CNS Homework 1 - Crypto

B11902083 徐粲桓

1. CIA

Reference: All by myself.

Confidentiality

- Ensure that the information is accessible only by authorized individuals
- e.g. Student's personal data leaked in a recent attack to the student affairs information system in several schools.

Integrity

- Ensure that the data remains unaltered during storage, transmission and processing.
- Checksums are hashes of binaries, often provided along with the binaries itself to let user check the downloaded files' integrity.

Availability

- Ensure that the resource is accessible by authorized individuals when needed
- e.g. Online FPS game server undergoing DDoS attack causing the players to suffer high latency is a violation to the server's availability.

2. Hash function

Reference: All by myself.

One-wayness

- Given $h = H(x)$, finding x is computationally hard.
- e.g. Password is often stored in its hashed form, without one-wayness, an adversary can effectively compute the original password if the hashes are leaked.

Weak Collision Resistance

- Given x, it is computationally hard to find $x' \neq x$ such that $H(x') = H(x)$.
- e.g. The checksum mentioned in p1 can only work if the hash function has weak collision resistance, otherwise, an adversary can craft a malicious binary x' that also passes the checksum test.

Strong Collision Resistance

- It is computationally hard to find any x, x' such that $x \neq x'$ and $H(x) = H(x')$.
- e.g. SHAttered is an attack against the SHA-1 hash function which breaks the strong collision resistance and allows crafting a pair of two PDF files that gives identical SHA-1 digests.

3. Asymmetric Cryptography

Reference: **3.1 (d)** discussed with B11902054 周得壹

3.1 (a)

This subproblem only discuss about the case where the line passing two points intersects E at third point. For other cases that finishes definition satisfying the group laws, please refer to 3.1 (c).

Point addition

For intuition, one can first consider elliptic curve over the field of real number $\mathbb{R}$, and substitute the arithmetic with operations in the field $\mathbb{F}_p$. In other words:

- Take $\bmod p$ for all addition and multiplication.
- Consider division $a/b, a, b \in \mathbb{R}$ be multiplying modular inverse ab^{-1} where $bb^{-1} \equiv 1 \bmod p$.

Start with two points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ (assuming $P_1 \neq P_2$ and $P_1, P_2 \neq O$) on the curve $E: y^2 = x^3 + ax + b$. Consider the line L passing P_1, P_2, its slope m is:

$$m = (y_2 - y_1)(x_2 - x_1)^{-1}$$

and $L: y = m(x - x_1) + y_1$, to find the point P_3 where L, E intersects, we solve for cubic equation:

$$(m(x - x_1) + y_1)^2 = x^3 + ax + b \Rightarrow 0 = x^3 - m^2x^2 + \dots$$

By Vieta's formula (which can be generalized in integral domains and field $\mathbb{F}_p$ is a subset of integral domains, see Wikipedia - Integral domain) we know that the third solution x_3 is given by $x_3 = m^2 - x_1 - x_2$, taking value at x_3 and reflect it across x-axis gives us y_3:

$$x_3 = m^2 - x_1 - x_2 \text{ and } y_3 = m(x_1 - x_3) - y_1$$

where we have $P_1 + P_2 = P_3, P_3 = (x_3, y_3)$.

> *Remark: When $P_1 = -P_2$ we have $P_1 + P_2 = O$ in the problem description. Furthermore, since O is identity, group law defines that $P + O = P, \forall P \in E$.*

Point doubling

To find the slope m' of tangent line L' passing $P_0 = (x_0, y_0)$, we use implicit differentiation to determine that:

$$2y\frac{dy}{dx} = 3x^2 + a \Rightarrow m' = \frac{dy}{dx} = \frac{3x^2 + a}{2y}$$

Note that if $y_0 = 0$ we have a vertical line, in this case we would define $P + P = O$.
Otherwise, the equation of L' is:

$$y = m'(x - x_0) + y_0$$

where we solve for the cubic (using Vieta's formula) again, this time we have $P + P = P'_0 = (x'_0, y'_0)$ where:

$$x'_0 = m^2 - 2x_0 \text{ and } y'_0 = m(x_0 - x'_0) - y_0$$

3.1 (b)

Note that since the set G of integral points along with operator $+$ forms a group $(G, +)$, it satisfies associativity i.e. $\forall g_1, g_2, g_3 \in G, g_1 + (g_2 + g_3) = (g_1 + g_2) + g_3$. By this property, one can apply the fast power algorithm, in detail:

- Step 1: Pre-compute $P, 2P, \dots, 2^mP, 2^m \leq n$ in $O(m) = O(logn)$ using Point doubling formula, starting from P.
- Step 2: Transform (in $O(logn)$) n to its binary form $n_2 = \{0,1\}^{m+1}$ and iterate through the bits, for each 1 add (with Point addition formula) the corresponding pre-computed value to the result.
  - e.g. $11 = (1011)_2 = 8P + 2P + P$.

As both steps is an $O(logn)$ operation, and Point addition/doubling are $O(1)$ formulas, this is an $O(logn)$ algorithm.

3.1 (c)

Note that the line L we discussed in 3.1 (a) do not necessarily passes E in another point, which happens when one of P_1, P_2 is a tangent point and the other is not.
WLOG, let P_1 be the tangent point, in such case we have $(P_1 + P_2) + P_1 = O$ (since $P_1 + P_2$ is just reflecting P_1 across x-axis) To satisfy the group law of inverse element, we thus define $P_1 + P_2 = -P_1$.

3.1 (d)

The following work heavily referenced this bookby Kenneth H. Rosen. In conclusion, if an elliptic curve is singular the discrete log problem are easier to solve.

The main idea is when the curve $E : y^2 = x^3 + ax + b$ is singular i.e. discriminant $\Delta := -16(4a^3 + 27b^2)$ gives zero, we can reduce the curve into the following two forms:

- $y^2 = x^3$
- $y^2 = x^2(x + c), c \neq 0, c \in \mathbb{F}_p$

with appropriate change of variables (Claim 1).

For each of the cases an isomorphism can map the points on E onto additive group of $\mathbb{F}_p$ and multiplicative group of $\mathbb{F}_p$, respectively.

Proof of Claim 1:

$$\begin{aligned} 4a^3 + 27b^2 = 0 &\Rightarrow a = 3k^2, b = 2k^3 \\ &\Rightarrow y^2 = x^3 + ax + b = x^3 + 3k^2x + 2k^3 \\ &\Rightarrow \begin{cases} y^2 = x^3, k = 0 \\ y^2 = x^2(x - c), k \neq 0 \text{ (By translating the double root to 0)} \end{cases} \end{aligned}$$

> *Note: The proof for isomorphism is omitted but can be found in the Theorem 2.30 & 2.31 in the book above. In addition, the isomorphism ignores the singular points in E to give E_{ns}. The proof in the book also includes the part explaining that ignoring singular points still forms a group for E_{ns}.*

For the first form, consider the isomorphism $\phi_1 : E_{ns} \to \mathbb{F}_p^+$ defined by

$$\phi_1 : (x, y) \mapsto xy^{-1}, \infty \mapsto 0$$

which is a group isomorphism between $E_{ns}, \mathbb{F}_p^+$ regarded as an additive group $\mathbb{F}_p^+ = (\mathbb{F}_p, +)$. Discrete log problem on E can thus be perceived as discrete log problem in $\mathbb{F}_p^+$ which is reduced to $g^x = g + g + \cdots + g = xg = h, x = hg^{-1}$ and can be solved trivially.

For the second form, let $\alpha^2 = a$ and consider the isomorphism ϕ_2 defined by:

$$\phi_2 : (x, y) \mapsto (y + \alpha x)(y - \alpha x)^{-1}, \infty \mapsto 1$$

1. If $\alpha \in \mathbb{F}_p$ then ϕ_2 gives an isomorphism $E_{ns} \to \mathbb{F}_p^\times$ where $\mathbb{F}_p^\times = (\mathbb{F}_p, \times)$.
2. If $\alpha \notin \mathbb{F}_p$ then ϕ_2 gives an isomorphism:

$$E_{ns} \simeq \{u + \alpha v | u, v \in \mathbb{F}_p, u^2 - \alpha v^2 = 1\}$$

   where the RHS is a group under multiplication.

Although discrete log problem on multiplicative groups are harder than additive groups, it is still easier than elliptic group (by the isomorphism and ref.)

3.2 (a)

To prove RSA is correct, we want to show that:

- $c \leftarrow m^e \bmod n$
- $m \leftarrow c^d \bmod n$

i.e. $((m^e \bmod n)^d \bmod n) = m$ indeed.

Note that by basic modular arithmetic, $(ab \bmod m) = ((a \bmod m)(b \bmod m) \bmod m)$. Therefore, $((m^e \bmod n)^d \bmod n) = (m^e \times m^e \cdots \times m^e \bmod m) = m^{ed} \bmod m = m$. We discuss these in two cases on $\gcd(m, n)$.

Case 1: $\gcd(m, n) = 1$

By Euler's theorem, $m^{\phi(n)} \equiv 1 \bmod n$ and that $ed \equiv 1 \bmod \phi(n)$:

$$\begin{aligned} m^{\phi(n)} \equiv 1 \bmod n &\Rightarrow m^{ed} = m^{1+k\phi(n)} = m(m^{\phi(n)})^k \\ &\Rightarrow m^{ed} \bmod n = ((m \bmod n)(m^{k\phi(n)} \bmod n) \bmod n) = m \bmod n \end{aligned}$$

Case 2: $\gcd(m, n) \neq 1$

> *Claim:* $(x \equiv y \bmod p) \wedge (x \equiv y \bmod q) \Rightarrow (x \equiv y \bmod \operatorname{lcm}(p, q))$
> *Proof.* $p|(x-y) \wedge q|(x-y) \Rightarrow \operatorname{lcm}(p,q)|(x-y) \Rightarrow (x \equiv y \bmod \operatorname{lcm}(p,q))$.

By Claim and that $\operatorname{lcm}(p, q) = n$, we have:

$(m^{ed} \equiv m \bmod p) \wedge (m^{ed} \equiv m \bmod q) \Rightarrow (m^{ed} \equiv m \bmod n)$

it suffices to show the two on the LHS to prove the RHS.

Note that in this case, we have $km = n = pq$ and thus either $p = \gcd(m, n)$ or $q = gcd(m, n)$.
For the first case, let $m = kp$ and indeed:

- $(m^{ed} = (kp)^{ed} = k^{ed}p^{ed}) \equiv (m = kp) \equiv 0 \bmod p$
- $$\begin{aligned} m^{ed} &= m^{ed-1}m \\ &= m^{h(p-1)(q-1)}m \ (ed \equiv 1 \bmod \phi(n)) \\ &= [m^{\phi(n)}]^{h(p-1)}m \\ &= m \bmod q \ (\text{Euler's Theorem, } m^{\phi(q)} \equiv 1 \bmod q) \end{aligned}$$

Therefore, we have $m^{ed} \equiv m \bmod n$.

ECDSA (a)

To show that the scheme is correct, we show that in the verification, the result x' will satisfy $x' = r$. Consider all arithmetic done under field $\mathbb{F}_n$:

$$\begin{aligned}
x' &= u_1 \times (k^{-1}x) + u_2 \times (dk^{-1}x) \\
&= (H(m)s^{-1}k^{-1} + rs^{-1}k^{-1}d)x \\
&= (H(m)(H(m)+rd)^{-1}kk^{-1} + r(H(m)+rd)^{-1}kk^{-1}d)x \\
&= ((H(m)+rd)^{-1}(H(m)+rd))x \\
&= x \bmod n \\
&= r.
\end{aligned}$$

ECDSA (b)

Suppose an eavesdropper has captured two messages and signature pair signed with same nounce k, say $(m_1, \sigma_1), (m_2, \sigma_2)$ where $\sigma_1 = (r, s_1), \sigma_2 = (r, s_2)$. Consider all arithmetic done under field $\mathbb{F}_n$:
Then we have the system of equation:

$$\begin{cases} s_1 = k^{-1}(H(m_1) + rd) \\ s_2 = k^{-1}(H(m_2) + rd) \end{cases}$$

Subtract first equation by second, we get:

$$s_1 - s_2 = k^{-1}(H(m_1) - H(m_2)) \Rightarrow k^{-1} = (s_1 - s_2)(H(m_1) - H(m_2))^{-1}$$

Substitute k^{-1} in first equation:

$$s_1 = (s_1 - s_2)(H(m_1) - H(m_2))^{-1}(H(m_1) + rd)$$

Then, we can calculate private key d by:

$$d = r^{-1}[s_1(s_1 - s_2)^{-1}(H(m_1) - H(m_2)) - H(m_1)]$$

where s_1, s_2, m_1, m_2, H, r are all known variables.

4. Application of PRG

Reference: All problems discussed with B11902054 周得壹 and B11902163 羅翊庭.

4.1. (a)

For Alice to be able to open to both 0 and 1, she need to find some $x_0, x_1 \in \{0,1\}^*$ such that $G(x_0) \oplus G(x_1) = r$. Bob need to choose such r first for Alice to do this. WLOG, consider the upperbound of this probability by assuming that:

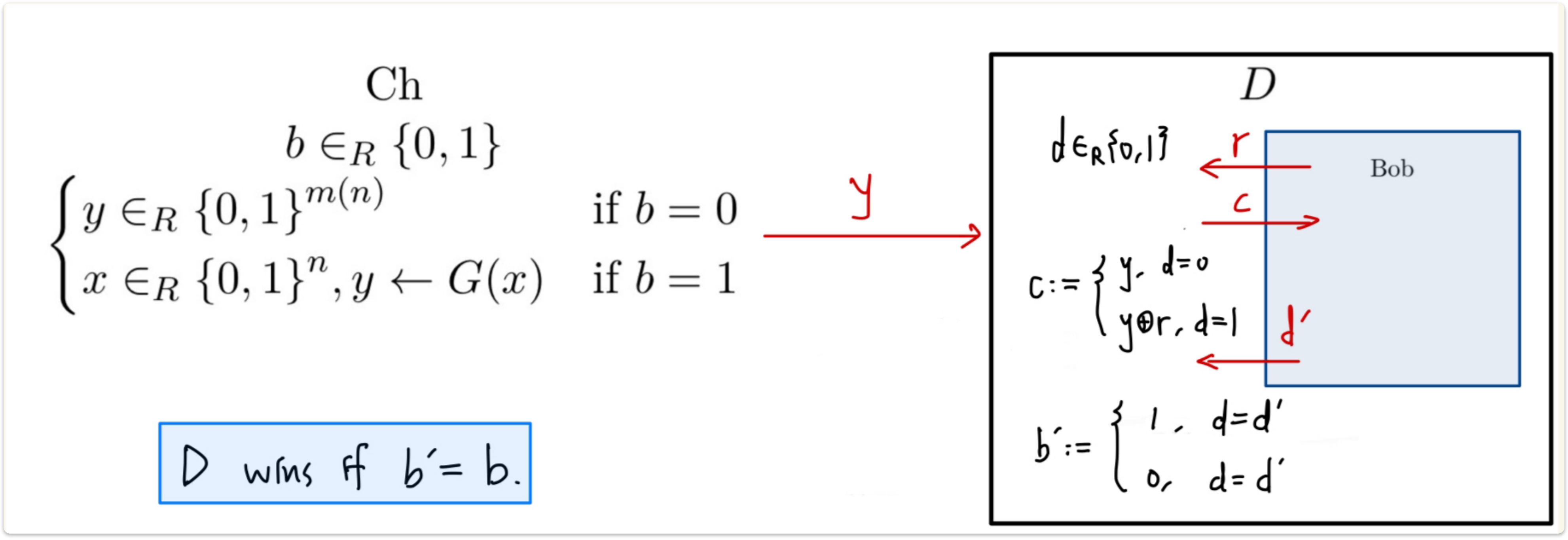
- $G(x_0) \oplus G(x_1) = r, r$ is unique.
  - This is not guaranteed as consider r first bit to be 0, $G(x_0), G(x_1)$ first bit can be both 0 or both 1. But we only consider the "worst case" here.

Under this assumption, there will be $2^n \times 2^n$ unique r and the probability for Alice to be able to open to both 0 and 1 (i.e. Bob gets one of the unique r) has an upperbound of:

$$\frac{2^{2n}}{2^{3n}} = 2^{-n}$$

4.1. (b)

Consider the following construction of D.



Ch
$b \in_R \{0,1\}$
$y \in_R \{0,1\}^{m(n)}$ if $b=0$
$x \in_R \{0,1\}^n, y \leftarrow G(x)$ if $b=1$
y
D
$d \in_R \{0,1\}$
r
c
Bob
$c := \begin{cases} y, & d=0 \\ y \oplus r, & d=1 \end{cases}$
d'
$b' := \begin{cases} 1, & d=d' \\ 0, & d=d' \end{cases}$
D wins iff $b' = b$.

The main idea is to construct a D that let's Bob determines y that is either randomly sampled from $\{0,1\}^{m(n)}$ or generated by $G(x)$, which is decided by $b \in_R \{0,1\}$.

- $b=1$: y is generated by $G(x)$ and by assumption, Bob can win the game with probability $1/2+\mu$ given that $y \leftarrow G(x), x \in_R \{0,1\}^n$ i.e. $b=1$.
- $b=0$: y is randomly sampled. There're two cases for Bob.
  - If y cannot be generated from $G(x)$, no matter what strategy he uses he can correctly guess only with $1/2$.
    - $\Pr[y=y_0] = \Pr[y = y_0 \oplus r], \forall y_0, r \in_R \{0,1\}^{m(n)}$ but gives opposite outcome, this means the probability for him to pass/fail on the guess is always $1/2$ for any r, which has nothing to do with his strategy.
  - Otherwise, it's the same as $b=1$.

As a result, we have:

$$\begin{aligned} \Pr(b'=b) &= \Pr(b'=b, b=1, y\in S) + \Pr(b'=b, b=0, y \in S) + \Pr(b'=b, b=1, y \notin S) \\ &= (\frac{1}{2}+\mu) \times \frac{1}{2} \times 1 + (\frac{1}{2}+\mu) \times \frac{1}{2} \times p + \frac{1}{2} \times \frac{1}{2} \times (1-p) \\ &> (\frac{1}{2}+\mu) \times \frac{1}{2} + \frac{1}{4} \\ &= \frac{1}{2} + \frac{\mu}{2} \end{aligned}$$

Where we let $S = \{G(x) : x \in_R \{0,1\}^n\}, 0 < p = 2^n/2^{m(n)} \leq 1/2 \text{ (By extension of G)}$.

4.2. (a)

$L \in \mathrm{NP}$

Consider the decision problem L be:

$$L = G(\{0,1\}^*) = \{G(x) : x \in \{0,1\}^*\}$$

We want to show that L is in NP i.e.

$$y \in L \iff \exists x \in \bigcup_{n=0}^{s(|y|)} \{0,1\}^*, V(y,x) = 1$$

Where V is polynomial time $(t(|y|,|x|), t$ is polynomial) verification algorithm and s is polynomial.

- By the extending property of PRG G, we can construct the following algorithm $V(y,x)$:
  - $y \in \{0,1\}^*$ is the string we're deciding.
  - $x \in \bigcup_{n=0}^{|y|} \{0,1\}^n$ is the string to put into G.
  - V simply computes $G(x)$ and returns

$$\begin{cases} 0, \text{ if } G(x) \neq y. \\ 1, \text{ if } G(x) = y. \end{cases}$$

- V is polynomial time as G is efficiently computable by definition and determining $G(x) = y$ is a polynomial time operation of $t(|y|,|x|) = max(|y|,|x|)$.
- For x, we can stop at y as G is extending i.e. $y \in \{0,1\}^{m(n)}$ must be generated by $x \in \{0,1\}^n$ where $m(n) > n$.

We now move on to the proof:

- ($\Rightarrow$) Given $y \in L$, by the extending property of G it suffices to use V to search $\bigcup_{n=0}^{|y|} \{0,1\}^n$ for such x and it must exist in this set.
- ($\Leftarrow$) If we have such $x \in \bigcup_{n=0}^{|y|} \{0,1\}^n$, then by definition $y \in L$ as it is generated by some $x \in \{0,1\}^*$ indeed.

Therefore by definition, $L \in \mathrm{NP}$.

Contradiction from $\mathrm{P} = \mathrm{NP}$

To finish the proof, we approach by proving with contradiction. Consider $L \in \mathrm{P} = \mathrm{NP}$, by definition it means there exists algorithm A returning $0, 1$ in at most $t(|y|)$ steps on input y, t is polynomial, such that:

$$y \in L \iff A(y) = 1$$

In short, A can determine whether y can be generated by G in polynomial time.

Now, using A we can break the PRG game:

$$\begin{array}{ccc}
\text{Ch} & & D \\
b \in_R \{0,1\} & & \\
\begin{cases} y \in_R \{0,1\}^{m(n)} & \text{if } b = 0 \\ x \in_R \{0,1\}^n, y \leftarrow G(x) & \text{if } b = 1 \end{cases} & & \\
& \xrightarrow{\quad y \quad} & \\
& & b' \leftarrow D(y) \\
& D \text{ wins if } b' = b. &
\end{array}$$

Table 1: PRG game

Consider the following strategy for D:

- Upon receiving y, D calculates $A(y)$ and:
  - If $A(y) = 1$, $D(y) \in_R \{0,1\}$.
    - D is not sure whether y happens to be in $\{G(x) : x \in \{0,1\}^*\}$ from random sampling in $\{0,1\}^{m(n)}$, or y indeed comes from G.
  - If $A(y) = 0$, $D(y) = 0$.
    - If y is not generated by G, it cannot be $b = 1$.

The winning probability $P(b' = b)$ is as follows, we can assume independence between choice of b and b':

- $y \in L, A(y) = 1$

$$\begin{aligned} P(b' = b) &= P(b' = b \cap b = 1) + P(b' = b \cap b = 0) \\ &= \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} \\ &= \frac{1}{2} \end{aligned}$$

- $y \notin L, A(y) = 0$

$$\begin{aligned} P(b' = b) &= P(b' = b \cap b = 1) + P(b' = b \cap b = 0) \\ &= 1 \times 0 + 1 \times 1 \\ &= 1 \text{ (We can always get a correct guess)} \end{aligned}$$

  Here, we consider the worse case where G do not have any "collision" and $G(x) : x \in \{0,1\}^n$ has size of 2^n. Then:

$$\begin{aligned} P(y \in L) &= \frac{1}{2} + \frac{1}{2} \times \frac{2^n}{2^{m(n)}} \\ &\le \frac{1}{2} + \frac{1}{4} \text{ (Extention of } G) \\ &= \frac{3}{4} \\ P(y \notin L) &\ge \frac{1}{4} \end{aligned}$$

  As a result:

$$\begin{aligned} P(b' = b) &= \frac{p}{2} + (1 - p), \frac{1}{2} < p \le \frac{3}{4} \\ &= 1 - \frac{p}{2}, \frac{1}{2} < p \le \frac{3}{4} \end{aligned}$$

  Or $\frac{5}{8} \le P(b' = b) \le \frac{3}{4}$ making D always wins with probability $\frac{1}{2} + \nu(n)$ where $\nu(n)$ cannot be neglected, thus the pseudo-randomness of G is broken, a contradiction.

Therefore by contradiction, if G is PRG, we have $\mathrm{P} \neq \mathrm{NP}$.

5. Simple Crypto

Reference: **(d)** discussed with B11902163 羅翊庭
Basically, most classical cipher follow a recognizable pattern and is computationally feasible to brute-force the original text. The code for each subproblem except for (b) is included in `code/code5.py`.

(a)

The name itself suggests that it's Affine substitution cipher. One can simply brute force $0 \leq a, b \leq 255$ until encrypting the two plaintext characters in the hint by $ax + b \bmod 256$ matches the ciphertext.

The flag is `CNS{51MP13_M47H_70_8r34K_4FF1N3_a3e4aa86dc09bf988d0596c4e83a6af2}`.

(b)

Searching the keyword in hint shows that it is probably encrypted in Rail Fence Cipher, using an existing decryption tool and adjust with the rail key on the decoded text, until yielding readable text as passphrase.

The flag is `CNS{8rU73_F0rC3_4r3_P0551813_70_8r34K_Z16Z46_50b012399a6eb9004fa2ae62d276cec2}`.

(c)

Combining the hint and the ciphertext's pattern, it should be Polybius cipher. (Although Playfair cipher also has the "Consider I/J together" rule, the ciphertext pattern do not match)

First I extracts 5 distinct characters in the ciphertext, then I used `itertools` to iterate all $2 \times 5!$ possibilities for rows and columns ($5!$ of them uses `i`, another $5!$ uses `j`). Decrypt by all the tables should yield one readable plaintext (Note the difference of character `i` and `j`) which is the passphrase.

The flag is `CNS{P01Y81U5_0N1Y_H4V3_5_UN1QU3_CH4r4C73r5_1N_C1PH3r73X7_ca59a45ed946684762b87dd9ef90eeba}`.

(d)

Decode using `xdd -p -r` yields ciphertext with the presence of `=`, indicating that this is very likely to be `base64` encoded (not an encryption) Pipe it to `base64 -d` yields the passphrases.

After putting the correct passphrase we will get the plaintext, hex encoded and encrypted using a 6-bytes one time pad. Since the flag prefix `CNS{` should be in the plaintext, we may brute-force OTP by:

1. Guessing the location of encrypted `CNS{` in ciphertext
2. XOR ciphertext with `CNS{` to yield first 4 bytes of OTP.
3. Guess all 256^2 possibilities for the last two bytes to construct a OTP.
4. Repeatedly applying OTP to reveal guessed plaintext.

This should be done in $O(n)$ by iterating through all possible position of `CNS{` and is thus feasible. Additionally, we can use simple filtering (e.g. decrypted characters must be readable text) to find the final plaintext and the flag:
`CNS{07P_4CH13V35_P3rF3C7_53CUr17Y_UND3r_455UMP710N5_b7dd544750014b538024ebe801788feb}. Don't tell anyone!`

6. Simple RSA

Reference: All by myself.
RSA can to meet security properties for badly chosen values. Here the main reason of failure is marked in bold, and the detailed code for each subproblem is in `code/code6.py`. (Note: I used external library for (c) and (d), but I will go through the idea in this writeup.)

(a)

Since **n is small**, we can simply use online factorization tool like this oneto factorize n for $n = pq$, the rest is following the RSA procedure and recover d to decrypt m.

The flag: `CNS{345Y_F4C70r1Z4710N}`.

(b)

The message m is encrypted using **same n and co-prime e_1, e_2** yielding c_1, c_2 respectively. By Extended Euclidean Algorithm, we can get s, t such that $e_1 s + e_2 t = 1$.
Then: $m^1 = m^{(e_1 s + e_2 t)} = (m^{e_1})^s \times (m^{e_2})^t = c_1^s \times c_2^t$.

The flag: `CNS{7W0_C1PH3r73X7_W111_8r34K_r54?!_79f17b6d9c07fb154c6dcdb92ab047a7}`.

(c)

Note that **e is small**, there are thus two approaches depending on m's size.

- $m^e < n$, we can efficiently recover $m = c^{1/e}$.
- $m^e > n$, which is the case here, we can either brute force $m^e + kn$, or since **we uses same $e = 7$ and different n to encrypt message to multiple parties**, we have system of congruence:

$$\begin{cases} m^e \equiv c_1 \bmod n_1 \\ \dots \\ m^e \equiv c_k \bmod n_k \end{cases}$$

By Chinese Remainder Theorem we can recover the general solution for m^e. Note that:

- n_i should be co-prime, but since we chooses p, q as sufficiently large primes, it should not collide for the equation we chose.
- CRT is already implemented in python library `sympy.ntheory.modular` as `crt()`.
- Note that the solution is in form of

  $$kN + \sum_{i=1}^{k} c_i t_i n_i$$

  where $t_i M_i \equiv 1 \bmod m_i$. Thus it makes sense to choose at least e equations as N should satisfy $N \geq min(n_1, \dots, n_k)^e$ (N need to be big enough so we can consider unique solution)

The flag: `CNS{CH1N353_r3M41ND3r_r3P347_8r04DC457_4774CK_edc38122a288aa9922617b5a6a08ac61}`.

(d)

Given the **small private exponent d** in this case, one can perform **Wiener's attack** to efficiently recover $\phi(n) = (p-1)(q-1)$. The attack is based on Wiener's Theorem stating the following: Let $N = pq$ with $q < p < 2q$, $d < \frac{1}{3}N^{1/4}$, and $ed \equiv 1 \bmod \phi(n)$. We can efficiently recover d by searching convergents of $\frac{e}{N}$.

- Note: The "convergents" means each of the rationals in the continued fraction expansion of a rational number.

As we have $ed = k\phi(n) + 1, |\frac{e}{\phi(n)} - \frac{k}{d}| = \frac{1}{d\phi(n)}$ where $\frac{1}{d\phi(n)}$ is small, searching the convergents $\frac{k}{d}$ allow us to compute some $\phi(n)$ and check its correctness by solving:

$$\phi(n) = (p-1)(q-1) = n - p - \frac{n}{p} + 1$$

$$p^2 + p(\phi(n) - n - 1) - n = 0$$

and check the roots p_1, p_2 satisfies $p_1 p_2 = n$ or not. Here I use the `owiener` (pip package link) package which supports finding d given e and n.

The flag: `CNS{816_e_4ND_W13N3r_4774CK_1N_7H3_NU75H311_0f2146bda49df4206f344cd29b0503fa}`

7. POA

Reference: **(a)** discussed with B11902163 羅翊庭

(a)

This problem is a classic POA using the padding oracle in choice (3), except the padding scheme here is slightly different from PCKS#7 padding, it pads 9-24 instead of 0-15. The full code is in `code/code7-1.py`, what it does is the following:

- For each block, we modify the last k bytes starting from `start_index` of `ciphertext_copy`.
  - `plain_block` saves the correctly guessed character.
  - The current guess is `ciphertext_copy[start_index]`, which simply iterates through 0-255.
- After guessing one block, the last block of ciphertext is discarded and the decrypted block is appended to `plain_block_list`.
- After guessing all except first plaintext block, concatenate them.

Although ignoring the first plaintext block m_1 (i.e. not recovering the IV) suffices to find the flag, I'll explain the idea here. To get m_1, we will decrypt c_1 with AES first (which we call d_1) then XOR d_1 with IV to recover m_1. It seems that both m_1, IV is unknown and it's impossible, but we can actually use the encryption service with a fixed prefix `Request nounce: ` just a size of a block as m_1, along with c_1 to recover IV using POA again.

The flag: `CNS{p4dd1n9_0r4cl3_4tt4ck_15_3v1l}`

(b)

The goal here is to forge a request `message` to pass two requirement:

- `message[:16] == 'Request nounce: '`
- `message == b'Sender: TA; Message: Please send over the 2nd flag'`

However, as the encryption service do not allow using `TA` directly in the sender field, we need to bypass this while fulfilling the requirement. This is done by the following steps in `code/code7-2.py`:

- `inj` is the item we wish to put in the sender field.
  - Let `inj` be `TA; Message: Please send over the 2nd flag`
  - Add the padding to block boundary, here it is to add 12 bytes of value 20.
- For the message field, input anything.
- Manually discard the ciphertext blocks after the padding and send it to decryption service, it should pass the tests and returns the flag.

The forgery works since blocks in CBC modes are not influences by blocks coming after it, so discarding trailing ciphertext blocks do not alter the plaintext in the beginning, thus, we can pass the test.

The flag: `CNS{f0r61n6_r3qu3575_4r3_5up3r_345y}`

8. CNS Store

Reference: All by myself.
Firstly, I solve the POW using a silly (but working) method by the following code snippet:

```
while True:
    ran_num = random.randint(0, some stupidly big number)
    hash_str = 'CNS2024' + str(ran_num)
    long_sha = sha256(hash_str)
    sha_val = long_sha[-6:]
    if sha_val == target:
        print(hash_str)
        break
```

It works but it may take some time. (A better solution is to keep track of existing hashes i.e. Birthday attack to break strong-collision resistance, but it wasn't much of a speed boost for me.)

(a)

The first problem is based on SHA-1 collisions. Here I use SHAttered, a SHA-1 collision attack that creates two different PDF files with identical hashes. The idea is as follows:

- Since the store allows any "product" with the name `CNS2024` in it, but the number of each product cannot exceed 10 and cannot be bought in two separate purchases. However, the second requirement is checked by hashing the product name and save it into a Python dictionary.
- Therefore, we can use SHAttered to create two colliding PDFs, **add `CNS2024` to the end of the file** (say, using online hex editor) This do not break the colliding property as SHA-1 is of Merkle–Damgård construction, and given existing collision we can extend the message and still preserve the collision.
- Sending these colliding files in bytes using pwntool can allow us to repeatedly trigger `shelf[h] -= amount` result in an negative amount of product count.
- The negative amount allow us to get extra money from lottery: `cash -= sum(shelf.values()) * 5` and let us buy flag.

The flag: `CNS{S1mp!e_H45h_C011!510n_@7tacK!}`

(b)

The second problem is almost identical to (a) but with SHA-256 suffix collision. We can simply do this using Birthday attack i.e. uses a dictionary to look up for any pair of colliding hashes in the hashes we generate, using the following code:

```
dic = {}
pair = []
while True:
    ran_num = random.randint(0, 100000000000)
    hash_str = contain_key + str(ran_num)
    sha_val = sha256byte(hash_str)[-4:]
    if sha_val in dic:
        pair.append(hash_str)
        pair.append(dic[sha_val])
        break
    else:
        dic[sha_val] = hash_str
```

The flag: `CNS{H@ppy_B!r7hDaY_t0_Y0U!}`

(c)

The third problem is length-extension attack of SHA-256. The main idea is the following:

- We are prompted with `userID`, the hash of hex of `IDstr`, and `userIdentity`, which forms `IDstr` by combining with session key `key`.
- `IDStr` is first compared against `userID`, then `userIdentity` is extracted to determine if we're admin.
- By length-extension attack, we can know $H(x||data||ext)$ from $H(x)$, without ever knowing x.
    - Here, ext will be `admin` so the last 5 characters of `IDstr` become `admin` allowing us to pass the admin test.

Here I used (and compiled, using its Makefile) this Github repository to perform length-extension attack written in C. The executable `./hash_extender` is required for `code/code8.py` to finish this subproblem. The command we call (in a subprocess) is `./hash_extender --data staff --secret <secret_len> --append admin --signature <original_hash> --format sha256`.

- Since `key` is a random bytes string of length 40-50, we iterate 40-50 to find the correct key length.
    - `hash_extender` supports `--secret` to assign the length $|x|$ for unknown secret x.
- `--data` assigns the $data$ `staff` which we wish to append ext after it.
- `--append` assigns ext `admin`.
- `--signature` is the original hash $H(x)$.

We search the range of key length 40-50, until we finds the flag in the server response.

The flag: `CNS{https://www.youtube.com/watch?v=YFJowRNfMGI}`