Michael Adams
CS 1699 – Project 1

Task W0

In this project I examined the execution speed of modular exponentiation. More specifically, I focused on how the runtime for this algorithm was affected by different input types and sizes. Formally, modular exponentiation (referred to as Problem P) can be stated as:

$$m = c^d \ (mod\ n) \qquad \text{(Eq. 1)}$$

As the equation shows, the base *c* will be raised to some exponent *d*. The output *m* is guaranteed to be within the range of [0, n) because the result of the exponentiation is taken modulo *n*.

Task W1

The notation used in Equation 1 was chosen purposefully as it represents the syntax presented for decrypting a message in RSA, a common application which depends on solving Problem P. RSA is a public-private key system currently considered one of the "best practice" in cryptography for securely sending messages. In this case, *c* is a ciphertext, *d* is a private key, the modulus *n* is part of the public key, and *m* is the plaintext message resulting from the decryption. To give the math a more tangible context, we will discuss Problem P in terms of its use in RSA throughout this report.

The naïve implementation of the algorithm has a runtime that is sensitive to its exponent *d*. Because *d* represents a private key, this variation in execution time presents a significant security risk. If an attacker can monitor the time it takes to solve Problem P, they may be able to draw conclusions about the value of *d*.

Task W4

As anticipated, the square-and-multiply process used in the naïve implementation of modular exponentiation (referred to as Algorithm A) caused a distinct variation in runtime depending on the size and value of d. The algorithm iterates over each bit in the exponent so we would expect to see the execution time increase with increasing private key length. To test this, I generated random 256, 512, and 1024-bit numbers to use as *d*. To limit confounding factors, I used the same *c* and *m* values for each exponentiation. To ensure consistency, I completed each exponentiation 5 times and averaged the values. The results are shown in Table 1. While key length has a strong effect on execution time, this does little to compromise the security of the private key *d*. Even if an adversary were able to determine the key length of the private key, the only way it would be beneficial is if the adversary discovered the user was using a private key short enough to brute-force.

Table 1: Effect of Key Size on Execution Time

Key Size	Average Execution Time
256-bit	3.56 sec
512-bit	9.69 sec
1024-bit	29.06 sec

A much more concerning security vulnerability is seen in keys of the same length. The naïve implementation of Problem P is affected by the value of the current bit of *d* being iterated over. If the current bit value is 0 (and the exponent is therefore even) the base value is squared. However, if the current bit value is 1, and additional multiplication is included. Figure 1 shows how this handled in my implementation of Algorithm A.

Figure 1: Additional multiplication in the case of odd exponent (starts on line 312)

To accentuate this difference, I generated three 256-bit keys to use for *d*. One key had mostly zeros (each byte = 00000001), one key was random, and one was all ones. As before, I used the same *c* and *n*, and averaged the results for each key over 5 exponentiations. The results are shown in table 2.

Table 2: Effect of Key Value on Execution Time

Key Value	Average Execution Time
Mostly Zeros	7.28 sec
Random	9.69 sec
Ones	13.92 sec

As we would expect, the additional multiplication required on every iteration for the Ones key produced an execution time nearly twice as long as the Mostly Zeros key, which only required a second multiplication once per byte. This leaves the algorithm vulnerable to a powerful timing attack technique. To execute this attack, an adversary would need to observe the runtime for the decryption and have access to the ciphertext *c*, and the modulus *n*. The adversary would use the RSA decryption algorithm along with the known ciphertext and modulus *n* to complete the operation in parallel with the user. Using statistics, the adversary would approximate the likelihood that the multiplication step was executed for the first bit, meaning that the bit value is 1. From there, the attacker would iterate along each bit predicting whether the bit value is 1 or 0. If the variance between the user's decryption timing and the attackers increases, this indicates that the adversary guessed the wrong bit. They could then backtrack and try a different value which would minimize the variance. This process allows the attack to be self-correcting which improves its ability to actually guess the exponent *d*.

Task W7

In an attempt to introduce some blinding and hopefully mitigate the timing attack discussed in W4, I looked to address the vulnerability in Algorithm A. The square-and-multiply approach is one of the simplest ways to solve Problem P, but the conditional multiplication (executed only when current bit of *d* is 1) effectively doubles the execution time for iterations in which it is used. If we could eliminate this conditional multiplication, each iteration of the exponentiation loop would always require only one operation regardless of current bit value. However, this would fundamentally change the output and is therefore not an option. Borrowing an idea from Montgomery Power Laddering, I decided to add an operation instead of removing one in order to balance execution time. Rather than skipping to the squaring step when the current value of *d* is 0, I added another conditional multiplication. However, we obviously don't want this extraneous multiplication to affect the output of the exponentiation, so the operation is stored in a temporary LargeInteger. The programmatic implementation is shown in Figure 2.

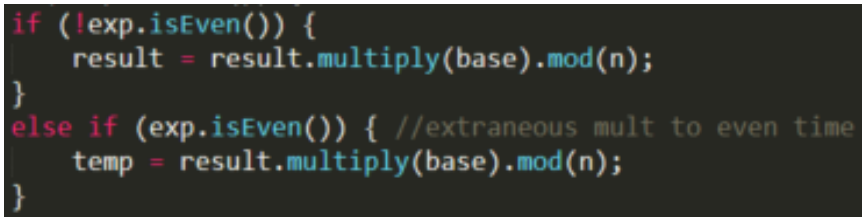

Figure 2: Additional multiplication in the case of either even or odd exponent (starts on line 342)

This straightforward approach significantly improved the consistency of the execution time for keys of the same length. Using the same methodology implemented in C3 and discussed in W4, I collected data on my improved algorithm (referred to as Algorithm B). As Table 4 shows, Algorithm B produces much more even execution times across the different key values.

Table 4: Effect of Key Value on Execution Time

Key Value	Average Execution Time
Mostly Zeros	10.15 sec
Random	13.76 sec
Ones	13.41 sec

Whereas Algorithm A had a 90%+ difference between Mostly Zeros, and Ones, Algorithm B reduced that difference to about 30%. Further, the execution time between the Random and Ones keys was improved to being nearly identical. The private key *d* should contain an approximately even number of 1s and 0s meaning that it should most closely resemble the Random test key. The much narrower execution time range in Algorithm B provides significant security improvements because even if the private key *d* should happen to contain a disproportionate amount of 1s or 0s, the execution time would not be significantly different. This mitigates the timing attack by lowering the variance between a 1 and 0-bit loop execution time, making it much harder to predict bits in the exponent and therefore guess *d*.

For consistency, I included the results from the different sized test keys in Table 5. Algorithm B was not intended to make a more constant execution time across different exponent sizes, and predictably had almost no effect.

Table 5: Effect of Key Size on Execution Time

Key Size	Average Execution Time
256-bit	3.56 sec
512-bit	9.69 sec
1024-bit	29.06 sec

One advantage to my implementation of Algorithm B is the minimal tradeoffs. The runtime does increase slightly relative to Algorithm A, and it requires more computational work because two multiplications are required for every iteration on *d*. However, this does not increase the asymptotic time complexity. Though the focus of this report was to discuss timing attacks, one added benefit of Algorithm B is power analysis attacks become much more difficult because the same amount of work is done on every iteration of the exponentiation loop. Finally, Algorithm B does not rely on any source of randomness or make any additional assumptions.

Task W8:

While my implementation of Algorithm B does result in much more consistent execution time for Problem P, it is not perfectly secure. As Table 4 shows, there is still some degree of difference between Random/Ones and Mostly Zeros. Therefore, I suppose it is possible to attempt a timing attack, though it would likely require many additional attempts and/or more advanced statistical methods like the Box Test.