Cryptography: Diffie-Hellman & RSA

CyberChallenge 2025

Maxim Kovalkov

Diffie-Hellman Key Exchange

The problem

We already know...

- symmetric ciphers
  → how Alice and Bob can protect their communications with a *shared secret key*

But... if Alice has a key, how does she get it to Bob?
By sending it over an encrypted channel, of course!

...you see the problem with that. There is no encrypted channel yet, that's the point!

The problem

So, it is inevitable that keys must be exchanged some other way. (Alice and Bob have to meet in person.)
...with *symmetric* ciphers of course

If there are N different people who want to communicate securely
...there is no choice but to have *everyone* meet with *everyone else* (pairwise) at least once
...and everyone must store $N-1$ keys, one for each interlocutor!
...the system creates $\frac{N(N-1)}{2} = O(N^2)$ distinct keys

This becomes quite unwieldy.

"Obvious" solution

Alice and Bob, and everyone in the Cryptography Kingdom, **trust** a central authority to manage the keys for all users.

All users remember one key to talk with the **Trusted Third Party (TTP)**.

Any time two users want to talk with each other, they ask the TTP to generate a random key.

This key is sent over a *secure channel* (because there are shared secret keys k_A and k_B for links TTP<->A and TTP<->B respectively)

"Obvious" solution... is "obviously bad"?

A TTP certainly has its uses — especially for smaller networks.
But there are problems:

- The TTP must be online and available at all times
- The TTP better be really *trusted*, and users' trust hinges also on how unlikely it is for the TTP to be *hijacked/compromised*
- But the TTP is also a *huge target*, as a hacker would be able to read *everyone's* messages

Key exchange over *insecure* channels?

It is actually possible to exchange keys **without** any trusted third party.

This is a problem solved by key exchange protocols, famously **Diffie-Hellman**.

...but to understand it, we need some number theory.

Diffie-Hellman

- Alice and Bob can publicly (!) choose a prime integer p and a generator $g \in \{2, 3, \ldots, p-1\}$
- Alice generates a random a and Bob generates a random b, such that $a, b \in \{2, 3, \ldots, p-1\}$
- Alice sends the value $A = g^a \mod p$ to Bob. Bob sends $B = g^b \mod p$

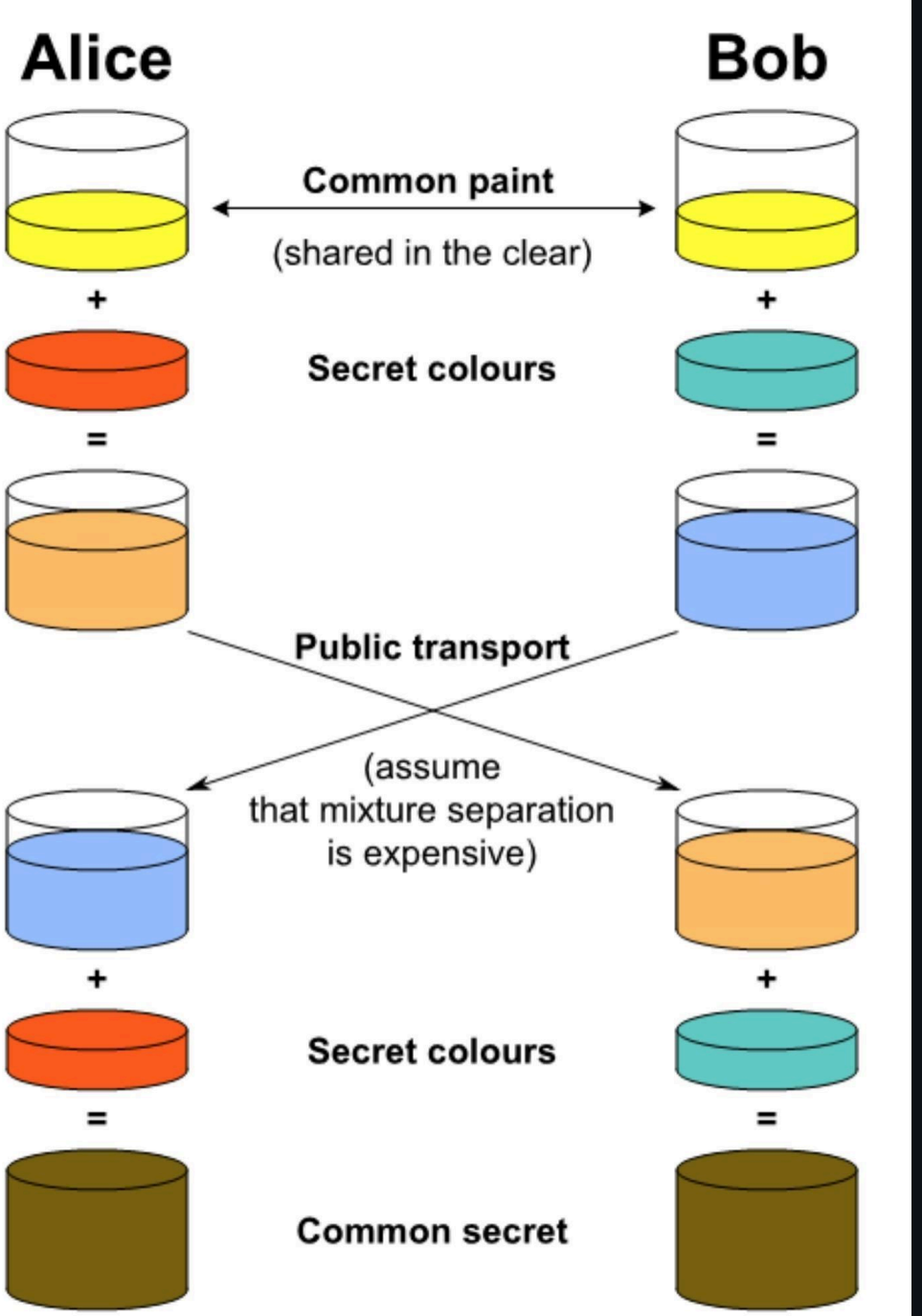
Diffie-Hellman

At this point...

- Alice knows a and B, so she can calculate $B^a = g^{ab} \mod p$
- Bob knows b and A, so he can calculate $A^b = g^{ab} \mod p$

This means we can pick g^{ab} as the shared secret!

Eve the eavesdropper **can't** get to g^{ab}, as she only has g^a and g^b!
(I know what you're thinking... $g^a \cdot g^b = g^{a+b}$, not g^{ab}!)


Alice
Bob
Common paint
(shared in the clear)
+
Secret colours
=
Public transport
(assume
that mixture separation
is expensive)
+
Secret colours
=
Common secret

Breaking Diffie-Hellman

Eve would need to recover a, given the values $A = g^a \mod p$, g, and p (or analogously with b)

She would need to compute an answer to the question "g to what exponent gives me A" — a **discrete logarithm**

We can not use the "actual" log function as we are in the realm of *modular arithmetic*!

This is a hard problem, with the best known algorithm being $O\left(e^{\sqrt[3]{n}}\right)$

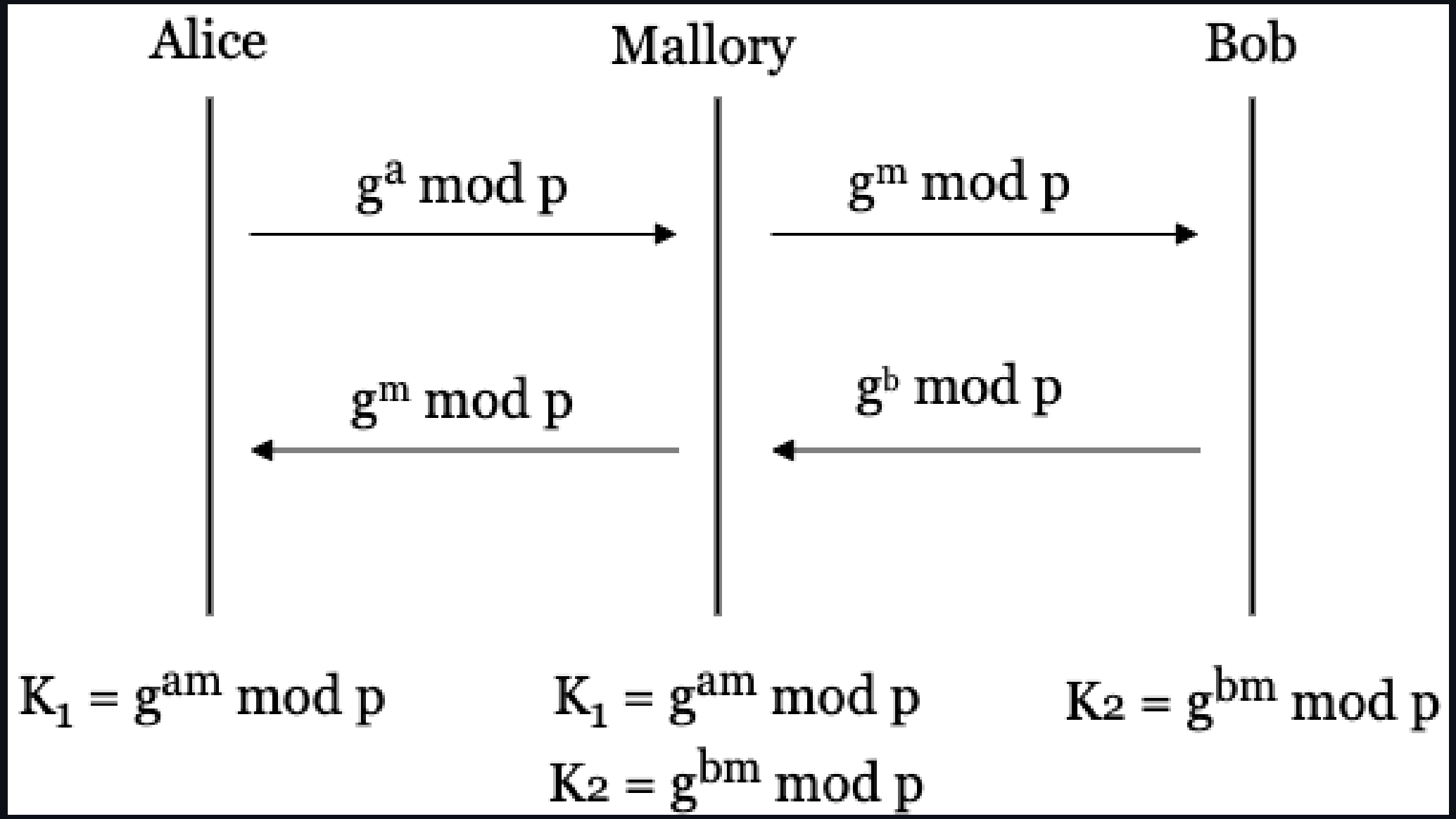
Breaking Diffie-Hellman

Diffie-Hellman	Equivalent
p of 1024 bits	$\approx$ 80-bit block cipher
p of 3072 bits	$\approx$ 128-bit b.c. (AES-128!)
...	
256-bit DH over Elliptic curves	$\approx$ AES-128 again

Man-in-the-Middle

Diffie-Hellman is vulnerable to a **MitM** attack

- If Eve can *interfere* with the communication channel, she can perform all the steps of DH, but *twice*: getting k_{AE} and k_{EB}.
- She can decrypt any message from A with k_{AE}, read it, then encrypt it with k_{EB} (& vice versa)
- But Alice thinks she is talking directly with Bob! (& vice versa)


Alice
Mallory
Bob
$g^a \bmod p$
$g^m \bmod p$
$g^m \bmod p$
$g^b \bmod p$
$K_1 = g^{am} \bmod p$
$K_1 = g^{am} \bmod p$
$K_2 = g^{bm} \bmod p$
$K_2 = g^{bm} \bmod p$

RSA

Recap

We have:

- *symmetric* ciphers, for when we can agree on a key beforehand
- *symmetric* ciphers + *key exchange*, to share the key without meeting in person

Now let's talk about:

- public key (*asymmetric*) encryption

Public key encryption

There are two keys:

- a *private key*, used for **decrypting** and **signing** messages
- a *public key*, used for **encrypting** messages and **verifying** signatures

Solves the problems of:

- *key distribution*: how to not trust all other users or a central authority
- *authentication*: verify that a message truly comes from the declared sender

Remember! The same *algorithm* is used for encrypting & decrypting, but *two different keys*!

Public key encryption

My public key is known by everyone, my private key is kept secret.

Of course, the private key should not be easily *derivable* from the public one. (One-way functions are very useful!)

Being able to use *both* keys complementarily for encryption/decryption lets us:

- guarantee **confidentiality** by *encrypting* with *public* key
- guarantee **authentication** by *encrypting* with *private* key
- even guarantee **integrity** with a clever mix of the above

RSA

Let's take a look at the most famous asymmetric algorithm.

Not unlike DH, it is based on *number theoretic* properties

In *very short* summary, the one-way operation is **exponentiation modulo N**

- cheap to compute (true of *modulus* exponentiation!)
- very hard to go back, requires **prime factorization** (slow!!)

Recall...

$$\varphi(n) = \text{“\# of positive integers up to } n \text{ that are coprime with } n\text{”}$$
$$= \#(\{x \in \mathbb{N} : x < n \wedge \gcd(x, n) = 1\})$$

Calculating $\varphi(n)$ is tricky, in general... but what is important to remember:

1. $\varphi(p) = p - 1;$ $\qquad\qquad$ $\varphi(ab) = \varphi(a)\varphi(b)$
2. $n = p \cdot q \quad \Longrightarrow \quad \varphi(n) = (p-1) \cdot (q-1)$
3. nothing is new under the sun 😉 (libraries!!!)

Some more Number Theory...

(Fermat-)Euler Theorem

If a and n are coprime, then

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

What this means (if you stare at the page a bit trying to make sense of the proof) is that

$$\text{if } a^x \equiv a^y \pmod{n}, \quad \text{then } x \equiv y \pmod{\varphi(n)}$$

RSA key generation

1. We randomly choose two *primes*: p, q
2. We compute $n = p \cdot q$, our modulus
3. We compute $\varphi(n) = (p-1)(q-1)$
4. Choose any $e > 1$, such that e is less than $\varphi(n)$ and shares no factors with it. e is our **public key**.
5. Find d by solving the equation: $e \cdot d \equiv 1 \pmod{\varphi(n)}$ (more on *how* later!). This is our **private key**.

RSA usage

We have $k_{pub} = \{e, n\}$ and $k_{priv} = \{d, n\}$.

M is the **plaintext** as a single integer.

My friend uses my public key e to compute $C = M^e \mod n$.

C is the **ciphertext** and gets sent to me.

I receive C and compute $C^d \mod n$, getting back M!

In other words, $M \equiv C^d \equiv (M^e)^d \pmod{n}$

RSA usage

Why is that true?

$(M^e)^d \equiv M^{e \cdot d} \equiv M^1 \equiv M \pmod{n}$

because $e \cdot d \equiv 1 \pmod{\varphi(n)}$, as we specifically picked such e and d

$$M^{e \cdot d} \equiv M^1 \pmod{n} \iff e \cdot d \equiv 1 \pmod{\phi(n)}$$

RSA in practice

- Specific values of e are used for greater efficiency:
  - exponents should be *smaller* and with few 1's in binary
  - 3 is the smallest but sometimes less secure
  - 65537 $(2^{16} + 1)$ is very common

Python

Lucky for us, Python gained very useful built-ins for our purposes:

- `int` type is of *arbitrary* length by default!
- efficient 3-argument pow builtin `pow(base, exp, modulus)`
- also computes the *modular inverse* if exponent is -1

```
from Crypto.Util.number import getStrongPrime
p, q = getStrongPrime(1024), getStrongPrime(1024)
n, phi = p*q, (p-1)*(q-1)
e, d = 65537, pow(65537, -1, phi)
def encrypt(pt): return pow(pt, e, n)
def decrypt(ct): return pow(ct, d, n)
```

Some extras

- Remember that many modern instances of RSA use *fixed* values of e to speed up encryption, like $65537 = (1.0000.0000.0000.0001)_{\text{bin}}$
  - easy enough – square 16 times & multiply: $((m^2)^{2\cdots})^2 \cdot m$
- We are not as lucky with d, since we don't pick it
- But we can use more Number Theory magic to go up to 4 times faster

Some extras

These steps are proven by the **Chinese Remainder Theorem** – another tool of NT (`sympy.ntheory.modular.crt()` if you ever need to solve CRT problems directly!)

$$
\begin{aligned}
&(\text{assume } p > q) \\
d_p &= d \mod (p-1) \\
d_q &= d \mod (q-1) \\
q_{\text{inv}} &= q^{-1} \mod p
\end{aligned}
\qquad
\begin{aligned}
m_1 &= c^{d_p} \mod p \\
m_2 &= c^{d_q} \mod p \\
h &= q_{\text{inv}} \cdot (m_1 - m_2) \mod p \\
m &= m_2 + h \cdot q \mod n
\end{aligned}
$$

We will not go into further detail: try it yourself, search the internet, find the formulae in the official CCIT slides or on Wikipedia

RSA — Attacks

Be careful with the modulus

In essence, all of RSA's security rests on the **difficulty of factoring $n = pq$**

...thus we use numbers of 1024+ bits to make brute-force *unfeasible*

i will link to software/services for when you need to factor directly (spoiler: rarely)

We must also **avoid (unintentionally) making it easy again**, through

- using *small* primes
- *reusing* the same prime
- using primes that are *close together*

Fermat factorization

What's wrong with primes that are close together?

If we express p, q as $p = a + b$ and $q = a - b$ we obtain a b that is small

$$N = pq = (a - b)(a + b) = a^2 - b^2$$

$$a^2 - N = b^2$$

Now we can look for **square numbers** (a^2) **close to** N,
and check if the difference between them is a small square.

This is called *Fermat's factorization method*, and might successfully brute-force a modulus

Common primes

What's wrong with sometimes reusing the same prime?

The answer is pretty straightforward.

- Suppose we generated two public keys $N_1 = p \cdot q,\ N_2 = p \cdot r$
- Remember from earlier explanations that `gcd(a, b)` is fast!
- We get $p = \gcd(N_1, N_2)$ and then divide to get q and r

If the target generates many keys, and prime recycling is likely, we can *keep testing pairs of moduli*; if we get a $\gcd(N_i, N_j) \neq 1$, then we broke both keys i and j

Common modulus

What's wrong with reusing the entire modulus?
...nothing, potentially! ;) but we must not use different exponents (rare in practice)

Our friend *Bezout's identity* comes to the rescue, aided by *extended GCD*

- Suppose we used keys $(N, e_1), (N, e_2)$ to encrypt the same plaintext: c_1, c_2
- Suppose that we get $\mathrm{xgcd}(e_1, e_2) = (1, u, v)$: this means $u \cdot e_1 + v \cdot e_2 = 1$ (we are assuming the GCD is 1)
- Now the magic trick: ${c_1}^u \cdot {c_2}^v \equiv m^{e_1 u} \cdot m^{e_2 v} \equiv m^1 \equiv m \pmod{n}$

We have now revealed the plaintext!

Oracle scenarios

Now let's check out some

Encryption oracle

We are given an encryption oracle, with a fixed secret key.
Can we recover the modulus?

- Choose messages x and y; ask the oracle for encrypted values C_x and C_y
- $C_x = x^e \mod N \Leftrightarrow C_x + k_1 N = x^e$; similarly $C_y + k_2 N = y^e$
- Compute "*actual*" values of x^e, y^e
- Compute $\gcd(x^e - C_x, y^e - C_y) = \gcd(k_1 N, k_2 N) = t \cdot N$

If you're lucky, t is either 1 or small (= brute-forceable); and you can always repeat the process

Decryption oracle 1

We are given a decryption oracle, which can decrypt any message but will *refuse* if it detects "*sensitive data*"

We can **trick** the oracle into decrypting the "forbidden" message $c \equiv m^e$

- Encrypt a chosen message x to get $(x^e \mod N)$
- Send the message $c' = (x^e \cdot c \mod N)$ for the oracle to decrypt
- The oracle will perform $c'^d \equiv (x^e m^e)^d \equiv (xm)^{ed} \equiv xm$
  $\rightarrow xm \neq m$ so it's not in the oracle's "blacklist": the result is sent back
- Now we just have to divide (multiply by the inverse) to recover m

Illustrates **homomorphic property** of RSA: $\mathsf{Dec}(a \cdot b) = \mathsf{Dec}(a) \cdot \mathsf{Dec}(b)$

Decryption oracle 2

We can access a *limited portion* of the decryption: only the *least significant bit*

Can we recover the plaintext? Yes — also bit by bit!

- Given an even a and an odd N, consider $r = a \bmod N$. If r is odd, then a "*overflowed*" ($a > N$), otherwise it didn't ($a < N$)
- Decrypting $2^e c$ and checking the LSB (even/odd) tells us whether $2m$ overflowed modulo N
  $\Leftrightarrow$ whether $0 < m \leq N/2$ or $N/2 < m < N$
  $\Leftrightarrow$ what is the *most significant bit* of m
- Repeat this with $4^e, 8^e, \ldots$ and recover the rest of the message

Small public exponent

Finally, let's see the reasons for why the exponent $e = 3$ has been replaced by 65537.

1. **Short message m + small exponent 3**: if $m^3 < N$ then it's as if the modulus were never there; we can just compute the (exact) cube root of m^3
2. Hastad's **Broadcast attack**: Same message sent to 3 recipients with pub.keys $N_1 < N_2 < N_3$.
   Chinese Remainder Theorem gives us $m^3 \bmod N_1 N_2 N_3$.
   Since $m < N_1$, then also $m^3 < {N_1}^3 < N_1 N_2 N_3$, so we can take the cube root like in case 1!

Implementation attacks

Our target is no longer the *mathematical structure* of RSA but its concrete implementation

Timing attacks

The fast exponentiation algorithm loops over the *bits* of the exponent,
and takes *more time* if the bit is a 1.

If we can detect such timing differences (side channel attack!) we can extract private exponents!

Fault injections

While decrypting with CRT method, a **glitch** in **one** of the two decrypted parts lets us factor the modulus! (very specific! see Boneh et al., 1997)

Useful resources

Training

Keep solving the official CCIT challenges! They follow material covered in these slides.

CryptoHack.org for more Cryptography challenges (with tutorial sections & increasing difficulty)

Utilities

- Once again recommending **CyberChef.org** for quick'n'dirty data encoding and transformations
- General Number Field Sieve, most efficient factorization algorithm available at CADO-NFS: http://cado-nfs.gforge.inria.fr/
- FactorDB.com for *known* factorizations
  `factor` function in SageMath
  YAFU software for facoring numbers
- Python libraries: some highlights include
  `sympy.integer_nthroot(x, n) -> (n_root_x: int, is_exact: bool)`
  `sympy.ntheory.totient`, `sympy.ntheory.crt`

That's all, folks!

CyberChallenge.IT 2025 — Cryptography 2 — Maxim Kovalkov