EECS578 – Prof. Bertacco
Fall 2015

EECS 578 – RSA mini-project
Assigned: 11/04/15 – Due: 11/17/15

1. Overview

This mini-project focuses on the RSA (Rivest-Shamir-Adleman) cryptographic algorithm. The RSA algorithm is asymmetric, which means that there are two different keys: public and private keys. A public key is used when encrypting messages, and it can be distributed to everyone. A private key is used when decrypting messages, but it is never distributed to anyone. A pair of public and private keys are generated using two large prime numbers. While it is theoretically possible to derive a private key from a public key, this process, called brute-forcing, can take tremendous time when the prime numbers are sufficiently large: for instance in 2010 a team of research managed to brute-force a 768-bit RSA private key after 1,500 computer-years of effort. Nowadays, most RSA implementations use 1,024 bits or more. Thus, attackers often try to guess a private key by observing the running time during decryption. These attacks are called **timing attacks**.

The key computation of the RSA decryption is a modular exponentiation. Here is an equation that computes the plaintext (M) from a ciphertext (C).

$$M = C^d \bmod n$$

where n is the product of two prime numbers, and d is a secret key.

In this mini-project, you need to evaluate a few implementations of modular exponentiation to determine if they are secure against timing attacks. We start with the simplest software implementation, and move to a sophisticated hardware implementation.

If you would like to learn the RSA algorithm in detail, check:
- https://simple.wikipedia.org/wiki/RSA_(algorithm)
- Chapter 9 "Public-key Cryptography and RSA" of the book "Cryptography and Network Security" by William Stallings.

2. Setting up for the mini-project

In this mini-project, you need to compile and run (1) C++ code and (2) Verilog code. For the C++ code, you can use any machine you want, including your laptop. For the Verilog code, however, you need to run VCS in either a CAEN machine or the host `oncampus-course.engin.umich.edu`. Note that this host is accessible only on campus.

To get started with this mini-project, download the file `mod_exp.tar.gz` from the class website and copy it to a local directory in your *engin* account. Unzip the file; it will create a directory `mod_exp`.

3. Part 1: Software implementations

If you go to a subdirectory `sw` in the directory `mod_exp`, you will find the file `mod_exp.cpp` that contains 4 different implementations of modular exponentiation: (1) a straightforward exponentiation (Section 3.1), (2) an exponentiation by squaring (square-and-multiply) (Section 3.2), (3) an exponentiation by Montgomery modular multiplication, paired with square-and-multiply (Section 3.3), and (4) another exponentiation by Montgomery modular multiplication, paired with Montgomery ladder (Section 4.3). You do not need to modify this file to complete the mini-project.

To compile the source code, go to the directory `mod_exp` and

```
make sw_impl (or make)
```

The above command will create the binary file `mod_exp` in the same directory. To run this binary,

```
./mod_exp [input file] [method index] [repeat count]
```

`[input file]` is a text file that contains a list of test vectors. Each test vector includes a base, an exponent and a modulus, separated by commas, and forms an exponentiation equation in the following: $base^{exponent} \bmod modulus$. These numbers are unsigned 32-bit, and must be less than 2^{32}. Please see examples in the file `input/sample.txt`. `[method index]` is an integer indicating which method will be used for exponentiation. It can be one of the four methods in the following table. `[repeat count]` is a positive integer indicating how many times the exponentiation computation should be repeated.

[method index]	exponentiation method
0	straightforward exponentiation
1	exponentiation by squaring (square-and-multiply)
2	exponentiation by Montgomery multiplication with square-and-multiply
3	exponentiation by Montgomery multiplication with Montgomery ladder

For instance, if you want to measure the total execution time of 10,000 executions of the straightforward exponentiation for test vectors in the file `input/sample.txt`,

```
./mod_exp input/sample.txt 0 10000
```

3.1. Straightforward method

The most straightforward calculation is to multiply the base number repeatedly and then perform a modulo operation afterwards. However, this calculation possibly generates values that are too large, due to the repeated multiplications. Instead, we apply the modulo operation after every multiplication.

$$\overbrace{(base \times base \times \cdots \times base}^{\text{exponent times}}) \bmod modulus$$
$$= (((base \times base \bmod modulus) \times base \bmod modulus) \times \cdots \times base \bmod modulus)$$

The function `straightforward()` in the source code `mod_exp.cpp` performs an exponentiation following the above equation.

Question 1 (8pts). What is the total elapsed time for 10,000 executions of each test vector in the file `input/sample.txt` when using this straightforward method? Use the command: "`make go0`".

Question 2 (12pts). In the file `input/question2.txt`, create a set of 10 test vectors that can help reveal a secret exponent in this implementation. Briefly explain how you can guess the secret exponent with your test vectors. Report the total elapsed time of 10,000 executions for each of your test vectors. Use the command: "`make q2`".

3.2. Square-and-multiply algorithm

The straightforward implementation in the previous subsection requires many multiplications depending on the exponent. The square-and-multiply algorithm, on the contrary, removes such repeated multiplications. Instead, this algorithm takes an iterative approach where each iteration corresponds to a bit of the exponent. The algorithm starts from the most significant bit, and moves to the least significant bit. At every iteration, it calculates a partial product by squaring the partial product obtained from the previous iteration, and then multiplies the partial product by the base number only if the current bit is 1. Below is a pseudocode of this algorithm.

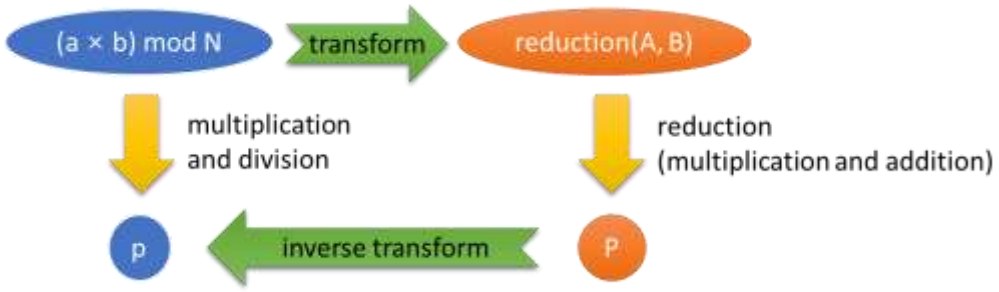
```
function square_and_multiply(b, e)  // b: base, e: exponent (e_{t-1} e_{t-2} … e_0)_2
   p ← 1
   for j = t – 1 downto 0 do
      p ← p×p
      if e_j = 1 then
         p ← p×b
   return p    // p = b^e
```

Question 3 (8pts). What is the total elapsed time for 10,000 executions of each test vector in the file `input/sample.txt` when using this square-and-multiply algorithm? Use the command: "`make go1`".

Question 4 (12pts). In the file `input/question4.txt`, create a set of 10 test vectors that can help reveal a secret exponent in this implementation. Briefly explain how you can guess the secret exponent with your test vectors. Report the total elapsed time of 10,000 executions for each of your test vectors. Use the command: "`make q4`".

3.3. Montgomery modular multiplication with square-and-multiply algorithm

Montgomery modular multiplication is a fast modular-multiplication method that is widely used in cryptography. Its speed-up comes from the fact that it can replace the given modulus (*e.g.*, 0x12345678) with another one (*e.g.*, 0x10000000) whose division and modular operation can be substituted with cheaper operations such as addition and multiplication. The below figure summarizes how it works, followed by brief descriptions for each step.



Step 1: Compute R^{-1}, and N'. (This step is not shown in the figure.)
To begin with, Montgomery modular multiplication needs a couple of extra values, which can be calculated from the original and new moduli (N and R). R must be chosen carefully so that these moduli are coprime. Using these moduli, we calculate the modular multiplicative inverse of R, called R^{-1} where $R \times R^{-1} \equiv 1 \pmod N$, using the extended Euclidean algorithm. Then, we calculate N' that satisfies $R \times R^{-1} - N \times N' = 1$.

Step 2: Convert multiplication operands into Montgomery form.
We convert two operands in multiplication, a and b, into their Montgomery form, A and B. This transformation is done by the following equation.

$$A = (a \times R) \bmod N,\ B = (b \times R) \bmod N$$

Step 3: Multiply the two transformed operands. $T = A \times B$. (This step is not shown in the figure.)
For your information, note that T is not equal to the product a × b in Montgomery form, because $T = A \times B = ((a \times b) \times R) \times R \bmod N \neq (a \times b) \times R \bmod N$. Thus, we need to multiply T by R^{-1} to remove the additional R in the equation, and we also need to perform a modulo operation to keep the product less than N. Fortunately, Montgomery reduction in the next step can reduce the product T into the Montgomery form without the modulo operation.

Step 4: Perform Montgomery reduction.
This step calculates the product of two operands in Montgomery form. It takes T from the previous step, and performs operations in the following pseudocode. Note that the two modulo operations and one division can be substituted with shift and logical operations.

```
function REDC(T)
    m ← (T mod R)×N' mod R
    P ← (T + m×N) / R
    if P ≥ N then
        return P – N
    else
        return P
```

Step 5: Convert the product back to the original form.

$$p = (P \times R^{-1}) \bmod N$$

Note that the product p must be equal to $(a \times b) \bmod N$.

Montgomery multiplication example. In this example, we perform a multiplication of two 32-bit integer operands: a=305,419,896 and b=2,271,560,481, with a 32-bit modulus N=4,292,870,399=65,519×65,521. Because a modulus can be as large as 2^{32}–1, R must be larger than 2^{32}–1. Let's pick R=0x100000000=2^{32} in this example. Note that R and N are coprime.
Step 1: R^{-1}=3,234,391,457 and N'=3,235,971,329. (Using the extended Euclidean algorithm, you can find R^{-1} where $R \times R^{-1} \equiv 1 \pmod{N}$, and you can also find N' where $R \times R^{-1} - N \times N' = 1$.)
Step 2: A=2,193,187,897 and B=1,027,920,224. (A=a×R mod N, and B=b×R mod N.)
Step 3: T=2,254,422,194,358,328,928. (T=A×B.)
Step 4: P=3,794,497,757 (P=REDC(T).)
Step 5: p=4,290,335,667 (p=P×R^{-1} mod N=a×b mod N.)

In this mini-project, we pick R=0x100000000=2^{32}, and keep it fixed for all the problems below. Thus, be aware that you need to carefully choose N, so that (1) R and N do not share any common divisor except 1, and (2) N is less than R.

Note: It is not required for you to understand details of the Montgomery modular multiplication in this mini-project. If you want to know more, please check:
- The original paper: "Modular Multiplication without Trial Division" by P. Montgomery (http://www.jstor.org/stable/2007970).
- https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

Exponentiation using Montgomery modular multiplication. You can use the square-and-multiply in Section 3.2 to perform the exponentiation in Montgomery form. To this end, you need to (1) convert a multiplier and a multiplicand into their Montgomery form, (2) perform Montgomery reduction, and (3) convert the result back to the original form, as shown in the following pseudocode.

```
function Montgomery_square_and_multiply(b, e)
   B ← Montgomery(b)   // b×R mod N (see Step 2)
   P ← Montgomery(1)   // 1×R mod N (see Step 2)
   for j = t – 1 downto 0 do
      P ← REDC(P×P)
      if e_j = 1 then
         P ← REDC(P×B)
   p ← Montgomery^-1(P)   // P×R^-1 mod N (see Step 5)
   return p    // p = b^e
```

Question 5 (8pts). What is the total elapsed time for 10,000 executions of each test vector in the file `input/sample.txt` when using the Montgomery algorithm? Use the command: "`make go2`".

Question 6 (12pts). In the file `input/question4.txt`, create a set of 10 test vectors that can help reveal a secret exponent in this implementation. Briefly explain how you can guess the secret exponent with your test vectors. Report the total elapsed time of 10,000 executions for each of your test vectors. Use the command: "`make q6`".

4. Part 2: Hardware implementations

In this part of the mini-project, you need to implement hardware modular-exponentiation modules. The hardware implementations should be secure against timing attacks. We provide (1) a Verilog source code for Montgomery multiplication (the steps 3 and 4 in Section 3.3), (2) a Verilog source code for unsecure Montgomery exponentiation using the square-and-multiply algorithm, and (3) Verilog testbenches for this source code. You can find the source code in the subdirectory `rtl`, and the testbenches in the subdirectory `testbench`.

For Sections 4.1 and 4.2, you can compile the source code and the testbench using the following command.

`make simv_square` (or `make simv`)

The above command generates a binary file `simv`. To run this binary,

`make run_square` (or `make run`)

In the testbench, the input file `input/sample.txt` is read line-by-line, and test vectors are fed into the exponentiation module one at a time, reporting the exponentiation result and the elapsed number of clock cycles in the terminal. Note that the testbench also performs an additional computation to create inputs suitable for the exponentiation module. For instance, it generates A, B, R^{-1} and N'. The exponentiation module performs computation only in the Montgomery domain.

4.1. Unsecure square-and-multiply implementation

The first hardware implementation is a hardware counterpart of Section 3.3 (Montgomery modular multiplication with the square-and-multiply algorithm). There are two Verilog files that will be used in this section: `montgomery_mult.v` and `montgomery_exp_square.v`. They can be found in the subdirectory `rtl`.

The file `montgomery_mult.v` defines a combinational-logic module that performs Montgomery modular multiplication. This module takes the following inputs: two integers (`A` and `B`) in Montgomery form, a modulus (`N`), and an N' (`N_prime`). It also uses a bit width defined in the file `defines.vh`, which is used to calculate $R = 2^{\text{bit width}}$. Again, in this assignment, R is set to 2^{32}. This module has one output: the multiplication result (`P`). You do not need to modify this file to complete the mini-project.

The file `montgomery_exp_square.v` defines a sequential-logic module that performs exponentiation using the square-and-multiply algorithm explained in Section 3.2. This module takes the following input signals:

- `base_mont`: a base converted in Montgomery form.
- `exponent`: an exponent.
- `N` and `N_prime`: a modulus and its corresponding N'.
- `one_mont`: the number 1 converted in Montgomery form.
- `start`: a start signal.

There are two output signals in this module: a finish signal (`finish`) and the exponentiation result (`exp_result`).

To check if this implementation is secure to timing attacks, you need to analyze the source code and measure the execution time for several test vectors in the file `input/sample.txt`. We provide a testbench `test_montgomery_square.v` that reads this file line-by-line, and performs exponentiation for each test vector, showing the elapsed time in the terminal. You do not need to modify the testbench to complete the mini-project.

Question 7 (8pts). How many clock cycles does this implementation take for each test vector in the file `input/sample.txt`?

Question 8 (8pts). Draw a state-transition diagram for the square-and-multiply exponentiation in the source code `montgomery_exp_square.v`. Note that there are four states: IDLE, SQUARE, MULT and FINISH.

4.2. Securing the implementation of the square-and-multiply exponentiation

From Question 7, you may have realized a way to guess the secret key (*i.e.*, the exponent in the exponentiation) by measuring the elapsed time. In this section, you need to modify the previous implementation so that its execution time does not vary depending on the secret key.

You should modify the file `montgomery_exp_square.v`. Please use the same file name in this section. (You might want to make a copy of the original file for your reference.)

Question 9 (8pts). Provide a high-level explanation of how to secure the original implementation against timing attacks. Include your new state-transition diagram if you have changed it. (In this question, do not use the Montgomery ladder, which will be explained in Section 4.3.)

Question 10 (8pts). Implement your secure module in Verilog. What is the execution time for each test vector in the file `input/sample.txt`? Your Verilog file should have the same name (`montgomery_exp_square.v`). Use only one Montgomery multiplier (*i.e.*, the module `montgomery_mult`).

4.3. Implementing a Montgomery ladder

In this section, you need to implement another hardware module that is also known to be secure against timing attacks, called Montgomery ladder. The Montgomery ladder is a method calculating exponentiation, which is similar to the square-and-multiplication algorithm. The difference is that the ladder always performs two Montgomery multiplications for every bit location. Also, its operands are more balanced than the square-and-multiplication algorithm. Here is a pseudocode. For your reference, you can also find a software implementation in the file `mod_exp.cpp`.

```
function Montgomery_ladder(b, e)
    B ← Montgomery(b)
    P0 ← Montgomery(1)
    P1 ← B
    for j = t – 1 downto 0 do
        if ej = 0 then
            P1 ← REDC(P0×P1)
            P0 ← REDC(P0×P0)
        else
            P0 ← REDC(P0×P1)
            P1 ← REDC(P1×P1)
    p ← Montgomery-1(P0)
    return p    // p = b^e
```

If you want to learn more about the Montgomery ladder in the context of RSA, please check: "The Montgomery Powering Ladder" by M. Joye and S.-M. Yen (http://link.springer.com/chapter/10.1007%2F3-540-36400-5_22).

We provide a skeleton Verilog file `montgomery_exp_ladder.v` that only contains definitions for inputs and outputs. You should modify this file to complete your implementation. We also provide a testbench `test_montgomery_ladder.v` that is similar to the previous testbench discussed in Section 4.1.

You can compile the source code and the testbench using the following command.

```
make simv_ladder
```

The above command generates a binary file `simv`. To run this binary,

```
make run_ladder
```

Question 11 (8pts). Implement your Montgomery ladder module in Verilog. What is the execution time for each test vector in the file `input/sample.txt`? Your Verilog file should have the name: `montgomery_exp_ladder.v`. Use only one Montgomery multiplier (*i.e.*, the module `montgomery_mult`). You do not need to show a state diagram for this implementation.

5. How to submit

You must use the answer sheet in the link:
http://eecs.umich.edu/courses/eecs578/eecs578.f15/miniprojects/RSAproject/RSA_answer.doc

To submit the project, follow the instructions on the submission webpage:
http://eecs.umich.edu/courses/eecs578/submit

You need to submit:

(1) The answer sheet (`RSA_answer.doc`)
(2) The test vector file for Question 2 (`question2.txt`)
(3) The test vector file for Question 4 (`question4.txt`)
(4) The test vector file for Question 6 (`question6.txt`)
(5) The modified exponentiation module in Section 4.2 (`montgomery_exp_square.v`)
(6) The Montgomery ladder module in Section 4.3 (`montgomery_exp_ladder.v`)

Please create a single archive file that includes all the deliverables, and upload it to the submission webpage. We only accept the following file extensions: tar, gz, tgz, bz2 and zip. The file name will be automatically changed to your uniqname by the submission webpage.