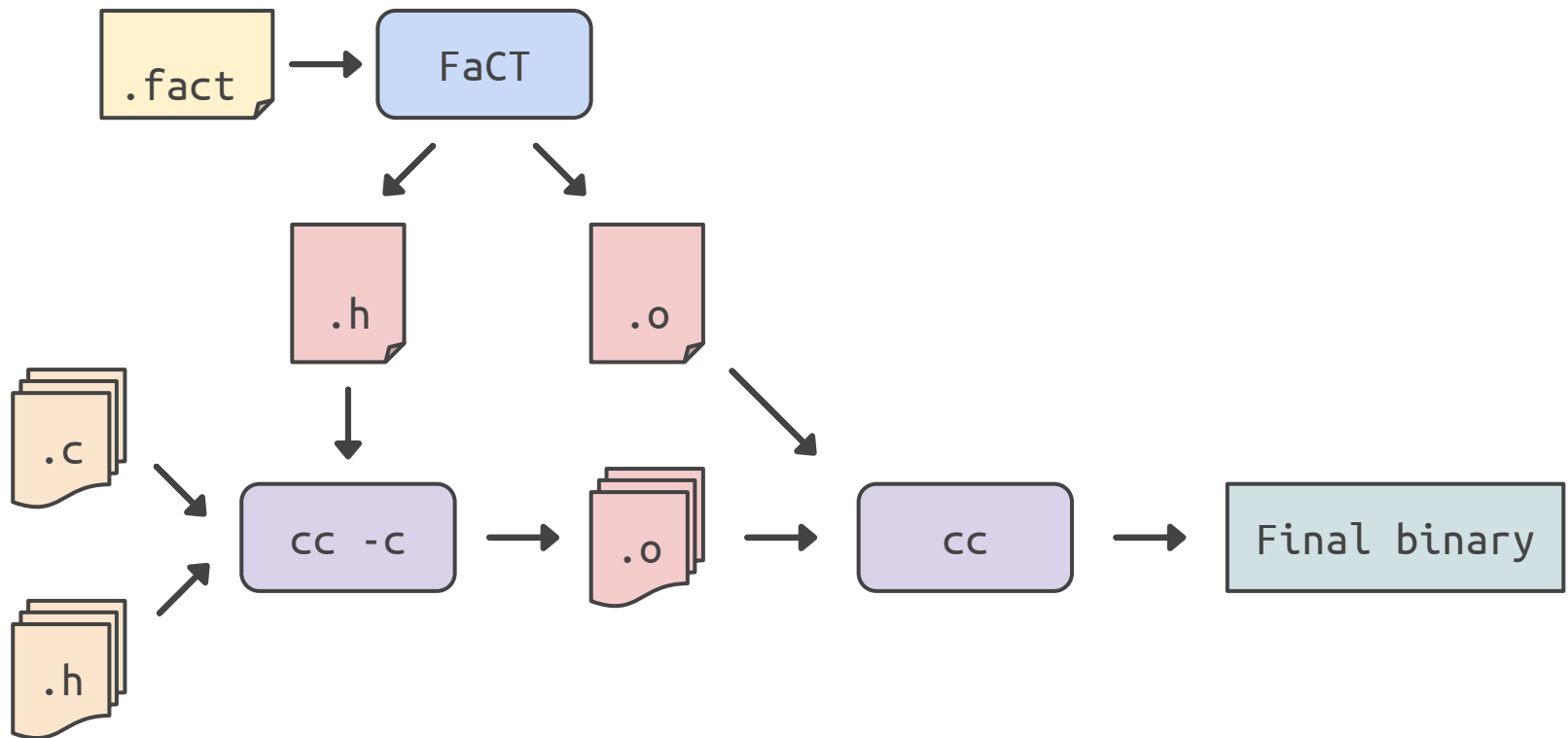
Constant problems with constant-time

- Can't use standard programming constructs
 - Manually keep track of secret vs. public
 - Write obfuscated code for computation on secrets
 - Difficult to write such code correctly
- Fighting the compiler
 - Need to prevent optimizer from undermining you
 - But now you don't produce efficient assembly
- Hard to maintain

We need a new language

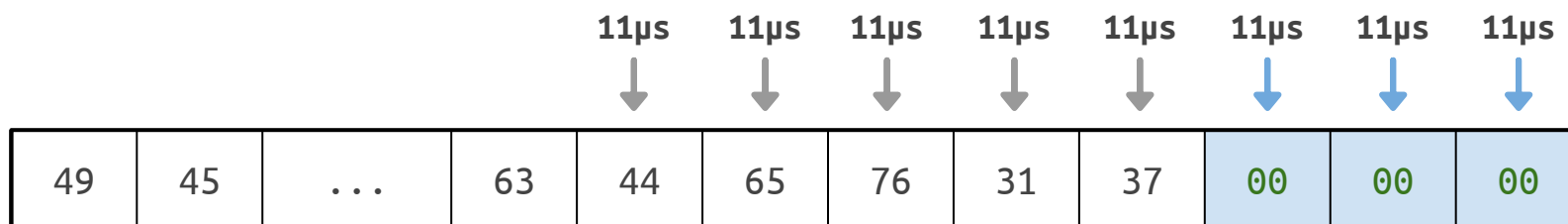
- Write clear code for computation on secrets
 - Helps you keep track of secrets vs. public values
 - Lets you use standard programming constructs
 - Ensures you write *correct* code
- Compiler that helps instead of hurts
 - Optimize your code as much as possible
 - But ensure code remains constant-time
- Simple to work with

FaCT

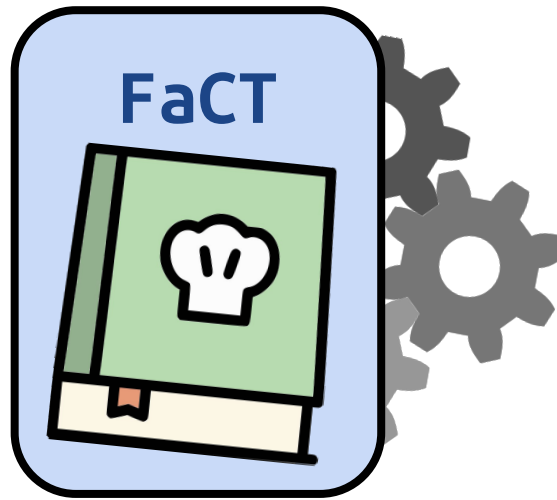


What does FaCT look like?

```
secret int32 remove_padding(secret mut uint8[] buf) {  
    uint8 padlen = buf[len buf - 1];  
    for (uint32 i from len buf - 256 to len buf) {  
        if (i >= len buf - padlen) {  
            if (buf[i] != padlen) {  
                return -1;  
            }  
            buf[i] = 0;  
        }  
    }  
    return padlen;  
}
```



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;  
}
```



```
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;
```



```
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

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



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

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



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

Useful language primitives

Add-with-carry

```
sum, carry = value1 + value2;
```

Byte packing

```
large_word = pack(a, b, c, d);
```

Byte unpacking

```
a, b, c, d = unpack(large_word);
```

Bit rotation

```
rotate_l = word <<< n;  
rotate_r = word >>> n;
```

Useful language primitives

Parallel vector types

```
type uint8x4 = uint8[4];
```

Vector operations


```
vec1 += vec2;  
vec1 ^= vec2;
```

Vector operations
with saturation

```
vec1 .+= vec2;  
vec1 .*= vec2;
```

Labels ensure proper transformations

- IFC to determine what is secret/public
 - Only transform secret computation
- Prevent secret expressions we can't transform
 - Loop bounds



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

Labels ensure proper transformations

- IFC to determine what is secret/public
 - Only transform secret computation
- Prevent secret expressions we can't transform
 - Loop bounds

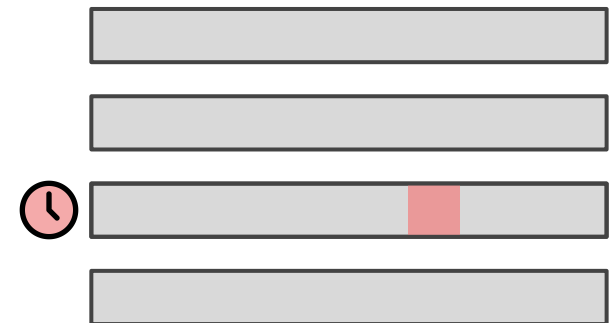
```
for (uint32 i from 0 to public_value) {  
    if (i < secret_value) {  
        do_operation();  
    }  
}
```

Labels ensure proper transformations

- IFC to determine what is secret/public
 - Only transform secret computation
- Prevent secret expressions we can't transform
 - Loop bounds
 - Array indices

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

Cache lines

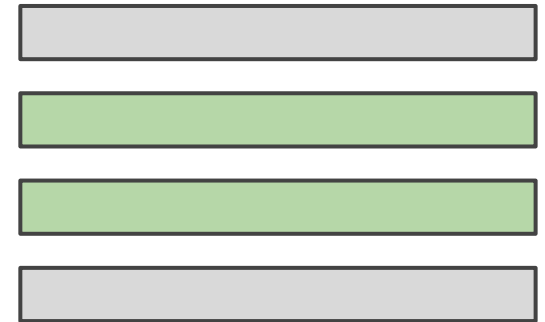


Labels ensure proper transformations

- IFC to determine what is secret/public
 - Only transform secret computation
- 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 smarter optimizations

- Public computations are fully optimized
 - It's public so make it as fast as possible
- Secrets are optimized safely
 - Only run specific LLVM optimization passes
 - No optimization passes that reintroduce leaks

Labels ensure constant-time code

- Use ct-verif¹ to verify constant-time
 - Pass annotated LLVM to ct-verif
- Use Z3 to prevent memory and arithmetic errors
 - Generate constraints while type checking
- Incorporated into FaCT compiler

¹Almeida et al. “Verifying constant-time implementations.” USENIX Security 2016.

FaCT

- DSL for constant-time code
- Compiler works with you, not against you
- Easily fits into your existing toolchain

