RSA Based Primality Test

New Mexico

Supercomputing Challenge

Final Report

April 4th, 2018

LAHS71

Los Alamos High School

Student:

Elijah Pelofske

Teacher:

Alan Didier

Abstract

Efficient and accurate primality testing is a key mechanism used to ensure digital security in the modern world. Ensuring accurate primality is critical for guaranteed accurate message encryption and decryption, accurate identity verification, as well as a secure cryptosystem, whose security is guaranteed by the discrete logarithm problem. There are two types of algorithms that determine whether an integer is prime or composite, known commonly as primality tests. The two types are, probabilistic which in general are faster but less accurate, and deterministic tests which are slower but guaranty accuracy.

The purpose of this project is to create a primality test that is based upon the asymmetric cryptosystem RSA. The RSA Based Primality Test works by using the first N primes respectively as both a prime factor of the modulus and encryption exponent. The integer being tested for primality is the other assumed prime factor of the modulus. A random message, M, that is less than N, is then encrypted and decrypted using the constructed cryptosystem. If that M is equal to original M, the test increments one on the count of true instances, otherwise it increments one on the count of false instances.

It was found that count of false instances for 2 factor RSA was the most efficient form of this test - for x supplied known primes between 2 and N, and y being the first odd counterexample, the line y = 3.3*x^2.7 describes the RSA Based Primality Tests properties with a correlation coefficient, r, of 0.97.

Purpose

The purpose of this project is to create a primality test algorithm which is based on the RSA cryptosystem. The advantage of this type of test, be it probabilistic or deterministic, is that it relies on proven hypothesis and concepts, and would not function the same, i.e. have the same flaws, as other types of probabilistic primality tests. Additionally, so far this exact type of test has not been implemented in this manner.

Introduction

The accurate and efficient primality testing is critical for constructing asymmetric cryptosystems. Accuracy of primality testing is important because encryption and decryption of every message less than N must be accurate. In addition to inaccurate encryption, inaccurate primality testing can result in cryptographically insecure keys. Efficiency of primality testing is important because cryptosystem construction must not take an intractable amount of computation time. Digital signatures, secure message transmission, and identity verification are all incredibly useful applications of asymmetric cryptography, as well as being a vital part of modern digital industry on the internet.

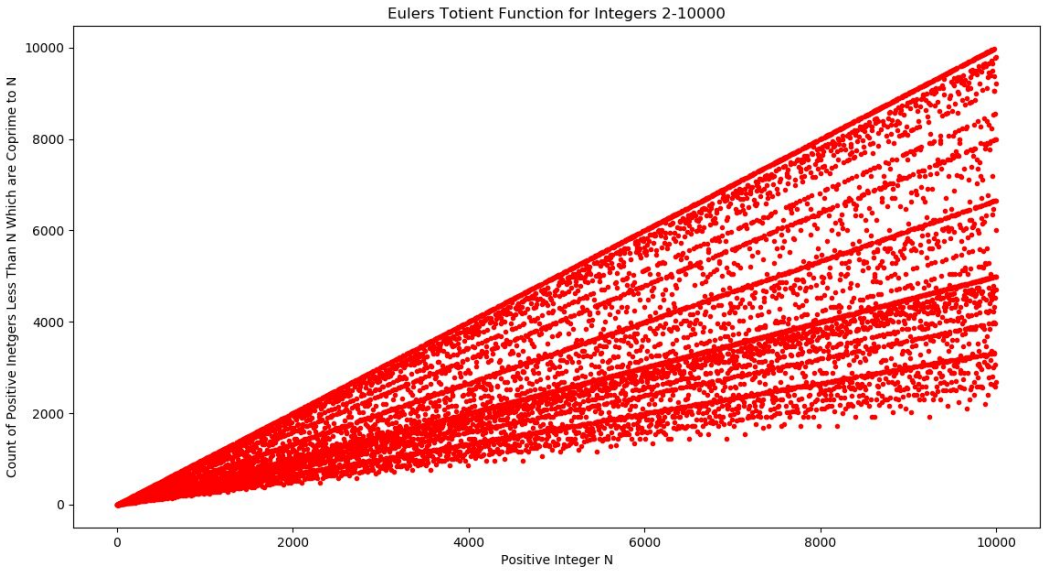
Examples of asymmetric cryptosystems (aka Public-Key cryptography) or key exchanges include RSA, Diffie-Hellman key exchange, ElGamal encryption, Paillier cryptosystem, and the Elliptic Curve cryptographic versions of all of these. In summation, efficient and accurate primality testing is critical for modern digital security.

The RSA cryptosystem on works using two key pairs; (e, N) and (d, N). The two key pairs are inverse permutations of each other via the simple function of raising the

message to encryption or decryption exponent (e or d) remainder division the modulus, N. As long as d is kept private, RSA can be used as identity verification as well as a secure cryptosystem. The operation of raising a number, M, to the power of e modulo (remainder division) N can be speed up on computers using a technique known as modular exponentiation. The modulus N is constructed from two or more known prime factors, and thus the Euler Totient function, denoted as φ(N), or the count of integers less than N which or coprime to N, can be constructed from the known prime factorization of the modulus, N. Figure 1 shows Euler's Totient function for all positive integers less than 10000. The public (encryption) exponent, designated as e, must be coprime to N and φ(N). The decryption exponent, d, must satisfy the equation 1=d*e modulo φ(N), and can be computed efficiently using the Extended Euclidean Algorithm.

Probabilistic primality tests do not guarantee a certificate of primality. Most probabilistic tests can determine if a number is definitively composite, but sometimes will output a false positive for primality. Other probabilistic primality tests simply use mechanisms which have not been fully proven to guarantee primality. Deterministic primality tests give a definitive output of either primality or compositeness.

Figure 1



Experiment/Model

The RSA Based Primality Test works by using the first N primes respectively as both a prime factor of the modulus and encryption exponent. The integer being tested for primality is the other assumed prime factor of the modulus. A random message, that is less than N, is then encrypted and decrypted using the constructed cryptosystem. If that M is equal to original M, the test increments one on the count of true instances, otherwise it increments one on the count of false instances. It should be noted that earlier and less reliable versions of the RSA Based Primality Test, such as the several possible multiprime implementations, are not included in this final version. There are 18 earlier versions versions which include some of these many elements that were found to

be not as useful. Figure 2 below is the final version of the RSA Based Primality Test coded in Python3.

Figure 2

```
import random
import time
from fractions import gcd
#The Miller Rabin primality test is used to find the known primes for the RSABPT
def miller_rabin(n):
        k = 10
        if n == 2:
                return True
        if n % 2 == 0:
                return False
        r, s = 0, n - 1
        while s % 2 == 0:
                r += 1
                s //= 2
        for _ in range(k):
                a = random.randint(2, n - 1)
                x = pow(a, s, n)
                if x == 1 or x == n - 1:
                        continue
                for _ in range(r - 1):
                        x = pow(x, 2, n)
                        if x == n - 1:
                                break
                else:
                        return False
        return True
def egcd(a,b):
        u, u1 = 1, 0
        v, v1 = 0, 1
        while b:
                q = a // b
                u, u1 = u1, u - q * u1
                v, v1 = v1, v - q * v1
                a, b = b, a - q * b
```

```
        return u
def mod_inverse(e,phi):
        return egcd(e,phi)%phi
def coprime_message(n):
        c = 1
        while c < n-1:
                c += 1
                m = random.randint(2, n-1)
                if gcd(m, n) == 1:
                        return m
def primegen(n):
    c = 0
    int = 1
    out = []
    while c < n:
        int += 1
        if miller_rabin(int) == True:
            c += 1
            out.append(int)
    return out
def rsabpt(q, max):
#RSA Based Primality Test Function
#q is the given integer being tested for primality
#max is the number of known primes used in the function
    true_primes = primegen(max)
    for prime in true_primes:
        for e in true_primes:
          n = q*prime
          phi = (q-1)*(prime-1)
          d = mod_inverse(e, phi)
          if e < phi:
            if gcd(e, n) == 1:
              if gcd(e, phi) == 1:
                if gcd(d, n) == 1:
                  if gcd(d, phi) == 1:
                     if gcd(n, phi) == 1:
                          m = coprime_message(n)
                          c = pow(m, e, n)
                          m1 = pow(c, d, n)
```

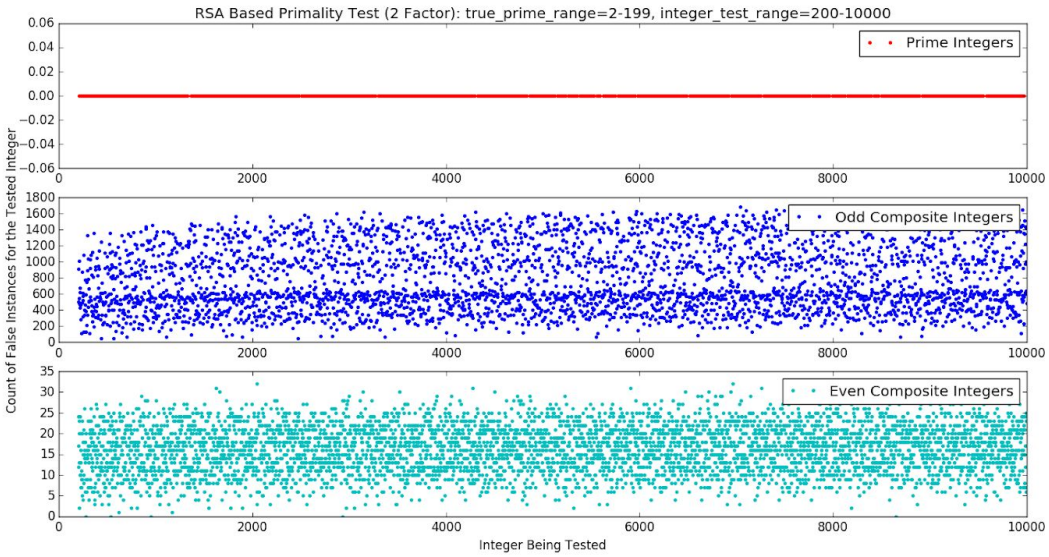
```
                    if m1 != m:
                      return False
    return True
start = time.clock()
p, t = rsabpt(65537, 20), time.clock() - start
print(p, t)
#Test example, 65537 is a fermat prime
#Also showing the time.clock() function for determining CPU seconds used
```

Results

The analysis of the functionality and capabilities of the RSA Based Primality test involved massive quantities of data analysis, therefore much of the benchmark results obtained are best illustrated in the form of graphs. All graphs were constructed using the Python library matplotlib. All CPU computation time benchmark tests were done on a Intel® Core™2 Duo CPU E4600 @ 2.40GHz × 2, and the operating system used was Kali Linux. All of the computations done for this project were completed across Windows 10, Kali Linux, and Ubuntu Linux machines, as well as online IDE's which ran the code on remote Linux machines. A total of 6 different computers were involved in gathering these results, including one which handled online IDE computations.

Figures 3, 4, 5, and 6 show some results of the 4 different possible conditions under which the RSA Based Primality Test could operate. The two defining aspects are 2 factor, or 3 factor, and whether the test outputs a count of true instances or a count of False instances. Figure 3 is the most promising form of the test, which is a 2 factor count of false instances test.

Figure 3



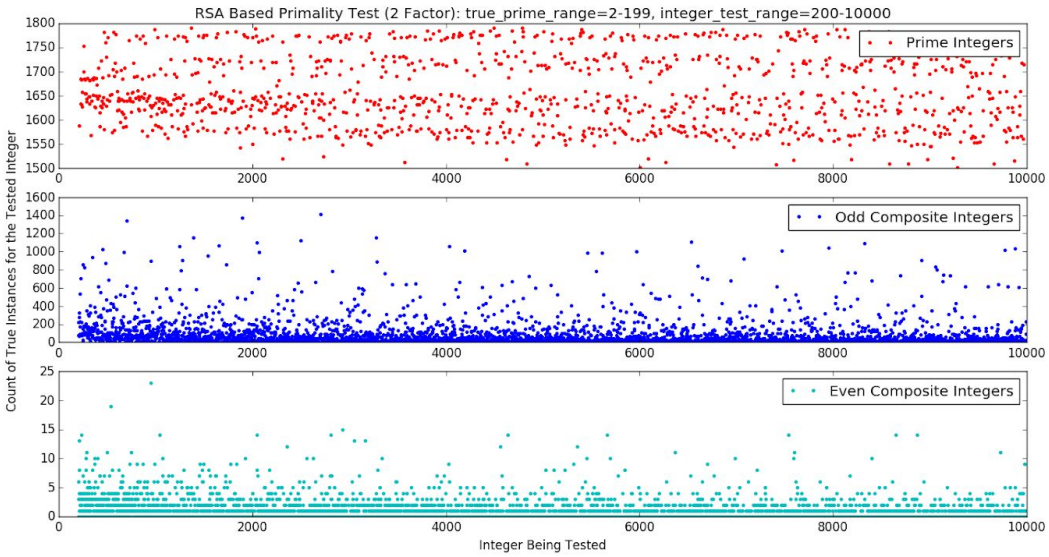
RSA Based Primality Test Minimums and Maximums for 2 Factor Count of False Instances

(max(false_prime), min(false_prime)) = (0, 0)

(max(false_odd), min(false_odd)) = (1683, 47)

(max(false_even), min(false_even)) = (32, 0)

Figure 4



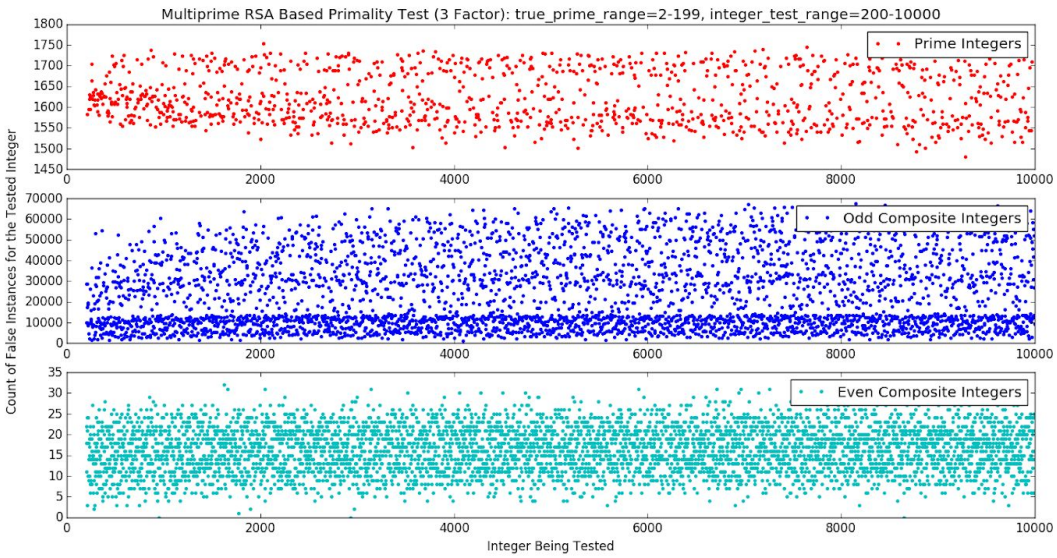
RSA Based Primality Test Minimums and Maximums for 2 Factor Count of True Instances

(max(true_prime), min(true_prime)) = (1790, 1502)

(max(true_odd), min(true_odd)) = (1412, 0)

(max(true_even), min(true_even)) = (23, 0)

Figure 5



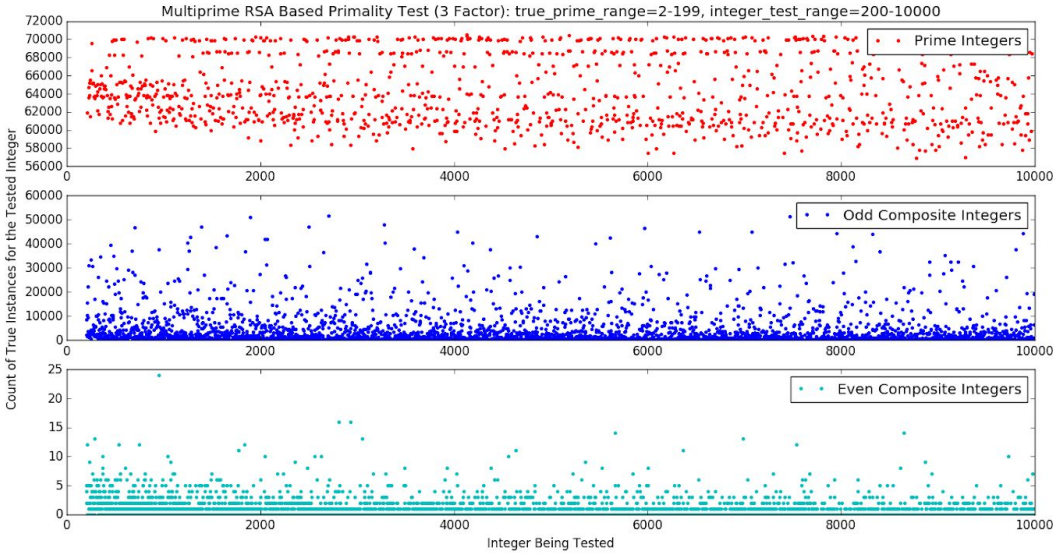
RSA Based Primality Test Minimums and Maximums for 3 Factor Count of False Instances

(max(false_prime), min(false_prime)) = (1748, 1489)

(max(false_odd), min(false_odd)) = (67465, 1299)

(max(false_even), min(false_even)) = (32, 0)

Figure 6



RSA Based Primality Test Minimums and Maximums for 3 Factor Count of True Instances

(max(true_prime), min(true_prime)) = (70483, 56913)

(max(true_odd), min(true_odd)) = (51426, 2)

(max(true_even), min(true_even)) = (24, 0)

Figure 7 below shows the percentage of incorrect instances for integers less than 10000 given an increasing number of known prime integers supplied to the RSA Based Primality Test, using count of false instances, 2 factor. An important note on the figure itself for clarity is that when x = 9, y = 0.

Meanwhile, Figure 8 shows percentage of incorrect instances for all positive integers. Again, this test was only done for the 2 factor, false count version of the RSA Based Primality Test.

Figure 7

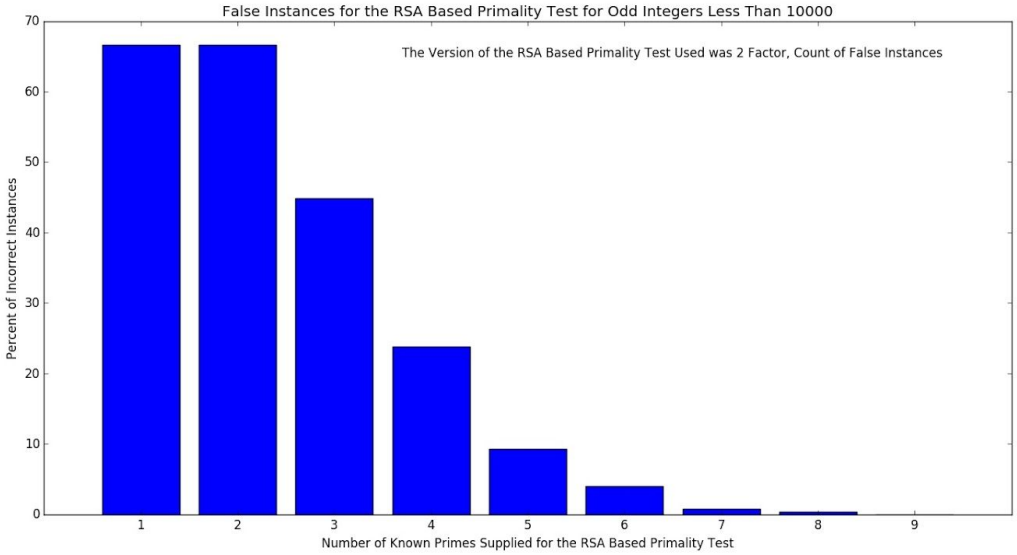


Figure 8

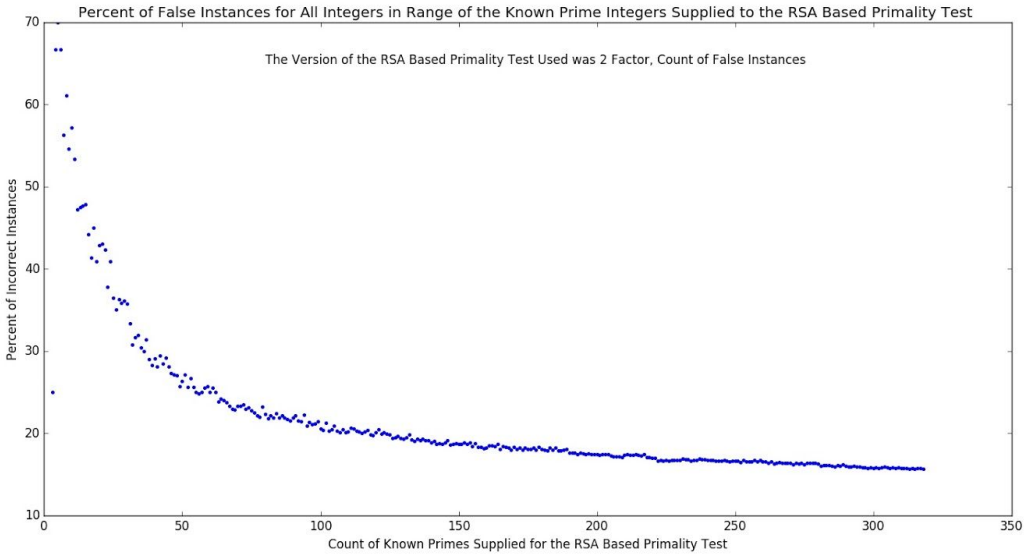
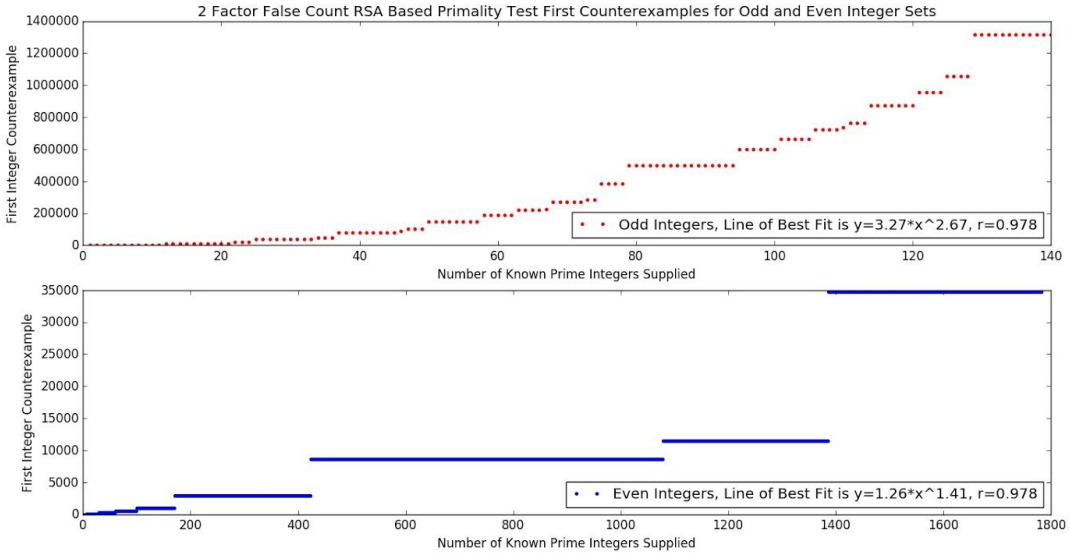


Figure 9 shows the curve that describes where the first counterexamples occur for each increment of known prime integers supplied to the RSA Based Primality Test 2 factor false count version, for both even and odd integers.

Figure 9



Figures 10, 11, and 12, are CPU computation time benchmark tests for a variety of differing bitlength prime integers.

Figure 10

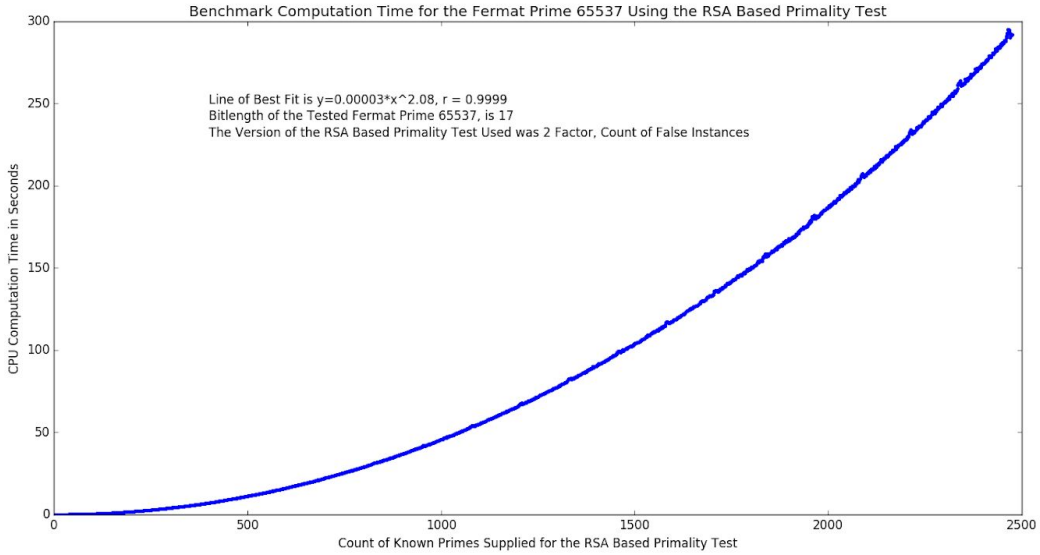


Figure 11

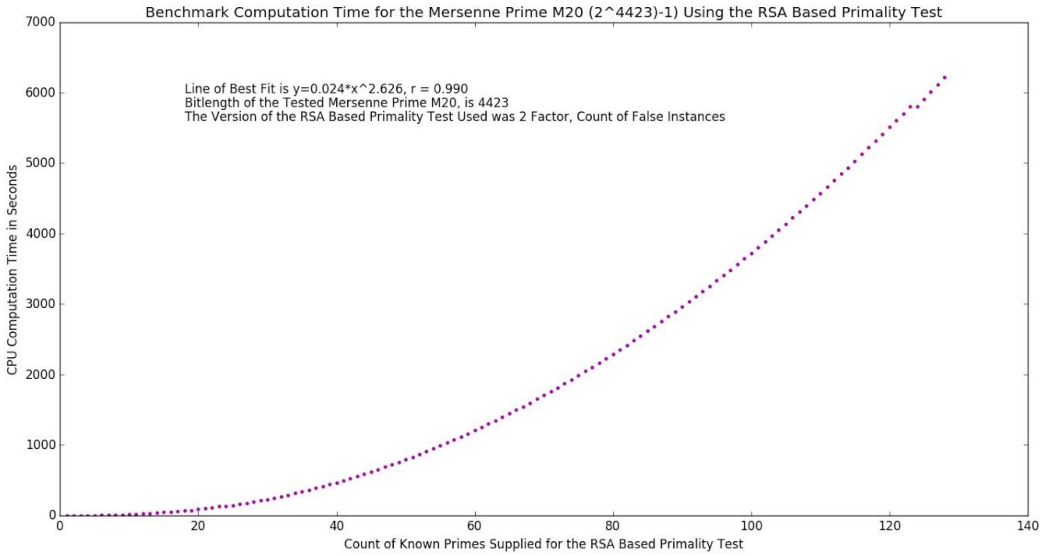
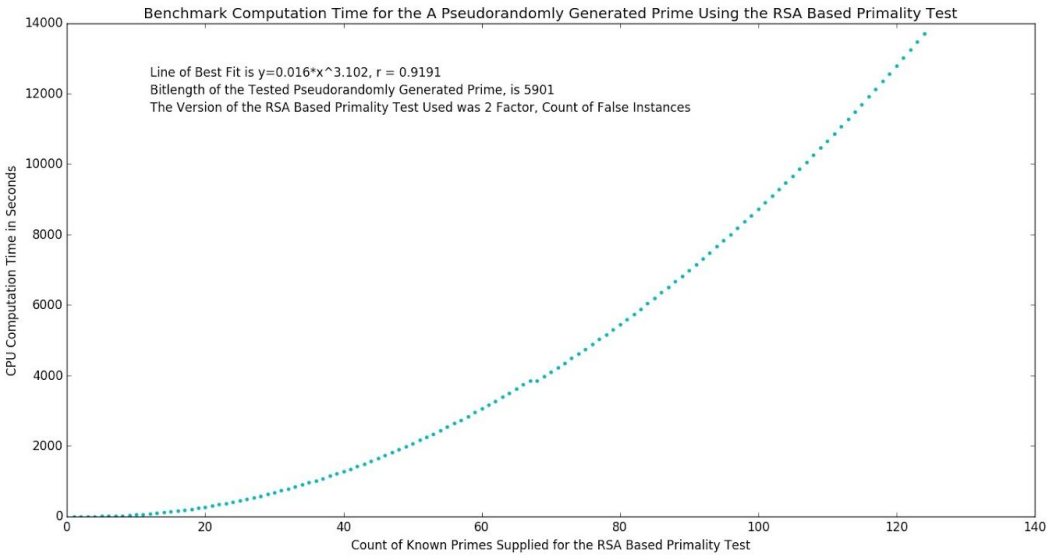


Figure 12



Analysis

Of the 4 types of the RSA Based Primality test, true and false count for 3 factor RSA (Figures 5 and 6), as well as true count of 2 factor (Figure 4) did not show clear differentiation of the bounds (minimums and maximums of the tested data set) for all numbers extending past the tested integer range, which in this case was 200-10000. However, the RSA Based Primality test for 2 factor false count (Figure 3) showed a clear differentiation between composite and prime integers (in this case using the known prime integer set between 2 and 199), making it a highly likely candidate for a primality test. It should be noted that the actual x value for these 4 tests is 46 - i.e. there

are 46 prime integers inclusively between 2 and 199.

The CPU computation time testing for primality was graphed for 3 different bitlength primes. The line of best fit and correlation coefficients for all datasets were very similar in their power lines of best fit, with consistently high correlation coefficients. The number of iterations for the RSA Based Primality Test, also referred to as the number of known prime integers supplied to the test, is limited for primality testing of a given integer N by the count of of prime integers that exist between 2 and N-1.

The first counterexample for false positives is described by the line y = 3.3*x^2.7 with a correlation coefficient of 0.97 for odd integers. The line y = 1.3*x^1.4 describes the same conditions described above for even integers with a correlation coefficient of 0.97.

The percentage of false positives from the RSA Based Primality Test decrease dramatically for increased number of known prime integers supplied to the RSA Based Primality Test. However, for larger bitlength prime integers (Bitlength(N) > 1024), the computation time increases as well, making the RSA Based Primality test usable as both a probabilistic and deterministic primality test depending on how much computation time is available. It should be noted that integers within the range of known primes used in the RSA Based Primality Test for the percentage of false positives graph were not tested, at least for Figure 7.

What is significant about figure 8 is that it shows that the test can be used as a low accuracy probabilistic primality test within the range of used primes, and therefore it seems infeasible to use the number of repetitions required to make the test definitive

within that integer range. Additionally, since the test is designed to use only the first N primes, using the RSA Based Primality test in this manner is simply useless - only large bitlength (Bitlength(N) > 1024) prime integers are useful for public key cryptography. An interesting aspect of this particular graph is that it showed that within the pool of used primes, a 2 factor false count RSA Based Primality Test outputs largely false positives, but also occasionally false negatives, which is not the case for integers outside of that set of known primes used in the test itself.

Based on previous RSA Based Primality Test versions and tests, it is possible to use differing pool of primes, besides just the primes that exist between 2 and N. However, this technique is not as efficient, nor does it have the same properties as the tested version used in all of the benchmark tests.

Conclusion

This implementation of a primality test based on the RSA cryptosystem is unique. The versatility of the RSA Based Primality test as being both probabilistic and deterministic based upon the input of known primes used in the code. The accurate primality testing of large integers is critical for accurate encryption and subsequent decryption, as well as secure asymmetric cryptosystems, whose applications include identity verification, and secure message transmission. Two factor RSA Based Primality Test with count of false instances is the most accurate form of the RSA Based Primality test, although the other forms (multiprime RSA and count of true instances) can be used as well, although with a lesser degree of accuracy.

Future experimentation for this project will include coding the RSA Based Primality Test in C++, specifically Nvidia's CUDA, which runs the code on GPUs rather than CPUs, and is consequently much faster. Then additional benchmark computation time tests with large bitlength primes will need to be conducted with this updated version of the code, given that computation time is the only factor that would be affected by coding the test in CUDA.

References

[1] A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. https://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] An Introduction to the Theory of Numbers. http://www.fuchs-braun.com/media/532896481f9c1c47ffff8077ffffffff0.pdf

[3] Asymmetric cryptography and practical security. David Pointcheval. https://pdfs.semanticscholar.org/f086/fc07648dd8277b0e14702718d3a86c8dc106.pdf

[4] Diffie-Hellman: Key Exchange and Public Key Cryptosystems. Sivanagaswathi Kallam. cs.indstate.edu/~skallam/doc.pdf

[5] Eulers Theorem. http://sites.millersville.edu/bikenaga/number-theory/euler/euler.pdf

[6] Great Internet Mersenne Prime Search. https://www.mersenne.org/

[7] "Primality Test". Wolfram Research inc. http://mathworld.wolfram.com/PrimalityTest.html

[8] Primality Randomization. Shubham Sahai Srivastava.

https://www.cse.iitk.ac.in/users/ssahai/talks/primality_randomization.pdf

[9] Primes in P. https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf

[10] Python Implementation of Miller Rabin Primality Test.

https://gist.github.com/Ayrx/5884790

[11] Python Implementation of Modular Inverse for RSA.

https://gist.github.com/ofaurax/6103869014c246f962ab30a513fb5b49

[12] RSA Attacks. Al Rasheed, Abdul Aziz and Fatim.

https://www.utc.edu/center-information-security-assurance/pdfs/course-paper-5600-rsa.pdf

[13] The Miller-Rabin Randomized Primality Test.

http://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf

[14] The Miller–Rabin Test.

http://www.math.uconn.edu/~kconrad/blurbs/ugradnumthy/millerrabin.pdf

[15] The Rabin-Miller Primality Test.

http://home.sandiego.edu/~dhoffoss/teaching/cryptography/10-Rabin-Miller.pdf

Acknowledgements

I would like to thank Alan Didier and the Los Alamos High School Computer Science Club for their help with computing power. I would also like to thank the wonderful online IDE, repl.it.