Stephen Daniel
sdani17

Project 2 Crypto Systems

Problem 1:

RSA System Implementation

For this RSA implementation, I chose to use the Python programming language. It was implemented in an object-oriented manner. The program is implemented entirely in the file `RSA.py`, including the demonstrations. I used no external libraries besides the basic Python standard libraries `time` and `math`. All the algorithms and data structures were hand-rolled by me.

For representing the message to be encrypted and decrypted, it is implemented as an object of class RSA, with a message field implemented as an array of int's (mapped from the characters of the input message), and a boolean flag to indicate whether the message is encrypted. Encryption and decryption operations are implemented as methods on these objects, and the public and private keys, respectively, are passed in as arguments.

The key generation function is implemented as a static function in the class RSA, taking two arguments: min and max, representing the minimum and maximum prime number, respectively, to use for p and q, and returns a dictionary containing two key-value pairs: `{'pub':[e, n], 'priv':[d, p, q]}`. These arrays are passed as a single argument to the encryption and decryption functions.

If all this seems a bit convoluted, I'll illustrate with an example. If we initialize our message and keys with the following code:

```
my_message = RSA("rsa")
keys = RSA.generate_keypair(1000, 10000)
```

The keys variable would then contain the dictionary

```
{'pub': [7, 1185137], 'priv': [506983, 1061, 1117]}
```

We could then use the following code to encrypt the message:

```
my_message.encrypt(keys['pub'])
```

To get the message from the RSA object, a method is implemented `get_message()` which maps the ints in the message array back to chars and concatenates these chars into a string which is then returned.

If the message is encrypted (i.e. the encrypted flag is True), the int's will be encoded as very large values and will not map to characters, even if we tried, so I added a check to see if the encrypted flag is set, and if it is the method will return the string "gobbledygook". Returning to our example code: we could execute the following code to decrypt our message and print it out:

```
my_message.decrypt(keys['priv'])
print(my_message.get_message())
```

Stephen Daniel
sdani17

Demonstration:
The full output of the program, including a demonstration of encryption and decryption is included in the file `rsa_output.txt`.

Helper functions:
The RSA class also implements several static helper functions to implement the RSA algorithm and to crack the algorithm: `is_prime()`, `prime_range()`, `factorize()`, `find_e()`, `find_d()`, `generate_keypair()`. These are left as separate static functions so they can be easily called from outside the class during the demonstration of cracking the RSA algorithm, which brings me to my next section:

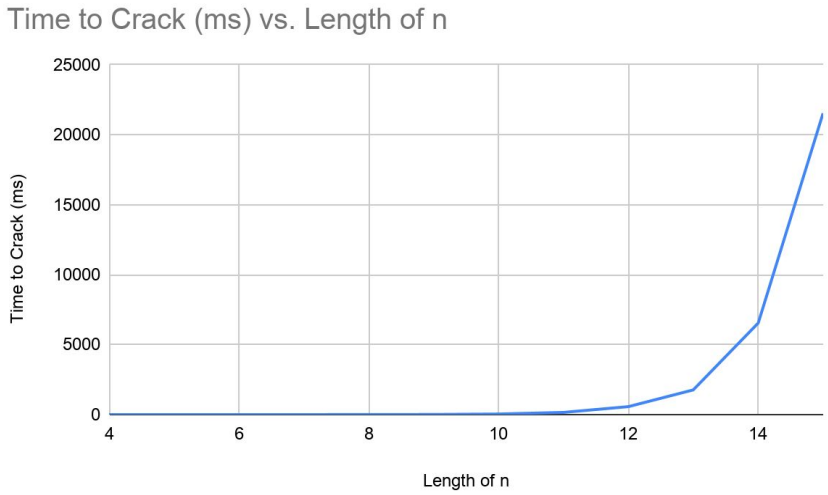
Time complexity of breaking RSA
Hand-rolling the RSA algorithm was instrumental in allowing me to investigate the time complexity, especially having access to all the helper functions. The hard part about breaking RSA, and the fact that it derives its security from, is the time complexity of factoring n, the product of two large primes, to find the factors, which then would enable us to trivially calculate d, the private key. To this end, I designed a helper method within RSA, `factorize(n)` that also times itself and returns that value at the end of an array of factors, and to build on that, a short routine that runs the `factorize()` function and the `find_d()` function on different sizes of n. The relevant snippet of code, and a table of the running times are included below for convenience:

```
# routine to test the time complexity of factoring the public key
print("***Demonstrating time complexity to break RSA***")
print("Generating list of primes...")
primes = RSA.prime_range(1, 100000000)
for i in range(1, 13):
    a = primes[int(10 ** (i / 2)) + 12] * primes[int(10 ** ((i + 1) / 2))]
    factors = RSA.factorize(a)
    print("\nn: {0}\nKey length: {1}".format(
          a,
          len(str(a))))
    print("Factors of n (p and q):", factors[:-1])
    totient = (factors[0] - 1) * (factors[1] - 1)
    print("Totient value for RSA algorithm: {0}".format(totient))
    start = time.time_ns()
    e = RSA.find_e(totient)
    d = RSA.find_d(totient, e)
    end = time.time_ns()
    time_taken = factors[2] + ((end - start) / 10 ** 6)
    print("d:", d)
    print("Total time taken to factorize n and calculate d: "
          "{0:.2f}ms".format(time_taken))
```

Stephen Daniel
sdani17

CSC 4700 Project 2
11/24/2019

Length of n	Time to Crack (ms)
4	0.09
5	0.06
6	0.20
7	0.66
8	2.44
9	9.47
10	41.76
11	156.61
12	569.11
13	1763.90
14	6539.24
15	21539.15



As can be seen from the table and chart, for every digit of n, the time to break it increases by roughly a factor of 4. It would quickly become infeasible to crack with n's of even only 30 or 40 digits, and production-quality RSA uses keys with over 100 decimal digits. Although I only used a naive factoring algorithm, even the more sophisticated algorithms are not fast enough to break production-quality RSA in a reasonable amount of time.

Problem 2:

For this crypto puzzle problem I also used Python 3. I used one non-trivial library: `hashlib`. Since this program is much shorter than the RSA implementation, I've included it here for convenience:

```
import hashlib
import string
import random

NUM_TRIALS = 25
# using hex digits throughout code instead of binary, for simplicity
# this is the puzzle P, just take different size slices of the same hex value for
# different sizes of B
B_BASE = '7d1a'
ALPHANUMERIC = string.ascii_lowercase + string.ascii_uppercase + string.digits
STR_LEN = 32

print("Using {0} trials for each B".format(NUM_TRIALS))

for b in [1, 2, 3, 4]:
    # Alice creates her puzzle
    B = B_BASE[:b]
    stats = []

    # average number of tries over many trials and store the stats in an array for analysis
    for i in range(NUM_TRIALS):
        tries = 0
        digest = ''
        # Bob tries to find a collision for the last (b * 4) bits
        # while the last b digits of the digest are not equal to B
        while digest[-b:] != B:
            tries += 1
            # create a new sha256 hash object, generate a random string, hash the string,
            # and get the digest in hexadecimal
            hash = hashlib.new('sha256')
            rand_string = ''.join(random.choice(ALPHANUMERIC) for i in range(STR_LEN))
            hash.update(rand_string.encode('utf-8'))
            digest = hash.digest().hex()

        stats.append(tries)
    print("Stats for B={0}: min:{1}  max:{2}  avg:{3}".format(
           b * 4, min(stats), max(stats), sum(stats) / len(stats)))
```

Stephen Daniel
sdani17

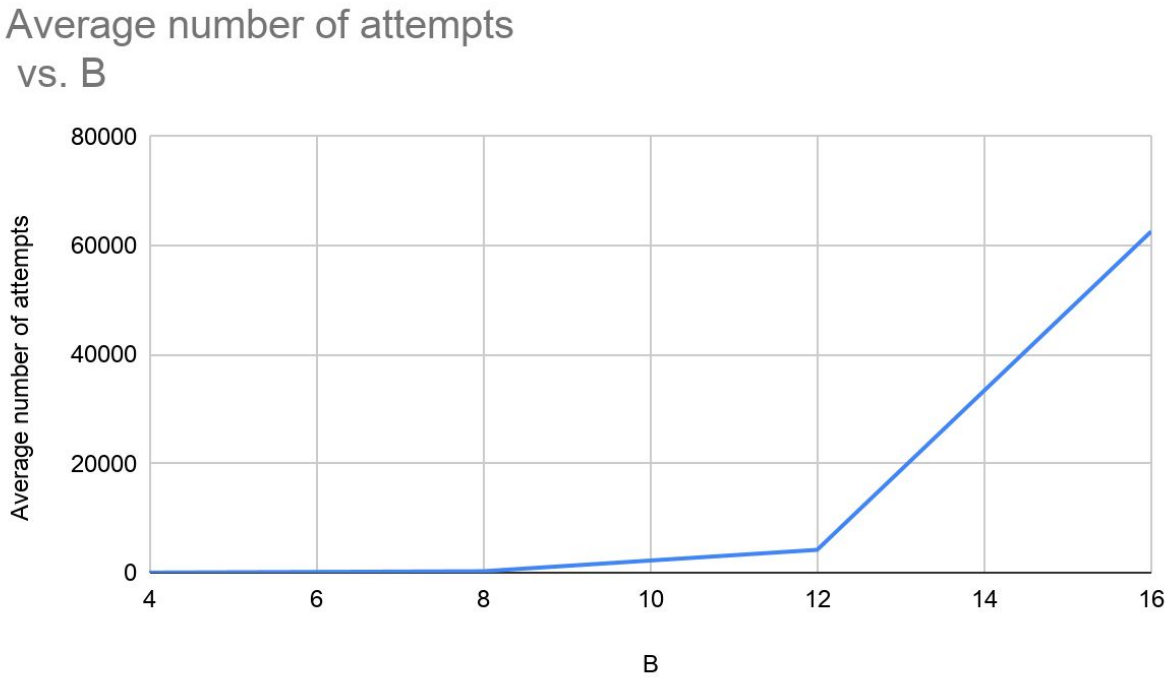
CSC 4700 Project 2
11/24/2019

Design

For this program, I implemented it as three nested loops.
Something worth mentioning here is that for simplicity, I operated on B and the hash digest in hexadecimal instead of bits (which would have required an extra library and/or lots of extra code). The end result is the same, I test the algorithm with B ranging from size 4 bits to 16 bits, I just do it in hexadecimal terms. That is why the outer loop iterates over [1, 2, 3, 4] instead of [4, 8, 12, 16].
The innermost loop is a while loop that iterates until the last b digits of the hash digest are equal to B, and counts the number of tries it takes.
The second nested loop solves the crypto puzzle a certain number of times (NUM_TRIALS) and appends the number of tries required to solve it in the inner loop to an array ‘stats’.
The outer loop iterates over different sizes of B and calculates and prints out the statistics from the inner loop.
The other aspects of the program are explained inline, in the comments.

Results:

```
Using 25 trials for each B
Stats for B=4: min:1  max:40  avg:11.24
Stats for B=8: min:3  max:774  avg:310.92
Stats for B=12: min:15  max:11649  avg:3959.96
Stats for B=16: min:3376  max:240247  avg:69319.32
```



As can be seen from this chart and the program output, for every additional 4 bits, the average time to crack the puzzle increases approximately 16 fold. This is fairly intuitive, since the output of the hash function is essentially random (for different inputs) as far as we're concerned. Therefore, to crack n bits of the hash, we're essentially rolling a 2^n sided die, and basic statistics tells us for m equally weighted possible random values, on average it will take m tries to get a specific value. Extrapolating this data, to crack all 256 bits of the hash, it would take on average 2^{256} tries, or about 10^{77}.