Constant time big integers

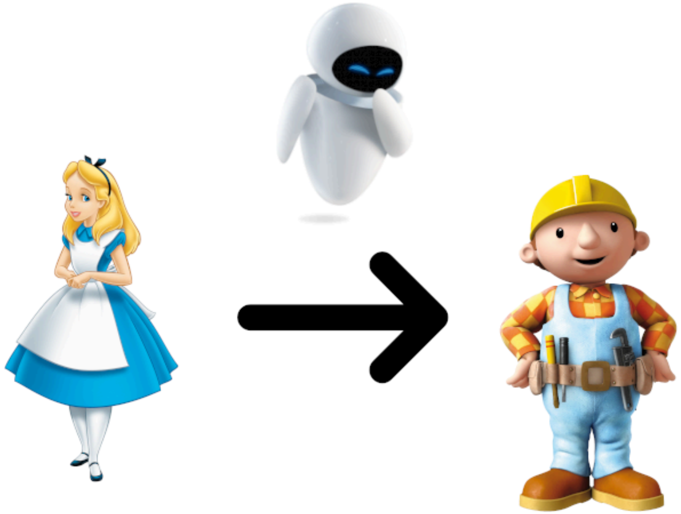
in Ada and SPARK

César SAGAERT

https://github.com/AldanTanneo/bigints

The need for constant time cryptographic primitives

- Avoid side channels in cryptography code
- Provide strong primitives for the Ada ecosystem



What I wrote

- Big integer library in Ada and SPARK, for the Ada ecosystem
- Design and algorithms heavily inspired by Rust's `crypto-bigint` [1]
- All functions in constant time unless explicitly named `*_Vartime`

*Disclaimer: unaudited code written by a single person, **do not** use for security critical applications* 😠

"Constant time" algorithms:

- no branching depending on secret values
- no memory accesses depending on secret values

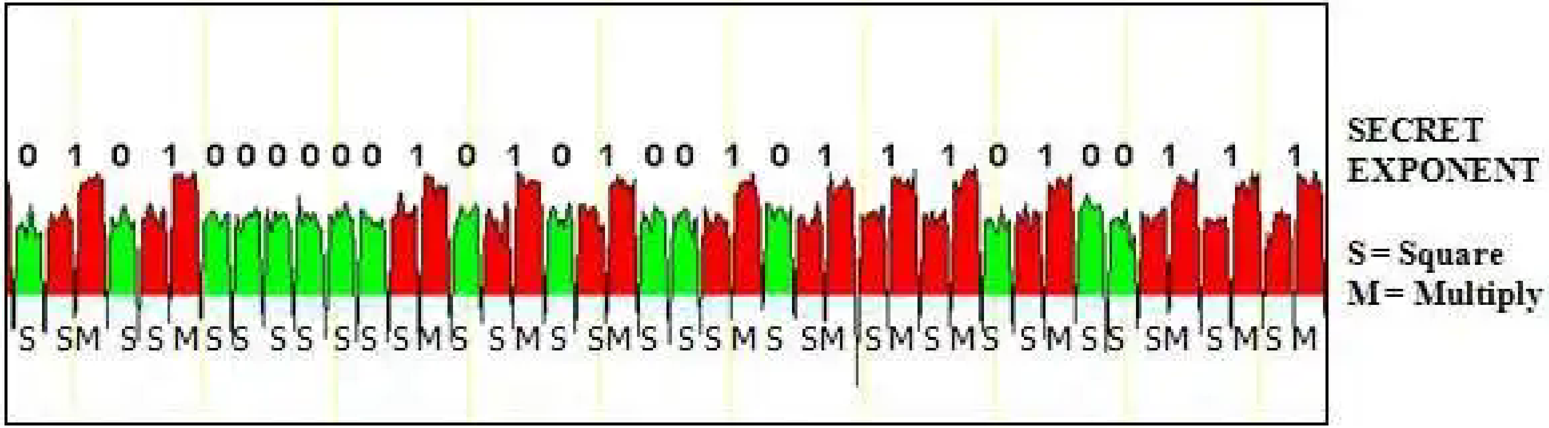
This includes hidden variable-time constructs like the `div` instruction on x86...

Side channels from non-constant time implementations

- timing / power consumption attacks
- cache access analysis

for ex: binary exponentiation leaks secret RSA exponent (power analysis) or number of 1 bits in exponent (timing analysis)

What does constant-time mean anyway



But:

- Code less readable, harder to audit
- Code is (usually) slower
- Optimizing compilers might recognize your constant-time algorithm and replace it with a faster, variable-time one! 😖

Binary exponentiation

Variable time

```
function Pow (A : Uint; N : Natural)
return Uint is
  Y : Uint := ONE;
  X : Uint := A;
begin
  for I in 0 .. BITS - 1 loop
    if Bit (N, I) then
      Y := Y * X;
    end if;
    X := X * X;
  end loop;
  return Y;
end Pow;
```

Constant time

```
function Pow (A : Uint; N : Natural)
return Uint is
  Y : Uint := ONE;
  X : Uint := A;
  B : Boolean;
begin
  for I in 0 .. BITS - 1 loop
    B := Bit (N, I);
    Y := Cond_Select (Y, Y * X, B);
    X := X * X;
  end loop;
  return Y;
end Pow;
```

The `Cond_Select` primitive is implemented with `xor`-ing and masking:

$$f(a, b, \mathrm{mask}) = a \oplus (\mathrm{mask} \cdot (a \oplus b))$$

Some care must be taken when transforming a boolean condition into a mask, so as to not have a hidden branch from compiler optimization.

Formal proof of contracts in SPARK

What SPARK brings to the table

- Guarantee no runtime errors
- Proof of bitwise operations used in CT algorithms

```
function Cond_Select (A, B : Uint; C : Choice) return Uint with
   Post => Cond_Select'Result = (if To_Bool (C) then B else A);

procedure CSwap (A, B : in out Uint; C : Choice) with
   Post => (if To_Bool (C) then B = A'Old and then A = B'Old
            else A = A'Old and then B = B'Old);
```

⚠️ We cannot prove that they effectively are in constant time

In fact, all constant-time code that is not written in raw assembly relies on best effort implementations

- Obfuscate intent from the compiler
- Add optimization barriers
  for ex: `System.Machine_Code.Asm ("", Volatile => True)`

To convince ourselves that our code is constant time

- Audit assembly output
- Measure execution time w. statistics on cycles, cache, power draw...

High level proofs

- Higher level proofs in the library? using Ada’s `Big_Integer` package to model own representation
- Proving for ex. that `A + B` gives the correct result: accumulate the result and prove the partial results are correct: ✅ 🎉
- Current **GNATProve** Alire package cannot prove these yet, so proof code is in a separate branch (`addition-proof`)

Little showcase

- Test suite includes small elliptic curve implementation using the library's modular integer facilities [2], [3]
- Define operations on a prime field, by instantiating a generic package:

```
package U256s is new Bigints.Uints (256);
subtype U256 is U256s.Uint;
use U256s;

P : constant U256 := Shl (ONE, 255) - From_U64 (19);
--  Prime P = 2^255 - 19

package GF_P is new Bigints.Modular (U256s, P);
--  Prime field GF(P), used in Curve25519 operations 😄
```

- Available in the Alire index!

References

[1] "RustCrypto: Cryptographic Big Integers." [Online]. Available: https://github.com/RustCrypto/crypto-bigint

[2] "Constant time big integers in SPARK." [Online]. Available: https://github.com/AldanTanneo/bigints

[3] "Elliptic Curves for Security." [Online]. Available: https://www.rfc-editor.org/rfc/rfc7748