# Project 3:
# CS6035

Zhaoran Yang
zyang625@gatech.edu

## 1 TASK 2

Even though hashing is seen as a form of one-way encryption as it's almost impossible to calculate the original string backwards, hackers can still decode the hash, either through brute-force or dictionary attacks, or by leveraging public resources where the most commonly used password hashes are stored, for example, rainbow table.

Salt values are a short random set of characters added to the end of a password before hashing, and salt values are stored along with the hashed password so that the system knows which salt value to use for verification. Though salts are useful for protecting passwords from rainbow tables, it's still vulnerable to brute-force and dictionary attacks.

Peppers can protect passwords further from brute-force attacks, by adding a single character (e.g., an upper- or lower-case alphabet letter) to the end of the password before hashing.[1] When trying to log in, the system iterates through every possible combination of password + pepper, hashes and compares it to stored hash – if there's a match, the user is allowed to login. Unlike salt, peppers are not stored, and it would take 52 times longer than without pepper for brute force attacks.

## 2 TASK 3

There's an alternative consensus mechanism called proof-of-stake (POS). POS uses a randomized process to determine who gets a chance to produce the next block.[2] Miners pledge their stake for a certain period to validate transactions. The more coins one pledges and the longer one owns the coins, the better odds one can create a new block.[2]

**Strength:**

1. **Less energy demand**: POW models verify transactions through trial and error and thus is a competitive mechanism that requires a significant amount of computing power and energy.[3] POS is much more energy efficient comparatively.
2. **More scalable**: POS mechanism puts higher weight on the number of coins one owns during the mining process, which incentivizes miners for maintaining the blockchain network. [3]
3. **More secure**: In POW network, due to lack of incentive above, a miner's goal is to profit, and thus attackers can fork in the blockchain, make alternative versions of blockchain that seem valid to earn coins, which disrupts the network**.** [3]

**Weakness:**

There's no obvious weakness compared to POW now. Even though POW is the most well-known model, POS is gaining popularity due to its ecological impacts mentioned above.

## 3 TASK 4

I will first check if the transaction string is in its proper format 'User1:User2:X' and convert the transaction string into an integer to get the integer value of the message m. Then I will use the get_factors method and get_private_key_from_p_q_e method to calculate the private key d.

After that, I will use the pow() method, pass m and both private and public keys to create the corresponding signature. The last step is to compare the derived signature to the given signature.

## 4 TASK 5

**Step 1**: I started with the get_factors method. I first ran my simple implementation of the method, which is finding the prime numbers p and q for n = p*q as p increases from 2 to sqrt(n). But it runs too slow. Therefore, I searched for more efficient algorithms - I found Pollard's rho algorithm particularly suitable for numbers with small factors, since we are given a public key with a small key size of 64 bits. So I implemented below pseudo code:

```
x ← 2
y ← 2
d ← 1

while d = 1:
    x ← g(x)
    y ← g(g(y))
    d ← gcd(|x - y|, n)

if d = n:
    return failure
else:
    return d
```

[4]

where g(x) = (x^2 +1) (mod n).

For a given composite n = p*q, where p and q are non-trivial prime factors. If we generate a sequence of x, starting x = 2, for x1 = g(x), x2 = g(g(x)), x3 = g(g(g(x))), where g(x) = (x^2 +1) mod(n).[4] It's going to collide with another sequence {xk mod(p)}. The d above is to check if such collision exists, and once we get a d that is other than 1 or n, it's our p. And q can be solved by q = n / p.

The brute-force method I implemented in the beginning, whose time complexity is O(sqrt(N)). For this algorithm, if x = g(x) occurring in the algorithm is an actual number, we can achieve half of the time in $O(\sqrt{p}) <= O(n^{1/4})$ iterations. [4]

**Step 2**: After solving the factorization problem, I started solving for the modular inverse for e^(-1) mod(φ(N)). Since pow(e, -1, φ(N)) is only supported by Python

3.8+, I searched for algorithms to get modular inverse and found Extended Euclidean Algorithm. I implemented the recursive algorithm using below pseudo code for get_private_key_from_p_q_e.

```
ExtEuclid (a,b) {
   // returns a triple (d,s,t) such that d = gcd(a,b) and
   // d == a*s + b*t

   if (b == 0) return (a,1,0) ;

   (d1, s1, t1) = ExtEuclid(b,a%b) ;
   d = d1 ;
   s = t1 ;
   t = s1 - (a div b) * t1 ;   // note: div = integer division
   return (d,s,t) ;
}
```

[5]

Euclidean Algorithm is an algorithm that finds the greatest common divisor (gcd) of two integers. The greater number repeatedly divides the divisor by the previous remainder until the remainder is 0.[6] By reversing the process of the Euclidean Algorithm, we can find integers p and s where p(a)+s(b) = gcd (a, b) = 1.[6]

## 5 TASK 6

- The vulnerability we discovered in this task is related to factorable RSA keys.

  In RSA, it takes an incredibly long time to factor a large key, e.g., 1024 bit, but it only takes microseconds to find the gcd of two 1024-bit integers.[7]

  This makes the key vulnerable in a way that if a large public key n is distinct, but it shares exactly one prime factor with other large public keys generated by the same random number generator (RNG), it's easy to obtain its private key by calculating the gcd between itself and the list of the public keys.

  This vulnerability might be introduced by insufficient randomness provided by the Linux RNG due to weak entropy sources, particularly the entropy pools, when the system is powered off long enough.[7] On the

first boot, during a window of vulnerability, aka a boot-time entropy hole, the Linux's /dev/urandom may be predictable.[7] Since entropy is extracted from the processes that read urandom, the entropy becomes predictable.

- To calculate the private key, I first need to extract p and q from n provided. I first passed it into the get_factors function I implemented for Task 5 but it takes too long. Then according to the vulnerability illustrated above, if I can find a common primary factor between n and any of the numbers in the public key list (p1, p2, …. pk) provided, I can also get p by calculating gcd between each pair (n, p1), (n, p2) …, (n, pk). Inspired by section 3.3, instead of calculating for each pair, I used the product tree method to expedite the search. Below are the specific steps to implement Task 6.

  **Step 1**: calculate the product of all items in public key list:

  P = p1*p2*…*pk

  **Step 2**: find gcd (n, P)

  **Step 3**: if gcd (n, P)! = 1, it means there's a prime factor that n shares with P, we can use it as our p, so our q = n / p.

  **Step 4**: pass p, q and e into get_private_key_from_p_q_e to derive private key d.

## 6 TASK 7

Broadcast RSA attack is often seen in the cases when the public exponent e is small or when partial knowledge of a prime factor of the secret key is available.[8] It's also known as Coppersmith's attack as it uses Coppersmith Theorem to derive the public key N.

When a message M is encrypted using a small public exponent e and the message M is sent to multiple recipients using different public keys, it's feasible to

compute M from M^e using the Chinese Remainder Theorem. In our case, all the cipher integers are c_1, c_2 and c_3, with the corresponding public keys n_1, n_2 and n_3 and the public exponent e = 3.

The application of Chinese Remainder Theorem in our case works like this: The theorem states that for an integer A, if we have the information of the remainders of the Euclidean division by several pairwise coprime integers, we can derive the remainder C of the division of A by the product of these remainders.[9] Then we can solve the message M using C. The mathematical steps are as follows:

- Given Ci ≡ M^3 (mod Ni) where for all i, j that gcd (Ni, Nj) = 1.
- From Ci to Cj, we can derive C such that C ≡ Ci (mod Ni) and C = M^3.
- Then we will have the message M as M = sqrt(C, 1/3).

The detailed solution steps for this task are as below:

1. I first calculate the product of all dividers for A as:

   N = n_1 * n_2 * n_3

2. Then for each i = 1 to 3, compute y_i = N / n_i
3. For each i = 1 to 3, calculate remainders via previously implemented egcd method to obtain remainders of Euclidean division for each

   zi ≡ y_i^(-1) (mod n_i)[9]

4. Make sure for all i, j, gcd(Ni, Nj) = 1, then calculate sum of c_i * y_i * z_i to x and obtain M^3 ≡ x (mod N)
5. Solve M via a self-implemented method get_cubic_root to retrieve M. I implemented the method by myself because the python method pow() returns a float and the number is beyond the float limit and thus fails to serve our purpose in this task. My implementation assumes that the solution m is an integer using binary search.

## 7 REFERENCES


1. Wikipedia, W. (2021, Nov 4). Pepper (cryptography). Retrieved March 20, 2022, from https://en.wikipedia.org/wiki/Pepper_(cryptography)

2. Naveen Joshi (2019, Apr 23). 8 blockchain consensus mechanisms you should know about. Retrieved March 20, 2022, from https://www.allerin.com/blog/8-blockchain-consensus-mechanisms-you-should-know-about

3. Amanda Reaume (2022, Feb 3). Proof Of Work Vs. Proof of Stake: Explained. Retrieved March 20, 2022, from https://seekingalpha.com/article/4468656-proof-of-work-vs-proof-of stake?external=true&gclid=Cj0KCQjwz7uRBhDRARIsAFqjulnAFwH7E2qoX8WnnrdL4F8diwOYPLDzlkEYt-yAETAtK4FGakjl1iQaAot-EALw_wcB&utm_campaign=14049528666&utm_medium=cpc&utm_source=google&utm_term=127999857095%5Edsa-1427141793820%5E%5E547854587978%5E%5E%5Eg

4. Wikipedia, W. (2022, Mar 9). Pollard's rho algorithm. Retrieved March 20, 2022, from https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

5. Richard Chang, University of Maryland Baltimore County (2009, Mar 3). The Extended Euclidean Algorithm. Retrieved March 20, 2022, from https://www.csee.umbc.edu/~chang/cs203.s09/exteuclid.shtml

6. University of Colorado Denver. The Extended Euclidean Algorithm. Retrieved March 20, 2022, from http://www-math.ucdenver.edu/~wcherowi/courses/m5410/exeucalg.html

7. Heninger, N., Durumeric, Z., Wustrow, E., & Halderman, A. J. (2012, Jul 11). Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices.

8. Wikipedia, W. (2022, Feb 15). Coppersmith's attack. Retrieved March 20, 2022, from https://en.wikipedia.org/wiki/Coppersmith%27s_attack

9. Brilliant.org. Chinese Remainder Theorem. Retrieved March 20, 2022, from https://brilliant.org/wiki/chinese-remainder-theorem/