Exponentiation

Tanja Lange

Eindhoven University of Technology

10 September 2020

Right–to–Left Binary

IN: Non-zero positive integers a, b, n, with $b = (b_{\ell-1} \dots b_0)_2$.
OUT: $c \equiv a^b \bmod n$.

```
1. $c \leftarrow 1, t \leftarrow a$,
2. for $i = 0$ to $\ell - 1$ do
    2.1 if $b_i = 1$ then $c \leftarrow c \cdot t \bmod n$
    2.2 $t \leftarrow t^2 \bmod n$
3. return $c$
```

Example
$42 = (101010)_2 = 2^5 + 2^3 + 2^1$, so $\ell = 6$ is minimal
We see the following intermediate states of (c, t):
$(1, a)$ initialization
$(1, a^2)$ no 2^0 contribution
(a^2, a^4) has 2^1
(a^2, a^8) no 2^2 contribution
(a^{10}, a^{16}) has 2^3
(a^{10}, a^{32}) no 2^4 contribution
(a^{42}, a^{64}) has 2^5 We could have skipped computing a^{64}.

Left–to–Right Binary

IN: Non-zero positive integers a, b, n, with $b = (b_{\ell-1} \ldots b_0)_2$.
OUT: $c \equiv a^b \bmod n$.

```
1. $c \leftarrow 1$
2. for $i = \ell - 1$ to 0 do
     2.1 $c \leftarrow c^2 \bmod n$
     2.2 if $b_i = 1$ then $c \leftarrow c \cdot a \bmod n$
3. return $c$
```

Example
$42 = (101010)_2 = 2^5 + 2^3 + 2^1$, so $\ell = 6$ is minimal
We see the following intermediate states of c:
1 initialization
a has 2^5
a^2 no 2^4 contribution
a^5 has 2^3
a^{10} no 2^2 contribution
a^{21} has 2^1
a^{42} no 2^0 contribution
Only 1 variable to update. Same number of squarings and multiplications.

Windowing methods

Windowing methods use precomputed powers of a, so this does not work with the data flow as in the first method – we would need to square all precomputed values at each step, which is much more work than what we save in the multiplications.

The exponentiation thus uses the left-to-right method.

```
IN: $a, b, n \in \mathbb{Z}_{>0}$, with $b = \sum_{i=0}^{\ell-1} b_i 2^i$ with $b_i \in \{0, 1, 2, \ldots, 2^w - 1\}$.
OUT: $c \equiv a^b \bmod n$.
  1. for $i = 0$ to $2^w - 1$ do
      1.1 $A[i] = a^i \bmod n$ // this takes 1 mult. as $A[i] = a \cdot A[i-1]$.
  2. $c \leftarrow 1$
  3. for $i = \ell - 1$ to $0$ do
      3.1 $c \leftarrow c^2 \bmod n$
      3.2 if $b_i \neq 0$ then $c \leftarrow c \cdot A[b_i] \bmod n$
  4. return $c$
```

Windowing methods

Windowing methods use precomputed powers of a, so this does not work with the data flow as in the first method – we would need to square all precomputed values at each step, which is much more work than what we save in the multiplications.

The exponentiation thus uses the left-to-right method.

To *compute* the coefficients it is easiest to process the bits of the exponent b starting from the least-significant bit.

Example
$42 = (101010)_2$

Windowing methods (fixed)

Windowing methods use precomputed powers of a, so this does not work with the data flow as in the first method – we would need to square all precomputed values at each step, which is much more work than what we save in the multiplications.

The exponentiation thus uses the left-to-right method.

To *compute* the coefficients it is easiest to process the bits of the exponent b starting from the least-significant bit.

Example
$42 = (\underline{10}\ \underline{10}\ \underline{10})_2 = (020202) = 2 \cdot 2^4 + 2 \cdot 2^2 + 2 \cdot 2^0$ for window width $w = 2$.
So we precompute $A = [a^3, a^2, a, 1]$ (only need a^2 in this example).
We see the following intermediate states of c, $\ell = 5$:
1 initialization
a^2 has $2 \cdot 2^4$
a^4 step after multiplication is only a squaring
a^{10} has $2 \cdot 2^2$
a^{20} step after multiplication is only a squaring
a^{42} has $2 \cdot 2^0$

Windowing methods (fixed)

Windowing methods use precomputed powers of a, so this does not work with the data flow as in the first method – we would need to square all precomputed values at each step, which is much more work than what we save in the multiplications.

The exponentiation thus uses the left-to-right method.

To *compute* the coefficients it is easiest to process the bits of the exponent b starting from the least-significant bit.

Example
$42 = (\underline{101}\ \underline{010})_2 = (005002) = 5 \cdot 2^3 + 2 \cdot 2^0$ for window width $w = 3$.
So we precompute $A = [a^7, a^6, a^5, a^4, a^3, a^2, a, 1]$ (only need a^5 and a^2 in this example).
We see the following intermediate states of c, $\ell = 4$:
1 initialization
a^5 has $5 \cdot 2^3$
a^{10} 1. step after multiplication is only a squaring
a^{20} 2. step after multiplication is only a squaring
a^{42} has $2 \cdot 2^0$

Windowing methods (sliding)

Windowing methods use precomputed powers of a, so this does not work with the data flow as in the first method – we would need to square all precomputed values at each step, which is much more work than what we save in the multiplications.

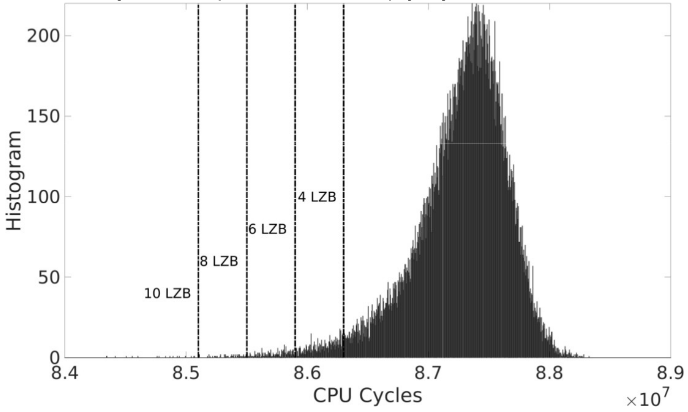
The exponentiation thus uses the left-to-right method.

To *compute* the coefficients it is easiest to process the bits of the exponent b starting from the least-significant bit.

Example
$42 = (\underline{1}\ 00\ \underline{101}\ 0)_2 = (100050) = 2^5 + 5 \cdot 2$ for sliding window $w = 3$.
So we precompute $A = [a^7, a^6, a^5, a^4, a^3, a^2, a, 1]$
We see the following intermediate states of c, $\ell = 6$:
1 initialization
a has $1 \cdot 2^5$
a^2 1. step after multiplication is only a squaring
a^4 2. step after multiplication is only a squaring
a^8 bonus squaring
a^{21} has $5 \cdot 2$
a^{42} 1. step after multiplication is only a squaring

Timings of using windowing method

Watch for side channels using timing. Do not use `if/else` but a pattern of steps independent of secrets, e.g. fixed windows with w squarings followed by a multiplication: multiply by 1 if the coefficient is 0.



This plot is from an attack paper (TPM-Fail) where the attack used that for some exponents the first bits are all 0, which is visible in the overall speed, if the number of loops in the exponentiation method varies with the exponent.