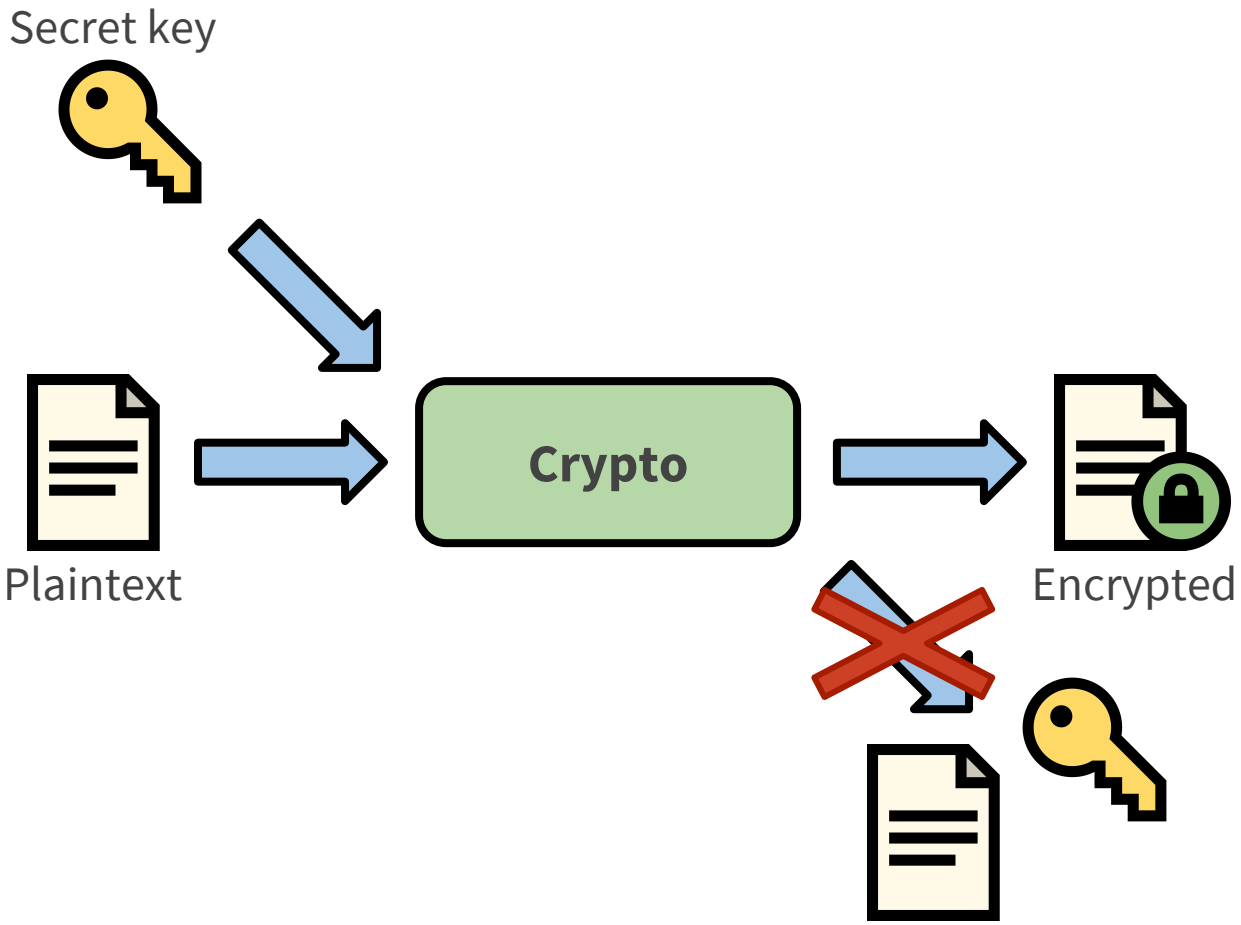
FaCT

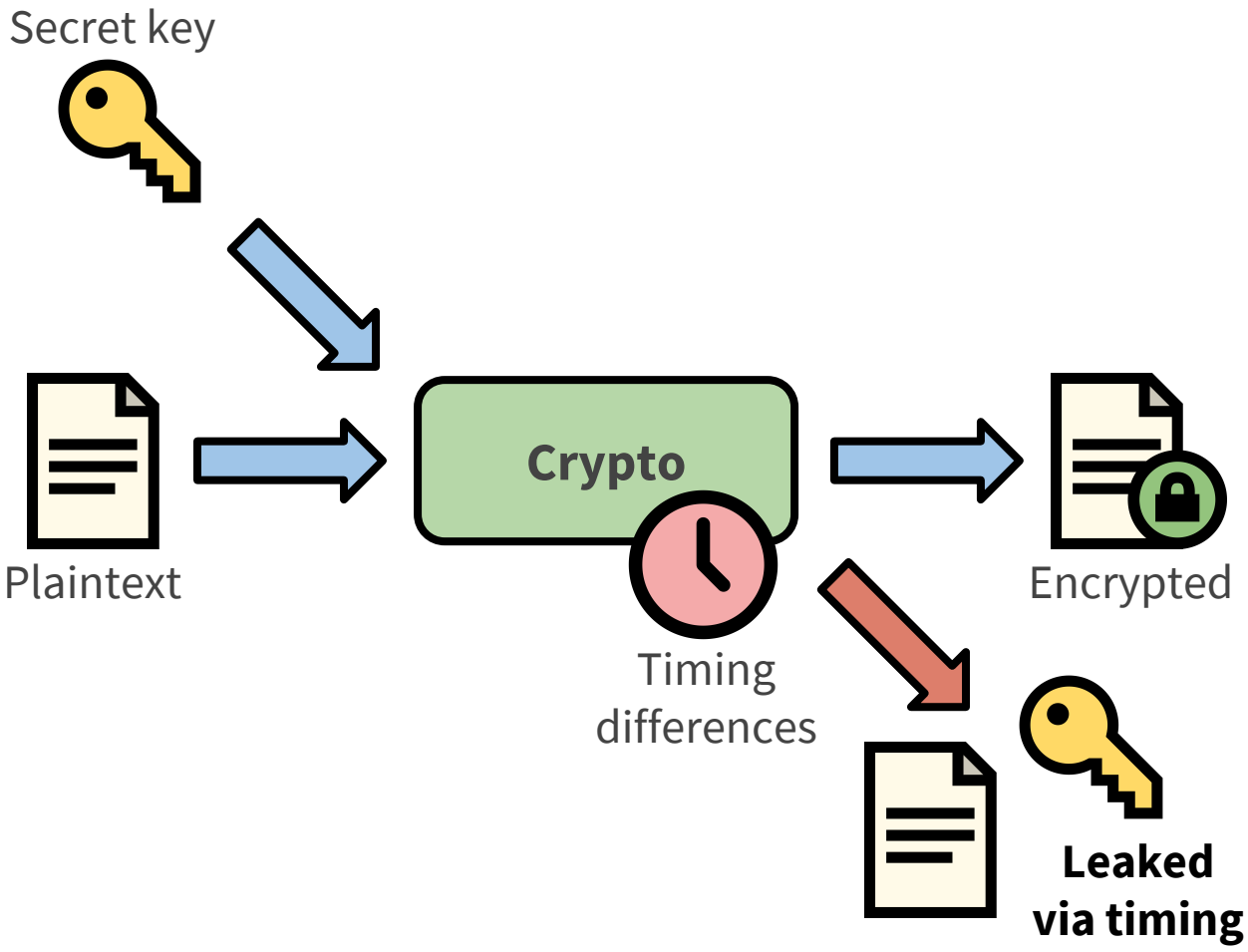
Sunjay Cauligi, Gary Soeller,
Fraser Brown, Brian Johannesmeyer,
Yunlu Huang, Ranjit Jhala, Deian Stefan

A Flexible, Constant-Time
Programming Language

Timing side channels

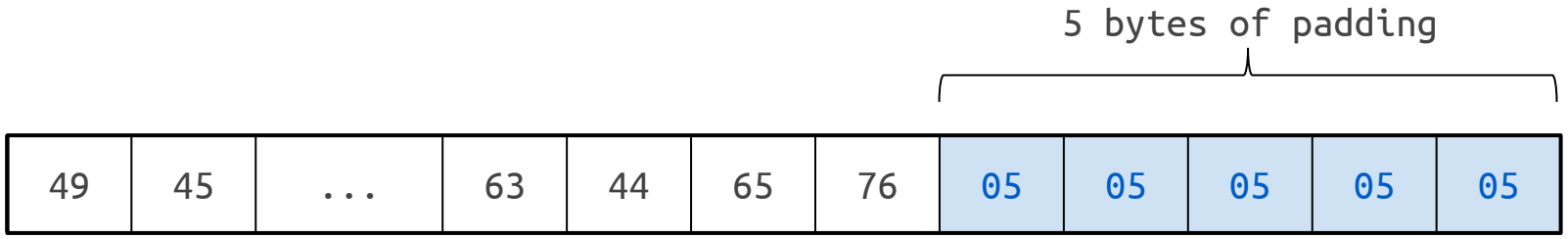


Timing side channels



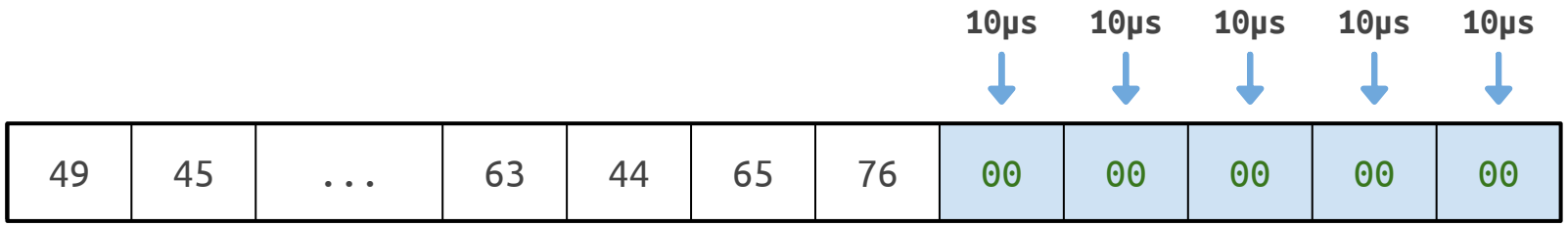
Writing secure code

- 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

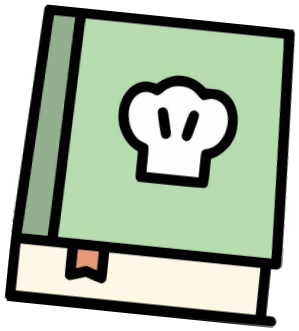
```
int32_t remove_padding(
    uint8_t* buf,
    uint32_t buflen) {

  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;
}
```



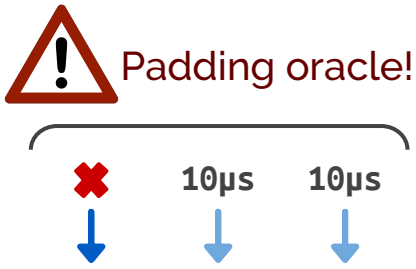
Writing secure code

```
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.

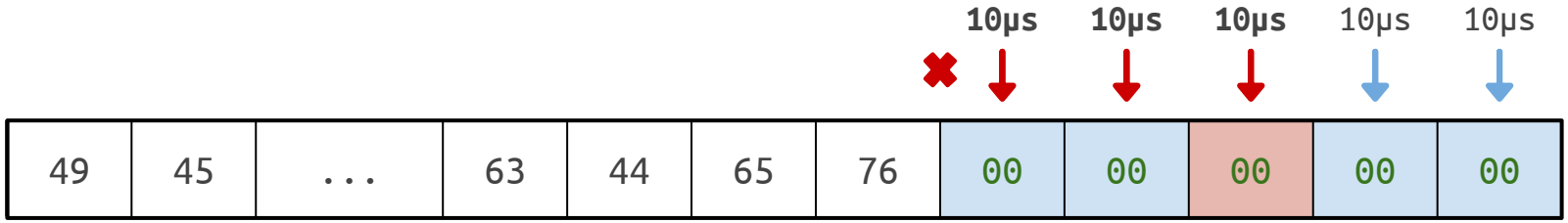


49	45	...	63	44	65	76	05	05	07	00	00

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;
}
```

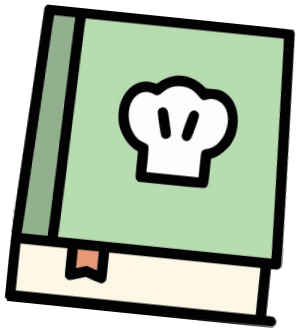
```
int32_t remove_padding2(
    uint8_t* buf,
    uint32_t buflen) {
  uint8_t ok = 1;
  uint8_t padlen = buf[buflen-1];
  uint32_t i;
  for (i = 0; i < padlen; i++) {
    if (buf[buflen-i-1] != padlen)
      ok = 0;
    buf[buflen-i-1] = 0;
  }
  return ok ? padlen : -1;
}
```



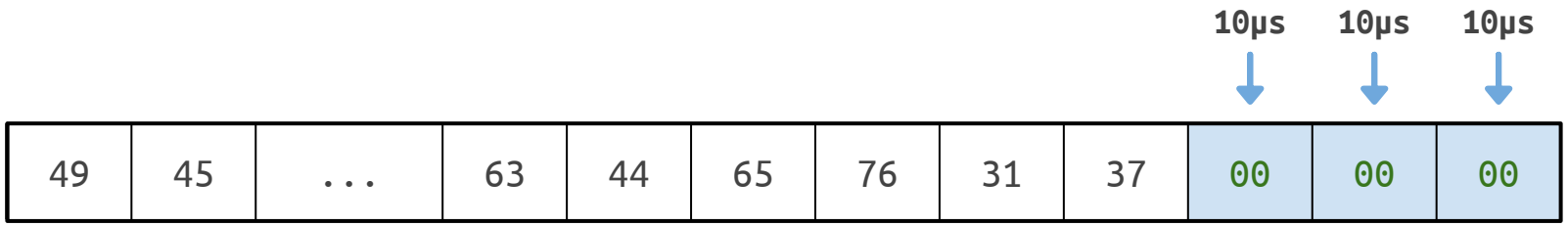
Writing secure code

```
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!



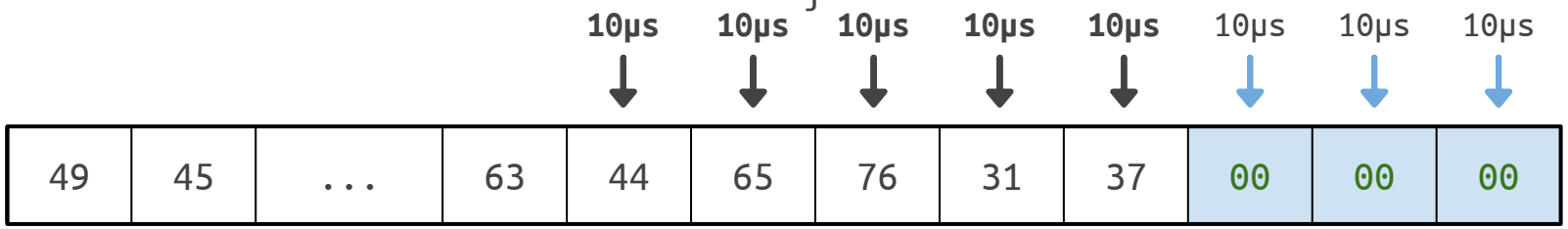
Use this instead.



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;
}
```

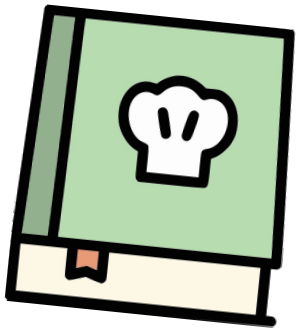
```
int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
  uint8_t ok = 1;
  uint8_t padlen = buf[buflen-1];
  uint32_t i;
  for (i = buflen-256; i < buflen; i++) {
    uint8_t b = buf[i];
    if (i >= buflen - padlen) {
      if (b != padlen)
        ok = 0;
      b = 0;
    }
    buf[i]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



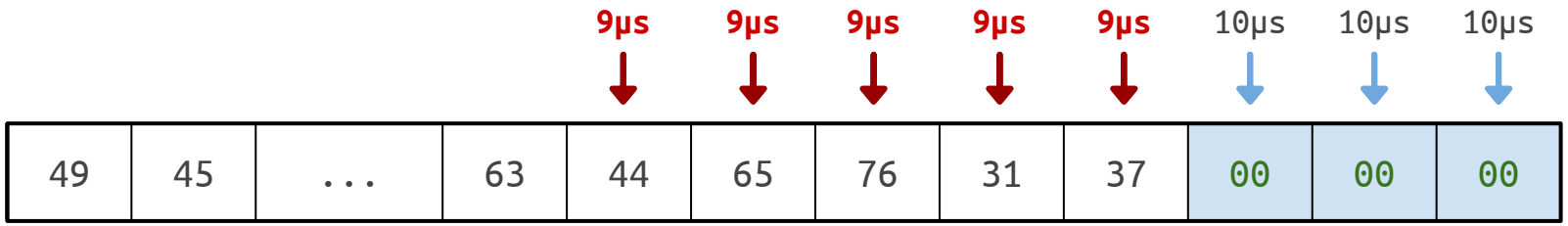
Writing secure code

```
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!



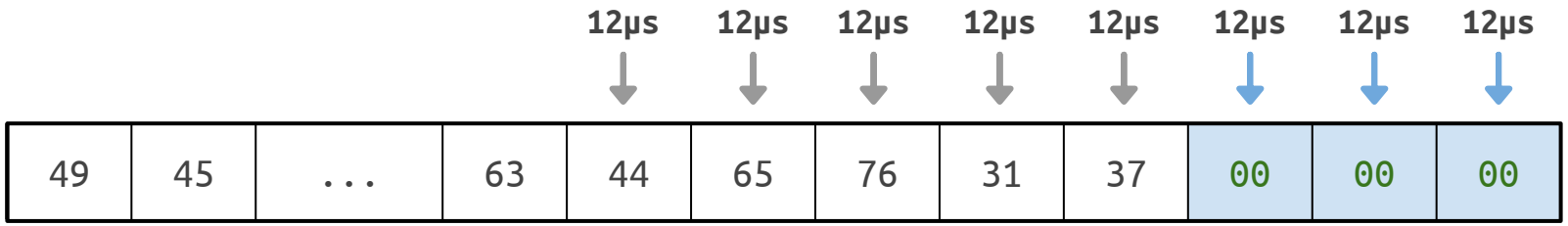
Use this instead.



Writing secure code

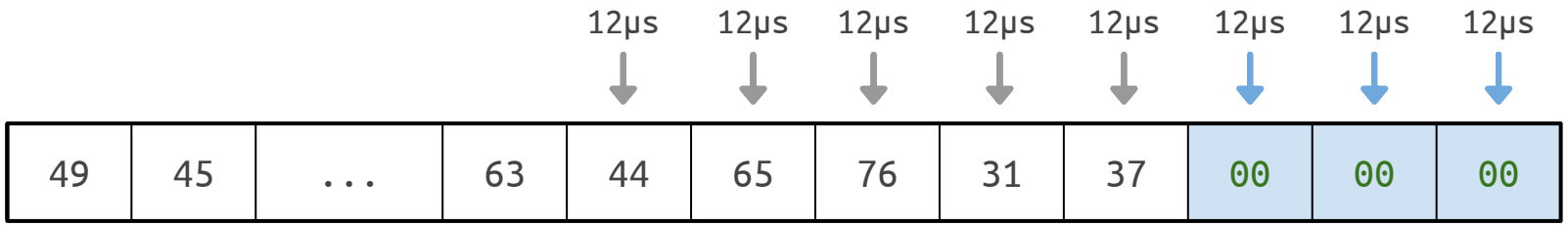

```
int32_t remove_padding3(
    uint8_t* buf,
    uint32_t buflen) {
  uint8_t ok = 1;
  uint8_t padlen = buf[buflen-1];
  uint32_t i;
  for (i = buflen-256; i < buflen; i++) {
    uint8_t b = buf[i];
    if (i >= buflen - padlen) {
      if (b != padlen)
        ok = 0;
      b = 0;
    }
    buf[i]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



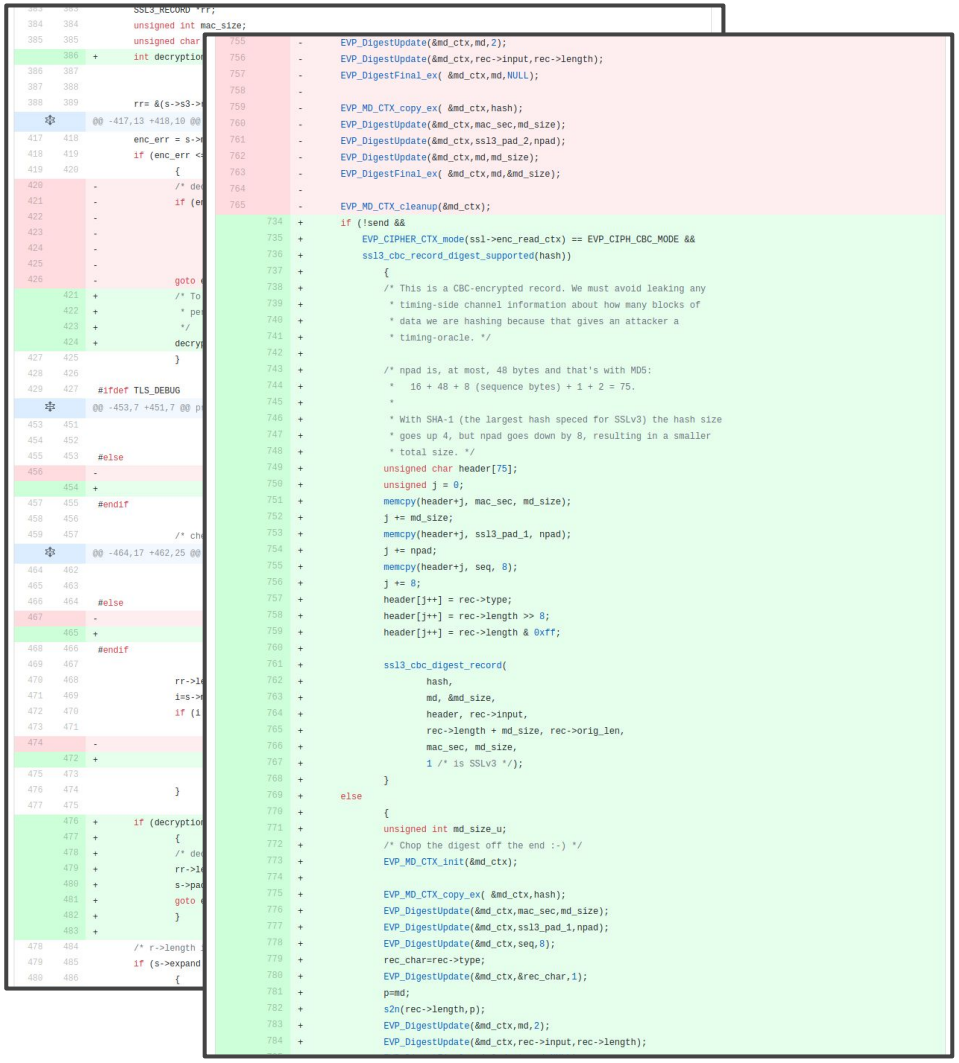
Error-prone in practice

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

        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)
+               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

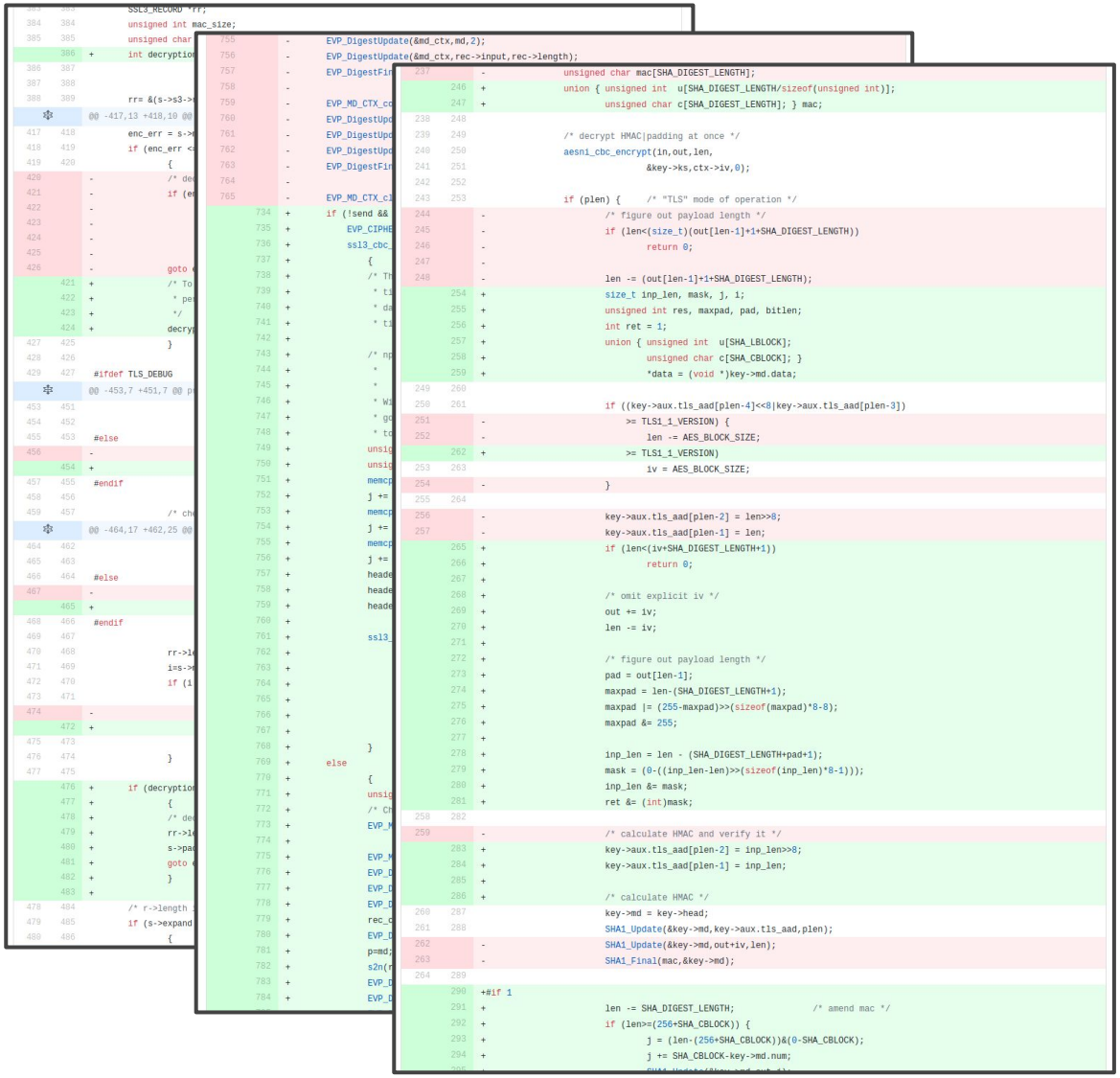


```
-    EVP_DigestUpdate(&md_ctx,md,2);
-    EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
-    EVP_DigestFinal_ex( &md_ctx,md,NULL);
-
-    EVP_MD_CTX_copy_ex( &md_ctx,hash);
-    EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
-    EVP_DigestUpdate(&md_ctx,ssl3_pad_2,npad);
-    EVP_DigestUpdate(&md_ctx,md,md_size);
-    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);
```

Lucky 13 timing attack

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

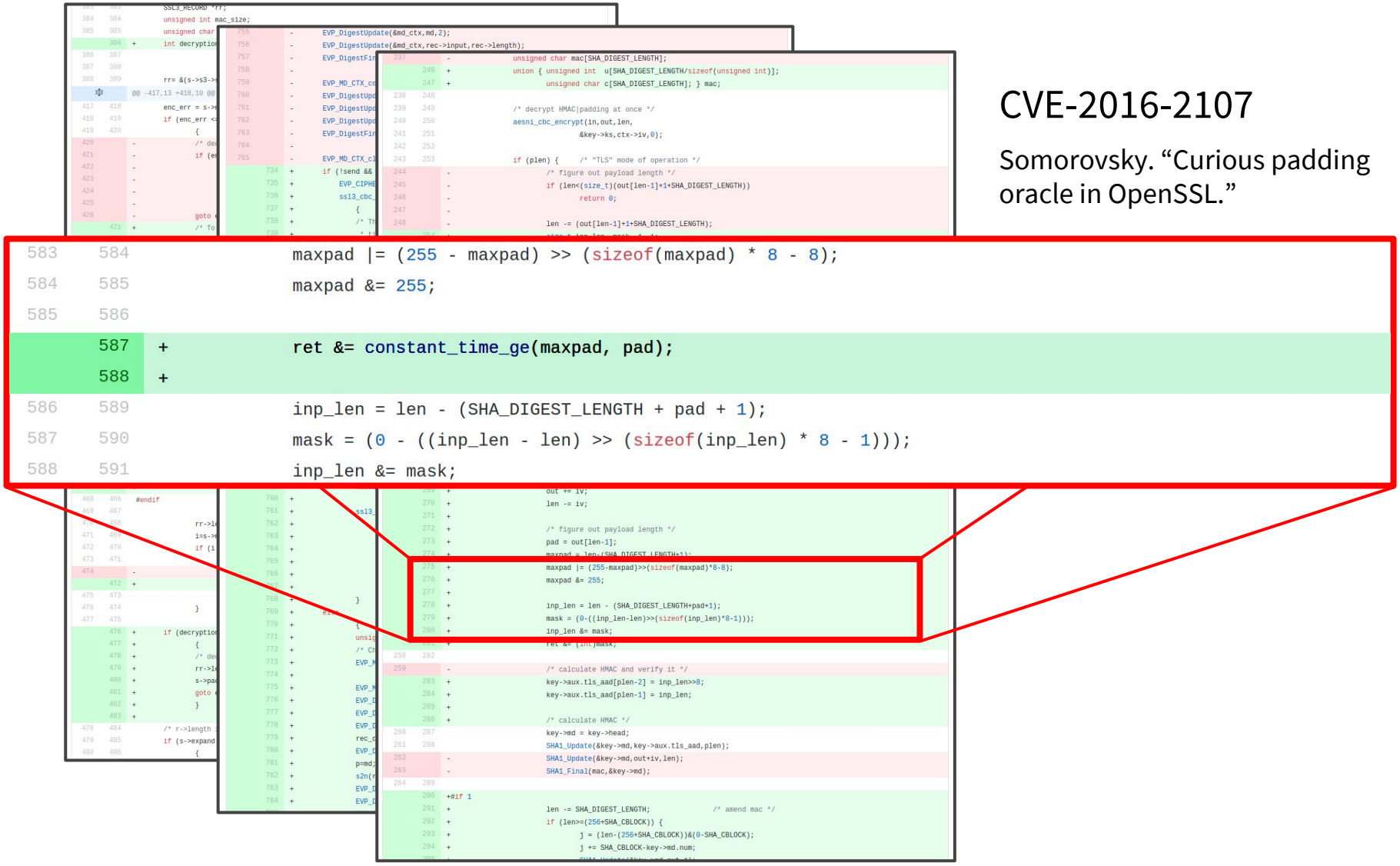
Error-prone in practice



Further refinements

Decryption path has no more measurable timing differences

Error-prone in practice



```
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;
```

CVE-2016-2107

Somorovsky. “Curious padding oracle in OpenSSL.”

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
```

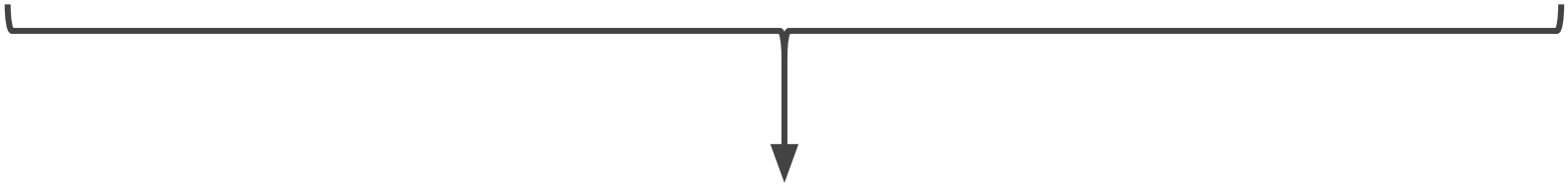
Compiler optimizations get in the way

```
/* 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);
```

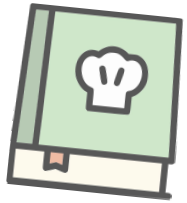


Assembly:

```
< 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 known [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
```

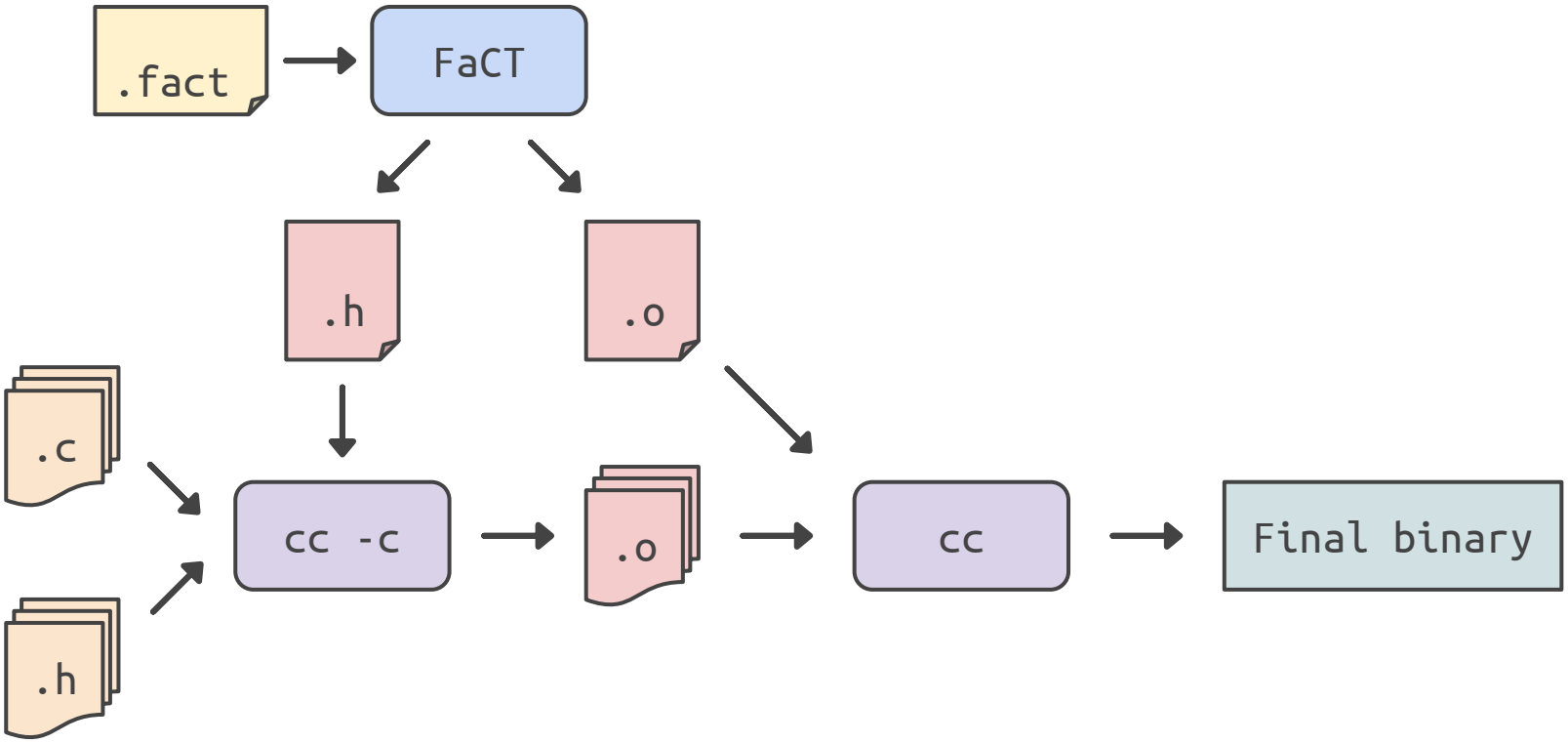
Constant problems with constant-time

- 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

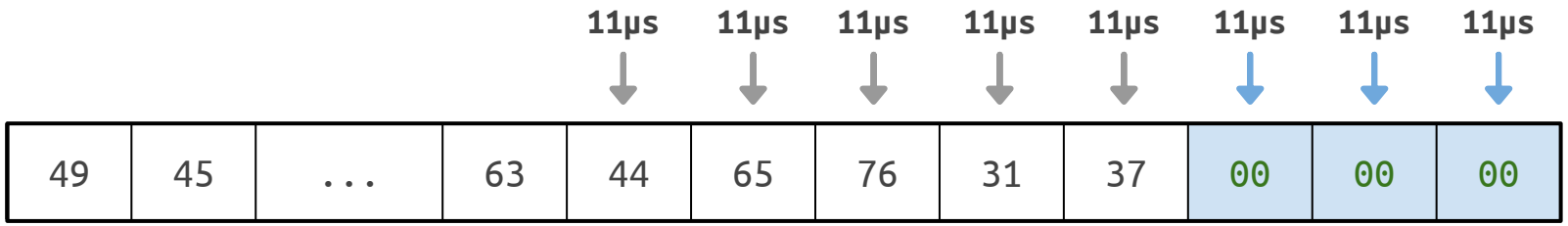
- Write clear code for computation on secrets
  - Helps you keep track of secrets vs. public values
  - Lets you use standard programming constructs
  - Ensures you write *correct* code
- Compiler that helps instead of hurts
  - Optimize your code as much as possible
  - But ensure code remains constant-time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

FaCT

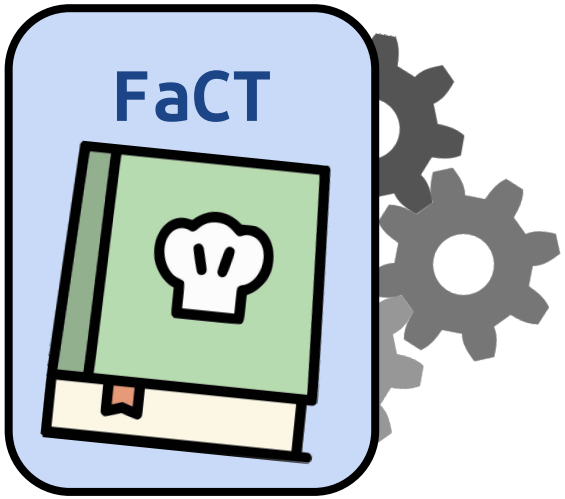


What does FaCT look like?

```
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

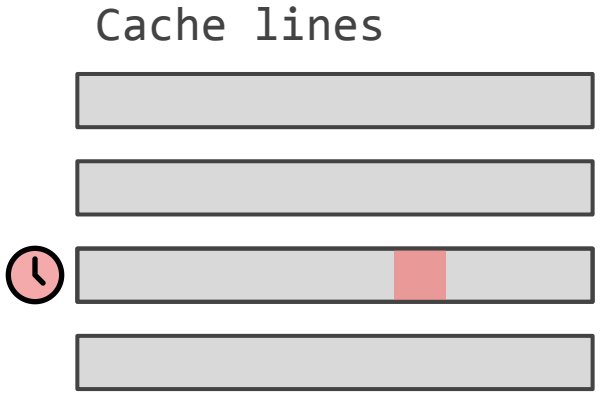
- IFC to determine what is secret/public
  - Only transform secret computation
- Prevent secret expressions we can't transform
  - Loop bounds

```
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

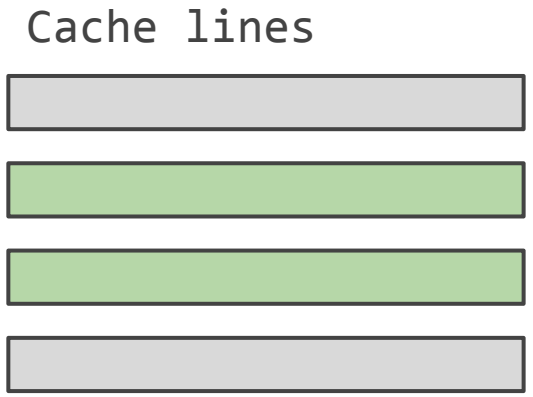
```
x = sensitive_buffer[secret_value];
```



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];
    }
}
```



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-verif[1] 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

[1]Almeida 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