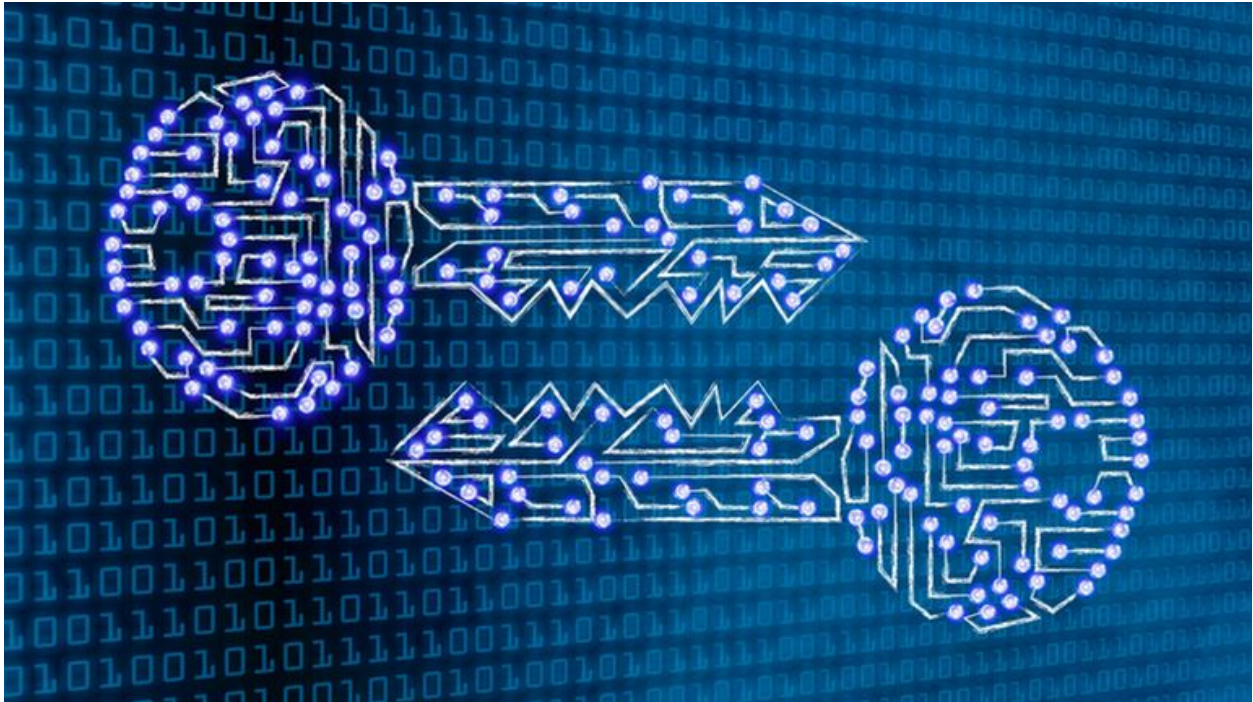


Hardware Security

Side-channel attack

Assignment #2



Introduction

Modification of the RSA implementation to use Montgomery modular multiplication and measuring the performance speed up with this modification . and implementing a side channel attack to this implementation.

Source Code : <https://github.com/khadijaAssem/Montgomery-modular-multiplication>

Part 1: Montgomery modular multiplication

Observation

Using Montgomery multiplication improved the time of execution since the normal multiplication implementation takes $O(n^2)$

Sample Runs

Time taken by unmodified RSA implementation 111

Time taken by montgomery modified RSA implementation 93

```
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
Original message = 4534e60556ac43e115c46f1485cbae63f02ab0b6680df2fe10af1b86b431cc139436036a051beb708bb9aab29d7f8220f3860911dff3edb0c259d4cdfed15b4eec6e4ed2b441
Ciphertext = 5db993695b5d54115fc4e1a844ebdcfef50b47fa4afd6e42a8e3c4913a25d348c953db3bc804d9532139ab31c2190cbeb9870b20997c6d53646e013881a83cc15376f5bafda838bbb
Decrypted message = 4534e60556ac43e115c46f1485cbae63f02ab0b6680df2fe10af1b86b431cc139436036a051beb708bb9aab29d7f8220f3860911dff3edb0c259d4cdfed15b4eec6e4ed2b44
Time taken by modular multiplication 111

Original message = 4534e60556ac43e115c46f1485cbae63f02ab0b6680df2fe10af1b86b431cc139436036a051beb708bb9aab29d7f8220f3860911dff3edb0c259d4cdfed15b4eec6e4ed2b441
Ciphertext = 5db993695b5d54115fc4e1a844ebdcfef50b47fa4afd6e42a8e3c4913a25d348c953db3bc804d9532139ab31c2190cbeb9870b20997c6d53646e013881a83cc15376f5bafda838bbb
Decrypted message = 4534e60556ac43e115c46f1485cbae63f02ab0b6680df2fe10af1b86b431cc139436036a051beb708bb9aab29d7f8220f3860911dff3edb0c259d4cdfed15b4eec6e4ed2b44
Time taken by montgomery multiplication 93
```

Time taken by unmodified RSA implementation 101

Time taken by montgomery modified RSA implementation 98

```
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
Original message = 3222ed37d8cbcd5683a1918f0f36c6dae5d21aef4706cd4924d11a9a9215962bc071f730bc4d62a1a9cada396445b9b0fa212f89e0170f8c68b5cde38b19856f4fe3549e2a90
Ciphertext = 6fc2338dbeeb1598b014af91fddeca3b21fbd79e82a89e70f093df4d464a4e433b55fa42072aac9b2ea627a1000fc5ea3572e9bec889115cd72519a4dedb35a26f0564eab87a1d43da
Decrypted message = 3222ed37d8cbcd5683a1918f0f36c6dae5d21aef4706cd4924d11a9a9215962bc071f730bc4d62a1a9cada396445b9b0fa212f89e0170f8c68b5cde38b19856f4fe3549e2a9
Time taken by modular multiplication 101

Original message = 3222ed37d8cbcd5683a1918f0f36c6dae5d21aef4706cd4924d11a9a9215962bc071f730bc4d62a1a9cada396445b9b0fa212f89e0170f8c68b5cde38b19856f4fe3549e2a90
Ciphertext = 6fc2338dbeeb1598b014af91fddeca3b21fbd79e82a89e70f093df4d464a4e433b55fa42072aac9b2ea627a1000fc5ea3572e9bec889115cd72519a4dedb35a26f0564eab87a1d43da
Decrypted message = 3222ed37d8cbcd5683a1918f0f36c6dae5d21aef4706cd4924d11a9a9215962bc071f730bc4d62a1a9cada396445b9b0fa212f89e0170f8c68b5cde38b19856f4fe3549e2a9
Time taken by montgomery multiplication 98
```

Time taken by unmodified RSA implementation 109

Time taken by montgomery modified RSA implementation 105

```
PlainRSADemo X
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
Original message = 287fb9733308ddf9d37a73c9476dafb2f5693c124c75dfed5bad19f6d9a0fa4587a22b4a2cb81ed053388aa388f9062d56d72291a8c0afe83ab6cc51cca94c22f0dd468b338a
Ciphertext = 65ce080c63bbaac25614e8f93009f82158bec2e89ad3df3564ef9d8035f68a2c49edf633d4af5ac5d4db872e7e749e7ee9698048a1699f621d34184e80a187c69aab905779c131c25e
Decrypted message = 287fb9733308ddf9d37a73c9476dafb2f5693c124c75dfed5bad19f6d9a0fa4587a22b4a2cb81ed053388aa388f9062d56d72291a8c0afe83ab6cc51cca94c22f0dd468b338
Time taken by modular multiplication 109

Original message = 287fb9733308ddf9d37a73c9476dafb2f5693c124c75dfed5bad19f6d9a0fa4587a22b4a2cb81ed053388aa388f9062d56d72291a8c0afe83ab6cc51cca94c22f0dd468b338a
Ciphertext = 65ce080c63bbaac25614e8f93009f82158bec2e89ad3df3564ef9d8035f68a2c49edf633d4af5ac5d4db872e7e749e7ee9698048a1699f621d34184e80a187c69aab905779c131c25e
Decrypted message = 287fb9733308ddf9d37a73c9476dafb2f5693c124c75dfed5bad19f6d9a0fa4587a22b4a2cb81ed053388aa388f9062d56d72291a8c0afe83ab6cc51cca94c22f0dd468b338
Time taken by montgomery multiplication 105
```

Time taken by unmodified RSA implementation 114

Time taken by montgomery modified RSA implementation 93

```
Original message = 57aa4a7b2fffc19b7aa446fe08c0702116a04987df7c8a74c92413c4b853b78b189ab8e0902f0e
Ciphertext = 802dff4150f0aa5959c5831d720f61715b39dbe87855cdef31359ce3a8ba70c82d277d8b503f31cc3ff6c
Decrypted message = 57aa4a7b2fffc19b7aa446fe08c0702116a04987df7c8a74c92413c4b853b78b189ab8e0902f0
Time taken by modular multiplication 114

Original message = 57aa4a7b2fffc19b7aa446fe08c0702116a04987df7c8a74c92413c4b853b78b189ab8e0902f0e
Ciphertext = 802dff4150f0aa5959c5831d720f61715b39dbe87855cdef31359ce3a8ba70c82d277d8b503f31cc3ff6c
Decrypted message = 57aa4a7b2fffc19b7aa446fe08c0702116a04987df7c8a74c92413c4b853b78b189ab8e0902f0
Time taken by montgomery multiplication 93
```

Time taken by unmodified RSA implementation 106

Time taken by montgomery modified RSA implementation 93

```
PlaintextDemo
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
Original message = 2a08295b377d68dd2549a1af5ca251393c5dfb9e64029c5095b76582513af5f7b7f76dfab0ffbad6bb4a280fc6feddbf3e0f6e6f51be080fcfb
Ciphertext = 969a3c248938467da9927d50684210c51e4eeclae74ba8b0bf868316c338e1c631afc35d48849657d419b8fcf85b3ce5306e1a9ae14004a2807467091b
Decrypted message = 2a08295b377d68dd2549a1af5ca251393c5dfb9e64029c5095b76582513af5f7b7f76dfab0ffbad6bb4a280fc6feddbf3e0f6e6f51be080fcfb
Time taken by modular multiplication 106

Original message = 2a08295b377d68dd2549a1af5ca251393c5dfb9e64029c5095b76582513af5f7b7f76dfab0ffbad6bb4a280fc6feddbf3e0f6e6f51be080fcfb
Ciphertext = 969a3c248938467da9927d50684210c51e4eeclae74ba8b0bf868316c338e1c631afc35d48849657d419b8fcf85b3ce5306e1a9ae14004a2807467091b
Decrypted message = 2a08295b377d68dd2549a1af5ca251393c5dfb9e64029c5095b76582513af5f7b7f76dfab0ffbad6bb4a280fc6feddbf3e0f6e6f51be080fcfb
Time taken by montgomery multiplication 93

Process finished with exit code 0
```

Part 2: Side-channel timing attack

Observation

This attack gives satisfying results as it succeeds to recognize the 2nd most significant bit many times with a good confidence ratio.

1st I tried implement the attack taking the advantage of the fact that the first MSB is one :

```
/* simulating bit two = 0 */
if (!One) {
    result = mont.multiply(result, result); // step 1 (2bd MSB)
    start = System.nanoTime();
    result = mont.multiply(result, result); // step 2 (3rd MSB)
    end = System.nanoTime();
    time = end-start;
    bool0 = mont.getBool();
    System.out.println(bool0);
}

/* simulating bit two = 1 */
else {
    result = mont.multiply(result, result); // step 1 (2bd MSB)
    result = mont.multiply(result, aBar); // step 1 (2bd MSB)
    start = System.nanoTime();
    result = mont.multiply(result, result); // step 3 (3rd MSB)
    end = System.nanoTime();
    time = end-start;
    bool1 = mont.getBool();
    System.out.println(bool1);
}
```

By doing so I got the results where no test failed and the program succeeded to recognize the 2nd MSB successfully for 20 times:

```
bit is one !!
of 20 iterations 0 faild
```


2nd I did what was mentioned in the requirements and assumed that the private exponent used by the target implementation is only the three most significant bits. and then the five MSB and then 20, 50, 100

```

start = System.nanoTime();
/* simulating bit two = 0 */
if (!One) {
    result = mont.multiply(result, result); // step 1 (2bd MSB)
    result = mont.multiply(result, result); // step 2 (3rd MSB)
    bool0 = mont.getBool();
    System.out.println(bool0);
}

/* simulating bit two = 1 */
else {
    result = mont.multiply(result, result); // step 1 (2bd MSB)
    result = mont.multiply(result, aBar); // step 1 (2bd MSB)
    start = System.nanoTime();
    result = mont.multiply(result, result); // step 3 (3rd MSB)
    bool1 = mont.getBool();
    System.out.println(bool1);
}

int expBitlength = exponent.bitLength();
/* Starts from expBitlength - 3 since the 1st bit is known to be
   1 and 2nd bit is already assumed before */
for (int i = expBitlength - 3; i >= expBitlength - 100; i--) {
    result = mont.multiply(result, result);
    if (exponent.testBit(i)) {
        result = mont.multiply(result, aBar);
    }
}
end = System.nanoTime();
time = end - start;
return mont.fromMSpace(result);

```

And the output was as follows:

Where :

- **avgBitOneNRed** : is the average of times taken when the bit is assumed to be **one** and it **didn't require** montgomery additional reduction
- **avgBitOneRed** : is the average of times taken when the bit is assumed to be **one** and it **required** montgomery additional reduction
- **avgBitZeroNRed** : is the average of times taken when the bit is assumed to be **Zero** and it **didn't require** montgomery additional reduction
- **avgBitZeroRed** : is the average of times taken when the bit is assumed to be **Zero** and it **required** montgomery additional reduction

Using 3 bits

```
Of 20 iterations 1 faild
```

```
In iteration 19 :  
avgBitOneNRed 41928  
avgBitOneRed 62607  
avgBitZeroNRed 67230  
avgBitZeroRed 89028  
bit is zero !!
```

Using 5 bits

```
In iteration 20 :  
avgBitOneNRed 61617  
avgBitOneRed 80757  
avgBitZeroNRed 81232  
avgBitZeroRed 91925  
bit is one !!  
Of 20 iterations 1 faild
```

Using 10 bits

```
bit is one !!  
Of 20 iterations 5 faild
```

```
In iteration 16 :  
avgBitOneNRed 187712  
avgBitOneRed 201721  
avgBitZeroNRed 211155  
avgBitZeroRed 225216  
bit is zero !!
```

```
In iteration 17 :  
avgBitOneNRed 133498  
avgBitOneRed 143525  
avgBitZeroNRed 146744  
avgBitZeroRed 159198  
bit is zero !!
```

Using 20 bits

```
In iteration 20 :  
avgBitOneNRed 168283  
avgBitOneRed 174909  
avgBitZeroNRed 174152  
avgBitZeroRed 178463  
bit is one !!  
Of 20 iterations 5 faild
```

```
In iteration 2 :  
avgBitOneNRed 338988  
avgBitOneRed 351511  
avgBitZeroNRed 348223  
avgBitZeroRed 372834  
bit is zero !!
```

```
In iteration 3 :  
avgBitOneNRed 384394  
avgBitOneRed 391714  
avgBitZeroNRed 404976  
avgBitZeroRed 408027  
bit is one !!
```

```
In iteration 4 :  
avgBitOneNRed 352118  
avgBitOneRed 367412  
avgBitZeroNRed 367147  
avgBitZeroRed 383067  
bit is zero !!
```


Using 50 bits

```
Of 20 iterations 6 faild
```

```
In iteration 1 :  
avgBitOneNRed 603848  
avgBitOneRed 582648  
avgBitZeroNRed 607138  
avgBitZeroRed 614863  
bit is zero !!
```

```
In iteration 2 :  
avgBitOneNRed 568440  
avgBitOneRed 573246  
avgBitZeroNRed 582213  
avgBitZeroRed 591788  
bit is zero !!
```

```
In iteration 3 :  
avgBitOneNRed 561375  
avgBitOneRed 552376  
avgBitZeroNRed 572014  
avgBitZeroRed 567425  
bit is zero !!
```

Using 100 bits

```
Of 20 iterations 10 faild
```

```
In iteration 2 :  
avgBitOneNRed 1258011  
avgBitOneRed 1250115  
avgBitZeroNRed 1262817  
avgBitZeroRed 1283709  
bit is zero !!
```

```
In iteration 3 :  
avgBitOneNRed 1237389  
avgBitOneRed 1254014  
avgBitZeroNRed 1244790  
avgBitZeroRed 1270670  
bit is zero !!
```

```
In iteration 4 :  
avgBitOneNRed 1270836  
avgBitOneRed 1254932  
avgBitZeroNRed 1268572  
avgBitZeroRed 1286181  
bit is zero !!
```

Summary

Conclusion

In part 1 : Using Montgomery multiplication made it faster to encrypt and decrypt the message.

In part 2 : As we increase the number of private exponent used by the target implementation bits the attack precision decreases.