UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Side Channel Attacks and Lazy Evaluation

Introduction to Side Channel Attacks, proof techniques, semantics
or: Don’t be intimidated, learn how to fail like a professional!

Marcel Fourné ■ 2019-08-21

Overview

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Cryptography, abstract

Side Channels

Necessary preliminaries for side channels, by example

CONSTANT TIME Criterion

Evaluation Order

Developing Proofs

cryptographic algorithms, computationally

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

```
1 op :: SecKey -> Data -> Result
```

- secret (key)
  - size of secret may be public knowledge
  - content of the secret must remain unknown to third parties
- non-secret data
- some operation, a function
  - takes time and computational resources to compute
    - may be observable, depending on the attacker (model)
    - may have other observable(!) side effects
  - total functions are most commonly used
  - $\neq$ total addition theorems, which are a mathematical implementation detail for later
- (public) result

What is a side channel?

Basically:

Any observation on a computation which allows inferences on the *content* of the secret key.

- Observations must be quantifiable.
- Inferences must be computable, but may be probabilistic.
  - partial bits
- 1 bit may be enough
- may be repeated (will be, a lot)
- may be preceded by an active attack

Popular side channels

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

- unexpected results, software/hardware induced errors
  - wrong results
  - specifically crafted results
  - error messages
  - correct results after a different time
- timing information (Kocher, 1996; Lipton&Naughton, 1993)
  - absolute values
  - relative differences
- power consumption (Kocher, 1998)
  - power consumption patterns (Simple Power Analysis)
  - power consumption pattern differences (Differential Power Analysis)
- memory access patterns and cache state (Percival, 2005; Bernstein, 2005)
- microarchitectural state (numerous since 2017, mainstream media famous since January 2018)

branching

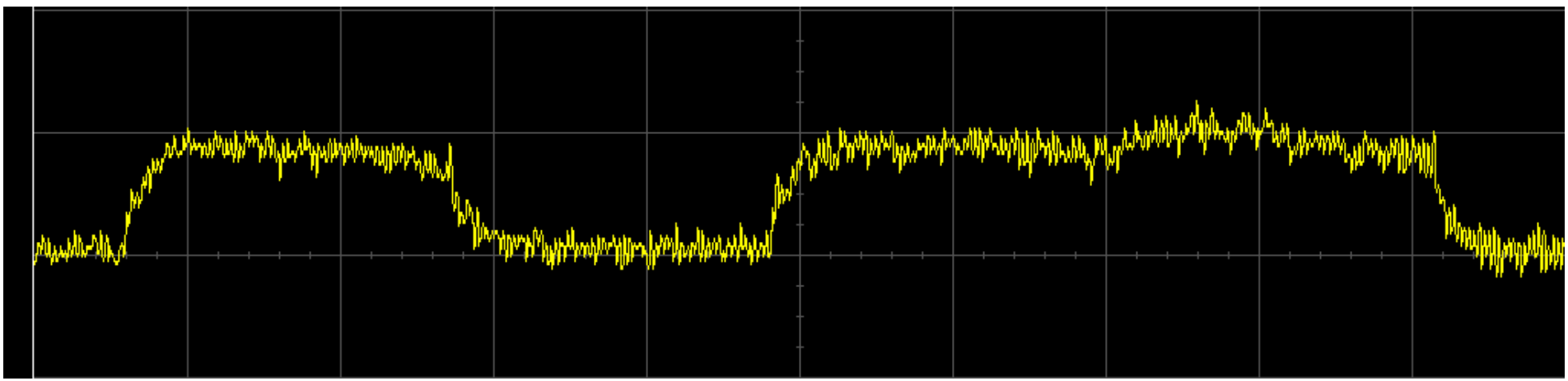
UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1  if(secret_key_bits[n] == 1) {
2    a * square(b)
3  }
4  else {
5    a * b
6  }
```

branching


```
1  if(secret_key_bits[n] == 1) {
2    a * square(b)
3  }
4  else {
5    a * b
6  }
```



branching, pt. deux

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1  if(secret_key_bits[n] == 1) {
2    a * square(b)
3  }
4  else {
5    a * square(c)
6  }
```

branching, pt. deux

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1  if(secret_key_bits[n] == 1) {
2    a * square(b)
3  }
4  else {
5    a * square(c)
6  }
```

branching, pt. trois

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1  if(secretKeyBits[n] == 1) {
2    a * square(1)
3  }
4  else {
5    a * square(0)
6  }
```

branching, pt. trois

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1  if(secretKeyBits[n] == 1) {
2    a ∗ square(1)
3  }
4  else {
5    a ∗ square(0)
6  }
```

secret address indices

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1       a + multiplicationTable [secretKeyBytes[n]]
```

secret address indices

UNIVERSITÄT DUISBURG ESSEN

Open-Minded

```
1       a + multiplicationTable [secretKeyBytes[n]]
```

on which level?

- What about cryptographic code which does not use secret values?
- What about higher level code?
  - The one which uses the cryptographic implementation... ?
  - composability, linking, address spaces
- runtime environment?
- Hardware?
  - a/b is almost never secure
  - value dependent latency multipliers
  - microcode implementations
- ...

Basically, anything observable which is not yet usable as a side channel may still become one.

So... no programs with value divergent behaviour?

The CONSTANT TIME Criterion

- branch free (PROGRAM COUNTER Model)
- no secret value dependent address indices

> "That's it? That's hard enough as it is for Turing Complete languages!"

- does *not* imply constant runtime behaviour (except informally)
  - garbage collection
  - operating system interaction
  - other nondeterministic behaviour
- may be overly prohibitive (see: Montgomery Multiplication)
  - this intuition has been disproven many times
  - formulae must be changed to be total and branch free
  - algebraic style line code instead of lookup tables
- may miss hardware side effects

garbage collection

- Pedersen, Askarov, timing sensitive garbage collection paper
  - requires non CT code
- CT code uses secrets in the same pattern, irrespective of GC interruptions
- in CT code memory may not be used dependent on secret values
- explicitly CT code is not vulnerable against this

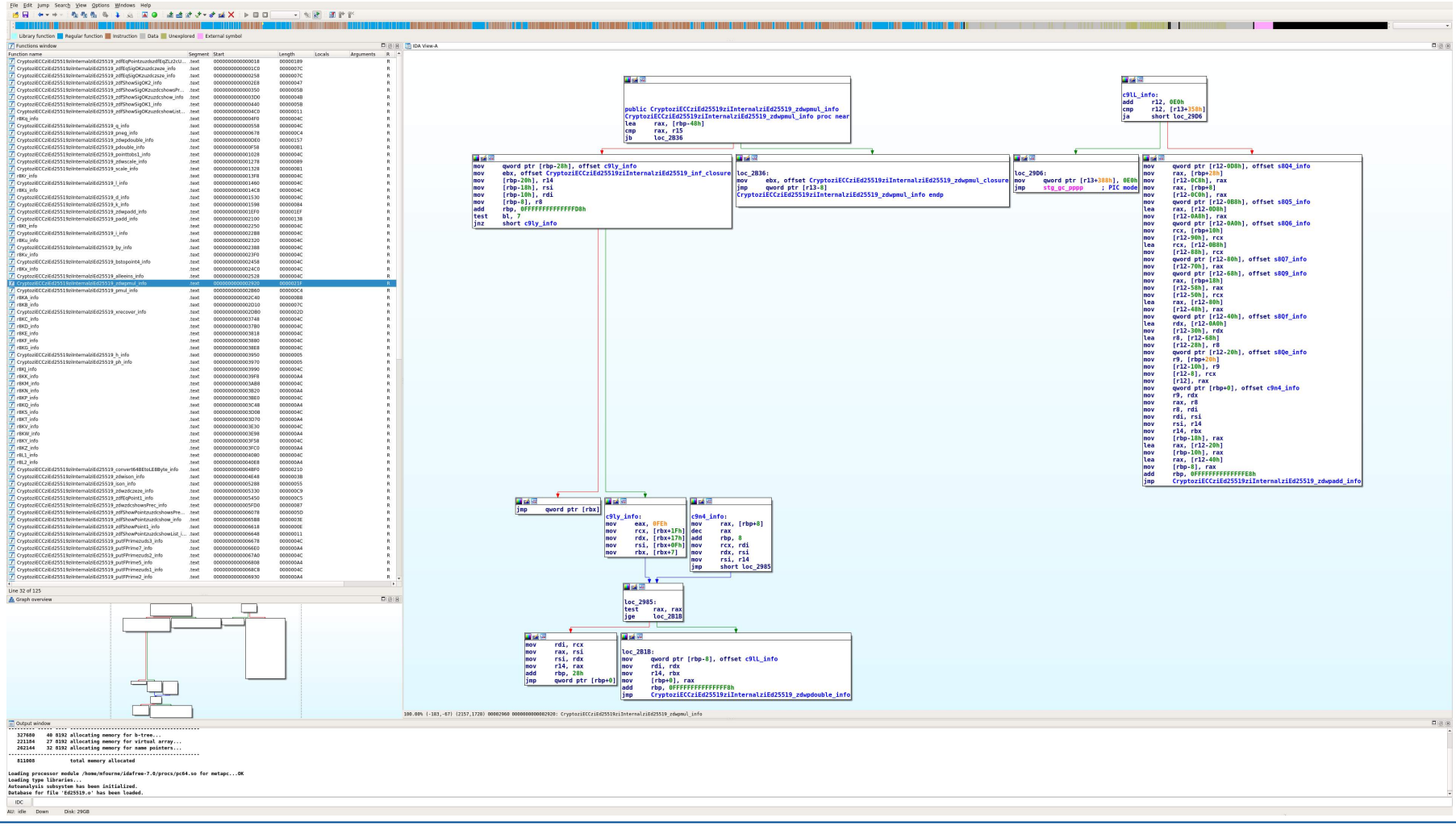
lazy evaluation semantics

- total functional programming implies same behaviour between strict and lazy evaluation
- intuition: if function *f* is called, then the result of *f* is produced by computation
- without branches (etc.) in *f*: if *f* is called, then CT behaviour happens in *f*
- necessary condition: no use (inspection) of secret values outside CT context
  - idea: have a verifiable subset of the program code and contain secret keys to this
  - implies control over copying during optimizations, which may be overly optimistic
- optimizations must be contained
  - let-downfloating?
  - let-upfloating!
  - Continuation Passing Style makes control flow explicit, but confuses IDA
- low-level calling conventions make more problems than evaluation order

manual proof, absence of required code gadgets

UNIVERSITÄT DUISBURG ESSEN

Open-Minded



mechanised

- branch-freeness via AST, type annotated functions
  - find type annotation at low-level Intermediate Language
  - separation of concerns if types are enforced to be created only in some set of modules
- knowest thou thine code generator, lest evil seepest in over yonder
  - type check at which level? pre desugaring? post desugaring?
  - GHC-specific: Tables-Next-To-Code, ghc-asm.lprl, runs after ASM codegen
  - modules with integrated C code, change of memory model
- maybe even scan generated assembly for problematic instructions... or is this too paranoid?

mechanised

- branch-freeness via AST, type annotated functions
  - find type annotation at low-level Intermediate Language
  - separation of concerns if types are enforced to be created only in some set of modules
- knowest thou thine code generator, lest evil seepest in over yonder
  - type check at which level? pre desugaring? post desugaring?
  - GHC-specific: Tables-Next-To-Code, ghc-asm.lprl, runs after ASM codegen
  - modules with integrated C code, change of memory model
- maybe even scan generated assembly for problematic instructions... or is this too paranoid?
- “When Constant-Time Source Yields Variable Time Binary: Exploiting Curve25519-donna Built with MSVC 2015” $\Rightarrow$ know your codegen and runtime!
  - verified C code with unverified runtime libraries yielded vulnerable code

use more Types

- linear
  - notion of “execute exactly once” seems ideal
  - “only once” is sufficient in lazy semantics
- dependent
  - state of the art approach used in implementation adopted by Mozilla Firefox
  - totality (we know this already for our code, but user code may be a different)
  - type inference may be hard
  - efficient code generation is hard, so some subsets are used which mostly one-to-one generate C or ASM constructs in a proof language

use more Types

- linear
  - notion of “execute exactly once” seems ideal
  - “only once” is sufficient in lazy semantics
- dependent
  - state of the art approach used in implementation adopted by Mozilla Firefox
  - totality (we know this already for our code, but user code may be a different)
  - type inference may be hard
  - efficient code generation is hard, so some subsets are used which mostly one-to-one generate C or ASM constructs in a proof language
    - proofs are not automatically transitive across compile chains due to different semantics, transitiveness is proven
    - but use of GCC (performance) vs. CompCert (verified)

Fin!

or: WIP