Cryptography

Communicate securely

Outline

- Symmetric-key encryption and one-time pad
- Cryptographic hash functions
- Public-key cryptography w. modular arithmetic
- Fast exponentiation and discrete-log. problem
- Diffie-Hellman, ElGamal enc., Schnorr signature

Slides available at it-course.ch/Cryptography.pdf

Some topics covered at ef1p.com/number-theory

Introduction

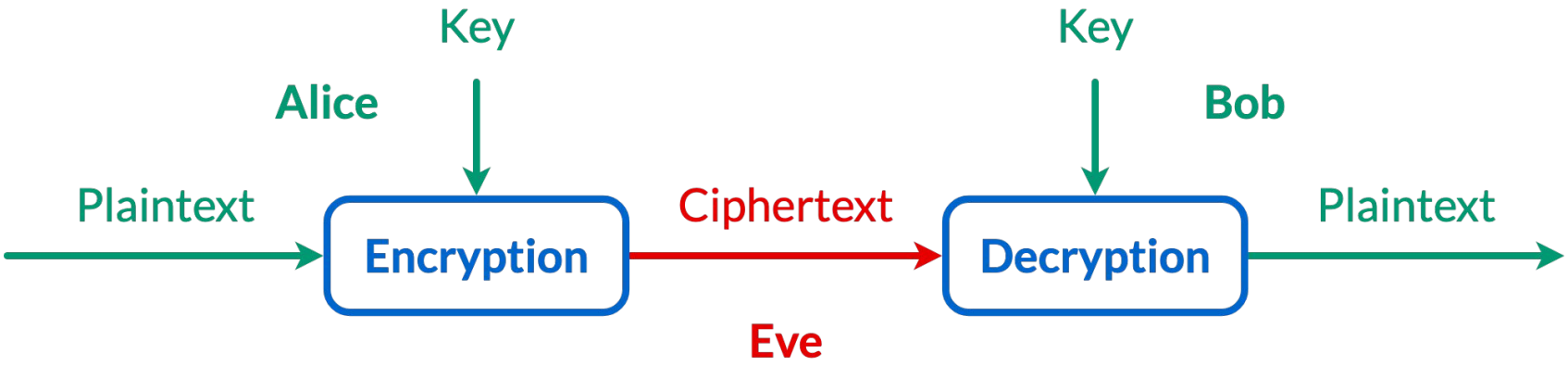
Cryptography from Greek κρυπτός (secret) and γράφειν (writing) is the science of secure communication in the presence of third parties.

For ages, encryption was more art than science.

Today based on computational hardness assumptions about problems like integer factorization and the discrete logarithm problem.

Computationally secure (hard to break) versus information-theoretically secure (can’t be broken).

Symmetric-key encryption



The sender and the receiver share the same key.

Only encr. method publicly known before 1976.

Prefix the plaintext with some random bytes to prevent the recognition of identical messages.

Perfectly-secure enc. (one-time pad)

The exclusive-or operator (XOR or $\oplus$):

It is commutative ($M \oplus K = K \oplus M$) and reversible (if $M \oplus K = C$, $C \oplus K = M$).

Bitwise application to 2 strings of bits:

M	0	1	0	0	1	1	0	1
$\oplus$								
K	1	1	1	0	1	0	0	1
=								
C	1	0	1	0	0	1	0	0

M ⊕ K = C		
0	0	0
0	1	1
1	0	1
1	1	0

Not practical: Key as long as message.

Perfectly secure because there's a K for every M.

Cryptographic hash functions

Preimage resistance:

Given y, infeasible to find x so that h(x) = y.

Collision resistance:

Infeasible to find distinct x and y so that h(x) = h(y).

Popular: SHA-256 (256-bit secure hash algorithm).

Hash functions have many interesting applications.

Public-key cryptography

Main problem with symmetric-key cryptography: Each pair of parties must share a different key or rely on a trusted third party to distribute one.

Public-key cryptography proposed in 1976: A pair of related keys (private key and public key), where it’s infeasible to derive the former from the latter.

Modular arithmetic

The modulo operator % returns the (positive) remainder of a division. For example: 8 % 3 = 2.

If two integers a and b have the same remainder modulo the same modulus m, we write $a =_m b$.

If $a_1 =_m a_2$ and $b_1 =_m b_2$, we have that

- $a_1 + b_1 =_m a_2 + b_2$
- $a_1 \cdot b_1 =_m a_2 \cdot b_2$

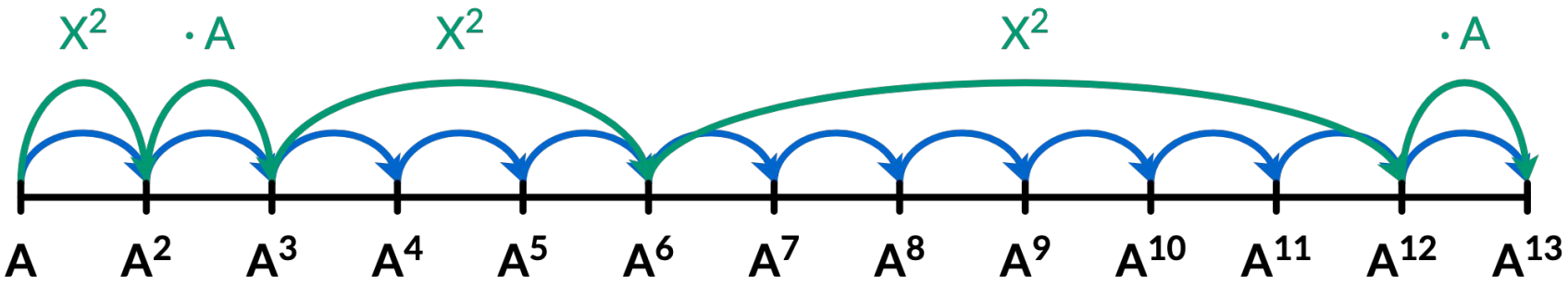
which can be seen by replacing $a_1 =_m a_2$ with $a_1 = a_2 + c_a \cdot m$ for some integer c_a (same with b).

Repeat an integer modulo a prime p

$A \cdot A =_p A^2$, $A^2 \cdot A =_p A^3$, $A^3 \cdot A =_p A^4$, and so on.

Instead of computing A^n in $(n - 1)$ steps, we can compute it faster by using recursion: If n is odd, compute $A^n =_p A^{n-1} \cdot A$. If n is even, $A^n =_p (A^{n/2})^2$.

This allows us to compute A^{13}, for example, in 5 instead of 12 steps (scales with bit length of n):



Discrete-logarithm problem (DLP)

Discrete-logarithm problem: Given $N =_p A^n$, no efficient algorithm is known to compute n from A and N (i.e. to determine how many times A has been repeated to get to N) for certain values of p.

Modular exponentiation is a **one-way function**.

This asymmetry has three important applications:

- Key exchange (or rather agreement)
- Asymmetric encryption
- Digital signatures

Diffie-Hellman key exchange

How to generate a secret over a public channel?

Alice	**Bob**
choose random $a < p-1$	choose random $b < p-1$
compute $A =_p G^a$	compute $B =_p G^b$
	$A \rightarrow$
	$\leftarrow B$
compute $K_A =_p B^a$	compute $K_B =_p A^b$

$$K_A =_p B^a =_p (G^b)^a =_p (G^a)^b =_p A^b =_p K_B$$

The shared key K can be used for symmetric enc.

ElGamal public-key encryption

Key generation:

- Bob chooses a prime p with a generator G
- Bob chooses $b < p - 1$ and computes $B =_p G^b$
- Bob publishes p, G, and B as his public key

Encryption:

- Alice chooses $a < p - 1$ and computes $A =_p G^a$
- Alice computes ciphertext as $C =_p M \cdot B^a$ and A

Decryption:

- Bob: $M =_p C \cdot A^{-b} =_p M \cdot B^a \cdot (G^a)^{-b} =_p M \cdot (G^b)^a \cdot (G^b)^{-a}$

How to compute a multipl. inverse?

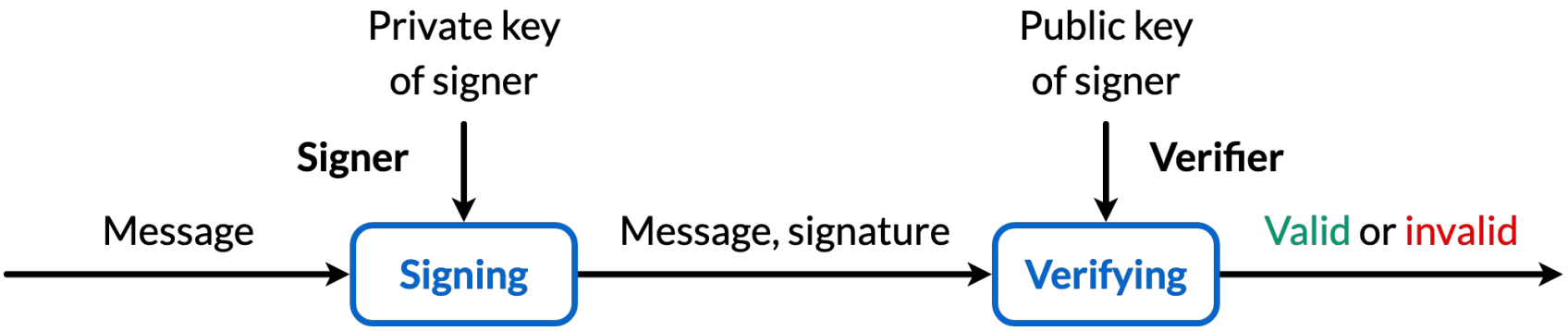
Use the extended Euclidean algorithm: Don't just subtract the numbers from each other, subtract whole equations from one another.

Example:

Quotient	Remainder	= … · 53	+ … · 25
	53	1	0
– 2 ·	25	0	1
– 8 ·	3	1	–2
– 3 ·	**1**	**–8**	**17**
	0	25	–53

Since $1 = (-8)\cdot 53 + 17\cdot 25$, $1 =_{53} 17\cdot 25$, $25^{-1} =_{53} 17$.

Digital signatures



Signatures are easy to produce and hard to forge.

Digital signatures must depend on the message.

For efficiency, sign only the hash of the message.

One use case: Have certificate authorities (CAs) sign certificates about who owns which public key.

Digital signature scheme

Digital signature schemes consist of 3 algorithms:

- **KeyGeneration**(entropy) → private k, public K (called key because you can unlock things like coins; k → K typically easy, K → k always hard)
- **Signing**(message, k) → signature (can only be produced by the person who knows the key k)
- **Verification**(message, K, signature) → true/false (anyone who knows the public key K can verify)

Computations with secret values

The idea behind many signature schemes is to use a linear one-way function f to hide the private key while still allowing the verifier to compute with it.

Example: You can compute $f(a \cdot b)$ if you know $f(b)$ and a without having to know b (s. Diffie-Hellman).

For the signature scheme discussed today, k is the private key and $K = G^k$ the public key.

A signature value s has to depend on the private key k and h = hash(message).

Zero-knowledge proofs

Goal: Convince another party of one's knowledge w/o revealing any information or leaving evidence.

Parties: Prover convinces verifier of knowing k. Trivial by revealing k but not zero-knowledge then.

- **Completeness** (successful proof): An honest verifier will be convinced by an honest prover.
- **Soundness** (proof of knowledge): Prover can fake knowledge only with negligible probability.
- **Zero-knowledge**: Verifier can fake transcript.

Knowledge of discrete logarithm

Prover		Verifier
knows k so that $K = G^k$		
choose random $r < \|G\|$		
compute $R = G^r$		
	$\xrightarrow{R}$	
		choose random $c < \|G\|$
	$\xleftarrow{c}$	
compute $s =_{\|G\|} r - k \cdot c$		
	$\xrightarrow{s}$	
		verify that $R = G^s \cdot K^c$

Evaluation of criteria

- **Completeness**: $G^r = G^{r - k \cdot c} \cdot (G^k)^c$.
- **Soundness**: By sending distinct challenges c and c' after the same R, the verifier can extract the secret k by computing $(s - s')/(c' - c)$ because $G^s \cdot K^c = R = G^{s'} \cdot K^{c'}$ and thus $G^s / G^{s'} = G^{s - s'} = K^{c'} / K^c = K^{c' - c}$.
- **Zero-Knowledge**: By choosing the random values c and s first, the verifier can compute $R = G^s \cdot K^c$, which results in a valid transcript.

Choice of the challenge

Since the verifier might choose the challenge c non-randomly, this protocol is only so-called **honest-verifier** zero-knowledge. A dishonest verifier can turn this into a signature scheme. (The verifier could commit to c beforehand.)

To be a proof of knowledge, the prover has to learn the challenge c **after** fixing the ephemeral key R. This dependency can be established either by a verifier or by a cryptographic hash function.

Schnorr signature scheme

Signer:

- Knows k so that $K = G^k$.
- Choose random $r < |G|$.
- Compute $R = G^r$, $c = \text{hash}(R, m, K)$, and $s =_{|G|} r - k \cdot c$ for message m.
- Share (c, s) as a signature.

Verifier:

- Verify whether $c = \text{hash}(G^s \cdot K^c, m, K)$.

Appendix of missing explanations

- $|G|$ is the so-called order of the generator G, which is the smallest integer n so that $G^n =_p 1$.
- Every integer A strictly between 0 and p is relatively prime to p, i.e. gcd(A, p) = 1.
- Extended Euclidean alg.: A^{-1} exists for every A.
- $A \cdot X =_p B$ has solution $X =_p A^{-1} \cdot B$ for every B.
- Given p – 1 possible X and B, solution is unique.
- Thus, A·X is a permutation of all possible values.
- $\prod X =_p \prod A \cdot X =_p A^{p-1} \cdot \prod X$ (product over all X).
- Cancel $\prod X$ on both sides: $1 =_p A^{p-1}$ for all A.