Modern Cryptography Homework 1

Yosef Israel Goren

April 2022

Let us denote:

$$\forall a, b : (a, b) = gcd(a, b)$$

GCD

1 Facts about GCD

1.1

Claim:

$$\forall a, b, n : (an + b, n) = (b, n)$$

Proof:
lemma: $(\alpha \mid n) \wedge (alpha \mid an + b) \Leftrightarrow (\alpha \mid n) \wedge (\alpha \mid b)$

$$(\alpha \mid n) \wedge (\alpha \mid an + b) \Leftrightarrow (\exists x : \alpha x = n) \wedge (\exists y : y\alpha = an + b))$$

$$\Leftrightarrow (\exists x : \alpha x = n) \wedge (\exists y : \alpha y = \alpha a x + b) \Leftrightarrow (\exists x : \alpha x = n) \wedge (\exists y : \alpha(y - ax) = b)$$

$$\Leftrightarrow (\exists x : \alpha x = n) \wedge (\exists z : \alpha z = b) \Leftrightarrow (\alpha \mid n) \wedge (\alpha \mid b)\blacksquare$$

from lemma:

$$\{\alpha : (\alpha \mid n) \wedge (\alpha \mid b)\} = \{\alpha : (\alpha \mid n) \wedge (\alpha \mid an + b)\}$$

$$\Rightarrow max\{\alpha : (\alpha \mid n) \wedge (\alpha \mid b)\} = max\{\alpha : (\alpha \mid n) \wedge (\alpha \mid an + b)\}$$

$$\Rightarrow (an + b, n) = (b, n)$$

$$\blacksquare$$

1.2

Claim:

$$\forall a, b, n : (a, b) \mid (ab, n)$$

Proof:
lemma: $\forall a, b, n : ((a, b) \mid ab) \wedge ((a, b) \mid n)$

$$\alpha = (a, b) \Rightarrow (\alpha a) \wedge (\alpha \mid n) \Rightarrow (\exists x : \alpha x = a) \wedge (\alpha \mid n)$$

$$\Rightarrow (\exists x : \alpha x b = ab) \wedge (\alpha \mid n) \Rightarrow (\exists y : \alpha y = ab) \wedge (\alpha \mid n) \Rightarrow (\alpha \mid ab) \wedge (\alpha \mid n) \blacksquare$$

let $\alpha = (a, n), \beta = (ab, n)$.
Now assume that: $\alpha \nmid \beta$.
This means that there must be at-least one 'extra' prime within α which does not exist in β. To be exact:

$$\exists p \in \mathbb{P}, k \in \mathbb{N}^{+} : (p^{k} \nmid \beta) \wedge (p^{k} \mid \alpha)$$

So from the transitivity of the division relation, and the lemma:

$$(p^{k} \mid ab) \wedge (p^{k} \mid n)$$

We can further deduce that:

$$\forall i \in \{0 ... k\} : (p^{i} \mid ab) \wedge (p^{i} \mid n)$$

Let $g = p\beta$.
for any prime within g which is not p, it must have come from β so it must be in ab and n. for the prime p, it's power within g must be smaller or equal to k, because $p^{k} \mid ab$. and we also know that for any such power (smaller or equal to k) - p^{i} divides both ab and n. This means that each prime instance of g divides ab and n, so g itself divides ab and n. This makes g a common denominator for ab and n, and it is also greater than β. This is a contrediction, as β is, by definition - the greatest common denominator.

$$\blacksquare$$

1.3

Claim:

$$\forall p \in \mathbb{P}, a, b \in \mathbb{N} : p \mid ab \Rightarrow p \mid a \wedge p \mid b$$

Proof:
$p \mid ab \Rightarrow \exists x : xp = ab$.
Now from the prime decomposition theorem, we know that each prime in the left expression, including p, must also be in the right expression. So if we look at a prime decomposition of ab we must see p with a power of at-least 1.
Let $a = \prod a_i^{k_i}, b = \prod b_i^{l_i}$ where a_i, b_i are all primes and the powers are all greater than one.

From the prior conclusion, p must be within $\prod a_i^{k_i} \cdot \prod b_i^{l_i}$. Since all of these are already primes, p must be one of them. meaning there is some a_i or b_i which is equal to p.
If $a_k = p$ than $p \cdot (a_k^{k_k - 1} \prod_{i \neq k} a_i^{k_i}) = ab$ so by definition $p \mid a$. In the same way if $p = b_i$ then $p \mid b$

■

1.4

Claim:

$$\forall a, b, n \in \mathbb{N} : ((a, n) = 1) \wedge ((b, n) = 1) \Rightarrow ((ab, n) = 1)$$

Proof:
lemma:
If a, n are co-prime it means that none of the primes that compose n are within a. In the same way we also know that none of the primes that comnpose n are in b. Since the set of primes composing a product, is the union of the primes that compose each part of the product (in this case the product is $a \cdot b$ and the parts are a, b); then if every prime in n is not in either a or b, it will not be in the union of the primes either
(in terms of group theory, this is because: $x \notin A \vee x \notin B \Rightarrow a \notin A \cup B$). Each prime composing n is not in the decomposition of ab. ■

Assume $(n, ab) > 1$.
Let p be some prime composing (n, ab) - there must be at-least one prime because (n, ab) is not 1.

$$p \mid (n, ab) \vee (n, ab) \mid n \vee (n, ab) \mid ab \Rightarrow p \mid n \vee p \mid ab$$

This contredicts the lemma. So the assumption is incorrect and $(n, ab) = 1$.

■

2 Extended Euclidian Algorithm

We can define the algorithm recursively:
$EEA(a, b)$:
if a or b are 0, return a, b.
Otherwise: find $k, m \in \mathbb{N}^+$ s.t. $a = b \cdot k + m$. (here we assume $a \geq b$).
Call: $x, y \longleftarrow EEA(b, m)$.
return $y, x - k$.
Note how m will be at most half of b, meaning that each recursive iteration is at worst - halfing the size of the input. This means that the complexity is logarithmic at worst. The complexity is $O(log(a + b))$.

3 Computing Inverses in $\mathbb{Z}_N^*$

We can use the Extended Euclidian Algorithm.
$InverseZ(a, N)$:
Call: $x, m \longleftarrow EEA(a, N)$.
return x.

Correctness: Given some $a \in \mathbb{Z}_N^*$, we want to find some $x \in \mathbb{Z}_N^*$ s.t.

$$ax = 1(modN)$$

In other words, we want to find $x \in \mathbb{Z}_N^*, k \in \mathbb{Z}$ s.t.

$$ax = 1 + Nk$$

If we change the equation:

$$ax - Nk = 1$$

this is the form given by the Extended Euclidian Algorithm and Indeed the gcd must be 1 from the definition of $\mathbb{Z}_N^*$.

To be more explicit, after we have run $EEA(a, N)$ - we got x, m s.t.

$$ax + Nm = (a, N) = 1$$

and so we can deduce:

$$ax = 1 - Nm \Rightarrow ax = 1(modN) \Rightarrow x = a^{-1}$$

$$\blacksquare$$

Modular Exponentiation

1 Generic Algorithm

In the next section we will show a full algorithm for doing that calculation.
The generic algorithm is the same, only where we see $a \cdot b(modN)$ - in the generic algorithm we can just abstract away and say it is $a \cdot b$ under the provided assumption.
Other than this - there is nothing spesific within the following algorithms which binds it to the group $\mathbb{Z}_N^*$.

2 Algorithm for $\mathbb{Z}_N^*$

The 'trivial' algorithm would be to calculate g^x would be to start with 1 and raise to the power x by taking the product of what we have thus far and g - x times, where in each iteration we only calculate the result $modN$. This would mean we never have to deal with numbers greater than N^2 which promises polynomial operations thorighout the algorithm. This would be correct because:

$$(((g) \cdot g(modN)) \cdot g(modN))...) \cdot g(modN) = g^x modN$$

The small problem with this algorithm is that it is polynomial with respect to x, which makes it exponential with respect to the input length!

While this disqualifies this algorithm, we can use these same ideas to make a new - polynomial - algorithm.
What we did before was count to x, where in each iteration we increment by 1; instead - we can count to x by incrementing each time by some 2^i for i going from 1 untill we have reached x. Also, instead of multiplying our current product by g^1, we will multiply the product by g^{2^i}.

To be more explicit, let us denote the binary representation of x:

$$x = \sum_{i=0}^{K} b_i 2^i : \forall i \in \{1...K\}, b_i \in \{0,1\}$$

This means:

$$g^x(modN) = g^{\sum_{i=0}^{K} b_i 2^i}(modN) = \prod_{i=0}^{K} g^{b_i 2^i}(modN)$$

$$= ((((b_0 2^0) \cdot b_1 2^1(modN)) \cdot b_3 2^3(modN))...) \cdot b_K 2^K(modN)$$

And this last expression is exactly what our algorithm will be caculating.
There are two parts to the algorithm;

1. Calculate the powers of g: $p_i = g^{2^i}(modN) : \forall i \in 1...K$.
   To do this efficiantly, start with $i = 0, p_i = g$; in each iteration:
   - $p_{i+1} \longleftarrow p_i^2(modN)$
   - $i \longleftarrow i + 1$
   - if $2^i \geq x$, finish
2. calculate the mod product:
   Start off with $result \longleftarrow 1$ and iterate over the the bits in the binary representation of x (these are b_i).
   In each iteration: $result \longleftarrow result \cdot p_i(modN)$.
3. return $result$.

What we are left with after these iterations is exactly the expression from before, which we have shown that is equal to $g^x(modN)$. It is also to see how these loops must terminate in a number of iterations which is linear to the number of bits in the input, as well as the operations in each iteration being polynomial. So number of logical operations, and arithmetic group operations done by the algorithm are both polynomial, as well as a polynomial amount of regualr integer operations. As we know, all of the above are computable with efficiantcy, accept the arithmetic operations which are within the group. Show that it too can be done efficiantly would prove the efficiantcy of our algorithm. Next we show how the operation of $\mathbb{Z}_N^*$ can be calculated efficiantly:

Multiplication Algorithm $a \cdot b(modN)$:

- Multiply $x \longleftarrow a \cdot b$
- Divide without remainder $k \longleftarrow \frac{x}{N}$
- return $x - k \cdot N$

Note how all of the prior operations are standard integer operations and are thus efficiant. In addition, $a, b \leq N - 1$ so $x \leq N^2$, meaning the binary representation of x is polynomial with resprect to the representations of a and b. This makes all of the integer operations polynomial w.r to a, b, which in turn means that our group operation is done efficiantly.

Data Processing Inequality

1 Statistical Distance

Claim:
For any pair of distributions X, Y over Ω on, and a function f over Ω:

$$SD(f(X), f(Y)) \leq SD(X, Y)$$

Proof:
Let $D = argmax_d(adv_d(f(X), f(Y)))$.
Let D' be a new distinguisher s.t. $\forall x : D'(x) = D(f(x))$.
This means:

$$adv_D(f(X), f(Y)) = \mid Pr_{x \leftarrow f(X)}[D(x) = 1] - Pr_{x \leftarrow f(Y)}[D(x) = 1] \mid$$

$$= \mid Pr_{x \leftarrow X}[D(f(x)) = 1] - Pr_{x \leftarrow Y}[D(f(x)) = 1] \mid = adv_{D'}(X, Y)$$

Thus:

$$SD(f(X), f(Y)) = dv_D(f(X), f(Y)) = adv_{D'}(X, Y) \leq max_d(adv_d(X, Y)) = SD(X, Y)$$

■

2 Computational Distance

Claim:
For any enasmble of distributions $X = \{X_n\}, Y = \{Y_n\}$ over $\{0,1\}^*$:

$$\forall f \in \{0,1\}^* \longrightarrow \{0,1\}^* : (X \approx^c Y \Rightarrow f(X) \approx^c f(Y))$$

Proof:
Let D be a distinguisher of these distributions and let $n \in \mathbb{N}$.
Let $D' = D \circ f$ (function composition).

$$X \approx^c Y \Rightarrow | Pr_{x \leftarrow X}[D'(x) = 1] - Pr_{x \leftarrow Y}[D'(x) = 1] | \leq neg(n)$$

$$\Rightarrow | Pr_{x \leftarrow X}[D(f(x)) = 1] - Pr_{x \leftarrow Y}[D(f(x)) = 1] | \leq neg(n)$$

$$\Rightarrow | Pr_{x \leftarrow f(X)}[D(x) = 1] - Pr_{x \leftarrow f(Y)}[D(x) = 1] | \leq neg(n)$$

So by definition, $f(X) \approx^c f(Y)$.

■

Semantic Security for Perfect Encryption

1

Claim:
For any perfectly secure encryption scheme (G, E, D), Advresery A, there exists an simulator S, such that for every message m: the distributions $S()$ and $A(E_k(m))$ are identical.

Proof:
Let $A : \Sigma^* \longrightarrow \Sigma^*$ (advresery).
Define $S : \varnothing \longleftarrow \Sigma^*$ (simulator): the simulator samples in the following manner:

- $m' \longleftarrow 42$.
- return $A(E_k(m'))$.

As we have seen in the tutorial, a definition for perfectly secure encryption is that for any two messages m, m': the distributions $E_k(m), E_k(m')$ are identical. This means that if we apply some (possibly stochastic) function A (the action of the advresery), These distributions are still the same. In other words: the distributions $A(E_k(m)), A(E_k(m'))$ are also identical. Now note that the latter distributions is actually the distribution of the simulator - directly from the simulator's definition.

■

Code Breaking

1 Crack The Code

The original text (spaces inserted):
"dear mr potter we are pleased to inform you that you have been accepted at hogwarts school of witchcraft and wizardry please find encloseda list of all necessary books and equipment term begins on september first we await your owl by no later than july thirty first yours sincerely miner vamcgonagal"

2 Explain The Algorithm

This is a ceaser cypher with a key of 64 on ascii encoding. This means that to encode a character: take it's ascii value, add 64 to it and convert it back to a character. Since the space of ascii values is bound with 256, any key larger than this would be the same as the same key mod 256. This means that the key space is with the size of 256.