Constant-time programming in FaCT

What is FaCT?

Domain specific language for writing constant-time code

- What's the difference between a DSL and a general-purpose language?

How do you use FaCT?

- Write your program in C
- Write your constant-time parts in FaCT

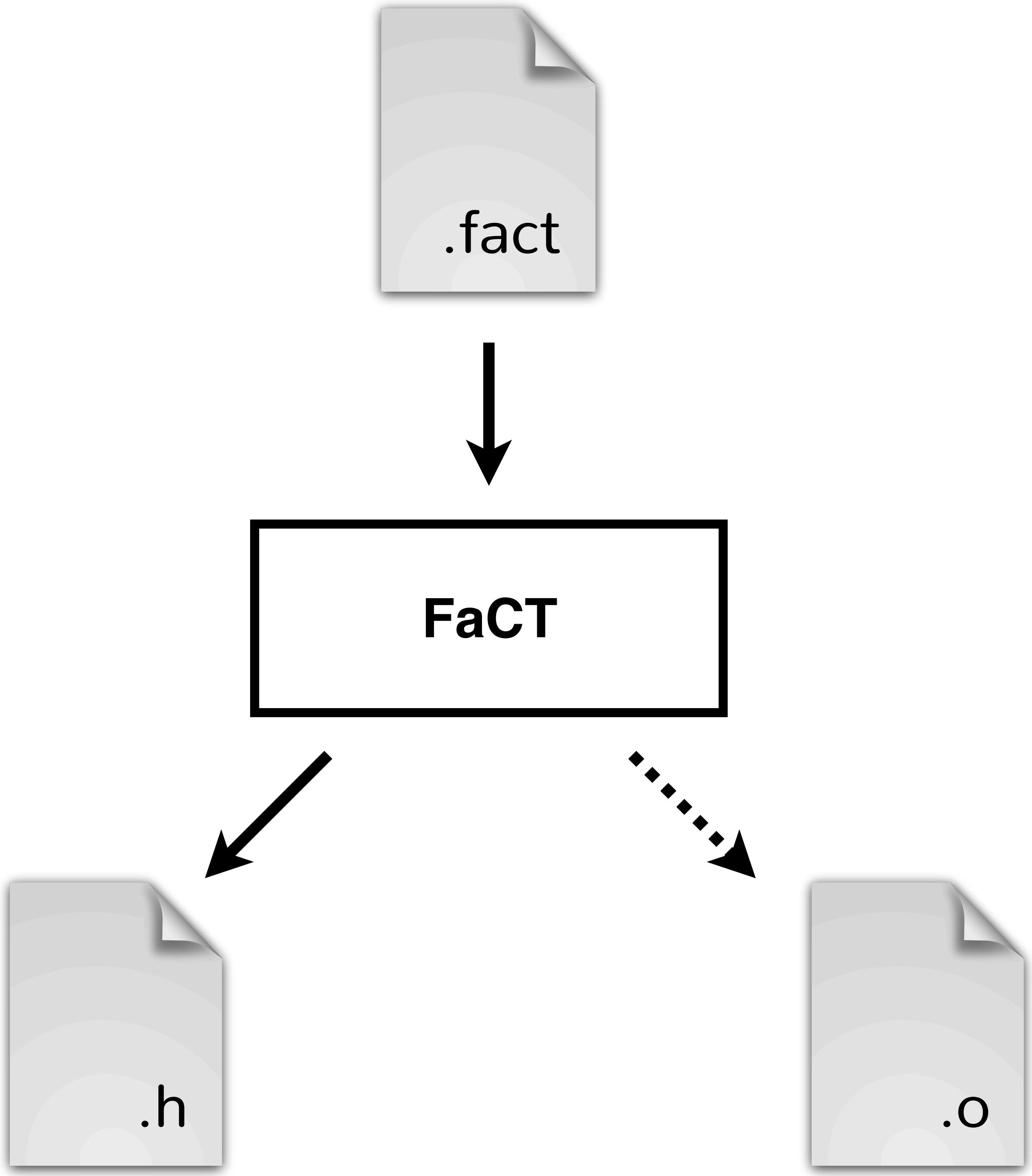
```c
int main() {
  uint8_t cond = 1;
  uint32_t x = 42;
  uint32_t y = 137;

  printf("cond: %u, x: %u, y: %u\n", cond, x, y);

  conditional_swap(&x, &y, cond);

  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, unsafe)


.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/Rust 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 = ...`

How does this help with CT?

Makes it easier to reason about what a function may do with a buffer argument

More importantly: 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`

Wait a second! What about this?

```
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;
  }
}
```

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 else 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
  - How?
- Should we do this for every potentially unsafe pattern?
  - A: yes, **B: no**

What does FaCT disallow?

No public assignments (in secret context)

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

Is this fundamental? **A: yes**, B: no

No calls to functions with public side-effects

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

Is this fundamental? A: yes, B: no

No % or / on secret data!

$\mathrm{sec}\ \%\ \mathrm{pub},\ \ \mathrm{pub}\ \%\ \mathrm{sec},\ \ \mathrm{sec}_1\ \%\ \mathrm{sec}_2$

Is this fundamental? A: yes, **B: no**

No secret-bounded loops

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

Is this fundamental? **A: yes**, B: no

No memory access at secret index

```
pub_arr[sec], sec_arr[sec]
```

Is this fundamental? A: yes, B: no

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;⟧
```

Slightly more complicated example

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

➡ Doesn't type check!

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;
  {
    secret uint32 tmp = x;
    x = ct_select(y, x, __branch1);
    y = ct_select(tmp, y, __branch1);
  }
  __branch1 = !__branch1;
  {...}
}
```

Slightly more complicated example

(the intuitive transformation)

```
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);
```

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);
}
```

Transforming C code to FaCT

```
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;
}
```

Transforming C code to FaCT

```
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;
        }
    }
}
```

What FaCT doesn't do (yet)

- OOB array access is possible
  - Why is this bad?
- Shifting beyond bit-width is possible
  - Why is this bad?
- Allows you to explicitly leave arrays uninitialized
  - Why is this bad?