Constant-time programming in FaCT

Deian Stefan

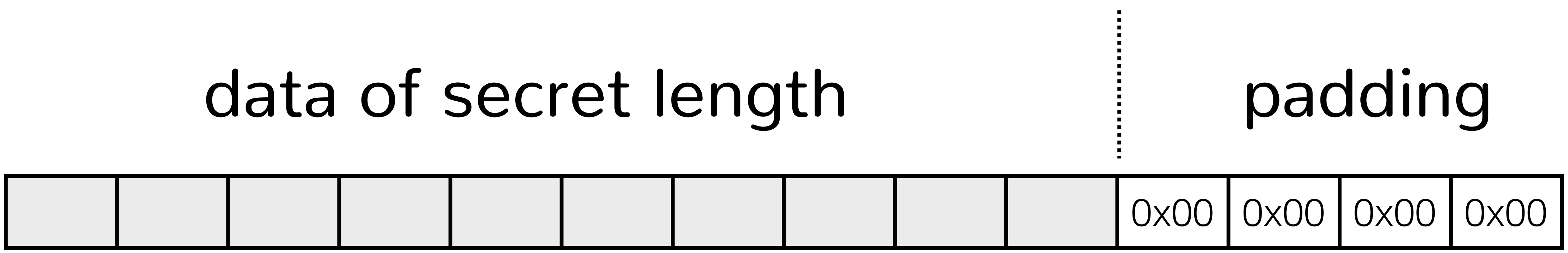
UC San Diego

Fraser Brown, Sunjay Cauligi, Ranjit Jhala, Brian Johannesmeyer,
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 hav been urged to do so.

My first remark is that, although the programmer's activit ends when he has constructed a correct program, the proces 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

M. Broy et al. (eds.), *Pioneers and Their Contributions to Software Engineering*

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)


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
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

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


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

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

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

OpenSSL padding oracle attack

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

Error prone in practice

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

```
-       EVP_DigestUpdate(&md_ctx,md,2);
-       EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
-       EVP_DigestFinal_ex( &md_ctx,md,NULL);
-
-       EVP_MD_CTX_copy_ex( &md_ctx,hash);
-       EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
-       EVP_DigestUpdate(&md_ctx,ssl3_pad_2,npad);
-       EVP_DigestUpdate(&md_ctx,md,md_size);
-       EVP_DigestFinal_ex( &md_ctx,md,&md_size);
-
-       EVP_MD_CTX_cleanup(&md_ctx);
+       if (!send &&
+           EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
+           ssl3_cbc_record_digest_supported(hash))
+               {
+               /* This is a CBC-encrypted record. We must avoid leaking any
+                * timing-side channel information about how many blocks of
+                * data we are hashing because that gives an attacker a
+                * timing-oracle. */
+
+               /* npad is, at most, 48 bytes and that's with MD5:
+                *   16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
+                *
+                * With SHA-1 (the largest hash speced for SSLv3) the hash size
+                * goes up 4, but npad goes down by 8, resulting in a smaller
+                * total size. */
+               unsigned char header[75];
+               unsigned j = 0;
+               memcpy(header+j, mac_sec, md_size);
+               j += md_size;
+               memcpy(header+j, ssl3_pad_1, npad);
+               j += npad;
+               memcpy(header+j, seq, 8);
+               j += 8;
+               header[j++] = rec->type;
+               header[j++] = rec->length >> 8;
+               header[j++] = rec->length & 0xff;
+
+               ssl3_cbc_digest_record(
+                       hash,
+                       md, &md_size,
+                       header, rec->input,
+                       rec->length + md_size, rec->orig_len,
+                       mac_sec, md_size,
+                       1 /* is SSLv3 */);
+               }
+       else
+               {
+               unsigned int md_size_u;
+               /* Chop the digest off the end :-) */
+               EVP_MD_CTX_init(&md_ctx);
+
+               EVP_MD_CTX_copy_ex( &md_ctx,hash);
+               EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
+               EVP_DigestUpdate(&md_ctx,ssl3_pad_1,npad);
+               EVP_DigestUpdate(&md_ctx,seq,8);
+               rec_char=rec->type;
+               EVP_DigestUpdate(&md_ctx,&rec_char,1);
+               p=md;
+               s2n(rec->length,p);
+               EVP_DigestUpdate(&md_ctx,md,2);
+               EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
```

OpenSSL padding oracle attack

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

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.

Error prone in practice

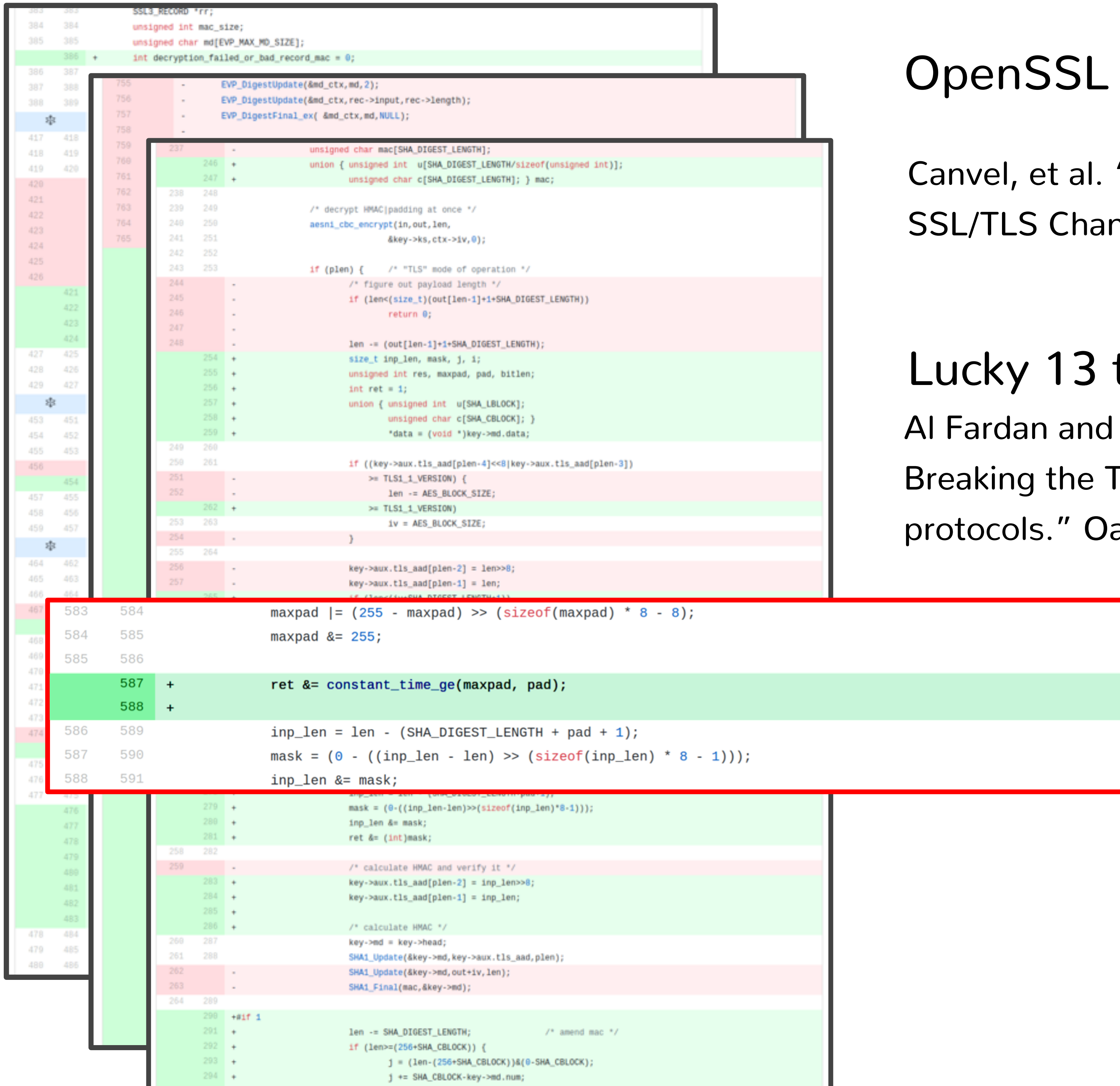
OpenSSL padding oracle attack

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.

Error prone in practice



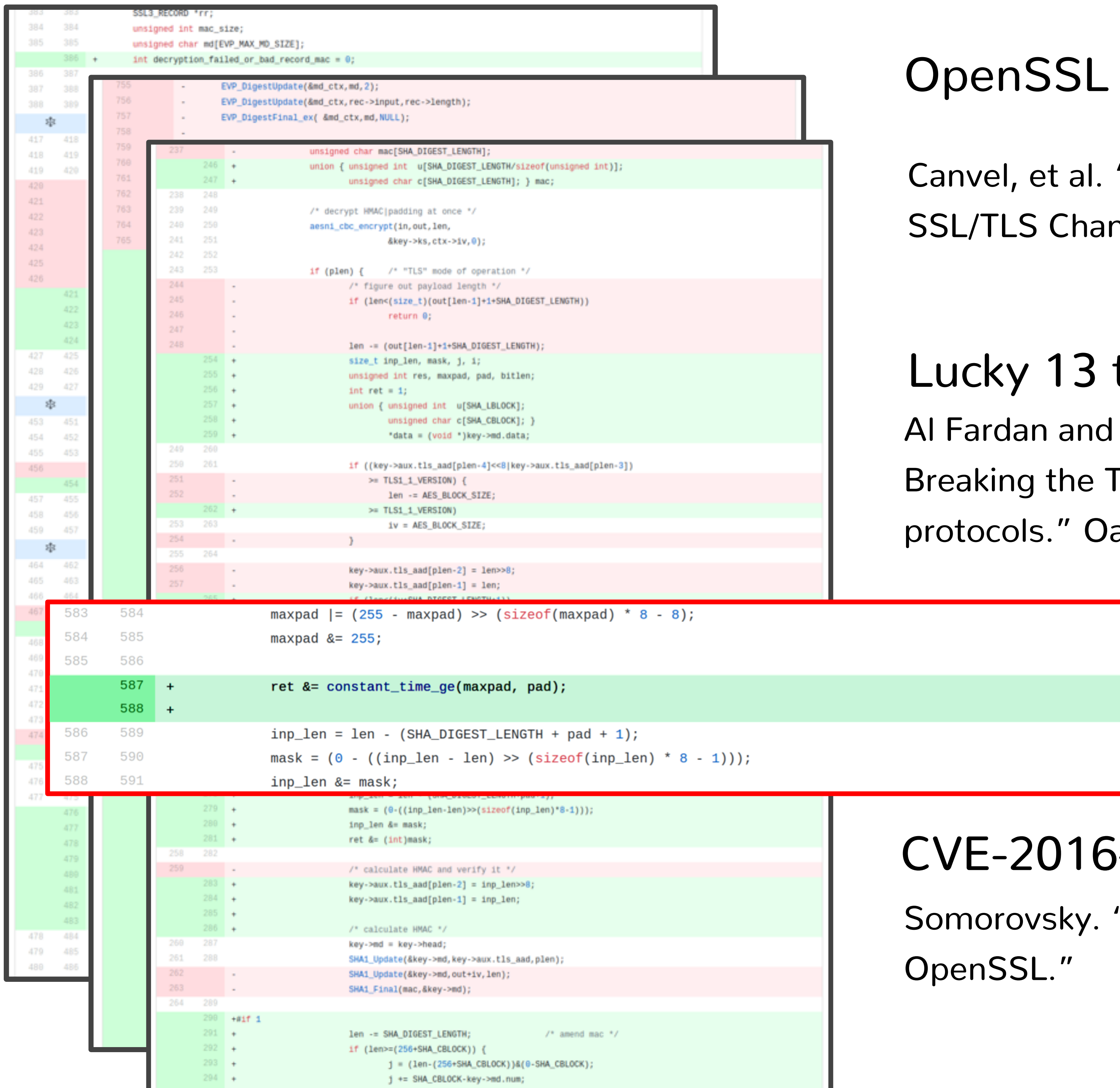
OpenSSL padding oracle attack

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.

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.”

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

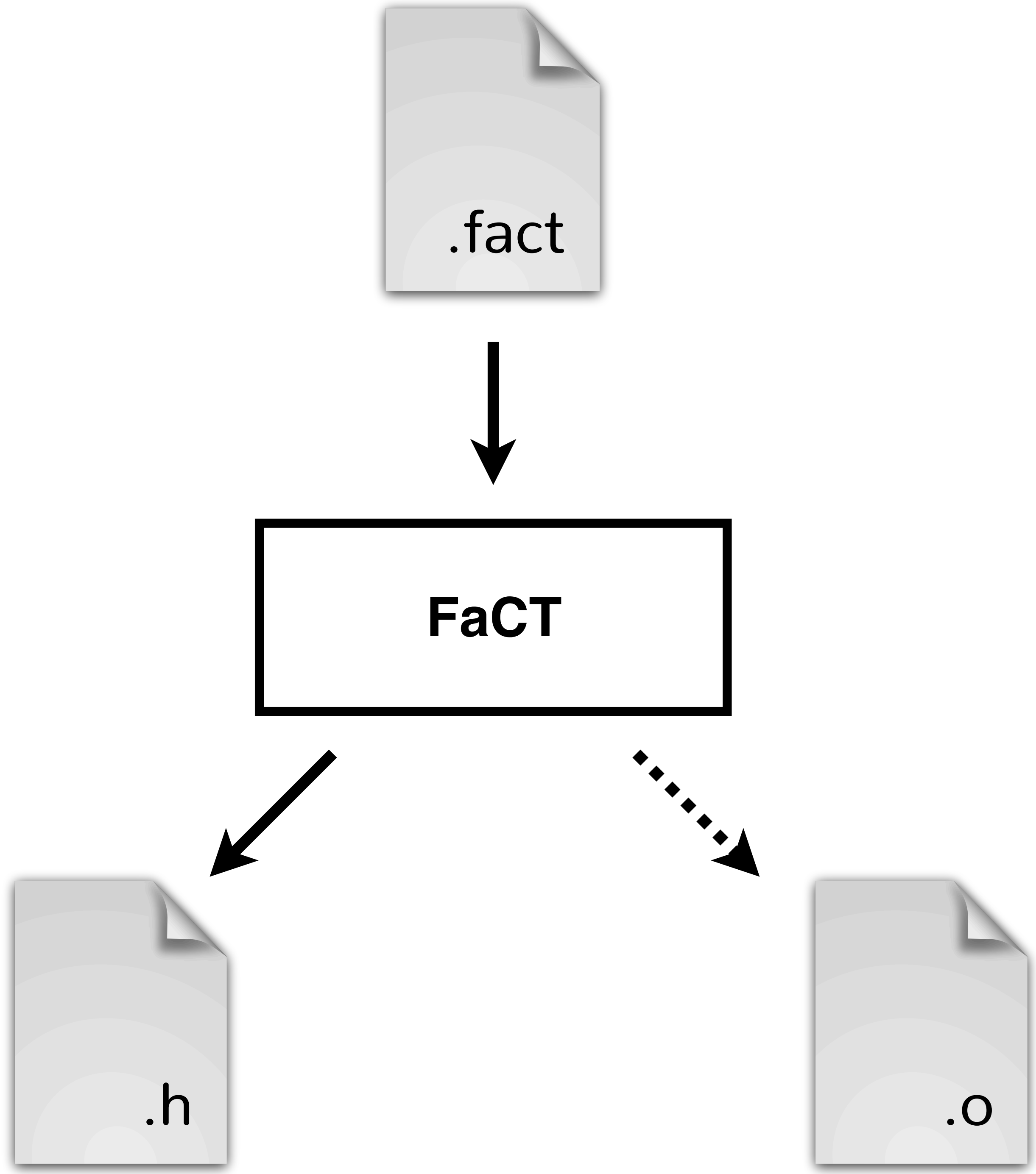
```c
int main() {
  uint8_t cond = 1;
  uint32_t x = 42;
  uint32_t y = 137;

  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;
}
```

```c
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)


.fact
FaCT
.h
.o

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, ii32* %_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, sec_1 % sec_2

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?

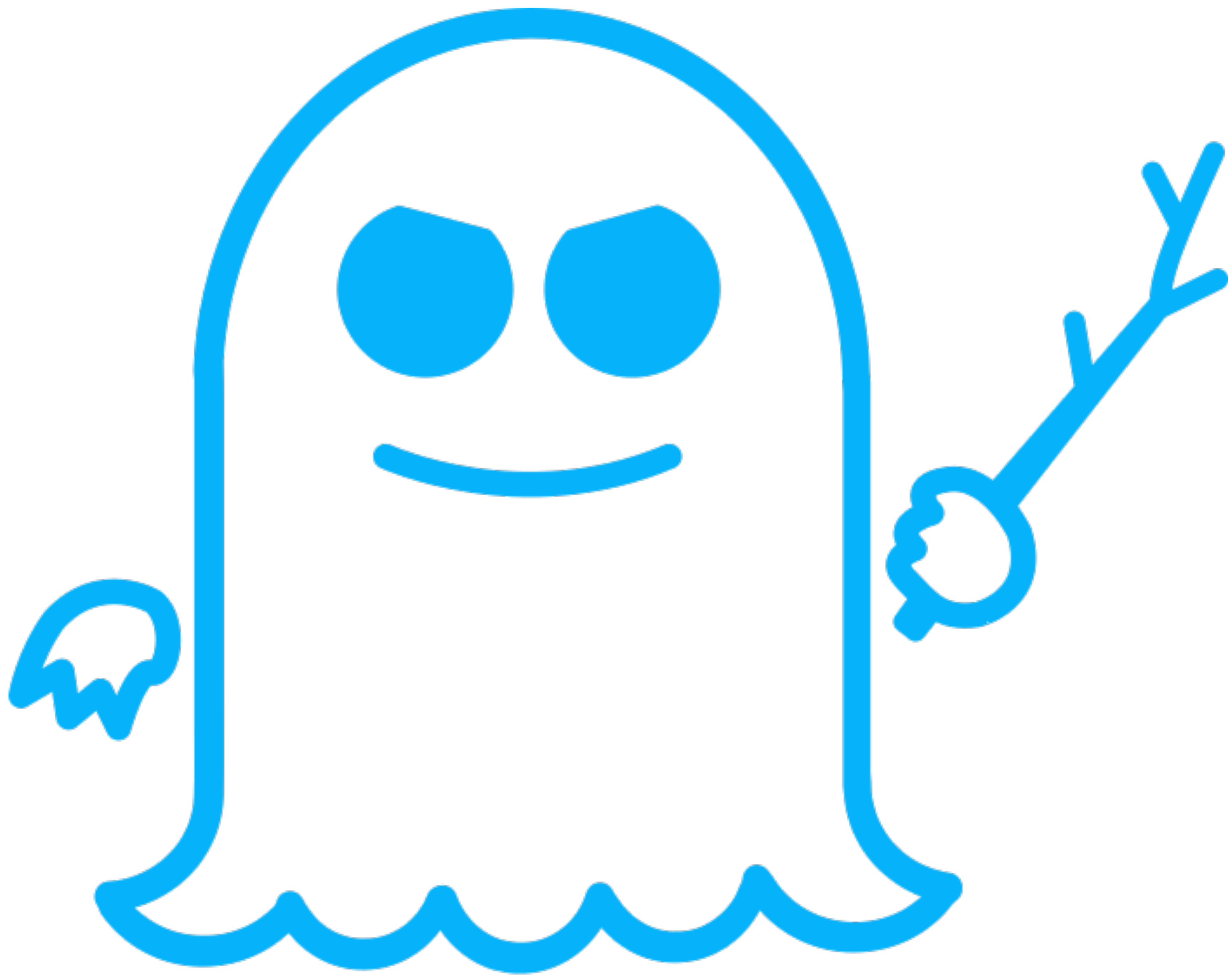
```
if (secret) {
    arr[32] = 55;
}
```

⬇

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

What makes transformations hard?

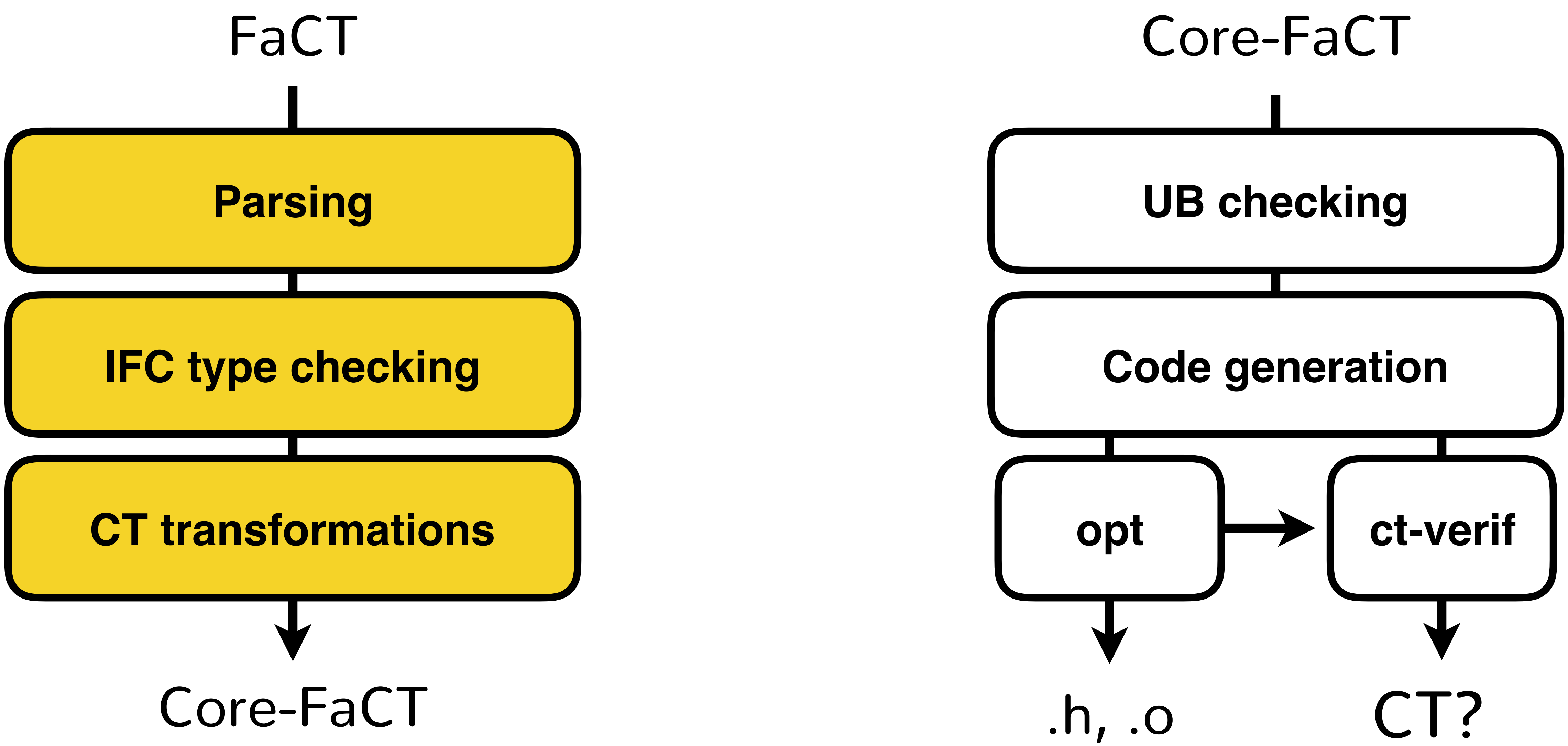
- Secret if-statements execute both branches
  - ➤ Not actual guards!
  - ➤ If not careful: can introduce OOB accesses!
- Similarly, code past secret return still executes!



Type system to rescue!

- Eliminate potential undefined behavior
  - ➤ All array accesses are in bounds
  - ➤ All bit shifts are less than bit with 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 $P \simeq P_0$ 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

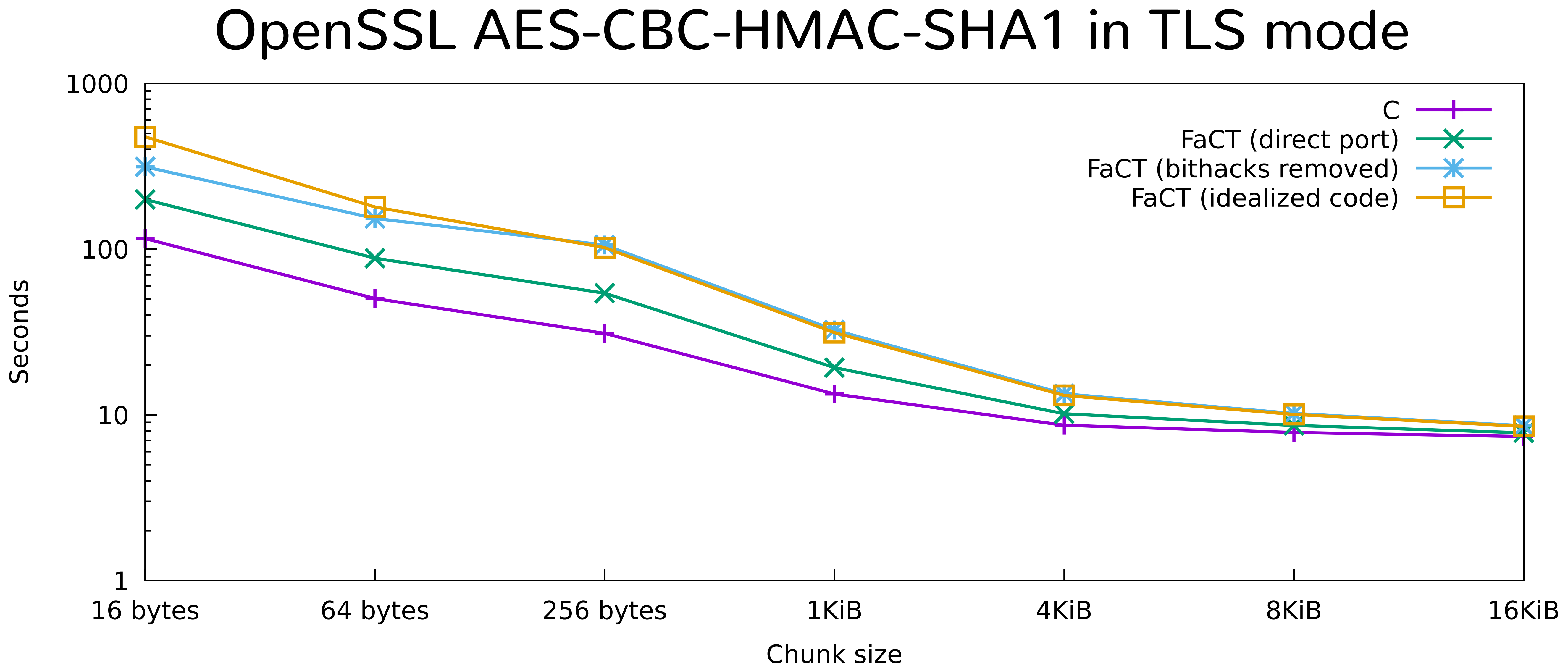
Secure compilation is not enough

- 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?

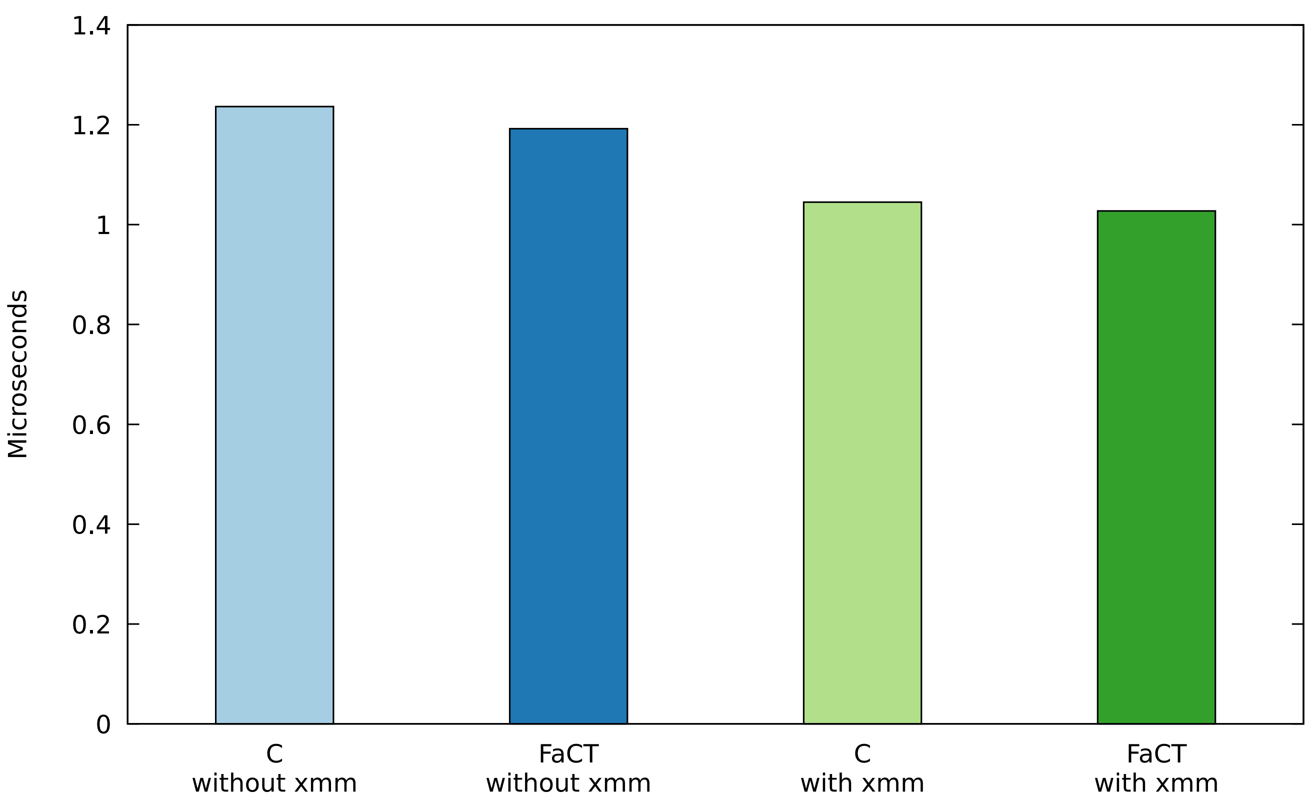
- libsodium’s secretbox
- OpenSSL’s AES-CBC-HMAC-SHA1
- mbedTLS’s Montgomery multiplication & sliding window exponentiation (used in RSA, DHM key exch)
- DJB/Langley’s curve-25519 Donna

Representative performance numbers

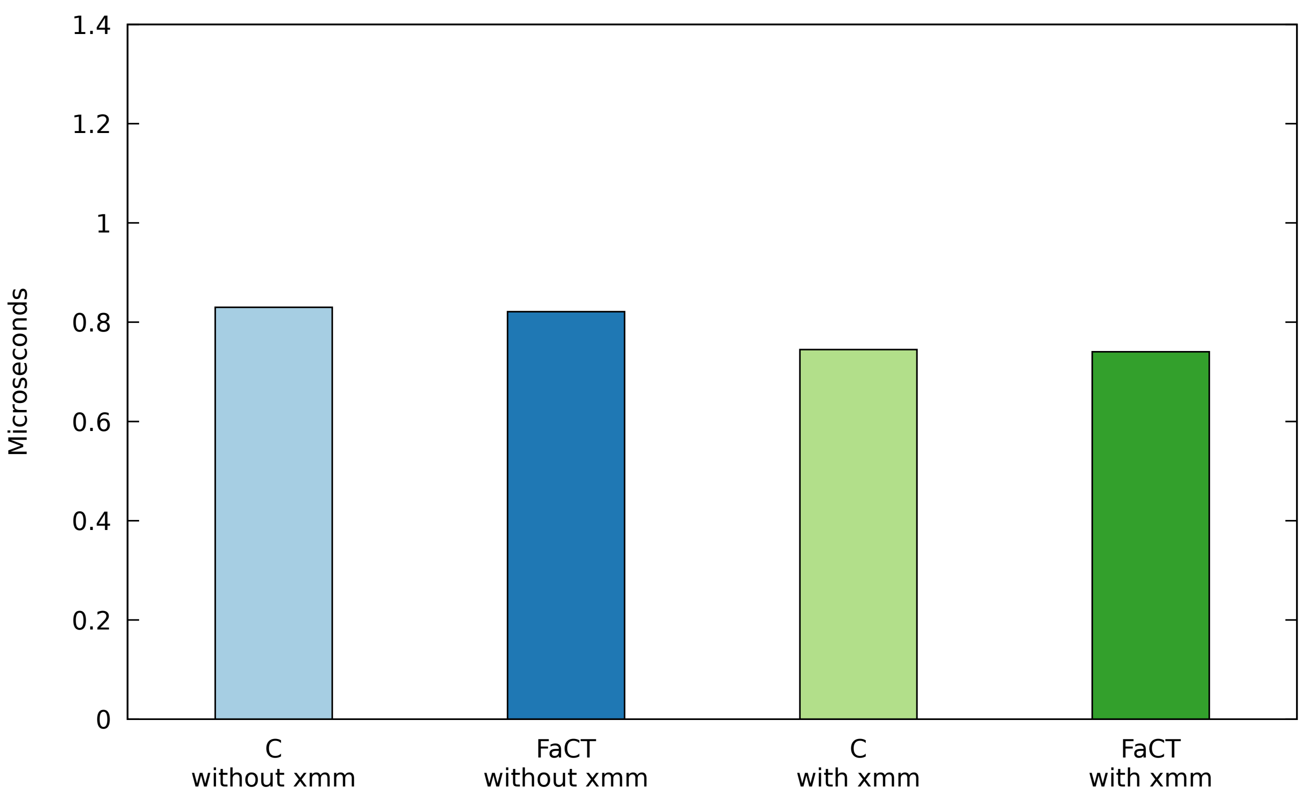


Representative performance numbers

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