Side-Channels in Cryptographic Software, the Haskell `case`

attacks, semantics, CT criterion, and compiler construction

Marcel Fourné ■ 2020-01-14

Who am I?

- studied both (pure) Informatics and IT-Security
- self-taught Haskeller since 2005
- free software Haskell ECC
- `https://hackage.haskell.org/package/eccrypto`
    - NIST Prime Curves
    - Ed25519
    - utility code, easy to modify
    - predecessor library has been used for Ripple prototype, alternative Bitcoin etc.
- some work in the Debian Haskell group
- some work in the Haskell cryptography community

Overview

1. Cryptography, abstract
2. Side-Channels
   - Preliminaries for side channels, by example
3. CONSTANT TIME Criterion
4. Haskell
   - Evaluation Order
   - Lazy evaluation side channels
   - Proving Haskell code side-channel silent

Cryptography, abstract

cryptographic algorithms, computationally

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

- secret (key)
  - size of secret may be public knowledge
  - content of the secret must remain unknown to third parties
- data (not secret)
- 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
- result (maybe public)

Side-Channels

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
- observations may be repeated
- may be preceded by an active attack

Popular side channels

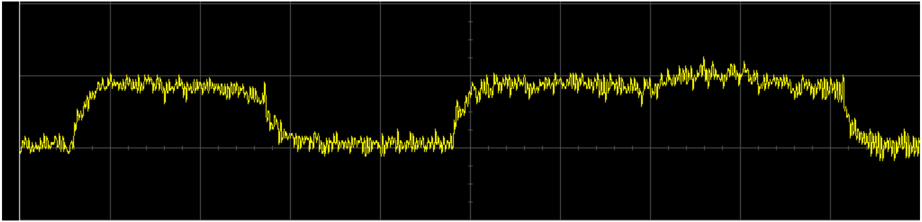
- unexpected results, software/hardware induced errors
  - wrong or specifically crafted results, error messages
  - correct results after a different time
- timing information (Kocher, 1996; Lipton&Naughton, 1993)
  - absolute values
  - relative differences
- power consumption (Kocher, 1998)
  - power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

branching

```
if(secretKeyBits[n] == 1) {
  a ∗ square(b)
}
else {
  a ∗ b
}
```

branching

```
if(secretKeyBits[n] == 1) {
  a * square(b)
}
else {
  a * b
}
```



branching, pt. deux

```
if(secretKeyBits[n]  == 1) {
  a ∗ square(b)
}
else {
  a ∗ square(c)
}
```

branching, pt. deux

```
if(secretKeyBits[n] == 1) {
  a ∗ square(b)
}
else {
  a ∗ square(c)
}
```

branching, pt. trois

```
if(secretKeyBits[n] == 1) {
  a * square(1)
}
else {
  a * square(0)
}
```

branching, pt. trois

```
if(secretKeyBits[n] == 1) {
  a ∗ square(1)
}
else {
  a ∗ square(0)
}
```

secret address indices

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

secret address indices

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

CONSTANT TIME Criterion

The CONSTANT TIME Criterion

We need to prove two things:

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

... but that is hard in a Turing Complete Programming Language

- *suggests* constant run-time behaviour, but:
  - garbage collection (no secrets)
  - operating system interaction and other nondeterministic behaviour
- may be overly prohibitive (see: non-CT variants of Montgomery Multiplication), but:
  - formulae can often be changed to be total and branch free
  - algebraic style line code instead of lookup tables or masked lookups
- may miss hardware side effects not dependent on program structure

on which level?

- CT only needed for code handling secret values
- higher level code not affected (if type system allows no memory accesses across boundaries)
  - composability, linking, address spaces, see C and its common exploits
- operating system interactions
- hardware effects
  - known-bad Assembly instructions
  - value dependent latency multipliers on certain archs
  - microcode implementation replacements

Results:

- anything observable which is not yet usable as a side channel may still become one
- How to prove CT program behaviour? Which program behaviour is still permissible?

CT Proof approaches

- correct by careful programming in Assembly
- manual analysis
- model in proof language, check with its type system
- generate implementation code from proof language
- check in implementation language type system

CT Proof approaches

- correct by careful programming in Assembly
  - no compiler, implementation effort for each platform
- manual analysis
  - error-prone
- model in proof language, check with its type system
  - equivalence of model to implementation, assumptions
- generate implementation code from proof language
  - different language semantics and compilers
- check in implementation language type system
  - often not as expressive as dedicated proof languages

Compilers may change optimisations, but their type system guarantees should hold.

Haskell

Haskell: a standardised lazy functional PL

- non-strict evaluation semantics (graph reduction)
- garbage collection, strictness analysis, HM-style type inference, expressive code
- pure functions, explicit side-effects, easy parallelisation, cross-module inlining
- efficient code can be generated with Instructions Per Cycle > 2; benchmarks rival optimised C
- used in industry (Facebooks anti-spam, Bluespec System Verilog, seL4. . .)
- influences other languages (STM, list comprehensions, monads, QuickCheck)
- active type system research community, committed to correctness, very friendly

Haskell: a standardised lazy functional PL

- non-strict evaluation semantics (graph reduction)
- garbage collection, strictness analysis, HM-style type inference, expressive code
- pure functions, explicit side-effects, easy parallelisation, cross-module inlining
- efficient code can be generated with Instructions Per Cycle > 2; benchmarks rival optimised C
- used in industry (Facebooks anti-spam, Bluespec System Verilog, seL4...)
- influences other languages (STM, list comprehensions, monads, QuickCheck)
- active type system research community, committed to correctness, very friendly

But:

- harder to reason about resource usage (garbage collection and lazy evaluation)
- reputation as hard to learn/too research-centric language

simplified laziness example

```
f 0 x _ = x
f n x y = f (n−1) y x

main = do
  x <− readLn
  let y = f x (ackermann 4 2) (ackermann 0 1)
  print y
```

important constructs: let, case

let: just like stating some unordered lemmas in mathematics before using them in a formula

```
g x = let i = m
          k = x + 2
          m = k + 2
      in i * k * m
```

case: need to evaluate its condition before choice of branch

```
h x = case x of
        0 -> 0
        1 -> x + 1
        _ -> x + 2
```

garbage collection

- Pedersen, Askarov, “From trash to treasure: timing-sensitive garbage collection” S&P’17 paper
  - allocates arrays differently based on secret values
  - requires non CT code
- in CT code memory may not be accessed dependent on secret value content,

  same for allocations
- CT code uses secrets with non-differing memory access patterns
- CT code is not vulnerable against this attack

GHC native compilation chain, simplified

Parsing, *Renaming*, *Typechecking*, and *Desugaring*, which produces `core`

⇓

Simplification works on `core` and it is where most optimisations happen

⇓

`core` is a minimal Haskell with explicit types; `case` for evaluation and branching, `let` for allocation

⇓

STG has thunks for laziness in `let`; three local optimisations after that

⇓

Cmm adds an explicit stack

⇓

RTS.a + the generated *Assembly*, which we analysed to check CT preservation

strict vs. lazy

- 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 subset
  - implies control over copying during optimisations, which may be overly optimistic
- optimisations must be contained
  - Continuation Passing Style makes control flow explicit, but confuses IDA
- low-level calling conventions make more problems for manual analysis than evaluation order

Montgomery Multiplication

As an example, why lazy evaluation can make non-CT code easier to exploit:

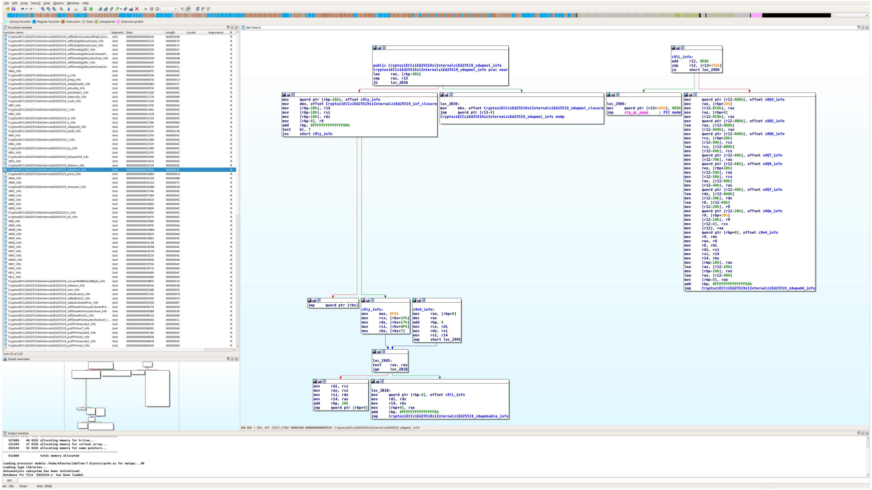
```
pmul :: EC -> Point -> Integer -> Point
pmul curve@(ECi l _ p _) b k  =
  let ex p1 p2 i
        | i < 0 = p1
        | condBit k i == 0 = ex (pdouble curve p1) (padd curve p1 p2) (i - 1)
        | otherwise        = ex (padd curve p1 p2) (pdouble curve p2) (i - 1)
  in ex b (pdouble curve b) (log2len k - 2)
```

Montgomery Multiplication

As an example, why lazy evaluation can make non-CT code easier to exploit:

```
pmul :: EC -> Point -> Integer -> Point
pmul curve@(ECi l _ p _) b k =
  let ex p1 p2 i
        | i < 0 = p1
        | condBit k i == 0 = ex (pdouble curve p1) (padd curve p1 p2) (i - 1)
        | otherwise        = ex (padd curve p1 p2) (pdouble curve p2) (i - 1)
  in ex b (pdouble curve b) (log2len k - 2)
```

manual proof, absence of required code gadgets



```
public CryptoziECCziEd25519ziInternalziEd25519_zdwpmul_info
CryptoziECCziEd25519ziInternalziEd25519_zdwpmul_info proc near
lea     rax, [rbp-48h]
cmp     rax, r15
jb      loc_2B36
```

```
mov     qword ptr [rbp-28h], offset c9ly_info
mov     ebx, offset CryptoziECCziEd25519ziInternalziEd25519_inf_closure
mov     [rbp-20h], r14
mov     [rbp-18h], rsi
mov     [rbp-10h], rdi
mov     [rbp-8], r8
add     rbp, 0FFFFFFFFFFFFFFD8h
test    bl, 7
jnz     short c9ly_info
```

```
loc_2B36:
mov     ebx, offset CryptoziECCziEd25519ziInternalziEd25519_zdwpmul_closure
jmp     qword ptr [r13-8]
CryptoziECCziEd25519ziInternalziEd25519_zdwpmul_info endp
```

```
c9lL_info:
add     r12, 0E0h
cmp     r12, [r13+358h]
ja      short loc_29D6
```

```
loc_29D6:
mov     qword ptr [r13+388h], 0E0h
jmp     stg_gc_pppp     ; PIC mode
```

```
mov     qword ptr [r12-0D8h], offset s8Q4_info
mov     rax, [rbp+28h]
mov     [r12-0C8h], rax
mov     rax, [rbp+8]
mov     [r12-0C0h], rax
mov     qword ptr [r12-0B8h], offset s8Q5_info
lea     rax, [r12-0D8h]
mov     [r12-0A8h], rax
mov     qword ptr [r12-0A0h], offset s8Q6_info
mov     rcx, [rbp+10h]
mov     [r12-90h], rcx
lea     rcx, [r12-0B8h]
mov     [r12-88h], rcx
mov     qword ptr [r12-80h], offset s8Q7_info
mov     [r12-70h], rax
mov     qword ptr [r12-68h], offset s8Q9_info
mov     rax, [rbp+18h]
mov     [r12-58h], rax
mov     [r12-50h], rcx
lea     rax, [r12-80h]
mov     [r12-48h], rax
mov     qword ptr [r12-40h], offset s8Qf_info
lea     rdx, [r12-0A0h]
mov     [r12-30h], rdx
lea     r8, [r12-68h]
mov     [r12-28h], r8
mov     qword ptr [r12-20h], offset s8Qe_info
mov     r9, [rbp+20h]
mov     [r12-10h], r9
mov     [r12-8], rcx
mov     [r12], rax
mov     qword ptr [rbp+0], offset c9n4_info
mov     r9, rdx
mov     rax, r8
mov     r8, rdi
mov     rdi, rsi
mov     rsi, r14
mov     r14, rbx
mov     [rbp-18h], rax
lea     rax, [r12-20h]
mov     [rbp-10h], rax
lea     rax, [r12-40h]
mov     [rbp-8], rax
add     rbp, 0FFFFFFFFFFFFFFE8h
jmp     CryptoziECCziEd25519ziInternalziEd25519_zdwpadd_info
```

```
jmp     qword ptr [rbx]
```

```
c9ly_info:
mov     eax, 0FEh
mov     rcx, [rbx+1Fh]
mov     rdx, [rbx+17h]
mov     rsi, [rbx+0Fh]
mov     rbx, [rbx+7]
```

```
c9n4_info:
mov     rax, [rbp+8]
dec     rax
add     rbp, 8
mov     rcx, rdi
mov     rdx, rsi
mov     rsi, r14
jmp     short loc_2985
```

```
loc_2985:
test    rax, rax
jge     loc_2B1B
```

```
mov     rdi, rcx
mov     rax, rsi
mov     rsi, rdx
mov     r14, rax
add     rbp, 28h
jmp     qword ptr [rbp+0]
```

```
loc_2B1B:
mov     qword ptr [rbp-8], offset c9lL_info
mov     rdi, rdx
mov     r14, rbx
mov     [rbp+0], rax
add     rbp, 0FFFFFFFFFFFFFFF8h
jmp     CryptoziECCziEd25519ziInternalziEd25519_zdwpdouble_info
```

```
00002960 0000000000002920: CryptoziECCziEd25519ziInternalziEd25519_zdwpmul_info
```

```
--------- ----- ---- --------------------------------------------
   327680    40 8192 allocating memory for b-tree...
   221184    27 8192 allocating memory for virtual array...
   262144    32 8192 allocating memory for name pointers...
-----------------------------------------------------------------
   811008            total memory allocated

Loading processor module /home/mfourne/idafree-7.0/procs/pc64.so for metapc...OK
Loading type libraries...
Autoanalysis subsystem has been initialized.
Database for file 'Ed25519.o' has been loaded.
```

Montgomery Multiplication

Also, how to find this at the assembly level:

```
pmul :: EC -> Point -> Integer -> Point
pmul curve@(ECi l _ p _) b k  =
  let ex p1 p2 i
          | i < 0 = p1
          | condBit k i == 0  = ex (pdouble curve p1) (padd curve p1 p2) (i - 1)
          | otherwise   = ex (padd curve p1 p2) (pdouble curve p2) (i - 1)
  in ex b (pdouble curve b) (log2len k - 2)
```

Compiles to this comparison for the branch criterion:

```
1  mov r8, [rbx+rdi*8+10h]
2  test r8, r8
3  jnz loc_3780
```

The address in line 1 contains a key content derived value, so our analysis flagged this branch non-CT.

mechanised

- know your code generator and run-time to prevent unintentional miscompilation of security mechanisms
    - type check at which level? pre desugaring? post desugaring?
    - after `core`: some graph optimisations on `STG` (Shared Term Graph/Spineless, Tagless G-machine)
    - GHC-specific: Tables-Next-To-Code, `ghc-asm.lprl`, runs after Assembly code generator
    - modules with integrated C code, change of memory model

mechanised

- know your code generator and run-time to prevent unintentional miscompilation of security mechanisms
  - type check at which level? pre desugaring? post desugaring?
  - after `core`: some graph optimisations on `STG` (Shared Term Graph/Spineless, Tagless G-machine)
  - GHC-specific: Tables-Next-To-Code, `ghc-asm.lprl`, runs after Assembly code generator
  - modules with integrated C code, change of memory model
- “When Constant-Time Source Yields Variable Time Binary: Exploiting Curve25519-donna Built with MSVC 2015”
  - verified C code executed with dependencies on unverified run-time libraries yielded vulnerable code

mechanised, Haskell future

- branch-freeness via Abstract Syntax Tree, type annotated functions
- find type annotation at low-level Intermediate Language (`core`)
- build AST of function using Template Haskell and find problematic constructs, check at compile-time
- separation of concerns if types are enforced to be created only in some set of modules
- maybe even scan generated assembly for problematic instructions

Further work (replacing `GMP`):

- typed Assembly-level functions: `timesWord2# :: Word -> Word -> (# Word, Word #)`
- reduction of field elements scheduled via type-level lifted overflow information ⇒provably non-overflowing field elements at compile-time

finding more interesting applications for type systems

- linear/affine type systems (Rust, Linear Haskell)
  - do not supplant CT-safety
  - may make more optimisations safe
- dependent types (Coq, Agda, Dependent Haskell)
  - state of the art approach used in implementation adopted by Mozilla Firefox
  - totality
  - type inference may be hard
  - efficient code generation is hard, so some subsets are used which mostly one-to-one generate C or Assembly constructs in a proof language

finding more interesting applications for type systems

- linear/affine type systems (Rust, Linear Haskell)
  - do not supplant CT-safety
  - may make more optimisations safe
- dependent types (Coq, Agda, Dependent Haskell)
  - state of the art approach used in implementation adopted by Mozilla Firefox
  - totality
  - type inference may be hard
  - efficient code generation is hard, so some subsets are used which mostly one-to-one generate C or Assembly 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!