BASIC TRAINING ON CRYPTOGRAPHY FOR CTFs

16th October 2021

ENISA 2nd Bootcamp Team EU — Turin, Italy

Manuel Goulão and *Filipe Casal*
(manuel.goulao@tecnico.ulisboa.pt)

Introduction

Beginner to intermediate cryptography **CTF** challenges

Most **common cryptosystems** and hardness assumptions

Some **typical attacks** to each assumption/cryptosystem

~1h per topic, followed by a small break

Practical **exercises** with Python and SageMath

Brief overview of some other less predominant topics in CTFs

Contents

1. **Basics**
    a. Encodings
    b. Cryptography
    c. Attack models

2. **Stream and block ciphers**
    a. Length extension
    b. Meet-in-the-middle (Birthday paradox)
    c. Modes of operation (ECB, CBC, GCM...)

3. **Diffie-Hellman**
    a. Man-in-the-middle
    b. Quadratic residuosity
    c. Pohlig-Hellman

4. **RSA**
    a. Malleability
    b. Padding
    c. Overview of common attacks

5. **Elliptic curve cryptography**
    a. Repeated *k* (DSA)
    b. Invalid curve attack
    c. Curves with easy DLog (Smart, MOV)

6. **Further topics**
    a. Lattice reduction
    b. Modern cryptography
    c. Relevant tools

Outline of most topics: https://en.wikipedia.org/wiki/Outline_of_cryptography

Basics — encodings

Encodings

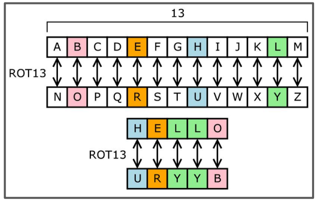
- Raw bytes
- Hexadecimal integer
- Base64
- …

```
>>> from Crypto.Util.number import bytes_to_long,long_to_bytes
>>> from base64 import b64encode
>>> pi = 31415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679
>>> long_to_bytes(pi)
b'\x05\xbe\xca\xc0\xca/\x1ar\xe7\xb7\x80\x1c\x1c\xdaY\xc5>x\x0f\xd8\xe5}\xe9[d\x9e\xa4~b\xbdL;\x9aE\xdb\xa1J7\xde\xefd\xcb'
>>> hex(pi)
'0x5becac0ca2f1a72e7b7801c1cda59c53e780fd8e57de95b649ea47e62bd4c3b9a45dba14a37deef64cb'
>>> b64encode(long_to_bytes(pi))
b'Bb7KwMovGnLnt4AcHNpZxT54D9jlfelbZJ6kfmK9TDuaRduhSjfe72TL'
>>>
```

Encoding π

Substitution ciphers

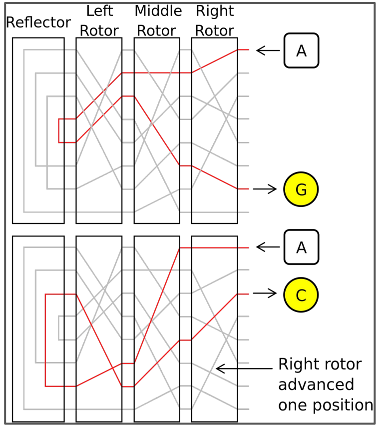
- Shift each character to another (Caesar cipher)
- Swap each character according to some key (Vigenère cipher)
- Pseudo-random substitution (Enigma machine)
- One-time-pad (Secure but not practical...)



ROT13 (Wikipedia)

Plaintext:	thequickbrownfoxjumpsoverthelazydog
Key:	LIONLIONLIONLIONLIONLIONLIONLIONLIO
Ciphertext:	EPSDFQQXMZCJYNCKUCACDWJRCBVRWINLOWU

Vigenère (Wikipedia)



Enigma (Wikipedia)

Basics — cryptography

One-way functions (candidates)

- Hash functions
- Discrete logarithm
- Factoring

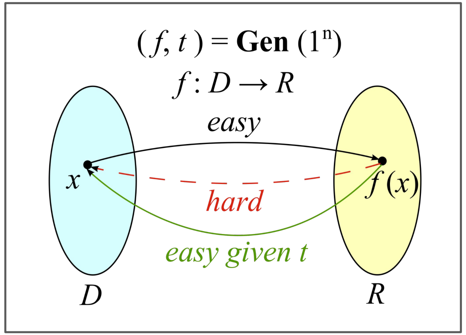
Symmetric-key cryptography

- Stream ciphers
- Block ciphers

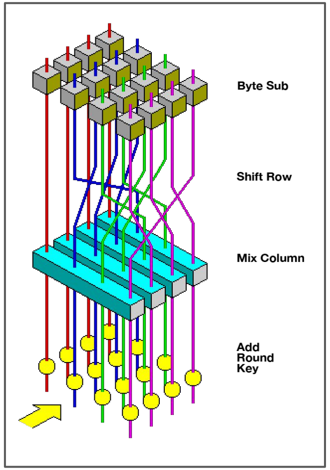
Public-key cryptography

- Key exchange
- Encryption scheme
- Signature scheme

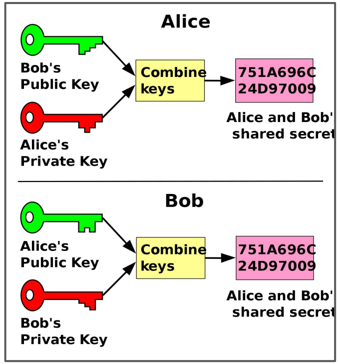
And so much more...



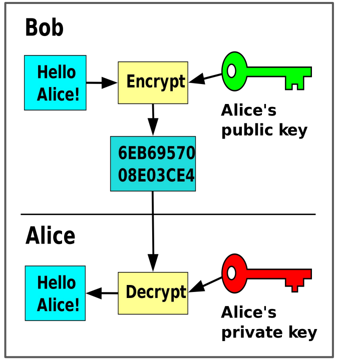
Trapdoor one-way function (Wikipedia)



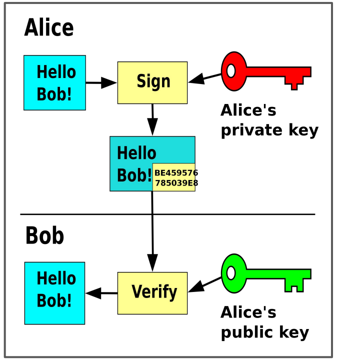
AES round function (Wikipedia)



Key exchange (Wikipedia)



Public-key encryption (Wikipedia)



Digital signature (Wikipedia)

Basics — attack models (for encryption schemes)

Ciphertext-only (COA)

- Access only to the ciphertext, no access to the plaintext.

Known-plaintext (KPA)

- Access to a number of pairs of plaintext and the corresponding ciphertext.

Chosen-ciphertext (CPA)

- Choose the plaintext to be encrypted, and receive the resulting ciphertext.

Chosen-plaintext (CCA)

- Choose arbitrary ciphertext, and have access to the plaintexts decrypted from it.

Adaptive chosen-ciphertext (CCA2)

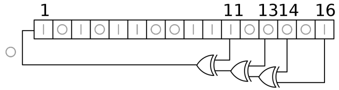
- Choose arbitrary ciphertext, and see the resulting plaintext. May use previous pairs to choose the next.

Other cryptographic schemes have different models, but follow the same setup:

"How much access to this cryptosystem do I have?"

And often, this kind of reasoning leads us to the right track to solve a challenge.

Stream and block ciphers — intro



$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

16-bit Fibonacci LFSR, circuit and feedback polynomial (Wikipedia)

$$X_{n+1} = (aX_n + c) \mod m$$

LCG recurrence (Wikipedia)

Hash functions

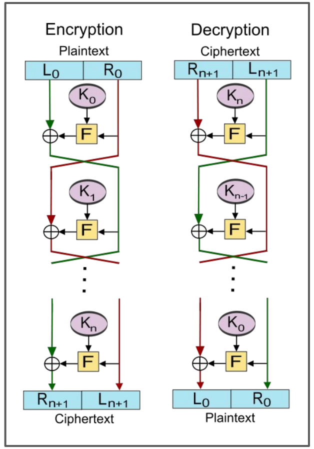
- Preimage resistance
  Given ***h***, hard to find ***m***, with ***h = HASH(m)***
- Second pre-image resistance
  Given ***m_1***, hard to find ***m_2***, with ***HASH(m_1) = HASH(m_2)***
- Collision resistance
  Hard to find ***m_1*** and ***m_2***, with ***HASH(m_1) = HASH(m_2)***

PRNGs

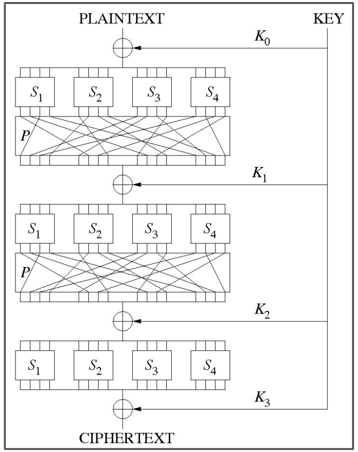
- **Mersenne Twister** — $2^{19937}-1$, 32-bit word length. Recovering internal state requires 624 32-bit outputs.
- **LFSR** — Berlekamp-Massey gives LFSR of minimal size.
- **LCG** — Lattice reduction attacks.

Block ciphers

- Feistel networks
- Substitution–permutation network



Feistel network (Wikipedia)



Substitution–permutation network (Wikipedia)

Stream and block ciphers — length extension

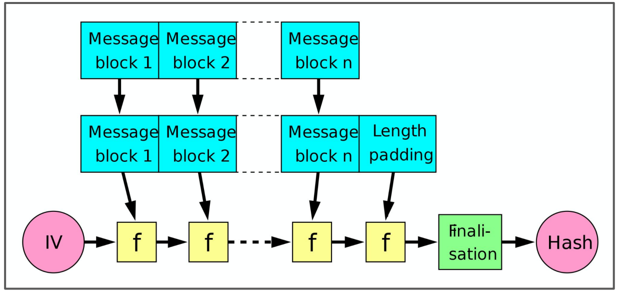
Use $\boldsymbol{H(m_1)}$ and the length of $\boldsymbol{m_1}$ to calculate $\boldsymbol{Hash(m_1 \mid m_2)}$ for an attacker-controlled $\boldsymbol{m_2}$, without needing to know the content of $\boldsymbol{m_1}$

MD5, **SHA-1** and **SHA-2** are susceptible to this kind of attack (Merkle–Damgård construction with a bad finalization function)

Reconstruct internal state from hash digest, then process the new data

The attack targets such hashes: ***HASH(key | message)***

Continue hashing to get the hash of ***(key | message | padding | more)*** one can choose ***more*** without any knowledge of the ***key***

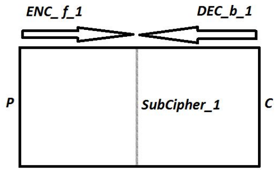


Merkle–Damgård construction (Wikipedia)

```
Original Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo
Original Signature: 6d5f807e23db210bc254a28be2d6759a0f5f5d99

Desired New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo&
waffle=liege

New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo\x80\x00\x00
          \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
          \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
          \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
          \x00\x00\x00\x02\x28&waffle=liege

New Signature: 0e41270260895979317fff3898ab85668953aaa2
```

Length extension example (Wikipedia)

Stream and block ciphers — meet-in-the-middle



1D meet-in-the-middle (Wikipedia)

Known plaintext attack, generic **space–time tradeoff**

Find keys by using both the range (ciphertext) and domain (plaintext)

Naive attack needs $\boldsymbol{2^{2k}}$ encryptions and ***O(1)*** space

MitM for key-size ***k*** uses only $\boldsymbol{2^{k+1}}$ encryptions/decryptions and $\boldsymbol{O(2^k)}$ memory

Prob. at least two people share birthday (Wikipedia)

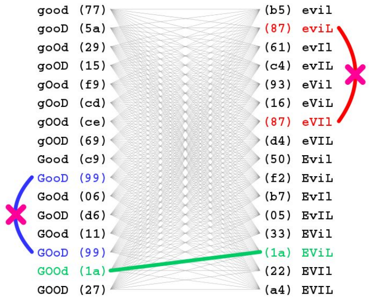
The attacker can compute $\boldsymbol{ENC^{k1}(P)}$ for all values of $\boldsymbol{k_1}$ and $\boldsymbol{DEC^{k2}(C)}$ for all possible values of $\boldsymbol{k_2}$ (total of $\boldsymbol{2^{k1} + 2^{k2}}$ operations)

If any of $\boldsymbol{ENC^{k1}(P)}$ matches a result from $\boldsymbol{DEC^{k2}(C)}$, the pair of $\boldsymbol{k_1}$ and $\boldsymbol{k_2}$ is possibly the correct key (can be checked with different plaintext-ciphertext pair)

$$C = ENC_{k_2}(ENC_{k_1}(P))$$
$$P = DEC_{k_1}(DEC_{k_2}(C))$$

$$C = ENC_{k_2}(ENC_{k_1}(P))$$
$$DEC_{k_2}(C) = DEC_{k_2}(ENC_{k_2}[ENC_{k_1}(P)])$$
$$DEC_{k_2}(C) = ENC_{k_1}(P)$$

Vulnerable system (Wikipedia)



Collisions between two sets (Wikipedia)

Stream and block ciphers — modes of operation

ECB — **identical plaintext** blocks into **identical ciphertext** blocks…

CBC (padding oracle) — allows decrypting but also encrypting

- Guess **last byte** of **last block**, use oracle to check padding
- Guess the one before that, and so on…
- $\boldsymbol{256 \cdot N}$ vs. $\boldsymbol{256^N}$ bytes
- Need block $\boldsymbol{C_{i-1}}$ to retrieve $\boldsymbol{C_i}$ ($\boldsymbol{C_1}$ requires knowing $\boldsymbol{IV}$)

GCM (forbidden attack)

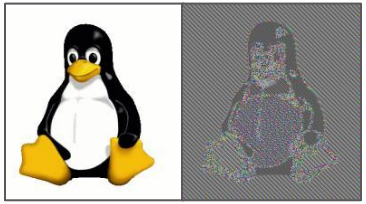
- Operations are done in $\mathbb{F}_{2^{128}}$
- ***Nonce/IV is reused*** in CTR (stream cipher)…
- Recover the authentication key $\boldsymbol{H}$ (depends on $\boldsymbol{K}$)

$$\boxed{\begin{aligned} T &= (C_1 * H + L) * H + AES(J_0) \\ T &= C_1 * H^2 + L * H + AES(J_0) \end{aligned}}$$

1st block of GCM
(BZDSJ16, presentation)

$$\boxed{\begin{aligned} T_1 &= C_{1,1} * H^2 + L_1 * H + AES(J_0) \\ T_2 &= C_{2,1} * H^2 + L_2 * H + AES(J_0) \\ \\ T_1 - T_2 &= (C_{1,1} - C_{2,1}) * H^2 \\ &\quad + (L_1 - L_2) * H \end{aligned}}$$

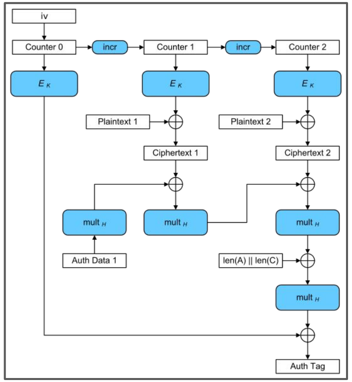
GCM nonce reuse
(BZDSJ16, presentation)



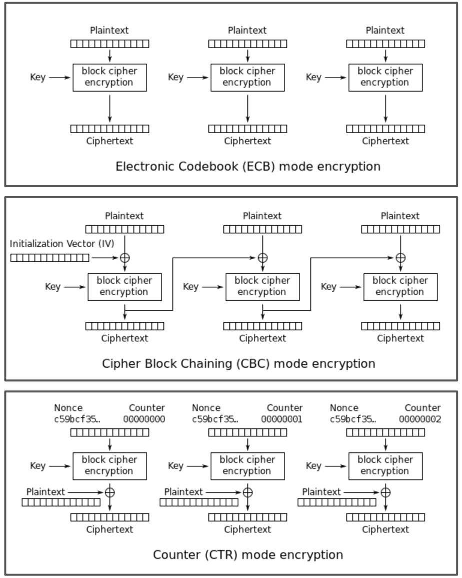
ECB encryption example (Wikipedia)

$$\boxed{\begin{aligned} P_i &= D_K(C_i) \oplus C_{i-1}, \\ C_0 &= IV. \end{aligned}}$$

CBC decryption (Wikipedia)



GCM AEAD mode (Wikipedia)



ECB, CBC, CTR modes (Wikipedia)

Stream and block ciphers — exercises

Diffie-Hellman — intro

Key-exchange — the parties compute shared (secret) key from only public information

DLog — easy to compute g^a *mod p*, hard to recover *a*

Here in a multiplicative group of integers *mod p*, but also for other finite cyclic groups where the DLog is hard (e.g. ECDH)

Key generation: private ***a***; public (***g^a mod p***)

No authentication (***man-in-the-middle*** attack)

How to choose ***p***?

Non-secret values in blue, and secret values in **red**.

1. Alice and Bob publicly agree to use a modulus $p = 23$ and base $g = 5$
2. Alice chooses a secret integer $\mathbf{a} = 4$, then sends Bob $A = g^{\mathbf{a}}$ mod p
   - $A = 5^{\mathbf{4}}$ mod $23 = 4$
3. Bob chooses a secret integer $\mathbf{b} = 3$, then sends Alice $B = g^{\mathbf{b}}$ mod p
   - $B = 5^{\mathbf{3}}$ mod $23 = 10$
4. Alice computes $\mathbf{s} = B^{\mathbf{a}}$ mod p
   - $\mathbf{s} = 10^{\mathbf{4}}$ mod $23 = 18$
5. Bob computes $\mathbf{s} = A^{\mathbf{b}}$ mod p
   - $\mathbf{s} = 4^{\mathbf{3}}$ mod $23 = 18$
6. Alice and Bob now share a secret (the number 18).

Both Alice and Bob have arrived at the same values because under mod p,

$$A^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p = B^a \bmod p$$

More specifically,

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$$

Example of DH for a very small *p* (Wikipedia)

Diffie-Hellman — man-in-the-middle

DH does not guarantee ***authentication***

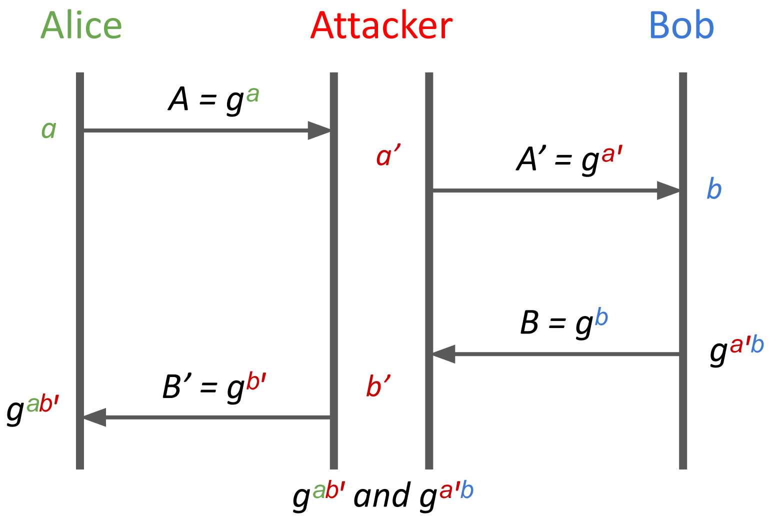
Attacker impersonates each party,
and communicates with the other on their behalf

Attacker shares one key with each party

Attacker relays all traffic,
decrypting and re-encrypting with the respective keys

Attacker can now read **EVERYTHING**

Alice and ***Bob*** are unaware of the ***Attacker***



Man-in-the-middle attack to DH

Diffie-Hellman — quadratic residuosity

Euler's criterion/Legendre symbol —
determine if an integer is a quadratic residue *mod p*

QR means congruent to a perfect square (***q = x² mod p***)

Legendre symbol of g^a reveals the *parity*, breaks **CPA**

Choose ***p*** with big subgroup of **quadratic residues**,
p = 2q + 1, ***q*** prime

g is then chosen to generate the order ***q*** subgroup
(usually *g = 2*)

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod{p} & \text{if there is an integer } x \text{ such that } a \equiv x^2 \pmod{p}, \\ -1 \pmod{p} & \text{if there is no such integer.} \end{cases}$$

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}.$$

Euler's criterion and Legendre symbol (Wikipedia)

$$\text{If } x = 2y, \quad g^{x^{(p-1)/2}} = g^{(x/2)(p-1)} = g^{(p-1)y} = g^{(p-1)^y} = 1^y = 1 \mod p$$

Parity leaking from Legendre symbol

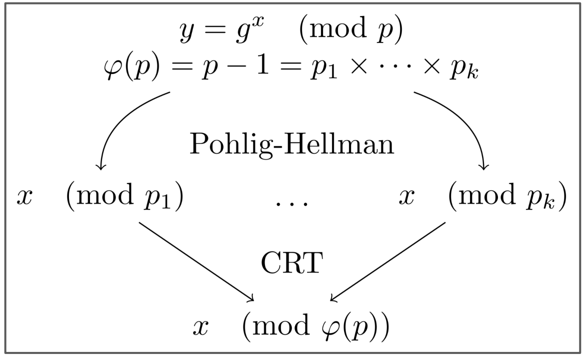
Diffie-Hellman — Pohlig-Hellman

Group order should have a **large prime factor** to prevent use of the Pohlig–Hellman algorithm

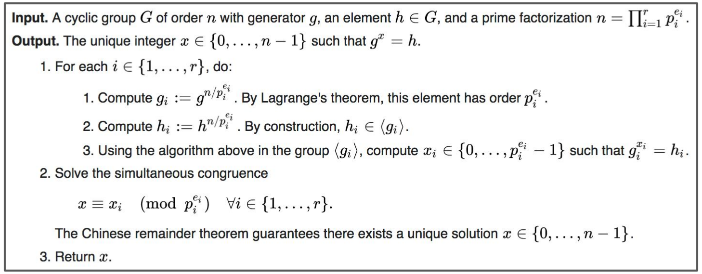
Choosing ***p = 2q + 1*** makes the order of the group only divisible by ***2*** and ***q***

Compute the **DLog** modulo **each prime in the group order**, use the CRT to combine them to the DLog in the full group

SageMath — sage.groups.generic.discrete_log



Pohlig-Hellman schematic (Wikipedia)



Input. A cyclic group G of order n with generator g, an element $h \in G$, and a prime factorization $n = \prod_{i=1}^{r} p_i^{e_i}$.

Output. The unique integer $x \in \{0, \dots, n-1\}$ such that $g^x = h$.

1. For each $i \in \{1, \dots, r\}$, do:
   1. Compute $g_i := g^{n/p_i^{e_i}}$. By Lagrange's theorem, this element has order $p_i^{e_i}$.
   2. Compute $h_i := h^{n/p_i^{e_i}}$. By construction, $h_i \in \langle g_i \rangle$.
   3. Using the algorithm above in the group $\langle g_i \rangle$, compute $x_i \in \{0, \dots, p_i^{e_i} - 1\}$ such that $g_i^{x_i} = h_i$.
2. Solve the simultaneous congruence

   $$x \equiv x_i \pmod{p_i^{e_i}} \quad \forall i \in \{1, \dots, r\}.$$

   The Chinese remainder theorem guarantees there exists a unique solution $x \in \{0, \dots, n-1\}$.
3. Return x.

Pohlig-Hellman pseudocode (Wikipedia)

Diffie-Hellman — exercises

RSA — intro

Encryption and **signature** scheme

Hard to factor product of two large prime numbers

Finding the ***e***-th roots of an arbitrary number, **mod *n***

Given the private exponent ***d*** one can efficiently factor ***n***

Given factorization of ***n***, one can obtain a private key

Key generation:

- Pick 2 random primes, ***p*** and ***q***, ***n = p ·q***
- Compute Euler totient function ***φ(n) = (p-1) ·(q-1)***, may also use Carmichael's totient function ***λ(n) = lcm(λ(p),λ(q))***
- Choose ***1 < e < φ(n)*** and ***gcd(e, φ(n)) = 1*** (usually *3* or *65537*)
- ***d ·e ≡ 1 (mod φ(n))***
- Public ***(n, e)***; private ***d***

$$m^e \equiv c \pmod{n}$$

Encryption of RSA

$$c^d \equiv (m^e)^d \equiv m \pmod{n}$$

Decryption of RSA

$$m^{ed} = m^{1+h\varphi(n)} = m(m^{\varphi(n)})^h \equiv m(1)^h \equiv m \pmod{n}$$

Correctness of RSA

$$h = \mathrm{hash}(m);$$
$$(h^e)^d = h^{ed} = h^{de} = (h^d)^e \equiv h \pmod{n}$$

Signing with RSA

RSA — malleability

$c_1 \cdot c_2 = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e \pmod N$

Blind signature — Alice obtains Bob's signature without Bob learning anything about the message

Decrypting a message by blind signing another message: never reuse key for encryption and signing

Last bit oracle (CCA) — decrypt an RSA ciphertext by having an oracle giving the parity of the plaintext

$$m' \equiv mr^e \pmod N$$
$$s' \equiv (m')^d \pmod N$$
$$s \equiv s' \cdot r^{-1} \equiv (m')^d r^{-1} \equiv m^d r^{ed} r^{-1} \equiv m^d r r^{-1} \equiv m^d \pmod N$$

RSA blind signing

1. Intercept $c = m^e$.
2. Send $2^e c$ to the parity oracle. $2^e c$ deciphers to $2m$.
3. If $2m$ is even, then $m \in [0, n/2)$,
4. Now iterate sending $(2^i)^e m$, use bisection to find m in logarithmic steps.
5. Next step — send $4^e c$ to the parity oracle, if it returns even, then $m \in [0, n/4) \cup [n/2, 3n/4)$.
6. Next step — send $8^e c$ to the parity oracle, if it returns even, then $m \in [0, n/8) \cup [n/4, 3n/8) \cup [n/2, 5n/8) \cup [3n/4, 7n/8)$.
7. ...

RSA last bit oracle

RSA — padding

RSA without padding (**textbook-RSA**) is **not CPA** secure (deterministic)

Malleability and **forgeability**

Compute ***e*-th root** of a small message ($m^e < N$, or too few laps)

Håstad's **broadcast** attack

Leak of **Jacobi** symbol

Padding must be random (**Coppersmith** — linear pad, short pad, …)

Padding oracle attack — **Bleichenbacher** Attacks (CCA)

$$c' = c \cdot s^e$$

Use padding oracle to find a valid *s*, for many s

00	02	padding string	00	data block

PKCS #1 block format for encryption (Bleichenbacher)

RSA — overview of common attacks

$$F(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

Find x_0 such that $F(x_0) \equiv 0 \pmod{M}$ for $|x_0| < M^{1/n}$

Coppersmith method
(sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots)

Low private exponent (**Wiener**)

- Uses the continued fraction method to find *d* when *d* is small ($d < (⅓)N^{1/4}$)

Low public exponent (**Coppersmith**)

- **Håstad's** broadcast — recover message encrypted to ≥*e* parties
- **Franklin-Reiter** related-message — recover two (known difference) related messages, encrypted for the same *N*
- **Coppersmith's** short-pad — recover a message sent twice with different (short) pad

Bad key generation

- **ROCA** (Coppersmith + Pohlig-Hellman):
  $M = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \ldots$ and $p = k \cdot M + (65537^a \mod M)$

Partial key exposure attack

Scheme	Secret information	Bits known	Technique	Section
RSA	$p \geq 50\%$ most significant bits		Coppersmith's method	§4.2.2
RSA	$p \geq 50\%$ least significant bits		Coppersmith's method	§4.2.3
RSA	p middle bits		Multivariate Coppersmith	§4.2.4
RSA	p multiple chunks of bits		Multivariate Coppersmith	§4.2.4
RSA	$> \log \log N$ chunks of p		Open problem	
RSA	$d \bmod (p-1)$ MSBs		Coppersmith's method	§4.2.7
RSA	$d \bmod (p-1)$ LSBs		Coppersmith's method	§4.2.7 and §4.2.3
RSA	$d \bmod (p-1)$ middle bits		Multivariate Coppersmith	§4.2.7 and §4.2.4
RSA	$d \bmod (p-1)$ chunks of bits		Multivariate Coppersmith	§4.2.7 and §4.2.4
RSA	d most significant bits		Not possible	§4.2.8
RSA	$d \geq 25\%$ least significant bits		Coppersmith's method	§4.2.9
RSA	$\geq 50\%$ random bits of p and q		Branch and prune	§4.3.1
RSA	$\geq 50\%$ of bits of $d \bmod (p-1)$ and $d \bmod (q-1)$		Branch and prune	§4.3.2
(EC)DSA	MSBs of signature nonces		Hidden Number Problem	§5.2
(EC)DSA	LSBs of signature nonces		Hidden Number Problem	§5.2
(EC)DSA	Middle bits of signature nonces		Hidden Number Problem	§5.2
(EC)DSA	Chunks of bits of signature nonces		Extended HNP	§5.2.4
EC(DSA)	Many bits of nonce		Scales poorly	
Diffie-Hellman	Most significant bits of shared secret g^{ab}		Hidden Number Problem	§6.2
Diffie-Hellman	Secret exponent a		Pollard kangaroo method	§6.3
Diffie-Hellman	Chunks of bits of secret exponent		Open problem	

Key recovery methods for public-key cryptosystems
(Gabrielle De Micheli, Nadia Heninger)

RSA — exercises

Compute MSBs of d:

$$ed = 1 + k\phi$$

$$d' = \frac{kN+1}{e}$$

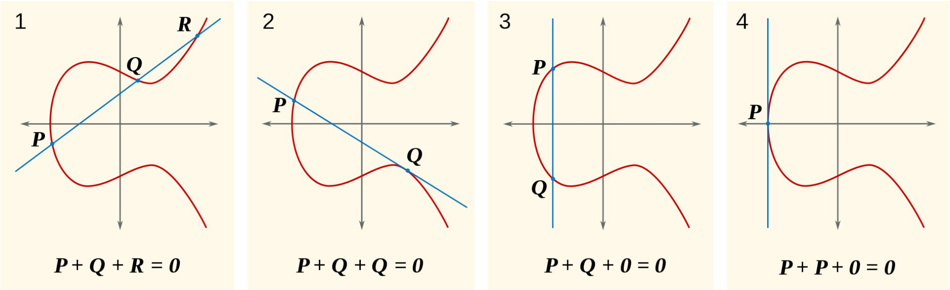
Bruteforce $k < e$.

Factoring given d:

1. Compute $e \cdot d$
2. $ed - 1 = k\phi \implies k \approx \frac{ed-1}{N} \implies \phi = \frac{ed-1}{k}$
3. $(p-1)(q-1) = \phi \implies p + q = N + 1 - \phi$
4. Solve $x^2 + (p+q)x + N = 0$

Elliptic curve cryptography — intro

- Elliptic curves over finite fields ***E(𝔽p)***
- Set of points satisfying $y^2 = x^3 + ax + b$ (Weierstrass equation) with the group operation, and **point at infinity** as identity
- Must be **non-singular** *Δ ≠ 0*, $\Delta = -16(4a^3 + 27b^2)$
- Best attacks (in "well-chosen" curves) are **generic attacks**
- **Point multiplication** can be done using *double-and-add*



EC group law (Wikipedia)

Point at infinity

$$\mathcal{O} + \mathcal{O} = \mathcal{O}$$
$$\mathcal{O} + P = P$$

Point negation

$$P + (-P) = \mathcal{O}$$
$$(x, y) + (-(x, y)) = \mathcal{O}$$
$$(x, y) + (x, -y) = \mathcal{O}$$
$$(x, -y) = -(x, y)$$

Point addition

$$P + Q = R$$
$$(x_p, y_p) + (x_q, y_q) = (x_r, y_r)$$

Assuming *E*, is given by $y^2 = x^3 + ax + b$:

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$
$$x_r = \lambda^2 - x_p - x_q$$
$$y_r = \lambda(x_p - x_r) - y_p$$

Point doubling

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

EC point operations (Wikipedia)

Elliptic curve cryptography — repeated *k* (DSA)

ECDSA

- Public parameters (curve, generator)
- Private key (*d*)
- Public key (*Q = d×G*)
- Secret nonce (*k*):

  if ***k* is repeated** for different signatures, one can **solve for *d***

Solve for d given two signatures (r, s), (r, s'):

Compute k from

$$s - s' = k^{-1}(z - z') \implies k = \frac{z - z'}{s - z'} \pmod n$$

Since $s = k^{-1}(z - rd)$,

then $d = \frac{sk-z}{r}$.

Repeated *k* attack

1. Calculate $e = \mathrm{HASH}(m)$.
2. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.
3. Select a **cryptographically secure random** integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.
5. Calculate $r = x_1 \mod n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \mod n$. If $s = 0$, go back to step 3.
7. The signature is the pair (r, s). (And $(r, -s \mod n)$ is also a valid signature.)

ECDSA signing (Wikipedia)

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$.
3. Let z be the L_n leftmost bits of e.
4. Calculate $u_1 = zs^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$.
5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
6. The signature is valid if $r \equiv x_1 \pmod n$, invalid otherwise.

ECDSA verification (Wikipedia)

$$C = u_1 \times G + u_2 \times Q_A$$
$$C = u_1 \times G + u_2 d_A \times G$$
$$C = (u_1 + u_2 d_A) \times G$$
$$C = (zs^{-1} + rd_A s^{-1}) \times G$$
$$C = (z + rd_A)s^{-1} \times G$$
$$C = (z + rd_A)(z + rd_A)^{-1}(k^{-1})^{-1} \times G$$
$$C = k \times G$$

ECDSA correctness (Wikipedia)

Elliptic curve cryptography — Invalid curve attack

- Adding points on ***E(𝔽p)*** do not consider coefficient ***b***
- ECs over ***𝔽p*** whose Weierstrass equation differs only in ***b*** have the same addition laws
- ***Attacker*** selects an invalid curve ***E'*** such that ***E'*** contains a point ***R*** of small order
- ***Victim*** computes ***K = dR***
- ***Attacker*** solves the DLog to recover ***d*** from ***K*** since the point ***R*** has low order

Point at infinity

$$\mathcal{O} + \mathcal{O} = \mathcal{O}$$
$$\mathcal{O} + P = P$$

Point negation

$$P + (-P) = \mathcal{O}$$
$$(x, y) + (-(x, y)) = \mathcal{O}$$
$$(x, y) + (x, -y) = \mathcal{O}$$
$$(x, -y) = -(x, y)$$

Point addition

$$P + Q = R$$
$$(x_p, y_p) + (x_q, y_q) = (x_r, y_r)$$

Assuming E, is given by $y^2 = x^3 + ax + b$:

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$
$$x_r = \lambda^2 - x_p - x_q$$
$$y_r = \lambda(x_p - x_r) - y_p$$

Point doubling

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

EC point operations (Wikipedia)

Elliptic curve cryptography — Easy DLog (Smart, MOV)

Smart's attack:

- Compute **DLog in linear time**
- Curves with trace of Frobenius equal to one
- I.e. when ***#E = p*** (***p*** is the order of the field)
- In *Sage* simply check if:

```
sage: E = EllipticCurve(GF(p), [a, b])
sage: E.order() == p
True
```

MOV attack:

- **Bilinear pairing**: function e that maps two points in $E(\mathbb{F}p)$ to an element in $\mathbb{F}p^k$, k is the embedding degree of E
- **DLog of** rP — compute $u=e(P,Q)$, $v=e(rP,Q)$ for any Q. From bilinearity, $v=e(P,Q)^r=u^r \Rightarrow$ **DLog in** $\mathbb{F}p^k$
- Usually, the embedding degree k is large, but for some curves it is small (supersingular curves $k \leq 6$)
- Embedding degree is the smallest $k \geq 2$ such that the order of the curve divides $p^k - 1$

```
sage: E = EllipticCurve(GF(p), [a, b])
sage: E.is_supersingular() # if true, k≤6, or just compute it:
sage: k = 1
sage: o = E.order()
sage: p = E.base().order()
sage: while not o.divides(p^k-1):
sage:     k += 1
```

Elliptic curve cryptography — exercises

Further topics — Lattice reduction

Lattice:

Subgroup of the additive group $\mathbb{R}^n$ (isomorphic to the additive group $\mathbb{Z}^n$) and which spans the real vector space $\mathbb{R}^n$.
I.e., for a basis of $\mathbb{R}^n$, the subgroup of **all linear combinations with integer coefficients** of the basis vectors forms a lattice.

$$\Lambda = \left\{ \sum_{i=1}^{n} a_i v_i \;\middle|\; a_i \in \mathbb{Z} \right\}$$

Lattice Λ (Wikipedia)

Write a problem with a solution as a “**short**” lattice vector, and use lattice reduction

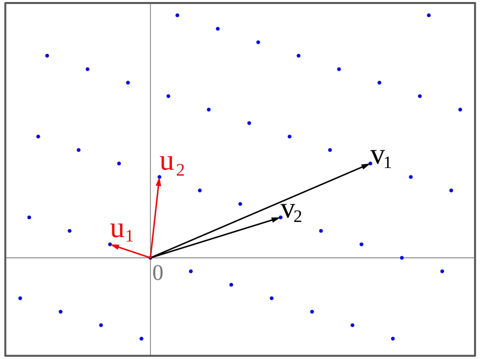
LLL — use rounded Gram-Schmidt coefficients (only integer linear combinations)

BKZ — generalizes LLL, solves SVP for lower dimension (parameter) blocks

Given an **integer lattice basis** as input, find a basis with short, nearly orthogonal vectors

Algorithms are included in *Sage*

Challenges — Subset sum, linear system with error, linear congruential generator, …



Lattice reduction in two dimensions (Wikipedia)

Further topics — Modern cryptography

- Zero-knowledge
- Secret sharing
- Threshold signatures
- E-voting
- Cryptocurrency
- Secure multiparty computation
- Homomorphic encryption
- Indistinguishability obfuscation
- Post-quantum cryptography (Lattice, Code, Multivariate, Hash, Isogeny based cryptography)
- Quantum cryptography (QKD, teleportation, Superdense coding, quantum money, …)

Further topics — Relevant tools

- PyCryptodome — https://pypi.org/project/pycryptodome
- SageMath — https://sagemath.org
- CyberChef — https://gchq.github.io/CyberChef
- Cryptogram solver — https://quipqiup.com
- Vigenère breaker — https://github.com/hellman/xortool
- Mersenne Twister PRNG cracker — https://github.com/icemonster/symbolic_mersenne_cracker
- Hash length extension attacks — https://github.com/bwall/HashPump
- Factors database — http://factordb.com
- Factorization calculator — https://www.alpertron.com.ar
- Bivariate Coppersmith — https://github.com/ubuntor/coppersmith-algorithm
- Multivariate Coppersmith — https://github.com/defund/coppersmith
- RSACtfTool — https://github.com/Ganapati/RsaCtfTool