LSE

my_prime

22^{nd} November 2022

Version 1.2

Martin GRENOUILLOUX <martin.grenouilloux@lse.epita.fr>

Maintainer : **Martin GRENOUILLOUX**
<martin.grenouilloux@lse.epita.fr>

Tag : **[RECRUT][CRYPTO]**

Submission : git
Repository url : login@git.cri.epita.fr:p/lse/my_prime-login
Deadline : 22^{nd} December 2022

Language : C (Except for part 2)

Required files:

AUTHORS
Makefile
README

Expected architecture for this project is as follows:

```
./my_prime
├── AUTHORS
├── README
├── exploration
│   ├── ...
│   ├── ...
│   └── report.pdf
└── generation
    ├── Makefile
    └── src
        ├── ...
        ├── ...
        └── ...
```

Contents

1 my_prime **5**

2 Generation **5**
2.1 Technical details 5
2.2 Examples 6

3 Exploration **7**
3.1 PSW Conjecture 7
3.2 Technical details 7

4 Contact **7**

5 Brainteaser **8**

1 my_prime

This is the 2025 LSE recruitment project for cryptology. It contains two main parts: Generation and Exploration. You are expected to do as much as possible on both parts. Notice they are independant from each other but it is strongly advised to start by part 1.

All work, even unfinished, will be taken into account for evaluation.

2 Generation

Prime numbers play a major role in modern cryptography. They offer a trapdoor function based on hardness of prime factorization and on the discrete logarithm problem. Although they are doomed to be replaced in cryptography with the uprising quantum computers, they still are relevant and a challenge is to find them as fast as possible.

Thus, you have to create a prime number generator. This can be done in three main steps.

- handling big integers for real cryptographic applications (you are allowed to use an extern library for operations on bigints)
- a primality testing algorithm of your choice
- an entropy pool for your PRNG (CS?)

In real life applications, primality testing code is executed a lot when generating key pairs. Your code has to be optimized: performance will then be part of evaluation.

2.1 Technical details

The code must be written in C. You are required to create all the cryptographic primitives needed for this project, thus you must not import them from existing libraries, with the exception of big integers. You can use a library that offers operations on big integers (mind the linking flags !) but you are strictly restricted to that. A suggestion of a good library is `openssl/bn.h` (see References).

You have to handle three options

- `-g <bits>` for generation of a N bits prime
- `--hex` used with `-g` to output the prime in hexadecimal form
- `-t <number>` for primality testing of a number in hexadecimal form

You are free to compile with whatever flags you want or need. Keep in mind that submitted code must be clean, compile with no errors, no warnings and must not leak memory.

No specific coding style is expected but the overall quality of the code does matter.

An example architecture is as follows

```
.
├── AUTHORS
├── README
└── src
    ├── main.c
    ├── Makefile
    ├── prime.c
    └── prime.h
```

A Makefile should be provided and produce an executable named `my_prime` upon compiling with the command `make` or `make all`.

2.2 Examples

```
$ ./my_prime -g 256
    67245145840401458372951609659399223694241814307864310846212
    663975918096155839

$ ./my_prime -g 256 --hex
    94ab675d8d079fe539ab9d6b3b0f617fc63949c8082c54ce6a9973576b1
    904bf

$ time ./my_prime -g 4096
    568937384720065961620186....14569469748969

real 0m8,667s
user 0m8,648s
sys     0m0,013s
# May vary

$ ./my_prime -t 94ab675d8d079fe539ab9d6b3b0f617fc63949c8082c54ce\
    6a9973576b1904bf
$ echo $?
    1

$ ./my_prime -t 3acf
$ echo $?
    0
```

3 Exploration

This part is independant from the first one but in its continuation. Because you are supposed to have implemented you own primality testing algorithm in part 1, you are now familiar with methods to estimate the compositeness of a number. One of them relies on Lucas sequences that you may not know, but you surely know one of its derivatives: the Fibonacci sequence. They provide strong divisibility properties that we use to compute if a number is a pseudoprime to a basis relative to certain properties or not. On this direction, an original conjecture has been announced by Pomerance, Selfridge and Wagstaff that we will call the PSW conjecture (it does not appear to have a name somehow).

3.1 PSW Conjecture

It states that for an odd number n congruent to $\pm 2 \ mod \ 5$, such that the two following equations state true:

$$2^{n-1} \equiv 1 \ mod \ n$$

$$F_{n+1} \equiv 0 \ mod \ n$$

then n is a prime number. In other words, if n is both a base-2 pseudoprime and a fibonacci pseudoprime congruent to $\pm 2 \ mod \ 5$, then it is a prime number[2].
You have to prove this. First try to reformulate the statement, split it into two different ones and see how you could link those two. And then try to find an angle of research to better apprehend this problem.
Trying to prove the opposite or find a counter-example also works, it is up to you. In this case, you will need to show a factorization (full or partial) of n.

3.2 Technical details

You are free to use the tools and languages of your choice. A `PDF` file exlaining your work and results is expected. Be aware that this conjecture remains unproven and evaluation will mostly be on your initiatives and your ability to apprehend a research project on modern cryptography.

4 Contact

Feel free to ask questions on the newsgroup `labos.lse` or by email at `recrutement@lse.epita.fr` with the tag [RECRUT][CRYPTO].

A channel is also available on the 2025 discord server: `#recrutements-lre-sys-secu` where we will be happy to answer questions and more generally exchange on the laboratory.

5 Brainteaser

If you want to take a break from the subject, here is a small RSA brainteaser.

We generated a RSA public key but the public exponent doesn't seem to be a unit mod n. Not a problem right ?

```
n = 76984458963924128591639791957323669240759827243945718971
9891637458722740115041246264837362983517
e = 218382435217703511591993997357505687929
c = 31050170958855674952940113455942905421640100487079822991
0210721885692988345670878493184185589283
```

References

[1] Primes and Prejudice, primality testing under advrsarial conditions https://eprint.iacr.org/2018/749.pdf

[2] Prime Numbers: A computational Perspective - Sections 3 and 3.9 http://thales.doa.fmph.uniba.sk/macaj/skola/teoriapoli/primes.pdf

[3] FIPS 186-4, Section C and F (but in general the whole document is just a reference) https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf

[4] OpenSSL cryptography and SSL/TLS toolkit - Bignums https://www.openssl.org/docs/man1.0.2/man3/bn.html