Corey Miner
Evan Weiler
ECE471
Ming Li
4 May 2018

Final Project Report

Abstract:

RSA and Diffie-Hellman are two staples of modern cryptography. Whether it is an online transaction or simply browsing your favorite website, you likely use both protocols without ever noticing it. This project aims to explore these two topics and attempts to recreate them. It also attempts to use existing Python libraries to make these processes much more efficient. While the size of the keys used is nowhere near what is used in actual practice, the protocols are still aptly demonstrated and the efficiency methods greatly reduced runtime.

Introduction:

We chose to create programs for breaking RSA as well as Diffie-Hellman man-in-the-middle attack. Both use the same idea of public and private keys and relate to each other. RSA is commonly used to protect Diffie-Hellman communications to prevent man-in-the-middle attacks. If we are able to break RSA and complete a man-in-the-middle attack on a Diffie-Hellman communication, we would be able to fully compromise the entire system. We know that both of these types of encryption use extremely large keys so it is unrealistic that anyone with current computing capabilities will be able to break the encryptions. However, we wanted to get a feel of where current day computing could break encryptions and how different optimizations could affect successful attack times for different key lengths. We decided to program the project in python because there are nearly unlimited resources and libraries that we could use to improve the efficiency of our code. We successfully created two programs that simulate an RSA break and a Diffie-Hellman meet in the middle attack. After these programs were created, we imported libraries and collected data on what improvements we saw. Overall, this project successfully allowed us to explore the logistics of breaking into a secure communication line, and two dynamic programs to break small key versions of both RSA, and Diffie-Hellman were created.

RSA Break (brute force):

For the first part of our project, we created a script that brute forced an RSA break and then developed strategies to make it more efficient and more viable. The script takes a user step by step showing asking them to input a certain number of bits that the two random prime numbers should be, generates the public key and then encrypts the a message using the public key. After the message is encrypted, the RSA breaking script factors the number and finds the two original primes based on the public

key. Then, the script finds the private key and is able to accurately decrypt the message.

When first creating the brute force method of breaking RSA, the script was extremely slow and could only handle prime numbers that were up to 10 or 11 bits long without taking an extremely large amount of time. We attempted to implement a few things to speed up this process. The first thing we did was implement python's multiprocessing utility. This utility allowed the code to run on multiple threads based on how many cores the computer running the program has. This increased the speed by about twice the speed it was before. As seen below, the commented code is the original brute force method of generating primes, and the non commented code shows full implementation of python generating primes with multiple threads.

```
def generatePrime(low, high):
    #primes = [i for i in range(low, high) if isPrime(i)]
    worker_pool = mp.Pool(processors)
    primes = [x for x in worker_pool.map(parallelF, range(low, high)) if x is not None]
    n = random.choice(primes)
    return n
```

Next we implemented gmpy2, a GNU GMP library that dramatically increased speed for doing arithmetic with extremely large integers. One big thing we were seeing with the code, is as soon as numbers got extremely large, it didn't matter how long it took to factor them. Simply encrypting and decrypting the message took the majority of the time. Implementing gmpy2 allowed these numbers to be calculated extremely quickly and allowed us to expand how many bits we could use for RSA breaking dramatically. Below is an example of the first brute force method, then the implementation using gmpy2's powmod functionality.

```
def encrypt(m, key):
    return m**key[0] % key[1]
```

```
def encrypt(m, key):
    return int(gmpy2.powmod(m, key[0], key[1]))
```

The next thing we implemented was another python library that increase time factoring large numbers called primefac. Below shows the original brute force method of finding the two original prime numbers, and then the implementation using the primefac library. This increased efficiency of the code and allowed us to break RSA with starting prime numbers as large as 50 or 60 bits within seconds.

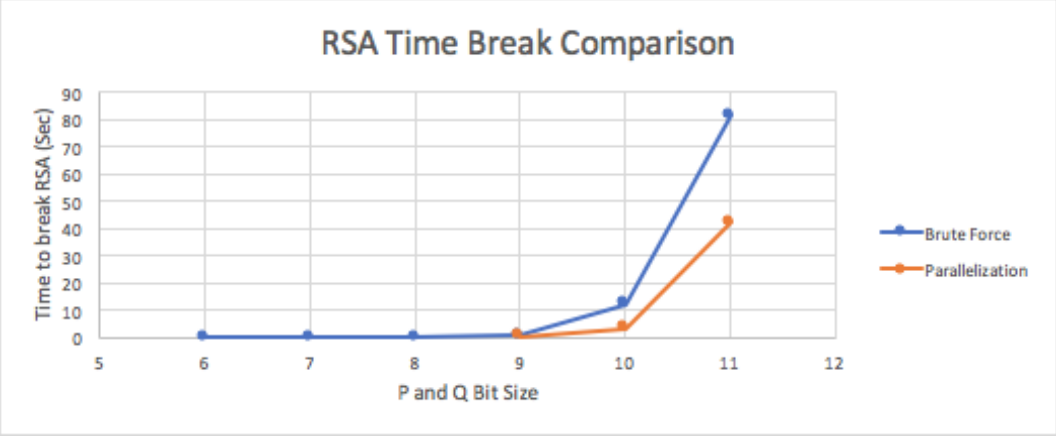
```
def findNums(pubKey):
    n = pubKey[1]
    for i in range(3,int(math.floor(n/3)+1)):
        for j in range(3, int(math.floor(n/3)+1)):
            if i * j == n and isPrime(i) and isPrime(j):
                return [i, j]
```

```
def findNums(pubKey):
    n = pubKey[1]
    nums = list(primefac.primefac(n))
    nums[0] = int(nums[0])
    nums[1] = int(nums[1])
    return nums
```

Speed of breaking RSA is the key measurement of how good an RSA breaker is. The things we did to attempt to improve efficiency showed dramatic improvements, however not nearly enough to break the real RSA 1024 bit encryption. Below are tables and graphs that illustrate this point.

Brute Force		Brute Force with parallelization	
P and Q bit length	Time to decrypt (seconds)	P and Q bit length	Time to decrypt (seconds)
6	0.000838041	9	0.417841911
7	0.009882927	10	3.433823824
8	0.160056114	11	42.34190087
9	0.970693827		
10	12.21452093		
11	81.33024311		

With all improvements			
P and Q bit length	Time to decrypt (seconds)	P and Q bit length	Time to decrypt (seconds)
15	0.000998974	35	0.746020079
16	0.002972126	40	2.678487062
17	0.00240922	45	3.398837805
20	0.006479025	50	7.367313147
25	0.004095078	55	12.70153213
30	0.071509123	60	21.10533094



As you can see, adding the GNU GMP libraries dramatically improved our RSA breaking capabilities. Without these, the simple brute force method would not have even been able to reach our original goal of breaking 16 bit keys. Overall, RSA breaking is a fairly simple process, however the key length of RSA is what keeps people from being able to break all communication encrypted with RSA. Our script was only able to efficiently break around 60 bit long primes. When the two prime numbers are much much larger, it becomes relatively impossible to break the encryption.

Diffie-Hellman:

For our second script we decided to do a demonstration of how the Diffie-Hellman key exchange protocol works. The script goes step-by-step through every aspect of the protocol, showing how each party involved ends up at the same secret key without exchanging either of their personal private keys. The script went through three iterations, the first using default Python functionality, simply following the Diffie-Hellman protocol. The second iteration implemented a Python library called "gmpy2" which has a function named "powmod()". Powmod(a, b, n) takes a to the power of b and takes the answer mod n. This function is much more efficient than the default implementation that we originally wrote. The first implementation of the power + modulo in Diffie-Hellman can be seen below:

```
115         aliceFirstMod = (g ** alicePrivate) % n
116         bobFirstMod = (g ** bobPrivate) % n
```

As you can see, the statement simply takes the public value "g" and raises it to the power of alice/bob's private keys, then takes that number mod n. This implementation is fine when dealing with small numbers, but when dealing with the massive numbers involved in Diffie-Hellman, it quickly becomes very inefficient. This is where powmod comes into play. The updated implementation can be seen below:

```
112         aliceFirstMod = gmpy2.powmod(g, alicePrivate, n)
113         bobFirstMod = gmpy2.powmod(g, bobPrivate, n)
```

Powmod takes advantage of the properties of modular arithmetic and makes this process much more efficient. This efficiency reduces the runtimes in some situations by factors of over 100. Efficiency numbers will be discussed later in this section.

The second area of efficiency that we implemented was in the area of prime number generation. This is used when determining the public values g and n for any given key exchange. At the beginning of the script, we generate a value for g (a small

prime number, so on the range of 2 - 19) and a value for n (a large prime number, so on the range of 1000000 to 2147483. The first implementation can be seen below:

```
48          g = generatePrime(2, 20)
49          n = generatePrime(2, 21474836)
```

```
32      def generatePrime(low, high):
33          primes = [i for i in range(low, high) if isPrime(i)]
34          n = random.choice(primes)
35
36          return n
```

This implementation simply iterates across a range from low to high and if a given number in that range is prime, it adds it to a list. A random entry is then taken from that list. This is inefficient as it requires to iterate across the entire range of numbers. The updated implementation takes advantage of a function from, the library “primefac” called “nextprime()”. This updated implementation can be seen below:

```
51          g = generatePrimeBetter(2, 20)
52          n = generatePrimeBetter(1000000, 2147483)
```

```
21      def generatePrimeBetter(low, high):
22          primeList = []
23
24          while(primefac.nextprime(low) < high):
25              primeList.append(primefac.nextprime(low))
26              low = primefac.nextprime(low)
27              print(low)
28
29          return random.choice(primeList)
```

Nextprime(n) does exactly what its name would suggest, it returns the next prime after a number n. This speeds up the process of generating a list of prime numbers immensely. The general principle of the function remains the same as the first implementation. Similar to powmod, efficiency numbers will be discussed later in this section.

The general flow of the Diffie-Hellman code is, like mentioned earlier, somewhat of a presentation rather than a straight calculation. The program first generates the values for g and n, then it introduces the program to the user, as seen below:

```
def main():
    cls()

    startTime = time.time()

    print("Generating public values g and n...")

    g = generatePrimeBetter(2, 20)
    n = generatePrimeBetter(1000000, 2147483)

    primeFacTime = time.time() - startTime

    g = 19
    n = 2147483647

    print("\nWe will now demonstrate how the Diffie-Hellman key exchange works")
    time.sleep(2)
    print("We will simulate two users, Alice and Bob who the user will control")
    time.sleep(2)
    print("The two public values will be g = {} and n = {}\n").format(g, n)
    time.sleep(2)
```

The program then asks the user for a value between 1 and n for each of Alice and Bob's private keys (with checks to ensure the value is as stated):

```
alicePrivate = raw_input("Please select the private key for Alice (between 1 and n): ") # must be between 1 and n
while wait:
    if alicePrivate.isdigit():
        try:
            alicePrivate = int(alicePrivate)
        except ValueError:
            pass

    if isinstance(alicePrivate, int) and alicePrivate < n:
        wait = False
    elif isinstance(alicePrivate, int) and alicePrivate >= n:
        alicePrivate = raw_input("Alice's private key must be less than the public value n, please try again: ")
    else:
        alicePrivate = raw_input("That is not a valid value (integer), please try again: ")

wait = True

bobPrivate = raw_input("Please select the private key for Bob (between 1 and n): ") # must be between 1 and n
while wait:
    if bobPrivate.isdigit():
        try:
            bobPrivate = int(bobPrivate)
        except ValueError:
            pass

    if isinstance(bobPrivate, int) and bobPrivate < n:
        wait = False
    elif isinstance(bobPrivate, int) and bobPrivate >= n:
        bobPrivate = raw_input("Bob's private key must be less than the public value n, please try again: ")
    else:
        bobPrivate = raw_input("That is not a valid value (integer), please try again: ")
```

After this the program then narrates through the values that Alice and Bob obtain:

```
print("\nAlice's private key is {} and Bob's private key is {}.").format(alicePrivate, bobPrivate)
time.sleep(2)
print("Alice and Bob will now both perform g^key(mod n)")
time.sleep(2)
print("i.e. Alice and Bob will raise g to the power of their key, all mod n")
time.sleep(3)

aliceFirstMod = gmpy2.powmod(g, alicePrivate, n)
bobFirstMod = gmpy2.powmod(g, bobPrivate, n)

print("\nThe value that Alice obtains is {}").format(int(aliceFirstMod))
time.sleep(2)
print("The value that Bob obtains is {}").format(int(bobFirstMod))
time.sleep(2)

print("\nAlice will now send her value to Bob, and Bob will send his number to Alice")
time.sleep(2)
print("Alice and Bob will then raise the values they receive to their private key, all mod n")
time.sleep(3)

aliceSecondMod = gmpy2.powmod(bobFirstMod, alicePrivate, n)
bobSecondMod = gmpy2.powmod(aliceFirstMod, bobPrivate, n)
```

These values are then outputted and further narration occurs, total runtime and time to generate the public values are also calculated and outputted:

```
print("\nThe value that Alice has now obtained is {}").format(aliceSecondMod)
time.sleep(2)
print("The value that Bob has now obtained is {}").format(bobSecondMod)
time.sleep(3)

print("\nAs you can see, the numbers that both Alice and Bob received are the same.")
time.sleep(2)
print("Their private keys were never transmitted over any medium, and they both have the same key.")
time.sleep(2)
print("This is the power of Diffie-Hellman")
time.sleep(2)

totalTime = (time.time() - timeAfterInput) + timeBeforeInput

print("\nRuntime: {} seconds").format(totalTime - 33)
print("PrimeFac time: {} seconds").format(primeFacTime)
```

Efficiency is a major aspect of using Diffie-Hellman. Obviously, we never achieved the high values that Diffie-Hellman usually obtains (up to 2000, sometimes 4000 bits). But, we can still demonstrate the power of efficiency using our code. Shown below is the excel table for increasing values of Alice's private key (Bob's private key is kept constant at 1 for simplicity):

g	n	bpriv	apriv	runtime(default)	runtime(gmpy2)
2	2147483647	1	65535	0.00400018692017	0.00200009346008
2	2147483647	1	131071	0.00600004196167	0.00300002098083
2	2147483647	1	262143	0.00999999046326	0.00200009346008
2	2147483647	1	524287	0.01799988746640	0.00400018692017
2	2147483647	1	1048575	0.03499984741210	0.00199985504150
2	2147483647	1	2097151	0.07300019264220	0.00200009346008
2	2147483647	1	4194303	0.14699983596800	0.00200009346008
2	2147483647	1	8388607	0.29800009727500	0.00300002098083
2	2147483647	1	16777215	0.66000008583100	0.00300025939941
2	2147483647	1	35554431	1.41400003433000	0.00199985504150
2	2147483647	1	67108863	2.93099975586000	0.00300002098083
2	2147483647	1	134217727	6.14299988747000	0.00200009346008
2	2147483647	1	268435455	12.67400026320000	0.00300002098083
2	2147483647	1	536870911	26.85099983220000	0.00100016593933
2	2147483647	1	1073741823	56.41699981690000	0.00499987602234
2	2147483647	1	2147483646	115.08399987200000	0.00300002098083

As you can see, gmpy2 greatly improves runtime, especially when it comes to much larger numbers of Alice's private key. Below are the values for the two different prime generation functions:

	primeFacTime
default	258.2229998
primefac	4.600999832
	*range: 2:21475836
primefac	48.71000004
	*range: 2:214758364

As you can see again, the primefac library greatly reduces runtime, further increasing the efficiency. Overall, the Diffie-Hellman process by itself is not very complicated, but making the process more efficient is the hardest part and something we definitely experienced when writing this code.

Appendix:

In order to run code, one must have pip and python 2 installed. Cd into the directory with the two python scripts in it, and run the following commands:

- Pip install -U pip
- Pip install primefac
- Pip install gmpy

Then the user can run either the Diffie-Hellman or RSA script by running

- Python "file name"