COMPSCI 4CR3 Assignment 4

Dev Mody

November 2024

Contents

1 Question 1

Given an element $g \in \mathbb{Z}_p^\times$ and the prime factorization $p-1 = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, the following algorithm determines whether g is a generator or not.

for $i = 1$ **to** k **do**

    if $g^{(p-1)/p_i} /= 1 \mod p$**, return False**

return True

1.1 a)

Show that the algorithm is correct; that is, it returns True if and only if g is a generator.

The problem concerns the group $\mathbb{Z}_p^\times$, the multiplicative group of integers modulo p, where p is a prime number. By definition, $\mathbb{Z}_p^\times$ consists of all integers g such that $1 \leq g < p$ and $gcd(g, p) = 1$. The group $\mathbb{Z}_p^\times$ has order $p-1$, as there are $p-1$ integers less than p that are co-prime to p. This group is cyclic as there exists a generator g such that every element of $\mathbb{Z}_p^\times$ can be expressed as $g^k \mod p$ for some k. In addition, $\mathbb{Z}_p^\times$ is a finite cyclic group since it is cyclic and there are a finite number of elements. To elaborate, the generator of $\mathbb{Z}_p^\times$ is an element with order $p-1$, where the order of an element g in a group is the smallest positive integer d such that $g^d \equiv 1 \mod p$. Lagrange's Theorem states that the order of any element of a finite cyclic group must divide the group's order, so any element in $\mathbb{Z}_p^\times$ must have order dividing $p-1$.

The algorithm in question tests whether $g \in \mathbb{Z}_p^\times$ is a generator by evaluating $g^{(p-1)/p_i} \mod p$ for each prime factor p_i of $p-1$. If $g^{(p-1)/p_i} \equiv 1 \mod p$ for any p_i, g is not a generator. If this condition does not hold for all p_i, then g is a generator. To formally prove the algorithm is correct, we must show that `The algorithm returns True` $\equiv$ g `is a generator`. This can be broken down into 2 cases which are:

1. g is a generator $\implies$ The algorithm returns True
2. The algorithm returns True $\implies$ g is a generator

Case 1: g is a generator $\implies$ The algorithm returns True

To start, since g is a generator of $g \in \mathbb{Z}_p^\times$, then it's order is $p-1$. From Lagrange's Theorem, the order of any element in a group must divide the group's order, so if g is a generator of $\mathbb{Z}_p^\times$, the order of g must divide $p-1$ and no smaller proper factor of $p-1$ can satisfy this. Now consider $g^{(p-1)/p_i} \mod p$ for any prime factor p_i of $p-1$. If $g^{(p-1)/p_i} \equiv 1 \mod p$, the order of g would divide $(p-1)/p_i$, which is less than $p-1$. This contradicts the assumption that g is a generator (since the generator's order must be $p-1$). Therefore, $g^{(p-1)/p_i} \mod p \neq 1$ for all p_i, and the algorithm correctly identifies g as a generator. By our algorithm, it only returns False when one of the prime factors of $p-1$, p_i holds in $g^{(p-1)/p_i} \equiv 1 \mod p$. If this doesn't occur, then it returns true. Hence, if g is a generator, then our algorithm returns True.

Case 2: The algorithm returns True $\implies$ g is a generator

Suppose the algorithm returns True, meaning $g^{(p-1)/p_i} \not\equiv 1 \bmod p$ for all prime divisors p_i of $p-1$. For contradiction, assume g is not a generator. This means the order of g, denoted as d, is a proper divisor of $p-1$, where $1 \leq d < p-1$. By Lagrange's Theorem, d must divide $p-1$, and hence it must also divide $(p-1)/p_i$ for some p_i. This implies $g^{(p-1)/p_i} \equiv 1 \bmod p$, contradicting the algorithm's condition for returning True. Thus, g must be a generator if the algorithm returns True.

As a result, this algorithm is proven to be correct.

1.2 b)

What is the complexity of the algorithm in terms of the number of multiplications? Express your answer using Big-O notation.

To derive the time complexity of the algorithm, we must first consider the complexity of the most important instruction being $g^{(p-1)/p_i} /= 1 \mod p$. We can notice that $g^{(p-1)/p_i}$ is in the same format as binary exponentiation, $A^B \mod p$ where $A = g$ and $B = (p-1)/p_i$. We know that the time complexity of Square-And-Multiply is $O(\log B)$ for $A^B \mod p$. Relating this back to our case, since $B = (p-1)/p_i$ and $p - 1 > p_i$ for all $i \in \{1, 2, ..., k\}$, we can express B as having order $O(p)$, giving a final time complexity for $g^{(p-1)/p_i} /= 1 \mod p$ being $O(\log p)$. Next, we see that the loop iterates k times based on the number of prime factors $p-1$ has. Each time we perform the loop, it performs the above instruction to see if an element generates a subgroup of the cyclic group, which we know takes $O(\log p)$ time. As a result, the time complexity of our algorithm is $O(k \times \log p)$.

1.3 c)

What is the output for the algorithm for $p = 899009829279928687167847647253$ and $g = 425044249325748129860331117047$?

The output was False for the above inputs.

2 Question 2

Let G be a group of order 3^n, i.e., $|G| = 3^n$ and suppose $g \in G$ is a generator of G. Recall that $g^{3^n} = g^{|G|} = 1$. Due to the specific structure of G, the Discrete Logarithm Problem (DLP) is computationally easy in G.

2.1 a)

Design efficient algorithm for DLP in G. Hint: Given $g^x = h$, you can compute one "digit" of x at a time. If

$$x = 3^{n-1}x_{n-1} + 3^{n-2}x_{n-2} + \cdots + 3x_1 + x_0 \tag{1}$$

is the base-3 expansion of x, you can determine x_0 by computing $h^{3^{n-1}}$. Include the pseudo-code of your algorithm here.

Before we get into the algorithm, we must first understand the problem fully and break it down into a more simpler problem. First we know that $h = g^x = g^{3^{n-1}x_{n-1}+\cdots+3x_1+x_0}$. Given from the hint in the question, we can raise $h^{3^{n-1}}$ to compute x_0 where we would obtain the following: $h^{3^{n-1}} = (g^x)^{3^{n-1}} = g^{x\times 3^{n-1}} = g^{3^{n-1}\times(3^{n-1}x_{n-1}+\cdots+3x_1+x_0)}$. Then we can rewrite this as $h^{3^{n-1}} = g^{3^{2n-1}x_{n-1}+3^{2n-3}x_{n-2}+\cdots+3^n x_1+3^{n-1}x_0}$. We know that for every 3^k where $k \geq n$ then we know that 3^n divides 3^k. Given that $g^{3^n} = 1$, we know that $g^{3^{2n-1}x_{n-1}+3^{2n-3}x_{n-2}+\cdots+3^n x_1+3^{n-1}x_0} \equiv g^{3^{n-1}x_0} \mod 3^n$. As a result, we must solve for a value of x_0 such that the following holds: $h^{3^{n-1}} \equiv g^{3^{n-1}x_0} \mod 3^n$. Since we're working with base-3 values, the only possible values of x_0 are $0, 1, 2$. In this scenario, the algorithm goes from least significant to most significant digits and performs the above computation for each x_i.

Algorithm 1 `Compute_Exponent`(g, h, n, p)

```
x ← 0 {Initialize the result exponent}
current ← h {Current value to reduce}
for i ← 0 to n − 1 do
    power_of_three ← 3^{n−1−i} {Compute current power of 3}
    mod_value ← current^{power_of_three} mod p {Compute modular exponentiation}
    for j ← 0 to 2 do
        if g^{j·power_of_three} mod p = mod_value then
            x ← x + j · 3^{n−1−i} {Update x with the current term}
            current ← current · g^{−(j·power_of_three)} mod p {Adjust current for the next iteration}
            break
        end if
    end for
end for
return x {Return the computed exponent}
```

2.2 b)

Analyze the running time complexity of your algorithm. Express your answer using Big-O notation.

In addition to the explanation provided above, the time complexity of my algorithm can be determined by the step-by-step analysis of the algorithm. It goes as follows:

- From line 3, we can clearly see that the loop initialized runs n times.
- The loop initialized on Line 6 runs a fixed amount of three times
- A key part of the algorithm to be analyzed is the modular exponentiations we're performing. For instance, we perform exponentiations on Lines 4-5, Line 7 and Line 9. Given that exponentiations has a run-time complexity of $O(loge)$, where e is the exponent, I provide the analysis for all cases of exponentiations as follows:
  - Line 4: Since we compute power_of_three $\leftarrow 3^{n-1-i}$, this line has the complexity $O(\log(n-1-i)) \approx O(\log n)$
  - Line 5: We compute mod_value $\leftarrow$ current$^{\text{power_of_three}} \mod p$, where here, has the run-time complexity of $O(\log(3^{n-1-i})) \approx O(\log(3^n)) = O(n\log(3)) \approx O(n)$
  - Line 7: We compute $g^{j\cdot\text{power\_of\_three}} \mod p = \text{mod\_value}$ which has the complexity $O(\log(j\times 3^{n-1}))$ where $j \in \{0,1,2\}$. Thus $O(\log(j\times 3^{n-1})) \approx O(\log(3^{n-1})) \approx O(n)$
  - Line 9: We compute current $\leftarrow$ current $\cdot\, g^{-(j\cdot\text{power\_of\_three})} \mod p$ which has the complexity $O(\log(-j\times 3^{n-1})) \approx O(n)$

As a result, since we perform the outer loop (line 3) exactly n times where Line 4 has complexity $O(\log n)$, and Line 5,7,9 have complexity $O(n)$, then the entire algorithm has a complexity of $O(n\log n + n^2 + n^2 + n^2) \approx O(n^2)$.

2.3 c)

Consider the following values for p and q: $p = 2060210292175507490794709453568 7$, $q = 150746928358503196354993776985 38$. For these values, g is a generator of a subgroup $G < \mathbb{Z}_p^\times$ of order 3^{65}. That means the numbers $g^0 \mod p, ..., g^k \mod p$ where $k = 3^{65} - 1$, form a group of order 3^{65}. Use your algorithm to find the discrete logarithm $x = log_g h$ for $h = 193412779505532697608485690260 15$.

The x value I found by performing my DLP algorithm is $x = 262151933815490313462445448196 0$.

3 Question 3

To prevent the existential forgery attack, we discussed in class how the RSA signature scheme uses a padding algorithm called EMSA. Suppose that instead of padding, we just use a Hash Function H.

3.1 a)

Write down the algorithms `Gen`, `Sig` and `Ver` for the new signature scheme that employs only H.

The modified signature scheme uses only a cryptographic hash function without any padding like EMSA (Encoding Method for Signature with Appendix). Assume that the Hashing Function used was SHA-256. We can describe the three main algorithms involved in this signature scheme:

1. **Key Generation (`Gen`)**

Algorithm 2 `Key_Generation()`

Choose two large prime numbers p and q
Compute the modulus $n = p \times q$
Compute the totient $\phi(n) = (p-1) \times (q-1)$
Choose a public exponent e such that $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$
Compute the private exponent d such that $e \times d \equiv 1 \mod \phi(n)$
Output:
    Public Key: (e, n)
    Private Key: (d, n)

2. **Signature Generation (`Sig`)**

Algorithm 3 `Signature_Generation(x)`

Compute the cryptographic hash of the message using the Hash Function H: $h = H(x)$
Use the private key (d, n) to sign the hash: $s = h^d \mod n$
Output: The signature s

3. **Signature Verification (`Ver`)**

Algorithm 4 `Signature_Verification(x, s)`

Compute the cryptographic hash of the message: $h = H(x)$
Use the public key (e, n) to compute: $h' = s^e \mod n$
if $h' = h$ **then**
    The signature is valid
else
    The signature is invalid
end if
Output: Return `True` if $h' = h$, otherwise return `False`

3.2 b)

Justify why the existential forgery attack is not applicable to the new scheme.

Before we continue, we must first understand what an existential forgery attack is. In simple terms, this attack is where the attacker tries to impersonate the sender of a message with their corresponding signature to try to send messages to the intended receiver. Here, the attacker chooses a random valid signature and computes an underlying message and sends this information publicly to the receiver. In this situation, the message would be verified by the receiver even though it was not the message that the sender originally sent.

In the case of the RSA Signature Scheme (not including the Hashing), an Existential Forgery Attack could be applied as the following. Say Bob is the Sender, Oscar is the Attacker and Alice is the Receiver. Bob has a private key of the integer d which acts as the modular inverse of e in his public key (n, e). He sends (n, e) over a insecure channel and Oscar chooses some signature $s \in \mathbb{Z}_n$ and computes a message $x \equiv s^e \bmod n$. Oscar sends the public key (n, e) and his impersonated message (x, s) over the insecure channel to Alice. Alice then performs the operation $s^e \equiv x' \bmod n$ where $x' = x$, meaning Alice recognizes the impersonated message as a valid signature.

If we implemented the RSA signature scheme using the Hash-only algorithm, an existential forgery attack would not be applicable to it. This is because the attacker now has to forge a signature for the Hashed message instead of the original message. Here, I am assuming it is computationally infeasible to reverse the hash function and to find two messages with the same hash values (due to possible Collision-Resistance) it becomes computationally infeasible to generate a valid signature for the hashed message. If its infeasible to find such a signature s, it is computationally infeasible to forge a fake signature and send it to the Receiver.

3.3 c)

Does using only H have a drawback compared to EMSA? Explain.

Using only the hash function H in the modified RSA signature scheme introduces certain drawbacks when compared to employing the Encoding Method for Signature Appendix (EMSA).

First, if the hash function H were to be reversed or if vulnerabilities such as hash collisions were exploited, an attacker could successfully forge a valid hashed signature and corresponding message. Hash collisions—where two distinct inputs produce the same hash output—undermine the integrity of the signature scheme, as they allow an attacker to generate falsified messages with the same hash value as a legitimate one.

Second, if the hash function H does not guarantee outputs longer than n bits (where n is the modulus bit-length in RSA), it becomes relatively easier for an attacker to forge signatures by brute-forcing smaller hash values.

On the other hand, EMSA incorporates a robust padding mechanism that provides additional security. Padding ensures that even if the hash function H is weak or partially compromised, the attacker would still face significant difficulty in guessing the correct format or forging valid signatures. Padding adds structured redundancy, making forgery substantially more challenging.

Finally, using only H exposes the scheme to **Length Extension Attacks**. In such attacks, an adversary can append additional data to the original message and produce a valid hash and signature without access to the private key. EMSA mitigates this risk by incorporating padding that invalidates any attempts to extend the input data without invalidating the signature.