Jacob Ingraham

CS556

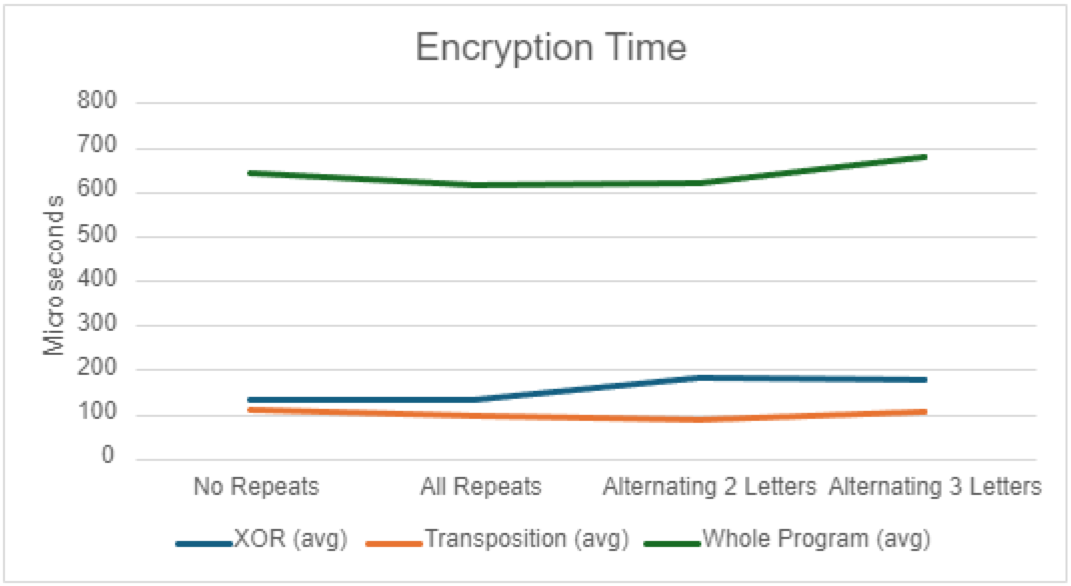
Encryption Evaluation

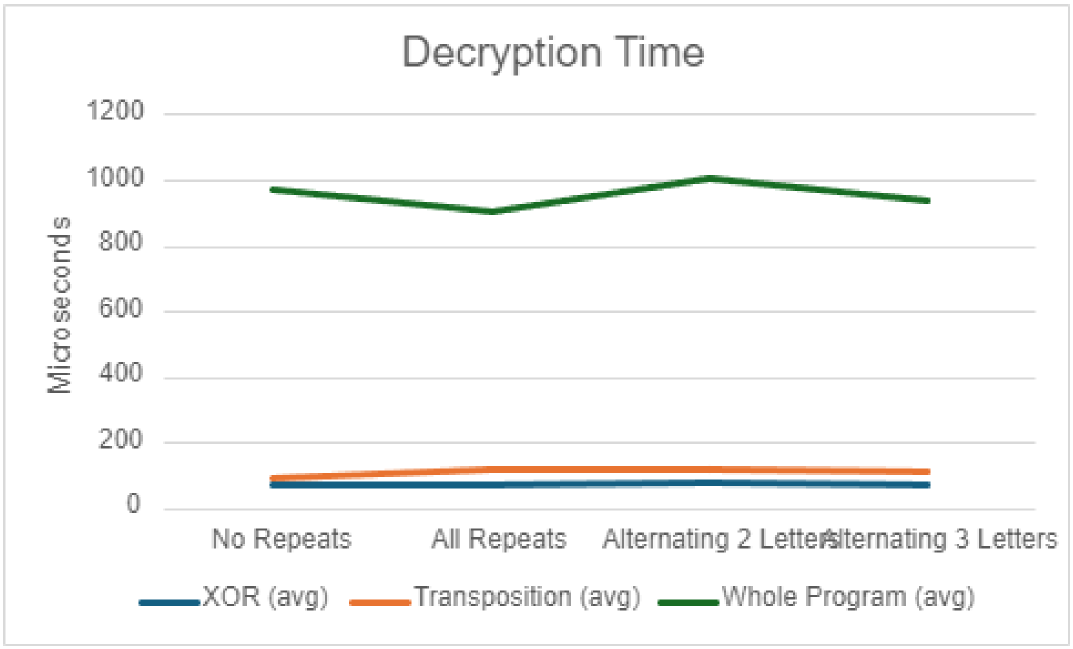
Part A (symmetric encryption):

In subdirectory PartA, the source code for this encryption algorithm is encrypt.cpp and decrypt.cpp. Please see the README for information on compiling and running the code.

This algorithm uses a simple combination of column-based transposition and bitwise XOR. It takes two 10-character strings to use as the keys. Since the transposition uses the strong alphabetical ordering of the keys, each key must have no duplicate characters within itself. Only the first key is used for the XOR, and the transposition is done for both keys.

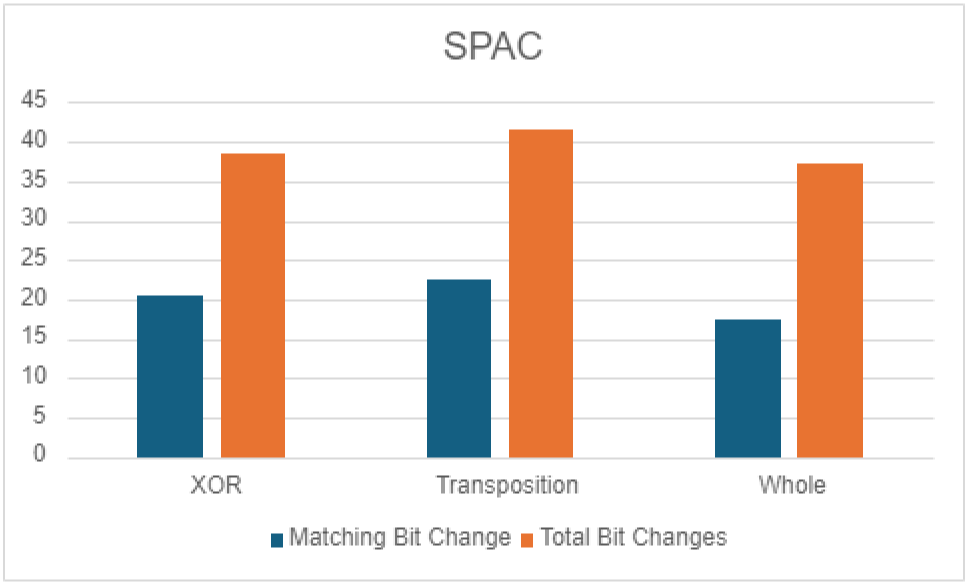
Question 1 – Test this algorithm's resilience to side-channel analysis timing attacks: When timing my encrypt and decrypt programs with different patterns in the keys I did not notice any significant patterns. The graphs are below (all averages are calculated from 5 runs). Although there is some variation between the keys, the lines on the graphs are mostly flat.



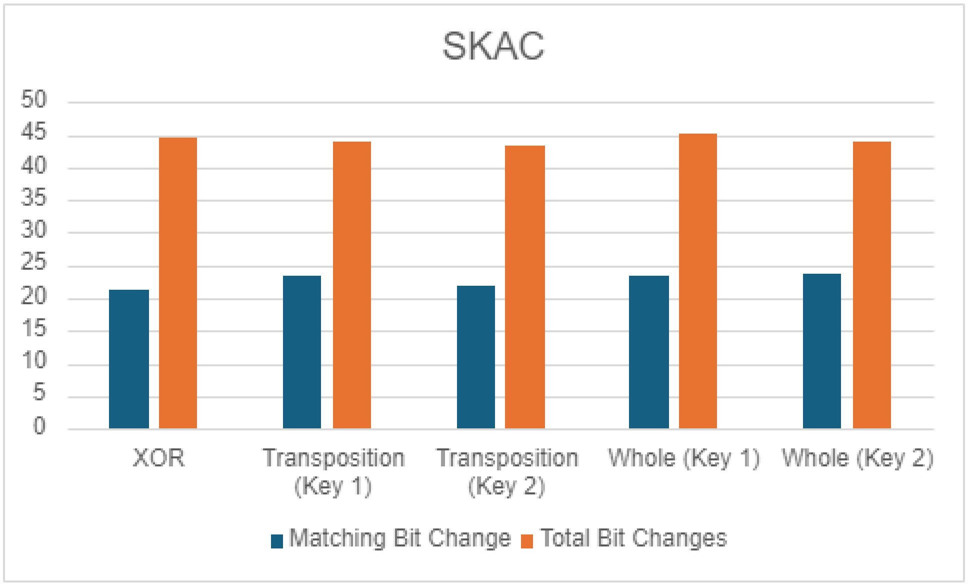


Question 2 – Test if this is a cryptographically strong algorithm based on SPAC and SKAC: The Strict Plaintext Avalanche Criterion (SPAC) and Strict Key Avalanche Criterion (SKAC) require there to be no correlation between the input or key bits and the output bits. The definition of SPAC states that any bit of the ciphertext should change (not be identical to the previous bit) with probability of 0.5 whenever the corresponding bit of the plaintext changes. Similarly, the definition of SKAC states that any bit of the ciphertext should change with probability of 0.5 whenever any bit of the key changes. Since SPAC references the plaintext's correlation with the ciphertext it is important for the key to be fixed when measuring this. The SKAC references the key's correlation with the ciphertext so the plaintext should be fixed when measuring this.

I calculated correlation between bit change in the plaintext and bit change in the ciphertext for SPAC in the following manner. While incrementing through the plaintext I incremented a 'total' counter every time a bit change was detected. I also checked the matching location of the ciphertext to determine if a matching bit change was present and incremented a 'same' counter if there was. I calculated the final values using an average of 10 runs (with the plaintext input changed every run and key values constant). The average probability for XOR was 0.53, for double transposition was 0.54, and both together was 0.47. Based on these values I believe that all three satisfy the SPAC. Below is a graph showing the average values of the 10 runs that I measured.



I calculated correlation between bit change in the key and bit change in the ciphertext for SKAC in a similar manner. Instead of iterating through the plaintext I compared with the keys. I also kept the plaintext constant while I changed the keys for every run. The average probability for XOR was 0.47, for double transposition was 0.53 (key1) and 0.51 (key2), for both together was 0.52 (key1) and 0.54 (key2). Based on these values I believe that all three satisfy the SKAC. Below is a graph showing the average values of the 10 runs that I measured.



Part B (asymmetric encryption):

In subdirectory PartB, the source code for this encryption algorithm is asymmetric_encr.cpp and asymmetric_decr.cpp. Also in subdirectory PartB, the source code for generating public and private keys is makeKeys.cpp. Please see the README for information on compiling and running the code.

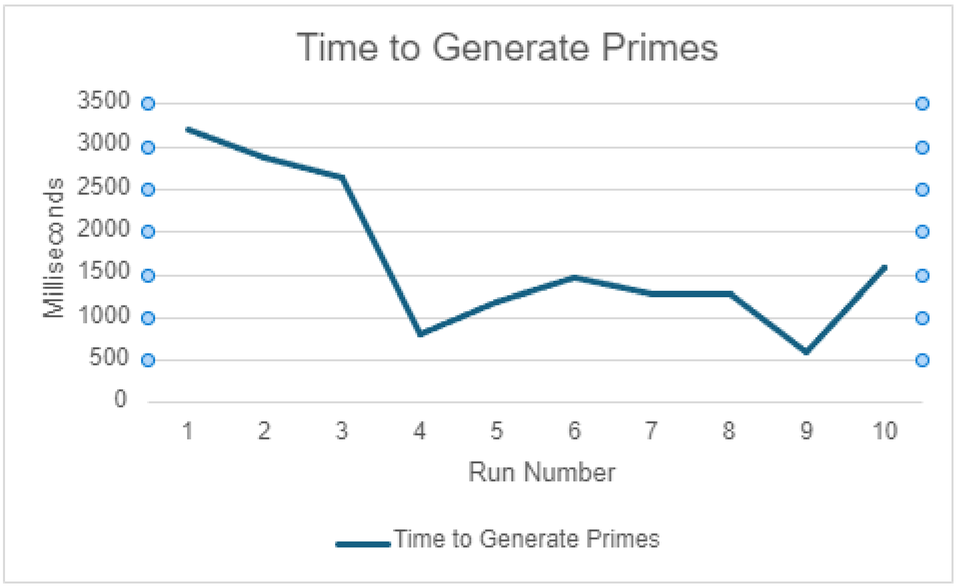
This encryption algorithm is an implementation of the RSA standard using 4096 bit keys. It uses the GMP (GNU Multiple Precision) library for C++ to handle the large integers.

Key Generation (The graphs below show the variation across 10 runs of the program):

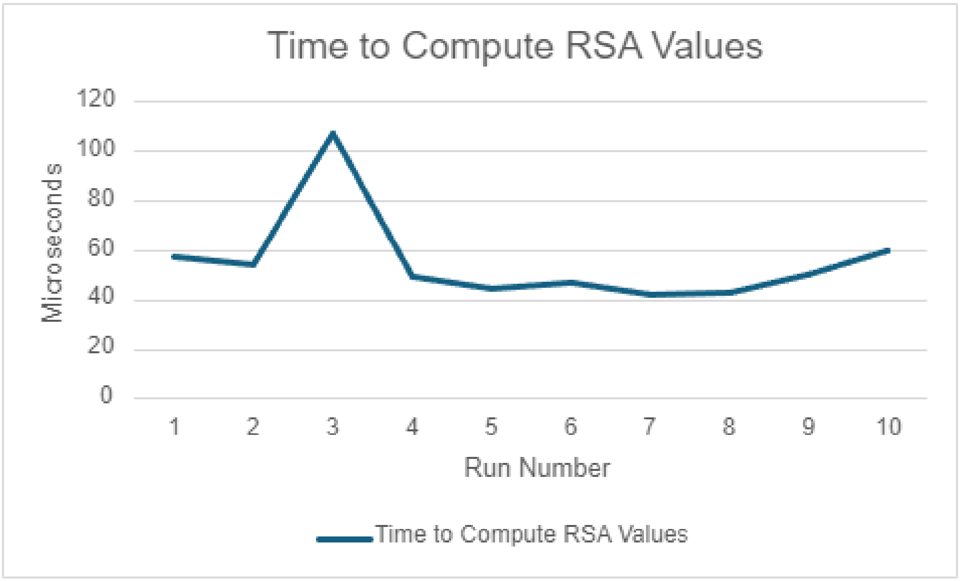
By far the most expensive part of this program is the generation of large primes. It is the only graph I created which needed to be scaled to milliseconds, and I could have made the y-axis measure seconds. I thought this was interesting because the key generation algorithm takes no input so it is all technically constant time. However, since it generates integers until two high probability primes are found, n can represent the number of integers it generates until the tests are passed.

Based on my tests this n is generally large (several hundred to several thousand) and the operations performed on each one of these integers are extensive. Some of them will fail the sieving process because they are divisible by one of the first prime numbers. But many of them will move on to the Fermat test (which will run <=30 times) and potentially the Miller-Rabin test (which will run <= 40 times). Both more advanced tests work with modular exponents and modular operations which can be very expensive, especially on these 2048-bit integers.

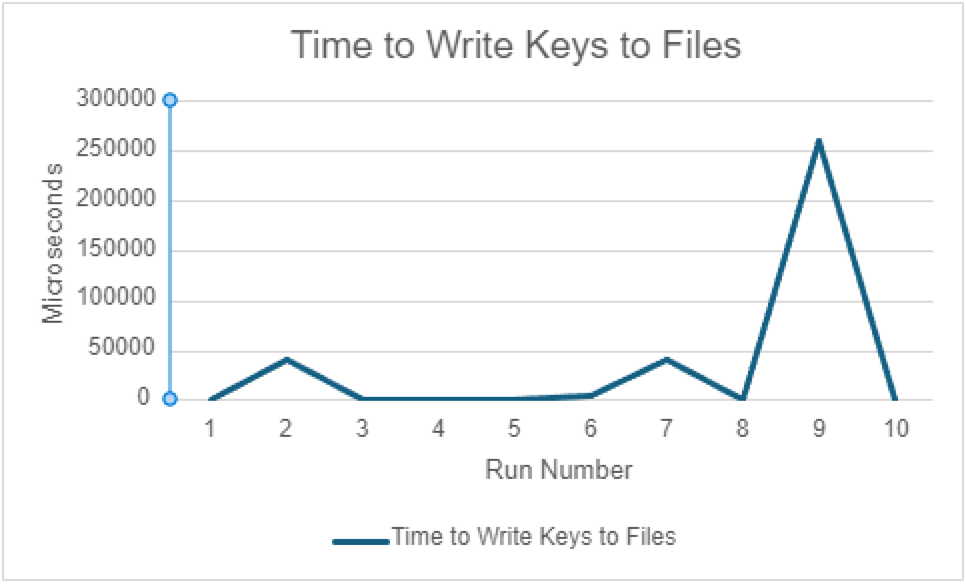
Although these operations can take a lot of time and resources, I don't think this function is greater than O(n) time. Only the Miller test has a nested loop, but it doesn't always trigger. Instead, I think this function is O(n) with a very large operational overhead.



The computation of RSA values such as: n, e, phi, and d. This was a very simple section of code containing a handful of constant operations. There is some overhead because the modulus and exponentiation of large values can be expensive, but it is still relatively cheap and bounded by O(1).



Writing keys to files. Although this is also a constant time piece of the program, I thought it was interesting that the 9th run took so much time. I looked at the key files and didn't notice anything out of the ordinary. This leads me to conclude that the longer time was caused by some type of operation in the CPU that was unrelated to this computation.

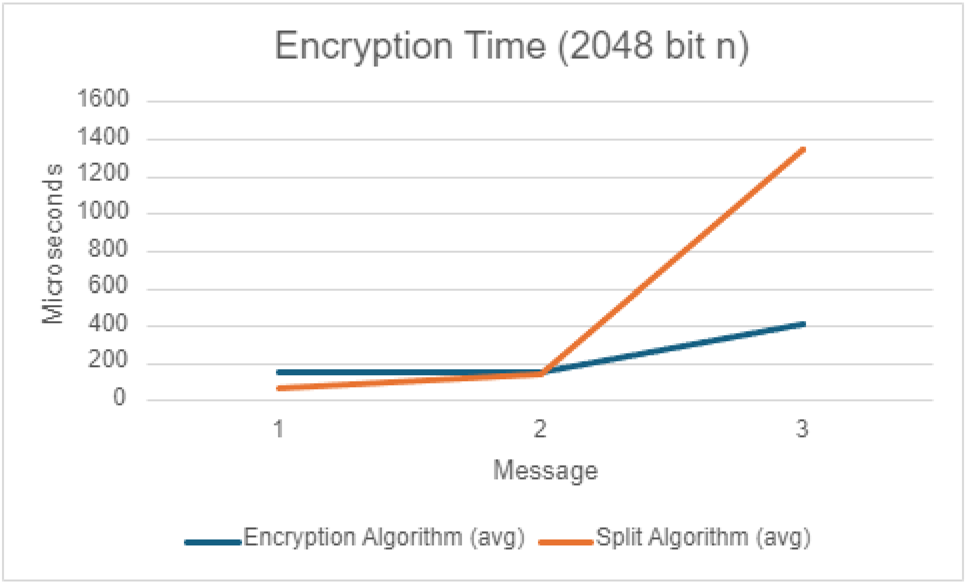


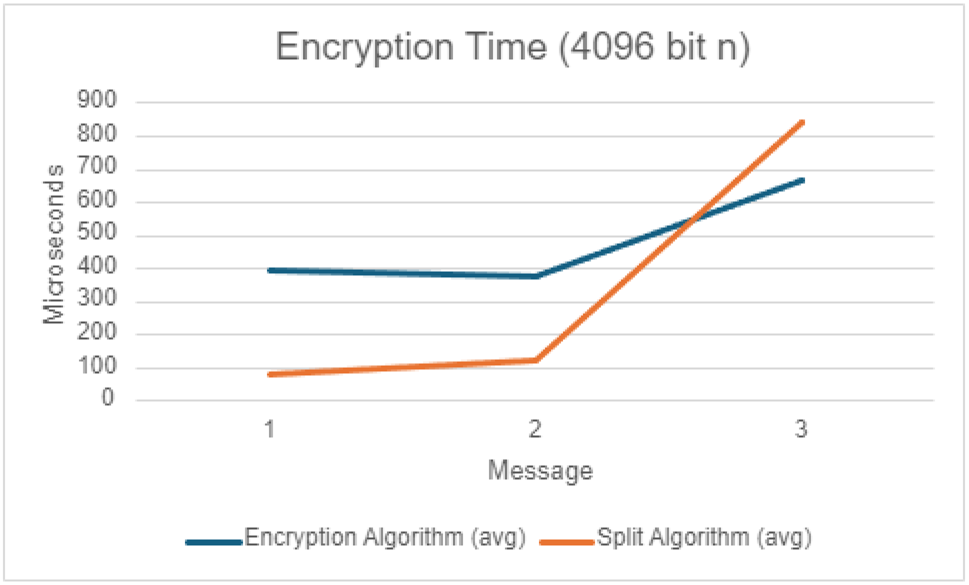
Encryption and Decryption (The graphs below show the averages for different message sizes. Message 1 = ~100 bits, Message 2 = ~1,000 bits, Message 3 = ~10,000 bits):

For this section I will compare the encryption and splitting algorithms within the RSA implementation. Since the specifications require us to split the message into pieces if m (size of message) is greater than n (size of key), I designed a recursive algorithm to split the message into chunks which are less than n.

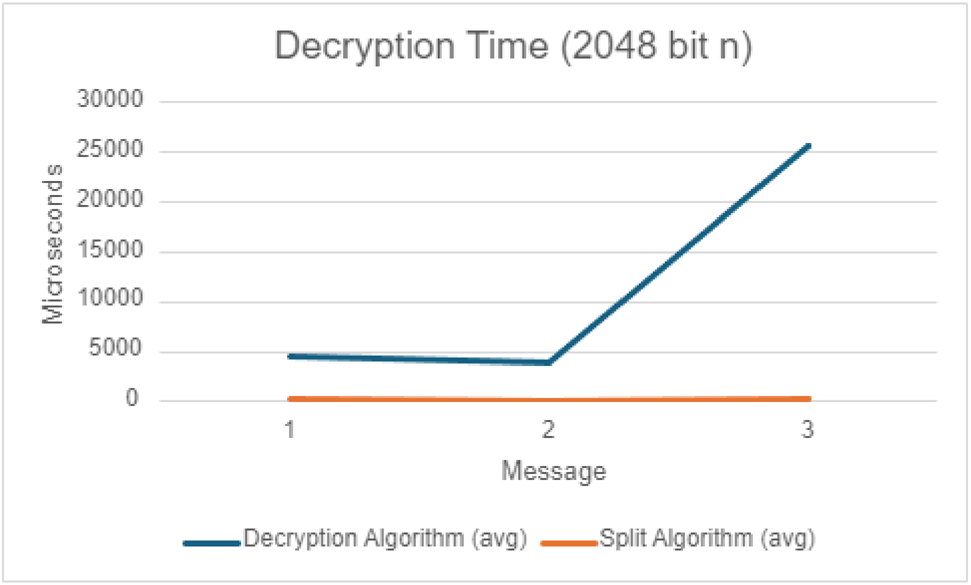
The encryption algorithm consists of one modular exponent operation for each section of the message. If m is very large and n is very small this could be O(n) but for the purposes of this program it is closer to O(1). Similarly, the splitting algorithm usually makes no (or very few) recursive calls and is O(1) in practice. Since it calls itself twice, I believe the real bound is O(2^n).

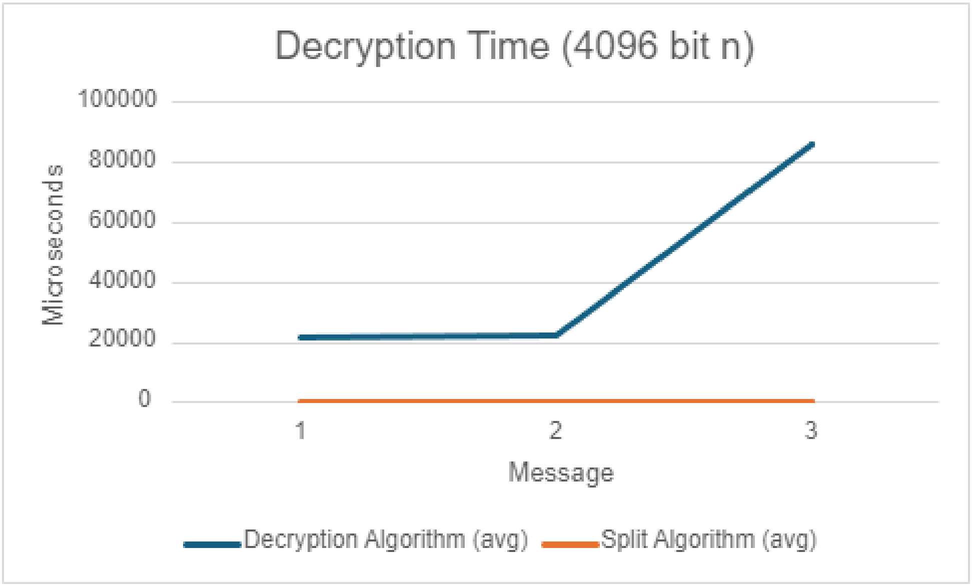
I ran these tests with both 2048-bit n and 4096-bit n to see how the algorithms behaved, and I found nothing surprising. The encryption almost perfectly doubled with larger n, which is how I would expect a constant time operation to behave when it is working with an integer of double the size. The splitting algorithm stayed about the same for messages 1 and 2 (because no splitting was done) but spiked less drastically when n was larger (because less splitting was done).





Although the encryption and decryption algorithms are the same, but use different keys, their performance is drastically different. This difference is even more obvious when 4096-bit n is used. I believe the reason for this increase in overhead is that the computation is working with much larger numbers. The ciphertext is much larger than the original message and the private key d value is much larger than its corresponding e. Despite this, it is still a constant time algorithm with bound of O(1).
The splitting algorithm for decryption is completely different, with no recursion involved. Since there is no way to know how large the block put into the encryption algorithm was, the output is padded with 0's to ensure each block will be the same size. I thought this might affect the security of the ciphertext, but experimentation showed that the padding was very rarely needed. The output of the modular exponent was usually very close to the size of n. Thus, the decryption splitting algorithm is constant time with bound of O(1) since it just pulls blocks of size < n.


Decryption Time (2048 bit n)
30000
25000
20000
15000
10000
5000
0
Microseconds
1
2
3
Message
Decryption Algorithm (avg)
Split Algorithm (avg)


Decryption Time (4096 bit n)
100000
80000
60000
40000
20000
0
Microseconds
1
2
3
Message
Decryption Algorithm (avg)
Split Algorithm (avg)