

Constant-time programming in FaCT

Deian Stefan

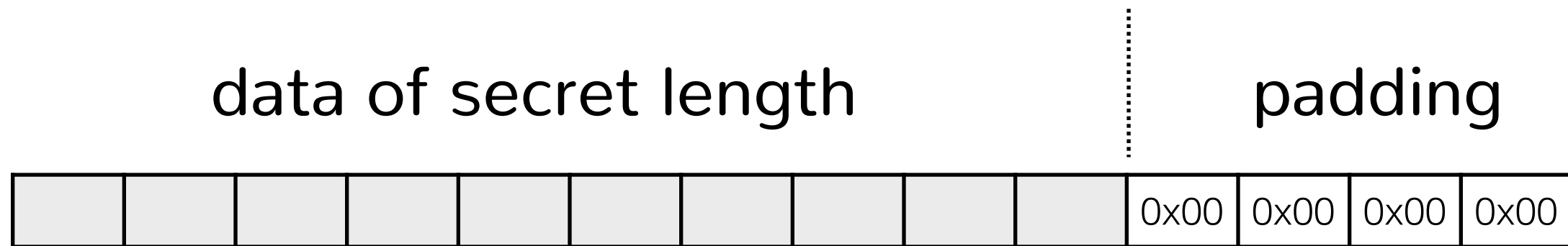
UC San Diego

Fraser Brown, Sunjay Cauligi, Ranjit Jhala, Brian Johannismeyer,
John Renner, Gary Soeller, Riad Wahby, Conrad Watt

What do we mean by constant-time?

time of computation should not depend on secret data

An example from mbedTLS



An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe?

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

A more complex example

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if (done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

A more complex example

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if (done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe?

A more complex example

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        *data_len = ct_select(i, *data_len, done & !prev_done);
    }

    return 0;
}
```

A more complex example

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        *data_len = ct_select(i, *data_len, done & !prev_done);
    }

    return 0;
}
```

Is this safe?

How are we constraining our programs?

- No variable-time instructions
 - No div, mod, floating-point
- No control flow based on secret data
 - No if-statements, short circuit operators, switch
 - No loops, procedure calls, early returns
- No memory access based on data

How are we writing these programs?

Folding control flow into data!

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the

Structured Programming with go to Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

Keywords and phrases: structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

CR categories: 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)



Error prone in practice

OpenSSL padding oracle attack

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

```
383 383 SSL3_RECORD *rr;
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 + int decryption_failed_or_bad_record_mac = 0;
386 387
387 388
388 389 rr = &(s->s3->rrec);
389 389
389 390 dtls1_process_record(SSL *s)
417 418 enc_err = s->method->ssl3_enc->enc(s,0);
418 419 if (enc_err <= 0)
419 420 {
420 - /* decryption failed, silently discard message */
421 - if (enc_err < 0)
422 - {
423 - rr->length = 0;
424 - s->packet_length = 0;
425 - }
426 - goto err;
421 + /* To minimize information leaked via timing, we will always
422 + * perform all computations before discarding the message.
423 + */
424 + decryption_failed_or_bad_record_mac = 1;
427 425 }
428 426
429 427 #ifdef TLS_DEBUG
429 430 dtls1_debug(SSL *s, "DTLS1: Processing record\n");
453 451 SSLerr(SSL_F_DTLS1_PROCESS_RECORD, SSL_R_PRE_MAC_LENGTH_TOO_LONG);
454 452 goto f_err;
455 453 #else
456 - goto err;
454 + decryption_failed_or_bad_record_mac = 1;
457 455 #endif
458 456 }
459 457 /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
464 462 SSLerr(SSL_F_DTLS1_PROCESS_RECORD, SSL_R_LENGTH_TOO_SHORT);
465 463 goto f_err;
466 464 #else
467 - goto err;
465 + decryption_failed_or_bad_record_mac = 1;
468 466 #endif
469 467 }
470 468 rr->length -= mac_size;
471 469 i = s->method->ssl3_enc->mac(s,md,0);
472 470 if (1 < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
473 471 {
474 - goto err;
472 + decryption_failed_or_bad_record_mac = 1;
475 473 }
476 474 }
477 475
476 + if (decryption_failed_or_bad_record_mac)
477 + {
478 + /* decryption failed, silently discard message */
479 + rr->length = 0;
480 + s->packet_length = 0;
481 + goto err;
482 + }
483 +
478 484 /* r->length is now just compressed */
479 485 if (s->expand != NULL)
480 486 {
```


Error prone in practice

OpenSSL padding oracle attack

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

```

384 384      unsigned int mac_size;
385 385      unsigned char md[EVP_MAX_MD_SIZE];
386 +      int decryption_failed_or_bad_record_mac = 0;
387
387 387      EVP_DigestUpdate(&md_ctx, md, 2);
388 388      EVP_DigestUpdate(&md_ctx, rec->input, rec->length);
389 389      EVP_DigestFinal_ex(&md_ctx, md, NULL);
390
417 418      EVP_MD_CTX_copy_ex(&md_ctx, hash);
418 419      EVP_DigestUpdate(&md_ctx, mac_sec, md_size);
419 420      EVP_DigestUpdate(&md_ctx, ssl3_pad_2, npad);
420 421      EVP_DigestUpdate(&md_ctx, md, md_size);
421 422      EVP_DigestFinal_ex(&md_ctx, md, &md_size);
422 423      EVP_MD_CTX_cleanup(&md_ctx);
423 424
424 425      if (!send &&
425 426          EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPHER_CBC_MODE &&
426 427          ssl3_cbc_record_digest_supported(hash))
427 428      {
428 429          /* This is a CBC-encrypted record. We must avoid leaking any
429 430             * timing-side channel information about how many blocks of
430 431             * data we are hashing because that gives an attacker a
431 432             * timing-oracle. */
432 433          npad is, at most, 48 bytes and that's with MD5:
433 434          * 16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
434 435          *
435 436          * With SHA-1 (the largest hash spec'd for SSLv3) the hash size
436 437          * goes up 4, but npad goes down by 8, resulting in a smaller
437 438          * total size. */
438 439          unsigned char header[75];
439 440          unsigned j = 0;
440 441          memcpy(header+j, mac_sec, md_size);
441 442          j += md_size;
442 443          memcpy(header+j, ssl3_pad_1, npad);
443 444          j += npad;
444 445          memcpy(header+j, seq, 8);
445 446          j += 8;
446 447          header[j++] = rec->type;
447 448          header[j++] = rec->length >> 8;
448 449          header[j++] = rec->length & 0xff;
449 450
450 451          ssl3_cbc_digest_record(
451 452              hash,
452 453              md, &md_size,
453 454              header, rec->input,
454 455              rec->length + md_size, rec->orig_len,
455 456              mac_sec, md_size,
456 457              1 /* is SSLv3 */);
457 458      }
458 459      else
459 460      {
460 461          unsigned int md_size_u;
461 462          /* Chop the digest off the end :-> */
462 463          EVP_MD_CTX_init(&md_ctx);
463 464          EVP_MD_CTX_copy_ex(&md_ctx, hash);
464 465          EVP_DigestUpdate(&md_ctx, mac_sec, md_size);
465 466          EVP_DigestUpdate(&md_ctx, ssl3_pad_1, npad);
466 467          EVP_DigestUpdate(&md_ctx, seq, 8);
467 468          rec_char = rec->type;
468 469          EVP_DigestUpdate(&md_ctx, &rec_char, 1);
469 470          p = md;
470 471          s2n(rec->length, p);
471 472          EVP_DigestUpdate(&md_ctx, md, 2);
472 473          EVP_DigestUpdate(&md_ctx, rec->input, rec->length);

```

Error prone in practice

OpenSSL padding oracle attack

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

Lucky 13 timing attack

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

```
383 383 SSL_RECORD *rr;
384 384 unsigned int mac_size;
385 385 unsigned char md[EVP_MAX_MD_SIZE];
386 + int decryption_failed_or_bad_record_mac = 0;

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);

734 + if (!send &&
735 +     EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
736 +     ssl3_cbc_record_digest_supported(hash))
737 + {
738 +     /* This is a CBC-encrypted record. We must avoid leaking any
739 +      * timing-side channel information about how many blocks of
740 +      * data we are hashing because that gives an attacker a
741 +      * timing-oracle. */
742 +
743 +     /* npad is, at most, 48 bytes and that's with MD5:
744 +      * 16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
745 +      *
746 +      * With SHA-1 (the largest hash speed for SSLv3) the hash size
747 +      * goes up 4, but npad goes down by 8, resulting in a smaller
748 +      * total size. */
749 +     unsigned char header[75];
750 +     unsigned j = 0;
751 +     memcpy(header+j, mac_sec, md_size);
752 +     j += md_size;
753 +     memcpy(header+j, ssl3_pad_1, npad);
754 +     j += npad;
755 +     memcpy(header+j, seq, 8);
756 +     j += 8;
757 +     header[j++] = rec->type;
758 +     header[j++] = rec->length >> 8;
759 +     header[j++] = rec->length & 0xff;
760 +
761 +     ssl3_cbc_digest_record(
762 +         hash,
763 +         md, &md_size,
764 +         header, rec->input,
765 +         rec->length + md_size, rec->orig_len,
766 +         mac_sec, md_size,
767 +         1 /* is SSLv3 */);
768 + }
769 + else
770 + {
771 +     unsigned int md_size_u;
772 +     /* Chop the digest off the end :- */
773 +     EVP_MD_CTX_init(&md_ctx);
774 +
775 +     EVP_MD_CTX_copy_ex( &md_ctx,hash);
776 +     EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
777 +     EVP_DigestUpdate(&md_ctx,ssl3_pad_1,npad);
778 +     EVP_DigestUpdate(&md_ctx,seq,8);
779 +     rec_char=rec->type;
780 +     EVP_DigestUpdate(&md_ctx,&rec_char,1);
781 +     p=md;
782 +     s2n(rec->length,p);
783 +     EVP_DigestUpdate(&md_ctx,md,2);
784 +     EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
```

Error prone in practice

OpenSSL padding oracle attack

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

Lucky 13 timing attack

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

```

384 384      unsigned int mac_size;
385 385      unsigned char md[EVP_MAX_MD_SIZE];
386 386      + int decryption_failed_or_bad_record_mac = 0;
387 387
388 388      - EVP_DigestUpdate(&md_ctx,md,2);
389 389      - EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
390 390      - EVP_DigestFinal_ex( &md_ctx,md,NULL);
391 391
392 392      -
393 393      -
394 394      -
395 395      -
396 396      -
397 397      -
398 398      -
399 399      -
400 400      -
401 401      -
402 402      -
403 403      -
404 404      -
405 405      -
406 406      -
407 407      -
408 408      -
409 409      -
410 410      -
411 411      -
412 412      -
413 413      -
414 414      -
415 415      -
416 416      -
417 417      -
418 418      -
419 419      -
420 420      -
421 421      -
422 422      -
423 423      -
424 424      -
425 425      -
426 426      -
427 427      -
428 428      -
429 429      -
430 430      -
431 431      -
432 432      -
433 433      -
434 434      -
435 435      -
436 436      -
437 437      -
438 438      -
439 439      -
440 440      -
441 441      -
442 442      -
443 443      -
444 444      -
445 445      -
446 446      -
447 447      -
448 448      -
449 449      -
450 450      -
451 451      -
452 452      -
453 453      -
454 454      -
455 455      -
456 456      -
457 457      -
458 458      -
459 459      -
460 460      -
461 461      -
462 462      -
463 463      -
464 464      -
465 465      -
466 466      -
467 467      -
468 468      -
469 469      -
470 470      -
471 471      -
472 472      -
473 473      -
474 474      -
475 475      -
476 476      -
477 477      -
478 478      -
479 479      -
480 480      -
481 481      -
482 482      -
483 483      -
484 484      -
485 485      -
486 486      -
487 487      -
488 488      -
489 489      -
490 490      -
491 491      -
492 492      -
493 493      -
494 494      -
495 495      -
496 496      -
497 497      -
498 498      -
499 499      -
500 500      -
501 501      -
502 502      -
503 503      -
504 504      -
505 505      -
506 506      -
507 507      -
508 508      -
509 509      -
510 510      -
511 511      -
512 512      -
513 513      -
514 514      -
515 515      -
516 516      -
517 517      -
518 518      -
519 519      -
520 520      -
521 521      -
522 522      -
523 523      -
524 524      -
525 525      -
526 526      -
527 527      -
528 528      -
529 529      -
530 530      -
531 531      -
532 532      -
533 533      -
534 534      -
535 535      -
536 536      -
537 537      -
538 538      -
539 539      -
540 540      -
541 541      -
542 542      -
543 543      -
544 544      -
545 545      -
546 546      -
547 547      -
548 548      -
549 549      -
550 550      -
551 551      -
552 552      -
553 553      -
554 554      -
555 555      -
556 556      -
557 557      -
558 558      -
559 559      -
560 560      -
561 561      -
562 562      -
563 563      -
564 564      -
565 565      -
566 566      -
567 567      -
568 568      -
569 569      -
570 570      -
571 571      -
572 572      -
573 573      -
574 574      -
575 575      -
576 576      -
577 577      -
578 578      -
579 579      -
580 580      -
581 581      -
582 582      -
583 583      -
584 584      -
585 585      -
586 586      -
587 587      -
588 588      -
589 589      -
590 590      -
591 591      -
592 592      -
593 593      -
594 594      -
595 595      -
596 596      -
597 597      -
598 598      -
599 599      -
600 600      -
601 601      -
602 602      -
603 603      -
604 604      -
605 605      -
606 606      -
607 607      -
608 608      -
609 609      -
610 610      -
611 611      -
612 612      -
613 613      -
614 614      -
615 615      -
616 616      -
617 617      -
618 618      -
619 619      -
620 620      -
621 621      -
622 622      -
623 623      -
624 624      -
625 625      -
626 626      -
627 627      -
628 628      -
629 629      -
630 630      -
631 631      -
632 632      -
633 633      -
634 634      -
635 635      -
636 636      -
637 637      -
638 638      -
639 639      -
640 640      -
641 641      -
642 642      -
643 643      -
644 644      -
645 645      -
646 646      -
647 647      -
648 648      -
649 649      -
650 650      -
651 651      -
652 652      -
653 653      -
654 654      -
655 655      -
656 656      -
657 657      -
658 658      -
659 659      -
660 660      -
661 661      -
662 662      -
663 663      -
664 664      -
665 665      -
666 666      -
667 667      -
668 668      -
669 669      -
670 670      -
671 671      -
672 672      -
673 673      -
674 674      -
675 675      -
676 676      -
677 677      -
678 678      -
679 679      -
680 680      -
681 681      -
682 682      -
683 683      -
684 684      -
685 685      -
686 686      -
687 687      -
688 688      -
689 689      -
690 690      -
691 691      -
692 692      -
693 693      -
694 694      -
695 695      -
696 696      -
697 697      -
698 698      -
699 699      -
700 700      -
701 701      -
702 702      -
703 703      -
704 704      -
705 705      -
706 706      -
707 707      -
708 708      -
709 709      -
710 710      -
711 711      -
712 712      -
713 713      -
714 714      -
715 715      -
716 716      -
717 717      -
718 718      -
719 719      -
720 720      -
721 721      -
722 722      -
723 723      -
724 724      -
725 725      -
726 726      -
727 727      -
728 728      -
729 729      -
730 730      -
731 731      -
732 732      -
733 733      -
734 734      -
735 735      -
736 736      -
737 737      -
738 738      -
739 739      -
740 740      -
741 741      -
742 742      -
743 743      -
744 744      -
745 745      -
746 746      -
747 747      -
748 748      -
749 749      -
750 750      -
751 751      -
752 752      -
753 753      -
754 754      -
755 755      -
756 756      -
757 757      -
758 758      -
759 759      -
760 760      -
761 761      -
762 762      -
763 763      -
764 764      -
765 765      -
766 766      -
767 767      -
768 768      -
769 769      -
770 770      -
771 771      -
772 772      -
773 773      -
774 774      -
775 775      -
776 776      -
777 777      -
778 778      -
779 779      -
780 780      -
781 781      -
782 782      -
783 78
```

Error prone in practice

OpenSSL padding oracle attack

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

Lucky 13 timing attack

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

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

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

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

421 421
422 422
423 423
424 424

427 425
428 426
429 427

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

464 462
465 463
466 464

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

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

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

290 + #if 1
291 + len -= SHA_DIGEST_LENGTH; /* amend mac */
292 + if (len>=(256+SHA_CBLOCK)) {
293 + j = (len-(256+SHA_CBLOCK))&(0-SHA_CBLOCK);
294 + j += SHA_CBLOCK-key->md.num;
295 + SHA1_Update(&key->md,out+iv,j);
296 + }
```

Error prone in practice

OpenSSL padding oracle attack

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

Lucky 13 timing attack

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

CVE-2016-2107

Somorovsky. "Curious padding oracle in OpenSSL."

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

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

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

238 248 /* decrypt HMAC(padding at once) */
239 249 aesni_cbc_encrypt(in,out,len,
240 250 &key->ks,ctx->iv,0);
241 251
242 252 if (plen) { /* "TLS" mode of operation */
243 253 /* figure out payload length */
244 254 if (len<(size_t)(out[len-1]*1+SHA_DIGEST_LENGTH))
245 255 return 0;
246 256
247 257 len -= (out[len-1]*1+SHA_DIGEST_LENGTH);
248 258
249 259 size_t inp_len, mask, j, i;
250 260 unsigned int res, maxpad, pad, bitlen;
251 261 int ret = 1;
252 262 union { unsigned int u[SHA_LBLOCK];
253 263 unsigned char c[SHA_CBLOCK]; }
254 264 *data = (void *)key->md.data;

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

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

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

258 262 /* calculate HMAC and verify it */
259 263 key->aux.tls_aad[plen-2] = inp_len>>8;
260 264 key->aux.tls_aad[plen-1] = inp_len;
261 265
262 266 /* calculate HMAC */
263 267 key->md = key->head;
264 268 SHA1_Update(&key->md,key->aux.tls_aad,plen);
265 269 SHA1_Update(&key->md,out+iv,len);
266 270 SHA1_Final(mac,&key->md);

290 +if 1
291 + len -= SHA_DIGEST_LENGTH; /* amend mac */
292 + if (len>=(256+SHA_CBLOCK)) {
293 + j = (len-(256+SHA_CBLOCK))&(0-SHA_CBLOCK);
294 + j += SHA_CBLOCK-key->md.num;
295 + SHA1_Update(&key->md,out+iv+1);
```

How can secure compilation help?

Can design language for writing crypto code and compiler to ensure that generated code is constant-time!

FaCT

- Domain specific language for writing CT code
 - Write your program in C, Python, Haskell
 - Write your constant-time parts in FaCT
- Secure compilation infrastructure
 - Generates constant time assembly
 - Generates glue code for C, Python and Haskell

Consider the following C program

```
int main() {  
    uint8_t cond = 1;  
    uint32_t x = 42;  
    uint32_t y = 137;  
  
    printf("cond: %u, x: %u, y: %u\n", cond, x, y);  
  
    conditional_swap(&x, &y, cond); // do it in CT  
  
    printf("after swap:\n");  
    printf("cond: %u, x: %u, y: %u\n", cond, x, y);  
  
    return 0;  
}
```

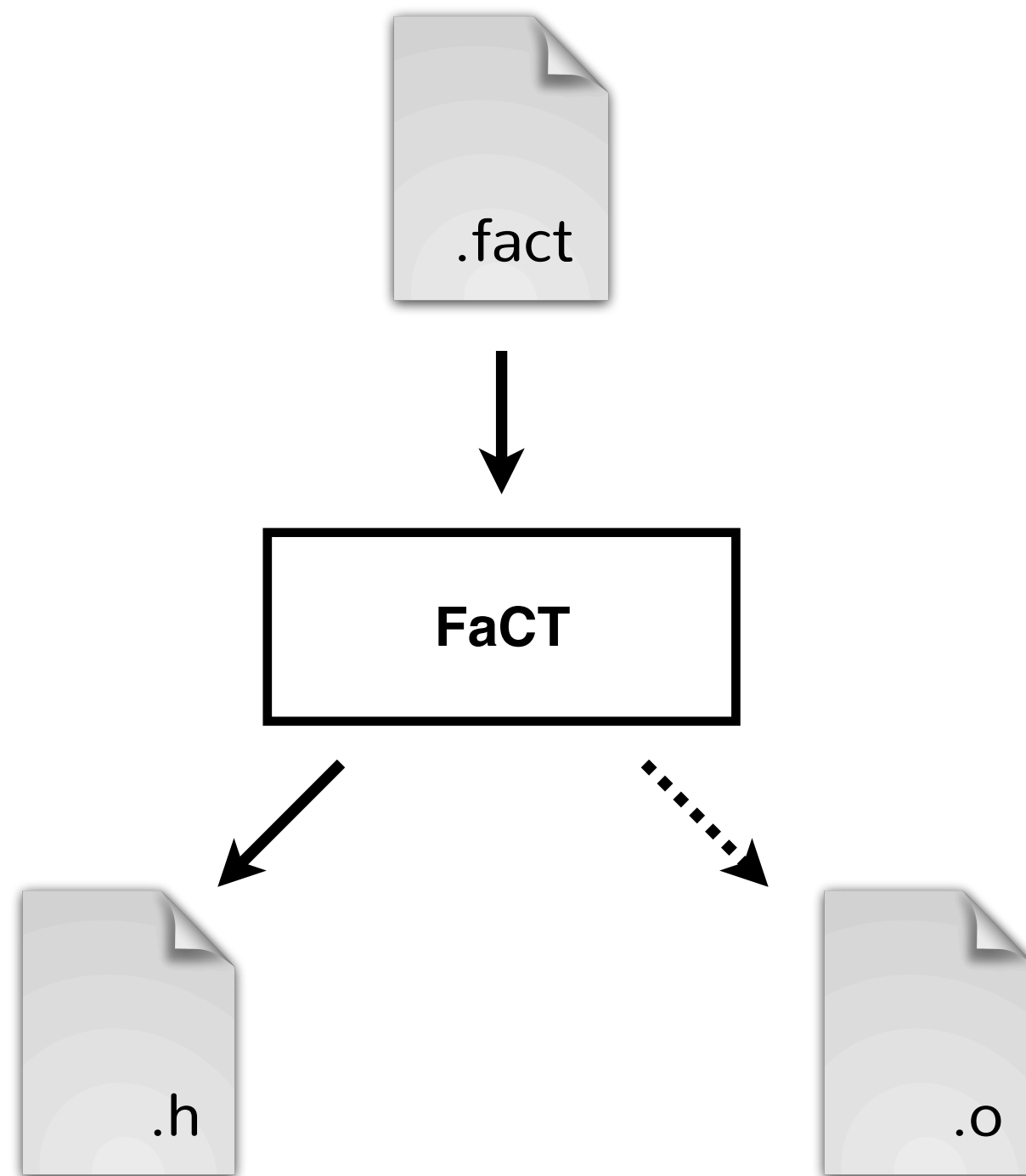


```
void conditional_swap(uint32_t* x,  
                     uint32_t* y,  
                     bool cond) {  
    if (cond) {  
        uint32_t tmp = *x;  
        *x = *y;  
        *y = tmp;  
    }  
}
```

(in C, unsafe)

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

(in FaCT, safe)



What's in the .h?

```
#ifndef __HELLO_WORLD_H
#define __HELLO_WORLD_H

void conditional_swap(
    /*secret*/ uint32_t * x,
    /*secret*/ uint32_t * y,
    /*secret*/ uint8_t cond);

#endif
```

What's in the .o?

```
define void
@conditional_swap(i32* %_secret_x, i32* %_secret_y, i1 %_secret_cond1) {
entry:
...
}
```

What do FaCT functions look like?

If you squint... kind of looks like C code

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

What do FaCT functions look like?

If you squint... kind of looks like C code

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

What do FaCT functions look like?

- C-like
 - statements + expressions
 - block scoping, local variables
 - C-like data types
 - C-like control flow constructs: if-statement, for-loops

What do FaCT functions look like?

- With some differences...
 - No floating point operations
 - No types that have implicit bit-width
 - No raw pointers
 - No heap allocation

How does FaCT help with CT?

- Mutability is explicit!
 - By default variables are constant
 - Must declare variable `mut` to mutate variables!
- E.g.,
 - Function args: `... fun(public mut uint32 y) { ...`
 - Local variables: `secret mut uint8[20] x = ...`

Secrecy is explicit!

- Every value must be labeled secret/public

- Function arguments and return values:

```
public int32  
conditional_assign(secret mut uint8[] x,  
secret uint8[] y, secret bool assign) { ... }
```

- Local variables:

```
secret mut uint8[20] local = arrzeros(20);
```

- FaCT propagates labels

- E.g., secret_x + public_y is labeled secret

How do labels help?

What introduces timing channels?

- Variable-time operators
 - E.g., /, %, ||, &&
- Control flow
 - If-statements, for-loops, early returns, function calls
- Memory access patterns
 - E.g., accessing memory based at secret index

Labels to the rescue!

- Labels are used to prevent information leaks
 - At compile time, label/type checking algorithm ensures that you cannot leak data labeled secret
 - E.g., the type checker disallows explicit assignment of `secret` data to `public` variables

How can we use labels?

- Restrict usage of variable-time operators
 - No secret operands to %, /, ||, or &&
- Restrict unsafe control flow
 - No branching on secrets, no secret-bounded loops
- Disallow leaky memory access patterns
 - No indexing based on secret data

Expressiveness :(

Can we do better?

- Yes! We can automatically transform statements that handle secrets to be constant time
 - Not everything, but many things!

First: What does FaCT disallow?

No public assignments (in secret context)

```
if (secret)  
    pub = ...
```

No calls to functions with public side-effects

```
if (secret)  
    fun(ref pub);
```

No % or / on secret data!

sec % pub, pub % sec, sec₁ % sec₂

No secret-bounded loops

```
for (uint32 i=0; i < sec; i+=1) {  
    ...  
}
```

No memory access at secret index

`arr[sec]`

What about everything else?

- Can use short circuit operators || and &&
- Can have control flow depend on secrets
 - E.g., if-statements, return, function calls

Compiler automatically transforms code

- If-statements transformed to execute both branches
 - Goal: preserve semantics of normal if-statement
 - Approach: keep track of control flow via a local variable (for each branch)

```
[[ if (cond) {  
    s1;  
} else {  
    s2;  
}  
]
```



```
secret mut bool __branch1 = cond;  
[[s1;]]  
__branch1 = !__branch1;  
[[s2;]]
```

Back to example

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



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

What about early returns?

- Goal: preserve semantics of early returns
- Approach:
 - keep track of control flow via a local variable
 - keep track of return value

```
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;
```

What about function calls?

- 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(secret mut uint32 x) {  
    x = 42;  
}
```



```
void fn(secret mut uint32 x, bool state) {  
    x = ct_select(state, 42, x);  
}
```

Back to mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                              size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if (input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

(in C, unsafe)

Back to mbedTLS

```
export
void get_zeros_padding( secret uint8 input[], secret mut uint32 data_len)
{
    data_len = 0;
    for( uint32 i = len input; i > 0; i-=1 ) {
        if (input[i-1] != 0) {
            data_len = i;
            return;
        }
    }
}
```

(in FaCT, safe)

What makes transformations hard?

```
if (secret) {  
    arr[32] = 55;  
}
```

What makes transformations hard?

```
if (secret) {  
    arr[32] = 55;  
}
```



What makes transformations hard?

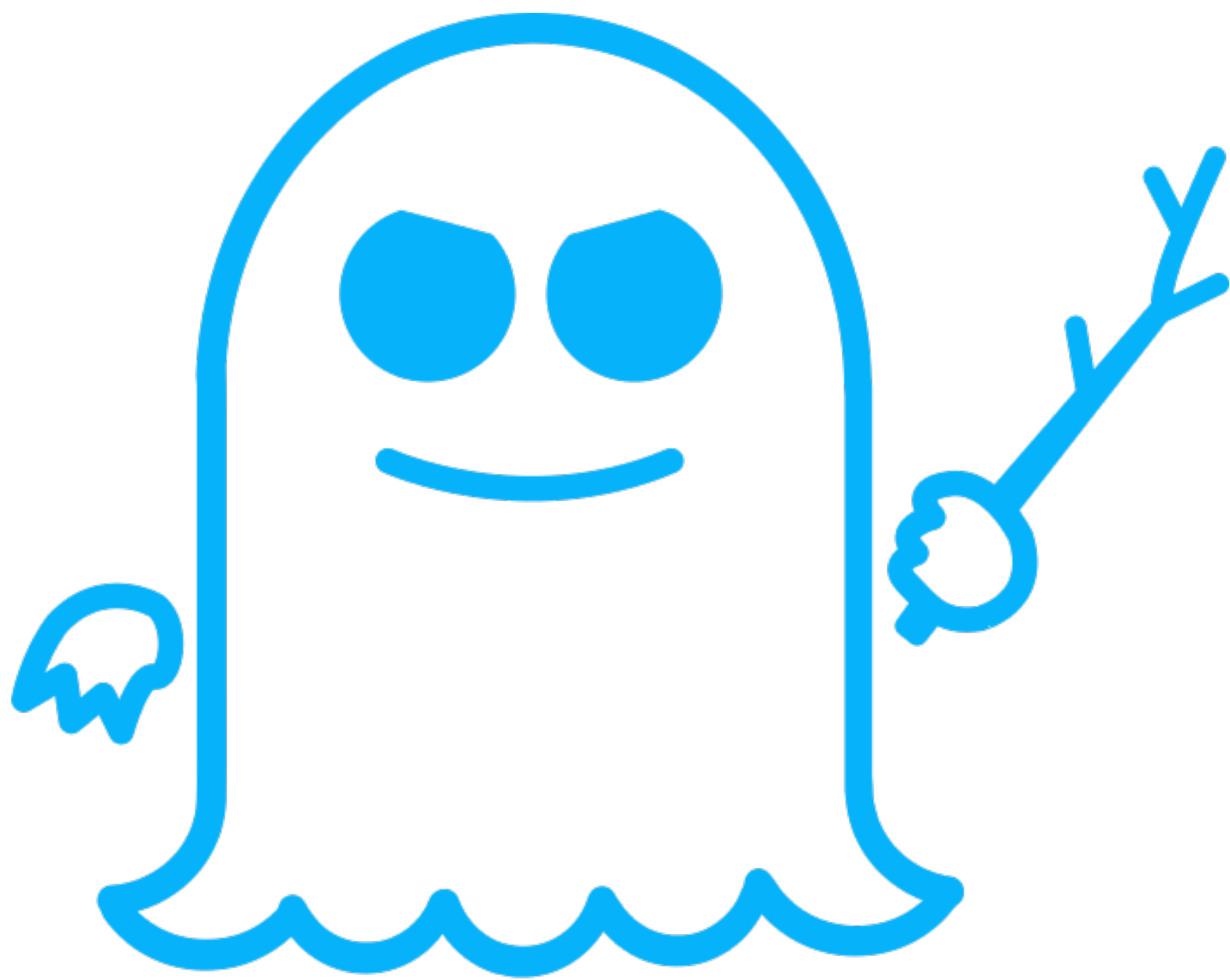
```
if (secret) {  
    arr[32] = 55;  
}
```



```
arr[32] = ct_select(secret, 55, arr[32]);
```

What makes transformations hard?

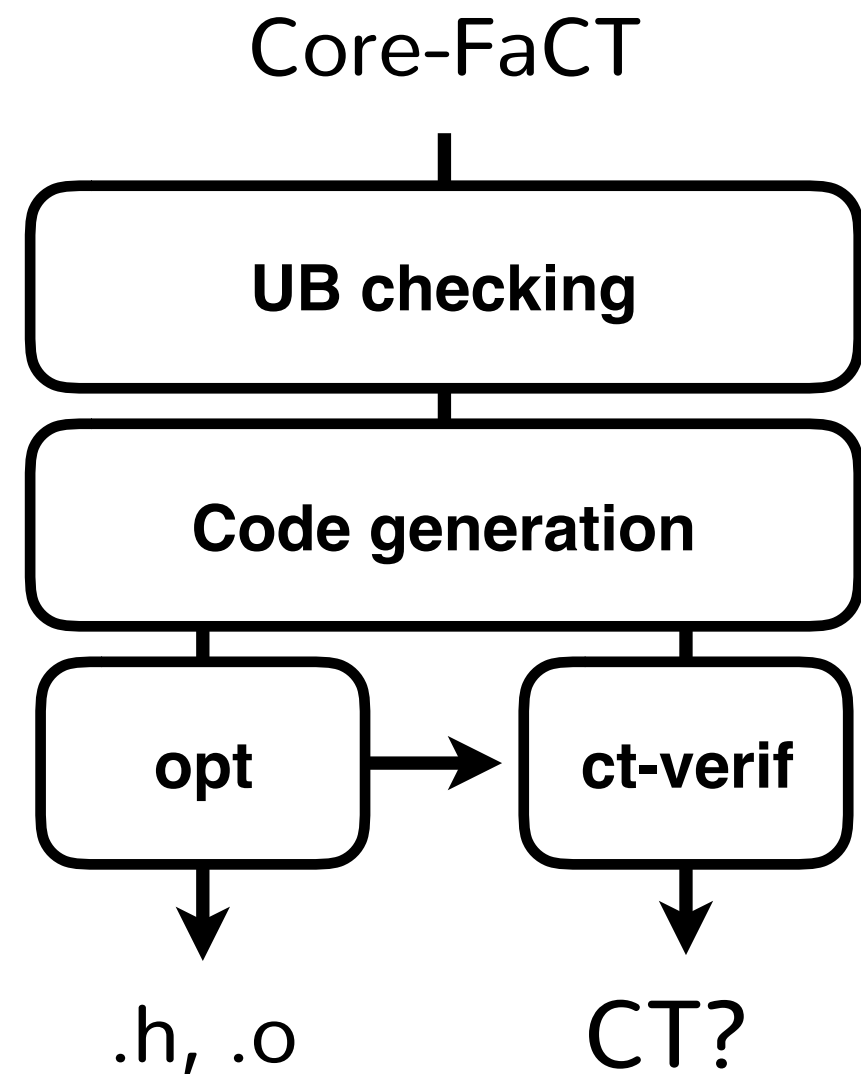
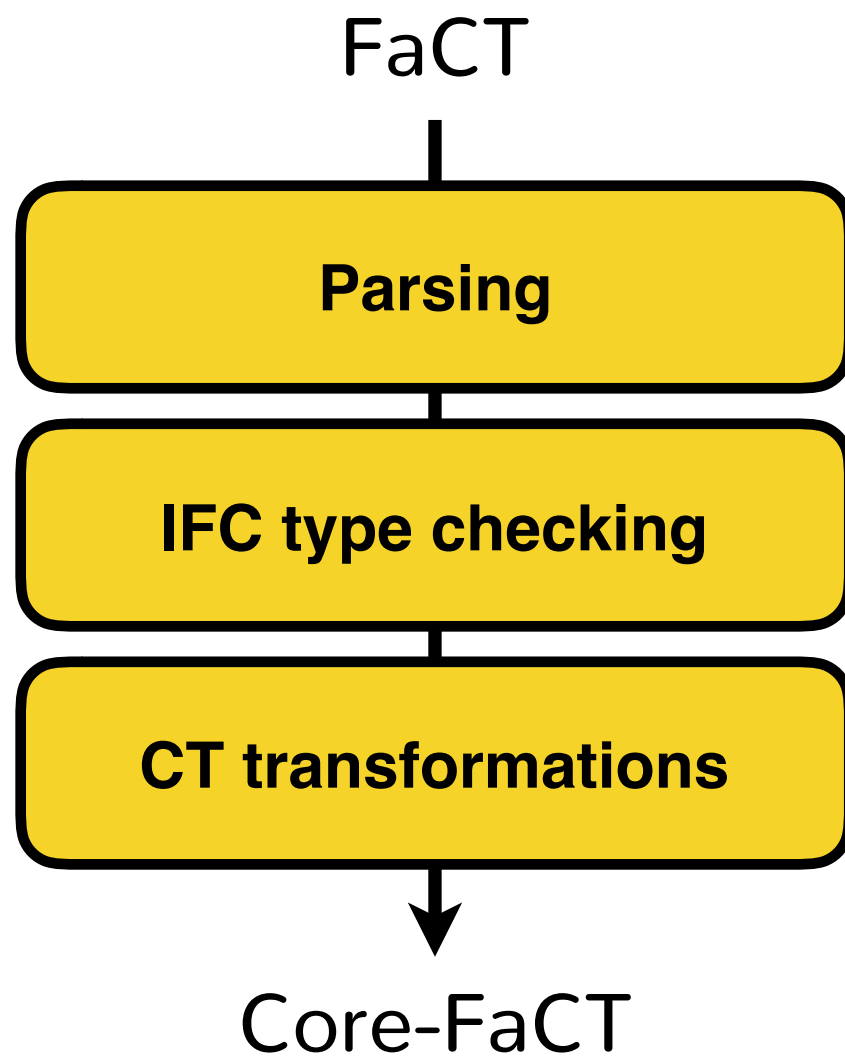
- Secret if-statements execute both branches
 - Not actual guards!
 - If not careful: can introduce OOB accesses!
- Similarly, code past secret return still executes!



Type system to rescue!

- Eliminate potential undefined behavior
 - All array accesses are in bounds
 - All bit shifts are less than bit width of value
 - No division by zero
 - All arrays and structs are initialized
- How do we do this?
 - Force developer to add checks/assertions

Is FaCT a secure compiler?



Correct, secure compiler?

- Prove that CT transformation is correct
 - Every well-typed FaCT program $P \approx P_0$ where $P_0 = CT(P)$
- Prove that any core-program P_0 is constant-time
 - Relatively easy: no secret control flow
- Prove that Core-FaCT compiler is correct/secure?
 - What about optimizations?

Correct, secure compiler?

- Target ct-wasm instead of LLVM
 - We already proved that any well-typed program in ct-wasm is constant-time
- What about ct-wasm optimizations?
 - Translation validation!



CTWASM

Secure compilation is not enough

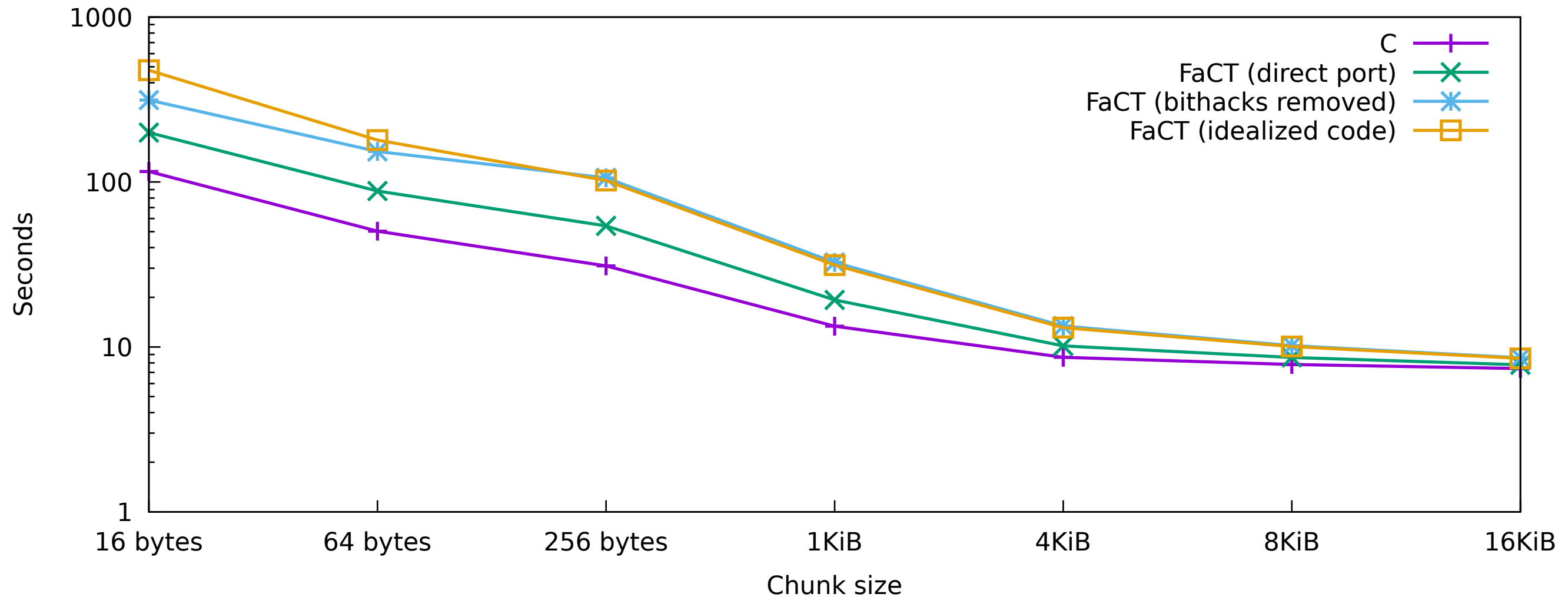
- Can we use it in real projects? Is it fast enough?
- Can developers actually understand/write code in FaCT better than in C?

What did we port?

- libsodium's secretbox
- OpenSSL's AES-CBC-HMAC-SHA1
- mbedTLS's Montgomery multiplication & sliding window exponentiation (used in RSA, DHM key exch)
- DJB/Langley's curve-25519 Donna

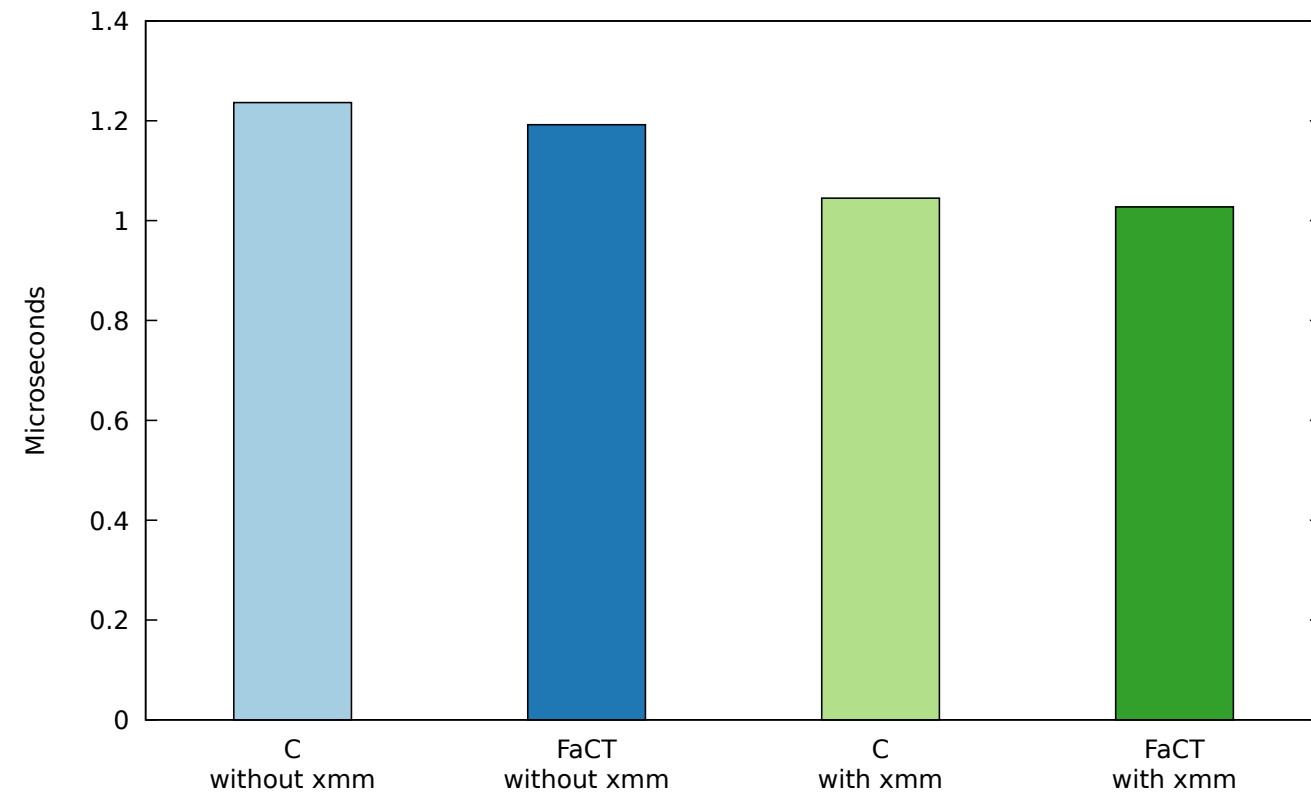
Representative performance numbers

OpenSSL AES-CBC-HMAC-SHA1 in TLS mode

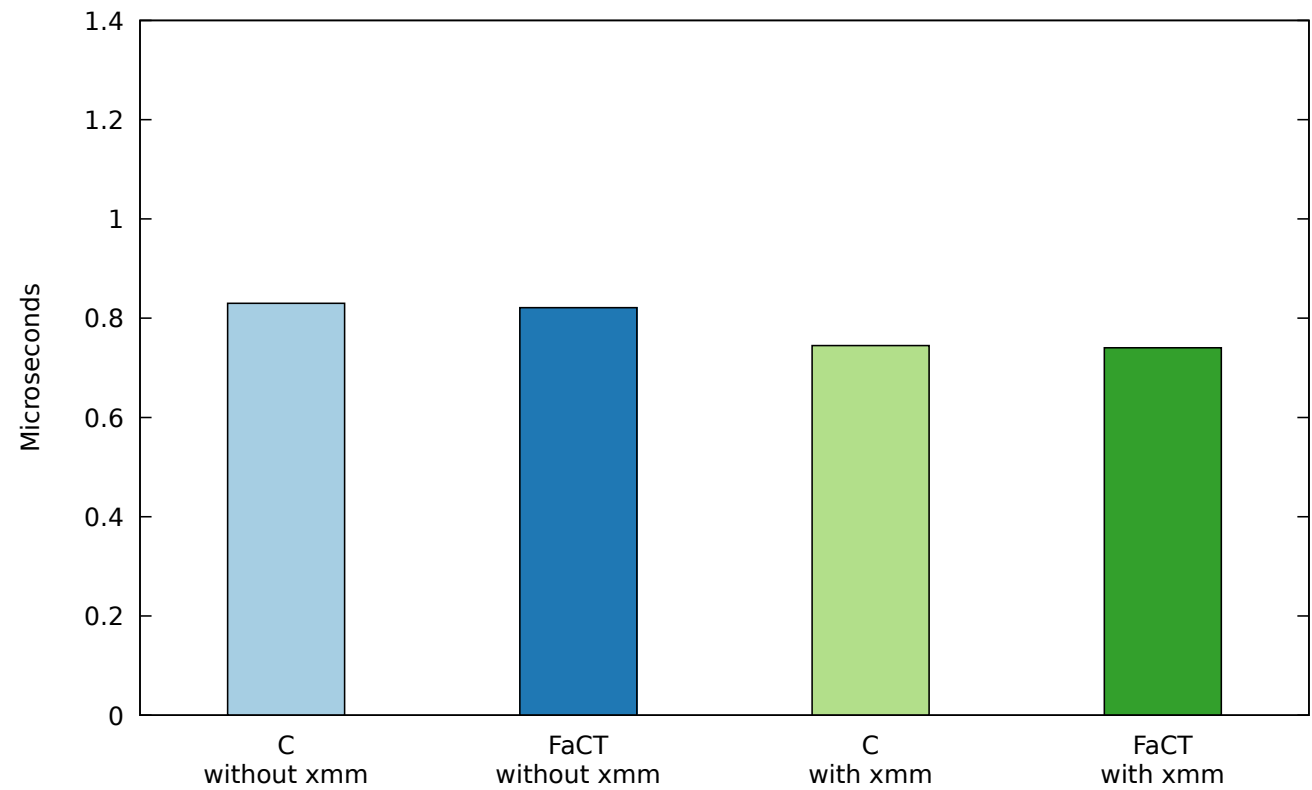


Representative performance numbers

secretbox encryption



secretbox decryption



Usability

- Conducted user study as part of undergrad PL class
 - Understanding what code is doing (C + FaCT)
 - Writing constant-time code (C + ct-verif, FaCT)
- Take away: yep!
 - Understanding syntax was harder than labels
 - Happy to show graphs and discuss results offline

Next steps

- Type families
 - Make it easier to abstract away implementation details to write high-level crypto algorithms
- Support for architectures beyond x64
- Translation validation for reasoning about CT across optimization passes
- Compiler verification