

Fast square and multiply modular exponentiation (FSMMAE)

Abstract

Modular exponentiation: $(a^b) \bmod n$ can be computed using square and multiply exponentiation algorithm¹. The present algorithm in this repository describes a faster variant stopping the computation when intermediary modulus is 0.

Observations

The modulus of a number by a number that begins with a number it begins with is the same as the modulus of the number by the number it ends with

$$2555 \bmod 25 = 55 \bmod 25$$

The number begins 25, so the modulus of 2555 by 25 is the same as the modulus of 55 by 25. Both are 5.

¹Christof Paar, Jan Pelzl, Understanding Cryptography, section 7.4



Why? Because $2555 = 25 * 100 + 55$ and modulus of $25 * 100$ by 25 is 0 , so the modulus of 2555 by 25 is the same as the modulus of 55 by 25 .

A generalization of the previous observation is that a modulus of a number is equal to the modulus of this number plus another number which modulus is equal to 0

$$5055 \bmod 25 = 55 \bmod 25$$

So can we compute only the smaller number modulus to be faster?

Yes, using the square and multiply modular exponentiation algorithm and storing the last modulus. If the current one is equal to 0 , the computation can stop, and the algorithm can return the last modulus stored.

Python implementation

Prototyping is done in Python for the ease of development and testing. The corresponding module is `pysma.py`. The reference implementation is `pow` Python built-in method.

C implementation

For a portable implementation and low-level profiling of the executables, C is used to test than new implementation. It is a transpile of the Python code. The classic square and multiply modular exponentiation is implemented in `sma.c`. The fast square and multiply modular exponentiation is implemented in `fsma.c`. Again the reference implementation is `pow` Python built-in method.

Dependencies

Compilation and tests need LLVM, cmake, Clang and Python3.

Compilation

Both sources are compiled as an executable to be used in CLI and shared library to be loaded using Python for testing purposes. The compilation is done using the following command:

```
make all
```

Testing

The tests are done using a number inferior to a C unsigned long integer, the type used in C programs, as base, exponent, and modulus. Running the corresponding Python module should end in return code 0 if the result is the same as the reference implementation `pow` otherwise an `assertionError` is raised.

Test results Both `sma.c` and `fsma.c` compiled shared libraries returned results equal to the reference implementation `pow` for all the numbers inferior to a C unsigned long integer tested.

Hypothesis

The FSMMEA algorithm is generally faster than the SMA algorithm because of the computation ending sooner in case a temporary modulus is equal to 0. The impact of storing the last modulus and comparing the current to 0 is negligible.

Performance analysis

Ongoing