Timing Side Channel Attack Report

This report shows the steps we have been through before getting to the final result. According to the actual chronological order those steps have been performed, the main sections are:

- Theoretical references
- Side channel attack
- Montgomery based RSA encryption
- Code development
- Big integer library
  - Data type
  - Operations
- RSA ecryption
- Code validation
- Data acquisition
- Bare metal Zybo Board acquisition
- OS acquisition
- Attack
- Attack algorithm
- Attack results
- Countermeasures
- Conclusions
- Practical references

Before starting, clone the git repository and get into it:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
$ cd some-where
$ git clone git@gitlab.eurecom.fr:roggero/tsca.git
$ cd tsca
```

Theoretical references

Side channel attack

The name side channel attack refers to any attack based on a certain amount of information obtained from a computer system on which statistics could be computed. Relating those probabilistic statistics to the actual knowledge of the internal operation of the system, secrets related to the internal operations themselves could be disclosed.

In our case, the attack will be a timing attack on an RSA crypto-computation (Montgomery modular algorithm based). This meas that, basing on the knowledge of the actual algorithm used by the encryption and exploiting its timing weaknesses, the attacker could obtain timing measurements on a series of encryptions and could analyze them, disclosing the secret key. More on the Montagomery based RSA ecryption is explained in the section Montgomery based RSA encryption, while for the attack algorithm employed refer to Attack algorithm.

Montgomery based RSA encryption

The RSA ecryption algorithm involves two steps:

- key pair generation
- modular exponentation and multiplication based ecryption

For the key pair generation, first two distinct large prime number (`p`,`q`) have to be found. Then, the modulus `n` is computed as the product of the two prime numbers. The Eulero's totient `t` is successively computed as the product

‘$t = (p-1) \cdot (q-1)$‘

and the public exponent `e` is chosen such that

‘$1 < e < t$‘

and

‘$gcd(e, t) = 1$‘ .

Finally, the secret exponent `d` is chosen such that

‘$d \cdot e = 1 \; mod \; t$‘ .

The pair `(n,e)` constitutes the public key, while the pair `(n,d)` the secret one.

To perform the encryption of a message `m` to obtain the ciphertext `c`, the following operation is performed:

‘$c = m^e \; mod \; n$‘ .

This computation consists of two main operations: modular multiplication and exponentiation. The implementation we adopted takes advantage instead of the Montgomery multiplication: the multiplier digits are consumed in the reverse order and no full length comparisons are required for the modular reductions (refer to Colin D. Walter paper on the topic). The basic pseudo-code it defines for the Montgomery modular multiplication is:

```
S = 0;
for i = 0 to nb-1 do
  qi  = (s0 + aib0 )(-m^-1 ) mod r;
  S   = (S + ai × b + qi × n) div r;
end for;
return S;
```

where nb is the total number of bits of the secret key, a and b are the first two operands which are determined according to the Montgomery exponentation function:

```
c = MM(k0,1,n);
s = MM(k0,m,n);
for i = 0 to nb-1 do
  if (ei = 1) then
    c = MM(c,s,n);  [multiply]
  end if;
  s = MM(s,s,n);    [square]
end for;
c = MM(c,1,n);
return c;
```

where MM() is the Montgomery multiplication defined by the previous algorithm and k0 is

'$k_0 = 2^n$'

where n is the modulus computed before.

Code development

The starting point to get a working implementation for the Montgomery based RSA encryption is having a library capable of managing integers on a large number of bits (such as 1024 or 2048), since this will be most likely the size that will be used by most of the main core variables (private and public key, for instance). Usually, standard C libraries support numbers up to 128 bits (long double), which is the minimum key size for an admissible time side channel attack on RSA encryption. Thus, an extra library is needed.

After such library is obtained, the pseudo-code presented above (Theoretical references) has to be ported into a real C implementation through the primary functions `Montgomery multiplication` and `Montgomery exponentation`.

Finally, both the library and the RSA encryption have to be checked against a reference and reliable implementation; in our case, it will be a Python one, since this programming language doesn't force any explicitly defined limit to a number object, which makes it an ideal candidate.

Big integer library

There are two main possibilities to manage large integer numbers: rely on an available library (such as GMP (GNU Multiple Precision Arithmetic Library) or OpenSSL) or create one from scratch.

The former implies to understand how it works and to deal with an optimized version of all the main operations. This code optimizations could lead to a significant reduction of the execution time related to data dependencies, making the final attack more complicated.

The latter, on the other side, requires more time to be implemented and tested, but theoretically should guarantee a higher data dependency. Thus, this option is the one used.

The custom library is implemented through the files bigint.h and bigint.c, which define: * the data type we will use to work on large integers * all the main operation needed to perform the Montgomery multiplication and exponentation.

Data type

In the file bigint.h the following parameter are free to be set:

- `VAR_SIZE`: it determines the basic unit to build the larger integer among 8 (uint8_t), 16 (uint16_t), 32 (uint32_t) and 64 (uint64_t) bits. Recommended size is 32.
- `INT_SIZE`: it determines the actual length of the data the system is working on (for instance, public and private key dimensions). Possible sizes are 64, 128, 256, 512, 1024 and 2048 bits.

As a consequence of these two parameter, the code defines the `bigint_t` data type as a struct containing a vector of `NUMB_SIZE` elements of size `VAR_SIZE`, where `NUMB_SIZE` is equal to

‘$NUMB\_SIZE = \frac{INT\_SIZE}{VAR\_SIZE} + 1$‘ .

Thus, the vector will always have and extra element, used to store possible carries due to intermediate operations. Instead of an extra element `VAR_SIZE` long, a couple of bits would have been enough but, to keep the operations implementation simpler and straightforward, the choice fell back on the first solution.

Data are saved in little endian format, i.e. the lowest address in the array contains the lowest chunk of data, and so on.

Operations

All the operations use parameter passed by value (no pointer usage). The library contains the following operations:

- Comparisons (if `first` is `eq`/`df`/`gt`/`ge`/`lt`/`le` to/than `second`, then the comparison returns `1` (true), otherwise returns `0` (false)):
- Equality:
  - int eq(bigint_t first, bigint_t second);
- Diversity:

  - int df(bigint_t first, bigint_t second);
- Greater than:
  - int gt(bigint_t first, bigint_t second);
- Greater or equal:
  - int ge(bigint_t first, bigint_t second);
- Lower than:
  - int lt(bigint_t first, bigint_t second);
- Lower or equal:
  - int le(bigint_t first, bigint_t second);
- Logicals (bitwise operation between `a` and `b`, except the `not`, which reverts `a`):
- Bitwise and:
  - bigint_t and(bigint_t a, bigint_t b);
- Bitwise or:
  - bigint_t or(bigint_t a, bigint_t b);
- Not:
  - bigint_t not(bigint_t a);
- Bitwise xor:
  - bigint_t xor(bigint_t a, bigint_t b);
- Shifts (bigint_t `a` is shifted of `pl` positions (logical shift semantic)):
- Logical shift right:
  - bigint_t lsr(bigint_t a, int pl);
- Logical shift left:
  - bigint_t lsl(bigint_t a, int pl);
- Arithmetics (`a` is `sum`/`sub`/`mul` with `b`):
- Sum:
  - bigint_t sum(bigint_t a, bigint_t b);
- Subtraction:
  - bigint_t sub(bigint_t a, bigint_t b);
- Multiplication (the result is casted to the `bigint_t` size):
  - bigint_t mul(bigint_t a, bigint_t b);
- Utility:
- Init: initialize a variable into the `bigint_t` structure passed a pointer, except the extra element (element `NUMB\_SIZE`):
  - bigint_t init(const char *s);
- Init_full: as `init`, but initialize also the extra element:
  - bigint_t init_full(const char *s);
- Sum_4_mul: special sum for the `mul` operation:
  - var_t sum_4_mul(var_t *a, var_t b, var_t *carry, int act);
- print_to_stdout: print the `bigint_t` number in hexadecimal format (0x..):
  - void print_to_stdout(bigint_t *a);
- rand_b: return a random `bigint_t` number:
  - bigint_t rand_b(void);

To check the actual implemetations of those functions, refer to the file bigint.c.

RSA ecryption

The Montgomery multiplication and exponentiation pseudo-codes (section Montgomery based RSA encryption) are ported instruction by instruction in C with the implemetations reported in the files pair mm.h, mm.c and me.h, me.c. More specifically, refer to the functions `MM_big` and `ME_big`.

Code validation

The `bigint` library and the Montgomery exponentation are now ready to be tested. As previously mentioned, Python is the programming language chosen both to launch several times the test operation and to provided the reference implementation. To compile the code, issue the following commands:

```
$ cd tsca
$ make test
```

It will generate a file called `main` in the same folder. Enter then the folder `test`:

```
$ cd test
```

It contains a set of python test programs which implement reliable versions of the very same C functions listed in C (section Operations). To run the tests on the `bigint` library, type:

```
# Comparisons:
$ python3 comp.py <operation> <numberoftests> <bit>
# Logicals:
$ python3 logic.py <operation> <numberoftests> <bit>
# Shifts:
$ python3 shift.py <operation> <numberoftests> <bit>
# Arithmetics:
$ python3 arith.py <operation> <numberoftests> <bit>
```

The chosen python script will check the custom implementation launching the executabe `main` aginst its intenal implementation. If `<numberoftest>` and `<bit>` are not defined, the program will automatically test for 10000 tests on 128 bits.

To run instead the test on the Montomery operations type:

```
# Multiplication
$ python3 modular.py mm <numberoftests> <bit>
# Exponentation
$ python3 modular.py me <numberoftests> <bit>
```

All the functions have been tested with the approach just shown for a number of tests between 10 millions and 100 millions each. Each iteration of any of the Python scripts uses random numbers generated runtime. Since at the time this report is written no error is detected, the library are supposed to be reliable from now on.

Data acquisition

It's time now to intensively run many Montgomery exponentation encryption on a bunch of messages using different sets of private exponent `e`, modolus `m` and `k0` and obtain the timing measurements associated to each set, to be able afterwards to mount an attack on them.

The different predefined sets are declared in the file cipher.c: the values for `VERSION` in cipher.h and having a predefined value for the key width (set in bigint.h with the parameter `INT_SIZE`) picks up a different set. The number of sets is limited since we don't have a code capable of generating them autonomously. Have a look at them and select one.

Two different codes are available to obtain timing measurements:

-
-

Bare metal Zybo Board acquisition

The folder zybo contains all the necessary files to define an hardware platform which is capable of running a custom code. Inside, the folder ZC010_wrapper_hw_platform_0 specifies a set of information useful to the first stage boot loader to initialize the hardware platform on which our code will run. The folder Test_sd_bsp contains the Xilinx libraries with a set of built-in functions for the board. Finally, the folder Test_sd contains the actual acquisition code (helloworld.c), together with the boot.bin and a set of other configuration files.

To run acquisitions on a OS-less system, in our case the Zybo board, two preliminary steps are necessary: * set the `VERSION` parameter in cipher.h; * set the `INT_SIZE` parameter in zybo bigint.h (the make-able version in the zybo folder, not the original one); * set the `TESTNUM` parameter in helloworld.c (number of total acquisitions).

As just mentioned, the actual acquisition code is contained in the file helloworld.c. Have a look at it:

```
$ cd ..
$ vim ./zybo/Test_sd/src/helloword.c
```

The `main()` function performs the following: * Creates two data file (`PLAIN.BIN` and `TIME.BIN`) that will be written on the same SD card we will plug in the zybo; the first contains the actual value of the message it has been encrypted, the second one the timing measurement related to that encryption; * Initializes the data structure `pair`, which contains the set of private key, public key and modulus; * Initialize the configuration of one led (`MIO7` on the zybo board) that will be turned on when the acquisition is concluded; * Starts the acquisition loop a number of times equal to `TESTNUM`: the message to be encrypted is randomly generated run-time and feeds one Montomery exponentiation, whose execution time in terms of clock cycles is recorded thanks to the Xilinx built in function `XTime_GetTime`, included in the library `xtime_l.h`; finally, write the two data files; * When the acquisition loop is over, the led is turned on and the `main()` returns.

To run an acquisition campaign, plug and SD card in the laptop/pc and type the following commands:

```
$ export PATH=/path-to-/Xilinx/SDK/2018.3/gnu/aarch32/lin/gcc-arm-none-eabi/bin/:$PATH
$ export PATH=/path-to-/Xilinx/SDK/2018.3/bin/:$PATH
$ cd ./zybo/Test_sd/Debug/
$ source set.sh <path-to-SD>
```

This will compile the whole bunch of files, create the boot.bin file, copy it to the SD card and unmount it. At this point, plug the SD card in the zybo board and power it on. When the `MIO7` led will turn on, the acquisition will be completed and the files ready to be read. Be careful, the zybo board is way less powerful than a PC microprocessor, setting a higher number of bits for the secret key and a higher number of acquisitions may need up to days! Setting 128 bits (`INT_SIZE`) and 10000 acquisitions (`TESTNUM`), the board should take around 3 minutes to generate the two files.

To visualize the results, copy the files `PLAIN.BIN` and `TIME.BIN` in the folder data and give them a new name. Then, type:

```
$ cd data
$ ./graph.py newname_TIME.BIN
```

It will appear the resulting distribution of samples versus the number of clock cycles. The following figure is most likely what you should obtain:

and is generated from 100000 samples on a 128 bits key. It is a Gaussian distribution, which shows clearly that, even without the intervention of an operating system, on average the computations take a different time. If we don't consider the uncertainty related to the time measuring functions, this behavior is clearly due to algorithm data dependencies.

OS acquisition

In the folder source we made also available a version capable of obtaining the very same measurements but on a system running an operating system. The file is timing.c, but before obtaining the acquisitions, the following parameters have to be set: * set the `VERSION` parameter in cipher.h; * set the `INT_SIZE` parameter in bigint.h; * set the `TESTNUM` parameter in timing.c; * set the `MODE` parameter in timing.c to decide if the timing measurements will be done on the whole modular exponentiation (`MODE = 0`) or only on the modulus squaring (`MODE = 1`).

Have a look at the file:

```
$ vim ./source/timing.c
```

The `main()`function performs the following: * Initialize the data structure `pair`; * Creates/opens two data files (PLAIN.BIN and TIME.BIN) in the folder data, which will be populated with a number of samples (plaintext ecrypted) according to `TESTNUM`; * For each plaintext, the timing measurements are taken `REPETITIONS` times (set to 10) and the minimum timing is chosen. In this way we try to reduce possible weird measurements due to OS scheduling policies, which may lead to clearly out-of-bound results. The functions on which the timings are taken (`ME_big()` and `MM_big()`) are the very same used in the zybo case. * The functions used for timing measurements are reported in time_meas.h; * A live progress status is printed to screen (it doesn't influence the timing measurements): when the execution is done, the two files are ready to be attacked.

To run the acquisition, type:

```
$ cd ..
$ make timing
$ ./timing
```

As for the zybo case, the time taken by this operation strongly depends on the width of the key chosen and the number of samples required. As a reference, on a single core running at 3.1 GHz, the time needed for 10000 measuremenets is around 4 minutes. The overhead with respect to the zybo is due to the factor 10 added when we look for the minimum.

As before, visualize the results:

```
$ cd data
$ ./graph.py newname_TIME.BIN
```

The following figure was obtained for 20000 samples on a 128 bits key.

As before, a Gaussian distribution is obtained, plus some spare samples with really high execution time, probably related to time windows in which the operating system preempted the script execution. Those will be filtered away during the attack phase, in order to rely only on significant data.

Attack

Attack algorithm

Attack results

Countermeasures

Conclusions

Practical references