WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF FUNDAMENTAL PROBLEMS OF TECHNOLOGY

FIELD:	Computer Science
SPECIALIZATION:	Computer Security

MASTER OF SCIENCE THESIS

Timing Attack Resistant Implementation of RSA on GPU.

AUTHOR:
Krzysztof Hamerski

SUPERVISOR:
dr Maciej Gębala

GRADE:

WROCŁAW 2017

Contents

Chapter 1

Introduction

Public key cryptography is the key factor in providing secure communication between two parties. Fast development of distributed system requiring not only security, but also integrity and non-repudiation has pushed cryptography to the limit. Since 1978[4] most commonly used cryptosystem is RSA, which provides asymmetric encryption, as well as generation of digital signatures. The security of RSA is mainly base on the bitwise key length. As computational power of modern CPUs arises, the minimal bit length of RSA key gets significantly bigger to provide sufficient security. At least 4096 bits long keys are considered secure nowadays. This leads to very high workload required to perform encryption/decryption. RSA is mainly based on modular arithmetics and simple computations become infeasible for moder CPUs, when dealing with so large integers.

One of the solution to this problem is to parallelize. GPGPU[7] (for General-Purpose computing on the Graphics Processing Unit) enables the use of GPU for parallel computation other than graphics. GPUs are designed to perform computations in parallel. Since every PC is equipped with some kind of GPU, one can easily exploit its capabilities. NVIDIA has made it even more accessible by creating CUDA (Compute Unified Device Architecture)[5]. It is a parallel computing platform and API, which exposes GPUs true potential.

1.1 Environment Specification

This project was developed in Microsoft Visual Studio 2015[3] with NSight plugin and CUDA API.[5] Source code is written mainly in C++ and inline assembly - PTX.[6] Table 1.1 more precisely illustrates GPU specification on which the program and all tests are run.

Table 1.1: Device specification

Device name:	GeForce GTX 960
CUDA Driver Version:	9.0
CUDA Runtime Version:	8.0
CUDA Capability version number:	5.2
Total amount of global memory:	4096 MBytes (4294967296 bytes)
GPU Max Clock rate:	1304 MHz (1.30 GHz)
Total amount of shared memory per block:	49152 bytes
Warp size:	32

Table below presents full platform and system information.

Table 1.2: PC specification

Operating System:	Windows 10 Education 64-bit
Motherboard:	Gigabyte Technology Co., Ltd. P55A-UD4
CPU:	Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz (4 CPUs), 2.7GHz
RAM Memory: :	4096MB

Chapter 2

Theory

This chapter basically presents minimal amount of theory required to fully understand the concepts, problems and solutions which were applied and implemented within this project.

2.1 RSA

The RSA algorithm was created by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1970s. The security of the algorithm is based on the problem of integer factorization.

- Key generation
  - Choose two large and distinct prime numbers p and q.
  - Compute the modulus n
    $$n = pq$$
  - Compute
    $$\lambda(n) = lcm(\lambda(p), \lambda(q)) = lcm(p-1, q-1)$$
    where λ is the Carmichael Totient Function[1]. This value must be kept in private.
  - Choose an integer e such that:
    $$1 < e < \lambda(n)$$
    and
    $$gcd(e, \lambda(n)) = 1$$
    This means e and $\lambda(n)$ are coprime.[8]
  - Choose value d for decryption and solve for d:
    $$de \equiv 1(mod\lambda(n))$$
    e is used for encryption. Usually this is done the other way around. e is chosen first so it has short bit length and small Hamming weight,[2] which provides for much faster encryption. In most cases $e =$ 3,5 or 7. High security is provided if larger number is used. Most common is Fermat's four: $e = 2^{16} + 1 = 65537 =$ 0x10001

  - Public key is then:
$$(e, n)$$
and the private key:
$$(d, n)$$

- Encryption

  For message m the ciphertext c is in relation:
$$c \equiv m^e \ \mathrm{mod} n$$

- Decryption
$$m \equiv c^d \ \mathrm{mod} n \equiv (m^e)^d \ \mathrm{mod} n \equiv m \ \mathrm{mod} n$$

2.2 Side channel attacks

2.2.1 Timing attacks

2.3 Parallel programming on CUDA

Chapter 3

Implementation

The code was written in C++ language. In order to minimize data movement between host and the device, most of logics and computation are executed fully on GPU. Implementation of RSA encryption requires only one function - modular exponentiation of a multi precision integer. Implemented Big Integer class provides much more functions, to properly handle data and validate results. Code listing below shows the header of class BigInteger.

```
class BigInteger
{
//fields
public:

// 4096 bits
static const int ARRAY_SIZE = 128;

private:
// Magnitude array in little endian order.
// Most-significant int is mag[length-1].
// Least-significant int is mag[0].
// Allocated on the device.
unsigned int* deviceMagnitude;

// same array allocated on the host
// provides faster access if nothing was changed
unsigned int* hostMagnitude;

// flag indicating if hostMagnitude matches deviceMagnitude
bool upToDate;

// Device wrapper instance diffrent for every integer
// to provide parallel execution
DeviceWrapper* deviceWrapper;

// methods
public:
BigInteger();
BigInteger(const BigInteger& x);
BigInteger(unsigned int value);
~BigInteger();
const unsigned int& operator[](int index);

// factory
static BigInteger* fromHexString(const char* string);
static BigInteger* createRandom(int bitLength);

// setters, getters
void set(const BigInteger& x);
unsigned int* getDeviceMagnitude(void) const;

// arithmetics
void add(const BigInteger& x);
void subtract(const BigInteger& x);
void multiply(const BigInteger& x);
void square(void);
void mod(const BigInteger& modulus);
void multiplyMod(const BigInteger& x, const BigInteger& modulus);
void squareMod(const BigInteger& modulus);
void powerMod(BigInteger& exponent, const BigInteger& modulus);

// logics
void shiftLeft(int bits);
void shiftRight(int bits);

// extras
bool equals(const BigInteger& value) const;
int compare(const BigInteger& value) const;
int getBitwiseLengthDiffrence(const BigInteger& value) const;
int getBitwiseLength(void) const;
int getLSB(void) const;
bool testBit(int bit);
void synchronize(void);
char* toHexString(void);
void print(const char* title);

//timer
void startTimer(void);
unsigned long long stopTimer(void);

// async calls
// must call synchronize to read from
void modAsync(const BigInteger& modulus);
void multiplyModAsync(const BigInteger& x, const BigInteger& modulus);
void squareModAsync(const BigInteger& modulus);

private:

void setMagnitude(const unsigned int* magnitude);
void clear(void);
void updateDeviceMagnitiude(void);
void updateHostMagnitiude(void);
static unsigned int random32(void);

/*
Parses hex string to unsigned int type.
Accepts both upper and lower case, no "0x" at the beginning.
E.g.: 314Da43F
*/
static unsigned int parseUnsignedInt(const char* hexString);
};
```

Listing 3.1: BigInteger.h

Big Integer class handles mathematical logic, validates input / output, and provides comfortable and readable interface.

The intermediary between Big Integer and GPU is another class - Device Wrapper. It handles GPU kernel launches, synchronization and data movement. Class definition is listed below.

```
class DeviceWrapper
{

private:

// main stream for kernel launches
cudaStream_t mainStream;

// lauch config
dim3 block_1, block_2, block_4;
dim3 thread_warp, thread_2_warp, thread_4_warp;

// 4 ints to help store results
int* deviceWords;

// auxiliary arrays
unsigned int* device4arrays;
unsigned int* device128arrays;
unsigned int* deviceArray;

unsigned long long* deviceStartTime;
unsigned long long* deviceStopTime;

public:

DeviceWrapper();
~DeviceWrapper();

// sync
unsigned int* init(int size) const;
unsigned int* init(int size, const unsigned int* initial) const;
void updateDevice(unsigned int* device_array, const unsigned int* host_array, int size) const;
void updateHost(unsigned int* host_array, const unsigned int* device_array, int size) const;
void free(unsigned int* device_x) const;

// extras
void clearParallel(unsigned int* device_x) const;
void cloneParallel(unsigned int* device_x, const unsigned int* device_y) const;
int compareParallel(const unsigned int* device_x, const unsigned int* device_y) const;
bool equalsParallel(const unsigned int* device_x, const unsigned int* device_y) const;
int getLSB(const unsigned int* device_x) const;
int getBitLength(const unsigned int* device_x) const;
void synchronize(void);

// measure time
void startClock(void);
unsigned long long stopClock(void);

// logics
void shiftLeftParallel(unsigned int* device_x, int bits) const;
void shiftRightParallel(unsigned int* device_x, int bits) const;

// arithmetics
void addParallel(unsigned int* device_x, const unsigned int* device_y)
    const;
void subtractParallel(unsigned int* device_x, const unsigned int*
    device_y) const;
void multiplyParallel(unsigned int* device_x, const unsigned int*
    device_y) const;
void squareParallel(unsigned int* device_x) const;
void squareParallelAsync(unsigned int* device_x) const;
void modParallel(unsigned int* device_x, unsigned int* device_m) const;
void modParallelAsync(unsigned int* device_x, unsigned int* device_m)
    const;
void multiplyModParallel(unsigned int* device_x, const unsigned int*
    device_y, const unsigned int* device_m) const;
void multiplyModParallelAsync(unsigned int* device_x, const unsigned int
    * device_y, const unsigned int* device_m) const;
void squareModParallel(unsigned int* device_x, const unsigned int*
    device_m) const;
void squareModParallelAsync(unsigned int* device_x, const unsigned int*
    device_m) const;

private:
void inline addParallelWithOverflow(unsigned int* device_x, const
    unsigned int* device_y, int blocks) const;
};
```

Listing 3.2: DeviceWrapper.h

Project also contains Test class to validate computations, measure times and simulate encryption. RSA class contains single "encrypt" function, which encrypts provided value. Full class diagram is presented on figure 3.1.

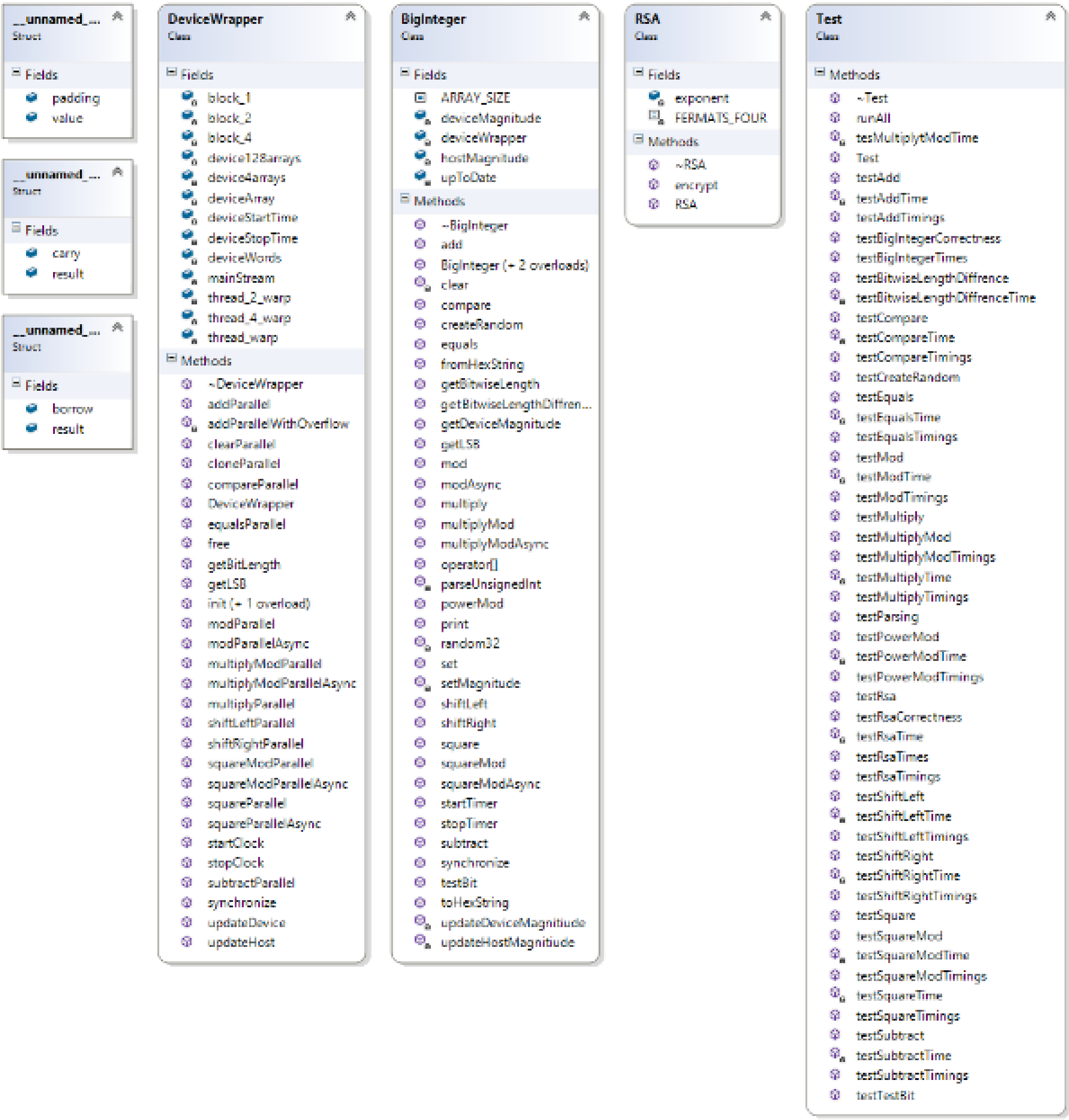

Figure 3.1: Class Diagram

3.1 Parallel equals

3.2 Parallel compare

3.3 Parallel bit length

3.4 Parallel left shift

3.5 Parallel right shift

3.6 Parallel add

3.7 Parallel subtract

3.8 Parallel multiply

3.9 Parallel modulo reduction

3.10 Parallel multiply modulo

3.11 Parallel power modulo

Bibliography

[1] Y. Ge. A note on the carmichael function. http://yimin-ge.com/doc/carmichael.pdf. Available: 2017-08-28.

[2] E. O. Iskra Nunez. Generalized hamming weights for linear codes. http://www.uprh.edu/~simu/Reports2001/NOU.pdf, 2001. Available: 2017-08-28.

[3] Microsoft. Visual stdio. https://www.visualstudio.com. Available: 2017-08-28.

[4] E. Milanov. The rsa algorithm. https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf, June 2009. Available: 2017-08-27.

[5] NVIDIA. Cuda zone. https://developer.nvidia.com/cuda-zone. Available: 2017-08-27.

[6] NVIDIA. Using inline ptx assembly in cuda. https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/ptx_isa_2.2.pdf. Available: 2017-08-28.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[8] W. Stein. Elementary number theory: Primes, congruences, and secrets. http://wstein.org/ent/ent.pdf, 2017. Available: 2017-08-28.

List of Figures

List of Tables