Cryptography

Fundamentals

Prof. Dr. Heiko Knospe

March 11, 2024

Mathematical Fundamentals

Modern cryptography relies on mathematical structures and methods.

We briefly discuss a number of fundamental topics from discrete mathematics, elementary number theory, computational complexity and probability theory.

More advanced *algebraic structures* will be discussed in a separate chapter.

Sets

Sets are the most elementary mathematical structure. Finite sets play an important role in cryptography.

Example: $M = \{0,1\}^{128}$ is the set of binary strings of length 128. Elements in M can be written in the form $b_1 b_2 \ldots b_{128}$ or

$$(b_1, b_2, \ldots, b_{128})$$

in vectorial notation. An element of M could, for example, represent one block of plaintext or ciphertext data. The cardinality of M is very large:

$$|M| = 2^{128} \approx 3.4 \cdot 10^{38}$$

Small and Large Numbers

It is important to help understand the difference between small, big and inaccessible numbers in practical computations. For example, one can easily store one terabyte (10^{12} bytes, i.e., around 2^{43} bits) of data. On the other hand, a large amount of resources are required to store one exabyte (one million terabytes) or 2^{63} bits and more than 2^{100} bits are out of reach.

The number of computing steps is also bounded: less than 2^{40} steps (say CPU clocks) are easily possible, 2^{60} operations require a lot of computing resources and take a significant amount of time, and more than 2^{100} operations are unfeasible. It is for example impossible to test 2^{128} different keys with conventional (non-quantum) computers.

Functions

Definition

A *function* or a *map*

$$f : X \to Y$$

consists of two sets (a *domain* X and a *codomain* Y) and a rule which assigns an output element (an *image*) $y = f(x) \in Y$ to each input element $x \in X$. The set of all $f(x)$ is a subset of Y called the *range* or *image* $im(f)$. Let $y \in Y$. Any $x \in X$ with $f(x) = y$ is called a *preimage* of y. Let $B \subset Y$. Then we call

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\}$$

the *preimage* or *inverse image* of B under f.

Injective, Surjective and Bijective Maps

Definition

Let $f : X \to Y$ be a function.

- f is *injective* if different elements of the domain map to different elements of the range: if $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$. Equivalently, f is injective if for all $x_1,\ x_2 \in X$:

$$f(x_1) = f(x_2) \Rightarrow x_1 = x_2$$

- f is *surjective* or *onto* if every element of the codomain Y is contained in the image of f, i.e., for every $y \in Y$ there exists an $x \in X$ with $f(x) = y$. In other words, f is surjective if $im(f) = Y$.
- f is *bijective* if it is both injective and surjective. Bijective functions are invertible and possess an inverse map $f^{-1} : Y \to X$ such that $f^{-1} \circ f = id_X$ and $f \circ f^{-1} = id_Y$.

Residue Classes modulo n

Residue classes play an important role in cryptography.

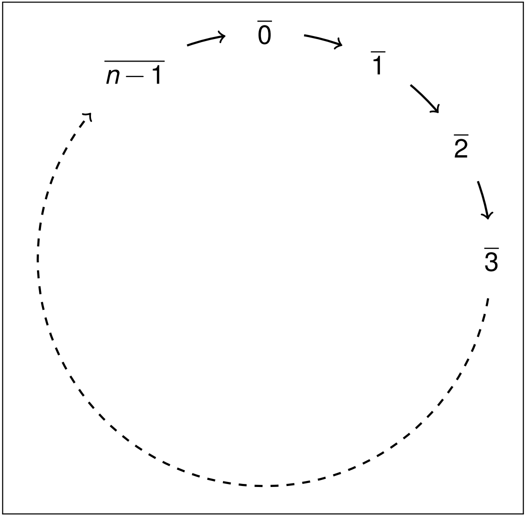
Let $n \in \mathbb{N}$ with $n \geq 2$. Then two integers $x, y \in \mathbb{Z}$ are called *congruent modulo n* if $n \mid x - y$, i.e. if the difference $x - y$ is divisible by n. This means that x and y have the same remainder when divided by n.

The residue class of $x \in \mathbb{Z}$ is the set

$$\overline{x} = \{\ldots, x - 2n, x - n, x, x + n, x + 2n, \ldots\}.$$

There are n *different* residue classes modulo n. The set the *residue classes modulo n* or *integers modulo n* is denoted by $\mathbb{Z}_n$ or $\mathbb{Z}/n\mathbb{Z}$. Each residue class has a *standard representative* in the set $\{0, 1, \ldots, n-1\}$ and one has $\mathbb{Z}_n = \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$.

Residue Classes modulo n



Example: $\mathbb{Z}_2$

$\mathbb{Z}_2 = \{\overline{0}, \overline{1}\}$ has only two elements. One has $\overline{-1} = \overline{1} = \overline{3}$ and $\overline{-2} = \overline{0} = \overline{2}$. The difference of two elements which are in the same class is divisible by 2 (i.e., their difference is even).

The standard representatives are 0, 1 and we have

$$\overline{0} = \{\dots, -4, -2, 0, 2, 4, \dots\},$$
$$\overline{1} = \{\dots, -3, -1, 1, 3, 5, \dots\}.$$

We may simple write 0 and 1 for these two classes.

XOR, AND, OR

Elements in $\mathbb{Z}_2$ can be added modulo 2, and addition is the same as the XOR operation on the bits 0 and 1. The multiplication modulo 2 corresponds to the AND ($\cdot$) operation.

$\oplus$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Table: *XOR and AND operations.*

The OR operation is given by $x\ OR\ y = x \oplus y \oplus x \cdot y$.

OR	0	1
0	0	1
1	1	1

Table: *OR operation*

Example: $\mathbb{Z}_{26}$

$\mathbb{Z}_{26} = \{\overline{0}, \overline{1}, \ldots, \overline{25}\}$ has 26 elements (i.e. there are 26 residue classes).

For example, one has $\overline{-14} = \overline{38}$, since $-14 - 38 = -52$ is a multiple of 26. The integers -14 and -38 are congruent modulo 26 and we write

$$-14 \equiv 38 \mod 26.$$

The standard representative of this residue class is 12 and

$$\overline{12} = \{\ldots, -14, 12, 38, 64, \ldots\}.$$

Computations with Residue Classes

Residue classes can be added, subtracted and multiplied. An arbitrary integer representative can be used, and it is reasonable to choose a small representative.

Examples: a) $79 - 180 \mod 26 \equiv 1 - 24 \equiv 1 + 2 = 3 \mod 26$.
b) $234577 \cdot 2328374 \cdot 2837289374 \mod 3 \equiv 1 \cdot 2 \cdot 2 \equiv 1 \mod 3$.

However, division is more tricky since **rational numbers $\frac{b}{a}$ are not representatives of residue classes**. We say that a is invertible modulo n if there exists $x \in \mathbb{Z}$ such that

$$ax \equiv 1 \mod n.$$

Then $x \equiv (a \mod n)^{-1}$.

Example: $(3 \mod 10)^{-1} \equiv 7$, since $3 \cdot 7 \equiv 1 \mod 10$.

Invertible Residue Classes

Proposition

An integer a is invertible modulo n if and only if $\gcd(a, n) = 1$*, i.e., if the greatest common divisor of a and n is* 1*.*

Example: 3 is invertible modulo 10, but 2 is not invertible modulo 10.

Definition

The invertible integers modulo *n* are called the units mod *n*. The subset of units of $\mathbb{Z}_n$ is denoted by $\mathbb{Z}_n^*$.

Example: $\mathbb{Z}_{10}^* = \{\overline{1}, \overline{3}, \overline{7}, \overline{9}\}$.

Prime Numbers

Definition

An integer $p \geq 2$ is called a prime number if p is only divisible by ± 1 and $\pm p$.

If p is prime, then

$$\mathbb{Z}_p^* = \{\overline{1}, \ldots, \overline{p-1}\}.$$

Prime numbers play an important role in public-key cryptography.
The *Prime Number Theorem* states that the density of primes among the first N integers is approximately

$$\frac{1}{\ln(N)}.$$

Euler's φ-Function

Definition

Let $n \in \mathbb{N}$. Then Euler's φ-function is defined by the cardinality of the units mod n, i.e.,

$$\varphi(n) = |\mathbb{Z}_n^*|$$

Examples: a) $\varphi(10) = 4$.
b) If p is a prime number, then $\varphi(p) = p - 1$.
c) If p and q are two different prime numbers, then

$$\varphi(pq) = (p-1)(q-1).$$

(Why?)

Extended Euclidean Algorithm

One of the key algorithms in elementary number theory is the *Extended Euclidean Algorithm*. The algorithm takes two nonzero integers a, b as input and computes $\gcd(a, b)$ as well as two integers $x, y \in \mathbb{Z}$ such that

$$\gcd(a, b) = ax + by.$$

The Extended Euclidean Algorithm is very efficient (also for large numbers) and can also be used to compute the multiplicative inverse of $a \mod n$. If $\gcd(a, n) = 1$ then the algorithm outputs $x, y \in \mathbb{Z}$ such that

$$1 = ax + ny.$$

Then

$$1 \equiv ax \mod n$$

and thus $x \equiv (a \mod n)^{-1}$.

Extended Euclidean Algorithm

Input: $a,\ b \in \mathbb{N}$
Output: $\gcd(a,b),\ x,\ y \in \mathbb{Z}$ such that $\gcd(a,b) = ax + by$
Initialisation: $x_0 = 1$, $x_1 = 0$, $y_0 = 0$, $y_1 = 1$, $sign = 1$

while $b \neq 0$ **do**
  $r = a \mod b$ // remainder of the integer division $a : b$
  $q = a/b$ // integer quotient
  $a = b$
  $b = r$
  $xx = x_1$
  $yy = y_1$
  $x_1 = q \cdot x_1 + x_0$
  $y_1 = q \cdot y_1 + y_0$
  $x_0 = xx$
  $y_0 = yy$
  $sign = -sign$
end while
$x = sign \cdot x0$
$y = -sign \cdot y0$
$gcd = a$
return $\mathrm{gcd},\ x,\ y$

Modular Exponentiation I

Modular exponentiation with a large basis, exponent and modulus plays an important role in cryptography. How can we efficiently compute

$$x^a \mod n\,?$$

If $a = 2^k$ then k-fold squaring modulo n gives the result:

$$x^a \mod n = ((((x^2 \mod n)^2 \mod n)^2 \mod n)^2 \ldots)^2 \mod n$$

For example, $x^{256} \mod n$ can be computed with only 8 squaring operations. After each squaring, one should reduce mod n in order to reduce the size of the (intermediate) result.

Modular Exponentiation II

If the exponent is not a power of 2, it can still be written as *a sum of powers* of 2. This gives a product of factors of type $x^{(2^k)} \mod n$, and each factor can be computed by k modular squarings. We call this the *Fast Exponentiation Algorithm*.

Example: Compute $6^{41} \mod 59$. We have $41 = 2^5 + 2^3 + 2^0$ and first compute the following sequence of squares:

$$
\begin{aligned}
6^2 &\equiv 36 \mod 59\\
6^4 &\equiv 36^2 \equiv 57 \mod 59\\
6^8 &\equiv 57^2 \equiv 4 \mod 59\\
6^{16} &\equiv 4^2 \equiv 16 \mod 59\\
6^{32} &\equiv 16^2 \equiv 20 \mod 59
\end{aligned}
$$

Then $6^{41} = 6^{32} \cdot 6^8 \cdot 6 \equiv 20 \cdot 4 \cdot 6 \equiv 8 \mod 59$.

Cardinality

Proposition

Let X and Y be finite sets of cardinality $|X|$ and $|Y|$, respectively. Then:

1. $|X \times Y| = |X| \cdot |Y|$ *and* $|X^k| = |X|^k$ *for* $k \in \mathbb{N}$.
2. *Suppose* $|X| = n$ *and* $k \leq n$. *Then the number of subsets of X of cardinality k is given by the binomial coefficient* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Example: There are $\binom{128}{2} = \frac{128 \cdot 127}{2} = 8128$ different binary words of length 128 with exactly two ones and 126 zeros.

Permutations

Definition

Let S be a finite set. A *permutation* of S is a bijective map $\sigma : S \to S$.

Example: The set $S = \{1, 2, 3\}$ has 6 permutations.

Proposition

Let S be a finite set and $|S| = n$. Then there are $n!$ permutations of S.

Note: the factorial increases very fast, for example

$$50! \approx 3.04 \cdot 10^{64}.$$

Permutations in Cryptography

Cryptographic operations often use permutations. A randomly chosen family of permutations (depending on a key) of a set such as $M = \{0,1\}^{128}$ would constitute an ideal block cipher. However, it is impossible to write down or store a general permutation of M since this set has 2^{128} elements. Much simpler (and much less secure) are *bit permutations*, which permute only the *position* of the bits.

Example: $(5\ 7\ 1\ 2\ 8\ 6\ 3\ 4)$ defines a permutation on $X = \{0,1\}^8$: a byte $(b_1, b_2, \ldots, b_8)$ is mapped to $(b_5, b_7, b_1, b_2, b_8, b_6, b_3, b_4)$. There are $8!$ bit permutations of X (a relatívely small number), but $(2^8)!$ general permutations (a very large number).

Big-O Notation

We often need to analyze the computational *complexity* of algorithms, i.e., the required resources (running time and space) as a function of the input size.

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ be two functions on $\mathbb{N}$. Then we say that g is an *asymptotic upper bound* for f, if there exists a real number $C \in \mathbb{R}$ and an integer $n_0 \in \mathbb{N}$ such that

$$|f(n)| \leq C\,|g(n)| \text{ for all } n \geq n_0.$$

One writes $f = O(g)$ or $f \in O(g)$.

Asymptotic Complexity: Examples

1. $f(n) = 2n^3 + n^2 + 7n + 2$. Since $n^2 \leq n^3$, $n \leq n^3$ and $1 \leq n^3$ for $n \geq 1$, one has $f(n) \leq (2+1+7+2)n^3$. Set $C = 12$ and $n_0 = 1$. Thus $f = O(n^3)$ and so f has cubic growth in n.
2. $f(n) = 100 + \frac{20}{n+1}$. Set $C = 101$ and $n_0 = 19$. Since $\frac{20}{n+1} \leq 1$ for $n \geq 19$, we have $f = O(1)$. Hence f is asymptotically bounded by a constant.
3. $f(n) = 5\sqrt{2^{n+3} + n^2 - 2n}$. Then $f = O(2^{n/2})$, and so f grows exponentially in n.

Complexity of Algorithms

Definition

If the running time of an algorithm is $f(n)$, where f is a *polynomial* and n is the input *size*, then the algorithm has *polynomial running time* and belongs to the complexity class **P**.

Polynomial-time algorithms are usually regarded as *efficient*. Problems that can be solved in polynomial time are *easy*. On the other hand, problems which cannot be solved by a polynomial-time algorithm are considered to be *hard*.

In computer science, one is usually interested in the *worst-case* complexity of algorithms. However, when looking at the complexity of attacks against cryptographic schemes, it is not sufficient that some instances are hard to attack. Instead, the *average-case* complexity of successful attacks is more important.

Complexity of Algorithms: Examples

- The functions in the above examples $\boxed{1}$ and $\boxed{2}$ are polynomial.
- The running time of the Extended Euclidean Algorithm on input $a, b \in \mathbb{N}$ is $O(\text{size}\,(a)\,\text{size}\,(b))$, so the algorithm is polynomial on the maximal input size.
- The running time of multiplying two numbers modulo n is $O(\text{size}\,(n)^2)$, which is polynomial.
- The running time of fast exponentiation modulo n is $O(\text{size}\,(n)^3)$, which is also polynomial.
- An algorithm which loops through $N = 2^n$ items has exponential running time in n.

Security Reduction

In complexity theory, a *reduction* is an algorithm which transforms one problem into another. Problem A *can be reduced in polynomial time to problem* B if A can be solved efficiently (i.e., in polynomial time) using an external algorithm for B as a sub-routine. We write

$$A \leq_p B,$$

i.e. there is a (polynomial-time) reduction from A to B. This means that A cannot be harder than B.

In cryptography, one often wants to show that a certain cryptographic problem B is hard. To this end, one gives a polynomial-time reduction from a well-known standard problem A to B. This shows that B is at least as hard as A. If we assume that A is a hard problem then the same holds true for B.

Example of a Security Reduction

Let p be a prime number, $g \in \mathbb{Z}_p^*$ and let a, b, c, d be secret integers between 1 and $p-1$. Suppose A is the following well-known computational Diffie-Hellman problem: given (g, p, g^a, g^b) find g^{ab}. All computations are modulo p.

Now let B be the following problem: given (g, p, g^c, g^d) find g^{c^2-cd} (modulo p). Is this new problem hard if c and d are unknown? To this end, we give a reduction from A to B:
Let g^a, g^b be given. We want to solve problem A. Set $x = g^a$ and compute $y = g^a(g^b)^{-1} = g^{a-b}$. This can be done in polynomial time. Now we use B as a sub-routine on input (g, p, x, y) and obtain

$$g^{a^2-a(a-b)} = g^{ab} \mod p.$$

Hence we have solved problem A. Note that we have not used the secret parameters! This shows that problem B is at least as hard as A.

Probability

We refer to textbooks on probability theory. We only consider *discrete probability spaces* and need the following notions:

- Probability space $(\Omega, \mathcal{S}, Pr)$, where Ω is a sample space, $\mathcal{S} = \mathcal{P}(\Omega)$ a set of events and $P : \mathcal{S} \to [0,1]$ a probability distribution.
- Independent events A, B, i.e., $P(A \cap B) = P(A) \cdot P(B)$, and mutually independent events $A_1, \ldots, A_n$.
- The conditional probability $P[A|B] = \frac{P(A \cap B)}{P(B)}$ of events A, B.
- Random variable $X : \Omega \to \mathbb{R}$.
- Probability mass function (pmf) $p(x) = P[X = x]$ of a random variable X.
- Expectation (expected value) $E[X] = \sum p(x)x$.
- Variance $V[X] = \sum p(x)(x - E[X])^2$.

Uniform Distribution and Random Bits

Definition

Pr has a uniform distribution if all elementary events have equal probability: $Pr[\{\omega\}] = \frac{1}{|\Omega|}$ for all $\omega \in \Omega$.

Uniform random bits (or random numbers) are quite important in cryptography (but difficult to generate).

Definition

A random bit generator (RBG) outputs a sequence of bits such that the corresponding random variables $X_1, X_2, X_3, \ldots$ satisfy

1. $P[X_n = 0] = P[X_n = 1] = \frac{1}{2}$ for all $n \in \mathbb{N}$ (uniform distribution), and
2. $X_1, X_2, \ldots, X_n$ are mutually independent for all $n \in \mathbb{N}$.

Negligible Functions

We need the notion of a *negligible* function in the context of the probability of successful attacks.

Definition

Let $f : \mathbb{N} \to \mathbb{R}$ be a function. We say that f is *negligible* in n, if $f = O(\frac{1}{q(n)})$ for all polynomials q, or equivalently, if $f = O(\frac{1}{n^c})$ for all $c > 0$.

Negligible functions are eventually smaller than any inverse polynomial. This means that $f(n)$ approaches zero faster than any of the functions $\frac{1}{n}, \frac{1}{n^2}, \frac{1}{n^3}, \ldots$

Example: $f(n) = 10e^{-n}$ and $2^{-\sqrt{n}}$ are negligible in n.
$f(n) = \frac{1}{n^2+3n}$ is not negligible, since $f(n) = O(\frac{1}{n^2})$, but $f \neq O(\frac{1}{n^3})$.

Birthday Paradox

Let $x_1, x_2, \ldots, x_n$ be a sequence in Ω. We say there is a *collision* if at least two elements are identical.

Proposition

Let Pr be a uniform distribution on a set Ω *of cardinality n. If we draw* $k = \left\lceil \sqrt{2\ln(2)n} \right\rceil \approx 1.2\sqrt{n}$ *independent samples from* Ω*, then the probability of a collision is around* 50%*.*

This fact is called *birthday paradox*: only $k = 23$ random birthdays (out of $n = 365$) are on average sufficient for a birthday collision.

For $|\Omega| = 2^n$, around $\sqrt{2^n} = 2^{n/2}$ independent samples probably give a collision.

Randomness and Entropy

In practice, true randomness is hard to achieve. If samples are not uniformly distributed, how can we measure the randomness in bits?

The *information entropy* measures the uncertainty of samples (e.g., messages, keys, passwords) in bits. The entropy of binary strings generated by a true random bit generator is equal to the length. In general, the entropy is less than the binary length.

Entropy

The notion of information entropy was introduced by Claude Shannon in his 1948 paper *A Mathematical Theory of Communication*.

Suppose Ω is a discrete probability space of messages with probability mass function $p(m)$. Then the *self-information* is a random variable I defined by

$$I(m) = \log_2\left(\frac{1}{p(m)}\right) = -\log_2(p(m)) \text{ for } p(m) > 0.$$

The expected value $E[I]$ defines the entropy $H(\Omega)$:

$$H(\Omega) = E[I] = \sum_{m\in\Omega} I(m)p(m) = -\sum_{m\in\Omega} p(m)\log_2(p(m))$$

The entropy measures the amount of uncertainty (in bits) and the average information content. If Ω has a uniform distribution and $|\Omega| = 2^n$ then $H(\Omega) = n$, the number of bits.

Examples

The following table shows the entropy per symbol for different symbol sets, if the symbols are uniformly distributed:

Symbol set	*Symbols*	*Entropy*
Octal number $(0-7)$	8	3
Arabic numeral $(0-9)$	10	3.32
Case sensitive latin alphabet (a−z, A−Z)	52	5.7
Case sensitive alphanumeric (a−z, A−Z, $0-9$)	62	5.95
ASCII printable character	95	6.57
An extended wordlist	$100,000$	16.61

For example, a randomly chosen password from a list of $100,000$ words has less than 17 bits entropy. With an appended number, 20 bits can be achieved. However, for short-term security against brute-force attacks at least 80 bits are needed.