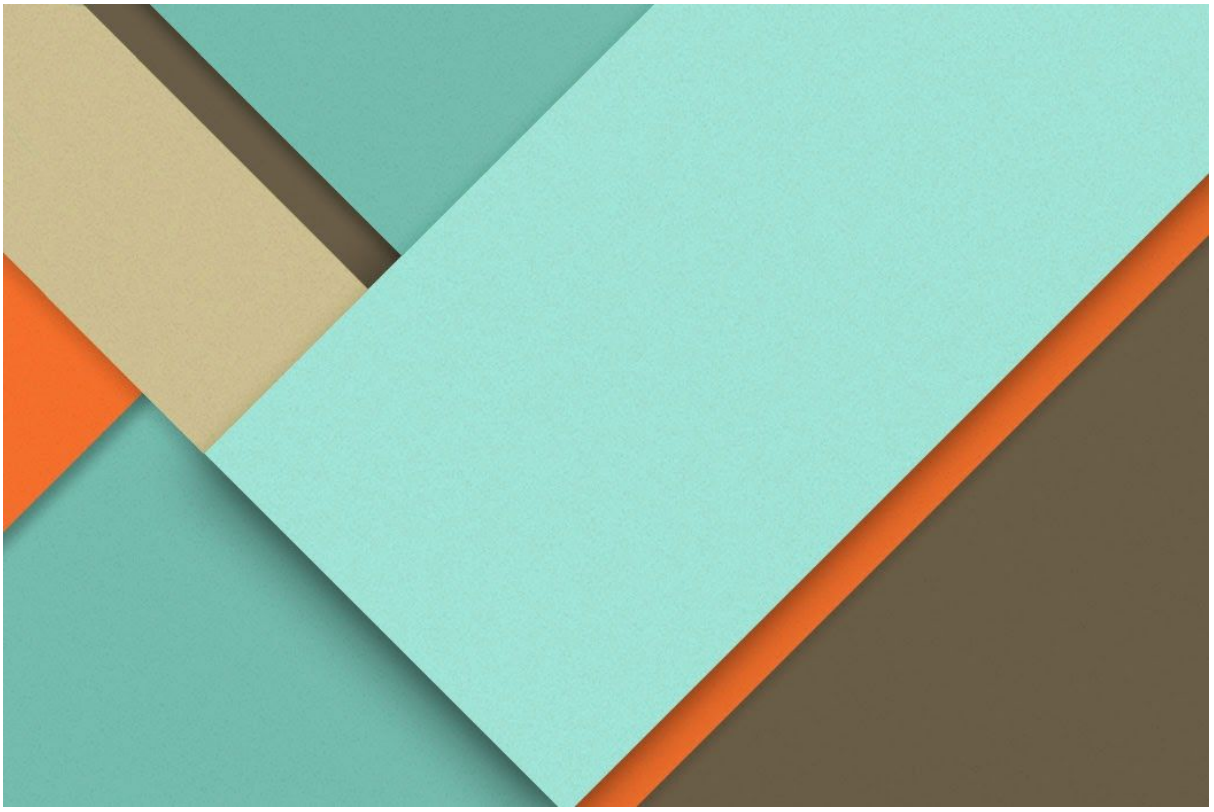


Assignment 2

Hardware Security

Mahmoud Mohamed Abdlaleem 59

Part I

Requirements:

to modify the given implementation to use Montgomery modular multiplication for better performance.

Modifications:

2 files (ModifiedRSA.java, Montgomery.java) were created besides the original code provided in PlainRSA.java.

First, Montgomery.java:

It provides two methods:

1. convertDomains : which convert to or from Montgomery Domain.
2. multiply: which performs Modular Multiplication of two BigIntegers in Montgomery Domain.

Second, ModifiedRSA.java:

it contains the same methods as the original code but with modifications in modExp implementations using Montgomery modular multiplication, with addition to init method to initialize the parameters needed by the Montgomery operations such as R and N' (Where R is the first power of two where $R > N \; and \; GCD(R, N) = 1$ And $N' = -N^{-1} mod \; R$).

Observations :

By running both PlainRSA and ModifiedRSA (in Montgomery, We are using the methods of BigInteger class such as divide and mod)

PlainRSA :

Just one run:

```
Total time is : 6.10535E7 nanoseconds
```

The average of 1000 runs:

```
Total time is : 3.0988206E7 nanoseconds
```

Since java is slow at startup, we can depend on the average of 1000 runs.

For ModifiedRSA:

Just one run:

```
Total time is : 1.582598E8 nanoseconds
```

The average of 1000 runs:

```
Total time is : 6.06821885E7 nanoseconds
```

We can see the PlainRSA is much better in performance. and we didn't get the expected boost from the Montgomery modular multiplication.

But we can take advantage of having **R as a power of two**, we can use And, shiftRight instead of mod and divide operations respectively.

```
// updated version (better performance taking advantage of R being power of 2)
BigInteger m = t.multiply(NPrime).and(RMinusOne);
t = t.add(m.multiply(N)).shiftRight(RMinusOne.bitCount());

// old version
BigInteger m = t.multiply(NPrime).mod(R);
t = t.add(m.multiply(N)).divide(R);
```

And after commenting the old version and calculating the average time for 1000 runs.

```
Total time is : 1.78827332E7 nanoseconds
```

The speedup is 1.7X $(3.0988206 * 10^7) / (1.78827332 * 10^7) = 1.733$.

Conclusion

We can see that Using Montgomery modular multiplication and taking advantage of having R as power of 2 speeds up the RSA decryption process.

Part II

Requirements:

Performing timing attack on Part I implementation to verify that the second most significant bit of the private exponent given in the above file is one.

Implementation:

Using the same exponent and the modulus from Part I, All the code needed for the attack, described in the paper, is in TimingAttack.java

TimingAttack.java

performs the following experiment using different portions of the private key, provided by

```
portions = new int[]{3, 5, 10, 20, 50, 100};
```

Each experiment is repeated 20 times, and check whether that attack successfully figure out that the second most significant bit is 1 or not

Experiment:

1. Generate 10000 decrypted messages.
2. For each message we first check if it needs reduction in both cases (bit is 0 or 1).
3. Then we run the decryption process of the ModifiedRSA from part I and calculate the running time.
4. Finally we calculate the average of running times of 4 sets (all combinations of second bit is 1 or 0 and needs reduction or not), let

   $1 \longrightarrow$ *average time of bit is* 1 *and needs reduction*

$2 \longrightarrow$ *average time of bit is* 1 *and doesn't need reduction*

$3 \longrightarrow$ *average time of bit is* 0 *and needs reduction*

$4 \longrightarrow$ *average time of bit is* 0 *and doesn't need reduction*

5. If the bit is 1 then:

   $1 > 2 \quad AND \quad 4 \approx 3$

   else

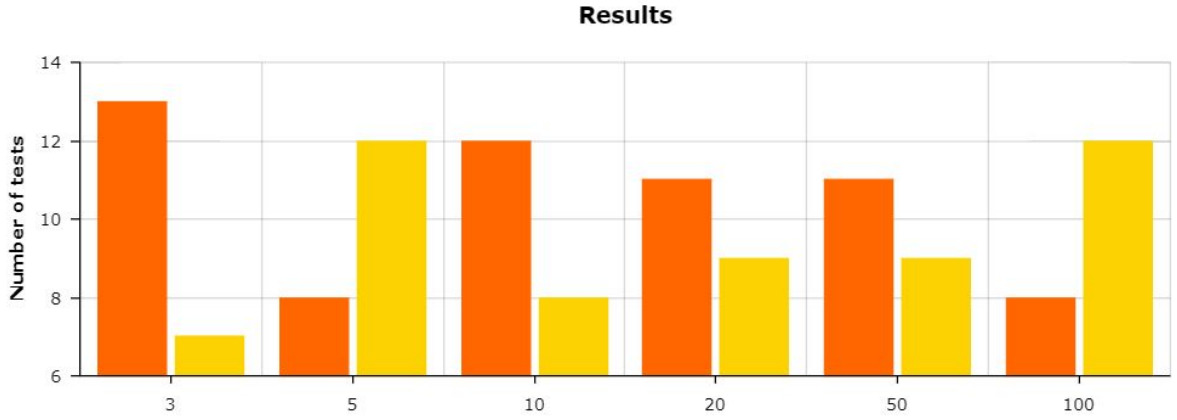
   $3 > 4 \quad AND \quad 1 \approx 2$

   and as we know that the second bit is 1 and we need to verify that fact, thus we just need to check if $(1 - 2) > (3 - 4)$.

Results: (amcharts.com is used to draw the charts in this section)

Let Orange bar refer to the number of experiments where the attack got the bit right, and Yellow bar refer to the number of experiments where the attack failed.

1. With no Amplifications in Reduction operation (Subtraction in Montgomery)



We can see from the chart that we failed to reveal the second most significant bit in nearly half of the tests (Accuracy : 52.5%).

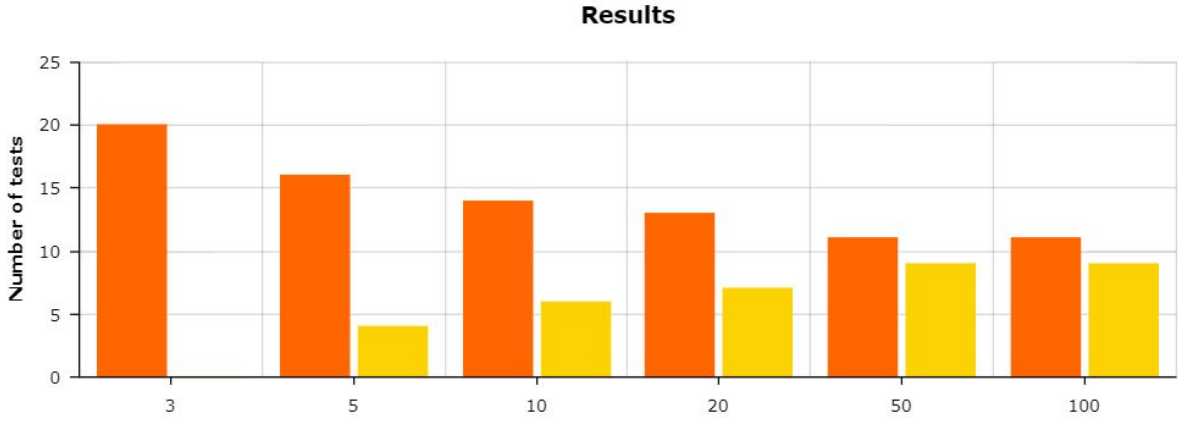
2. With Amplifications in Reduction operation (Subtraction in Montgomery)

with modifying the code in Montgomery to

```
if (t.compareTo(N) >= 0) {

    for (int i = 0; i < 10; i++) { // for attack
        t.subtract(N);
    }
    return t.subtract(N);
}
return t;
```

The result was :



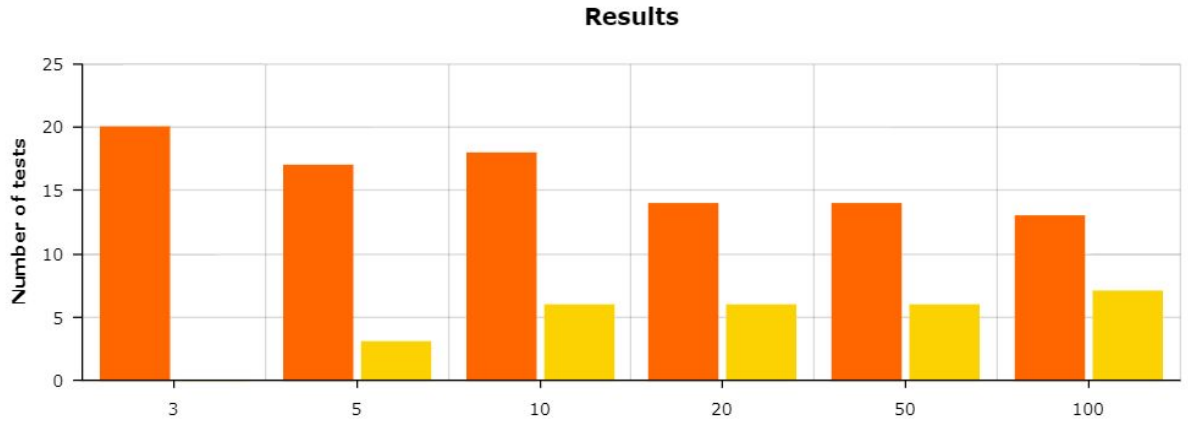
As we can see from the chart, it's easy to verify the second most significant bit in small portions and it's getting more difficult while increasing the length of the portion.

(We got Accuracy 70.833333%)

If we increase the amplification of reduction operation from 10 to 20 subtractions

```
if (t.compareTo(N) >= 0) {

    for (int i = 0; i < 20; i++) { // for attack
        t.subtract(N);
    }
    return t.subtract(N);
}
return t;
```



Better results as expected, since the reduction branch takes double time.
Accuracy: 80%

Conclusion:

In small portions the difference in time if there's a reduction branch is clear and easy to recognize, thus we can reveal the second most significant bit in small portions in most cases correctly. As portion size increases, it’s more difficult to reveal the bit correctly and it may need more samples and more precise and accurate measures of time.