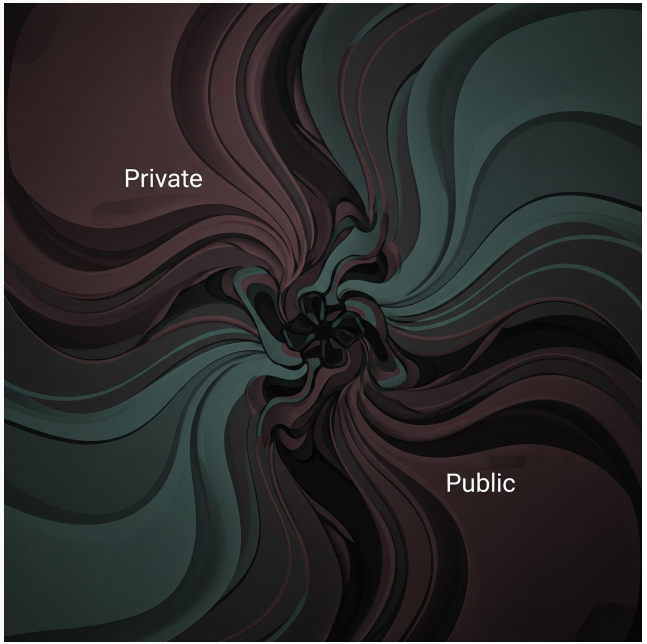
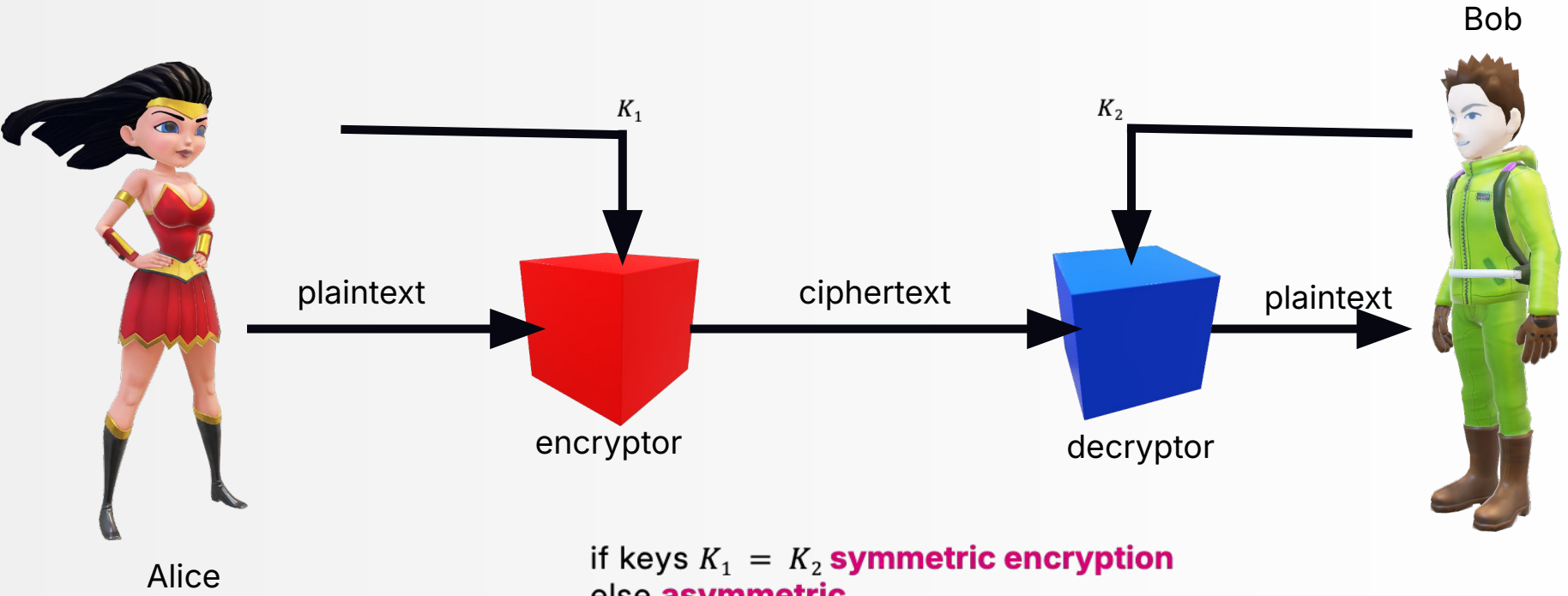
Asymmetric encryption: the Diffie-Hellman intuition



Reminder about the model



if keys $K_1 = K_2$ **symmetric encryption**
else **asymmetric**

Another definition

- **Key exchange** is the process of establishing a shared symmetric key between two parties over an insecure communication channel
    - classic challenge in enabling private communication between remote parties

- Asymmetric cryptography provides a solution to this problem by enabling secure key exchange mechanisms

Public-key encryption

A relevant example of asymmetric encryption
- when $K_1 \neq K_2$

is the public-key encryption

purposes
- integrity and non-repudiation
- key exchange
    - main example that requires confidentiality

Diffie and Hellman [1976] "New Directions in Cryptography"

Split the Bob's secret key K into two parts

- K_E to be used for encrypting messages to Bob
- K_D to be used for decrypting messages by Bob
- K_E can (must) be made public
    - essence of public key cryptography a type of asymmetric cryptography

Number of keys

Today we say that a pair of keys (private, public) is associated with each actor

- linear number of asymmetric keys
- quadratic number of symmetric keys

“New Directions in Cryptography”

- The Diffie-Hellman paper [IEEE Transactions on Information Theory, vol. IT-22, Nov. 1976] generated lot of interest in crypto research, both in academia and private industry http://www-ee.stanford.edu/%7Ehellman/publications/24.pdf
- Diffie & Hellman produced the revolutionary idea of public key cryptography, but did not have a proposed implementation (this came up two years later with Merkle-Hellman and Rivest-Shamir-Adelman)
- **In their '76 paper, Diffie & Hellman did invent a method for key exchange over insecure communication lines, a method that is still in use today**

Excerpt from RSA paper

- Historical paper: "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems R.L. Rivest, A. Shamir, and L. Adleman", Communications of the ACM 21, 1978 https://people.csail.mit.edu/rivest/Rsapaper.pdf
- ***The era of electronic mail may soon be upon us**; we must ensure that two important properties of the current paper mail system are preserved: (a) messages are private, and (b) messages can be signed. We demonstrate in this paper how to build these capabilities into an electronic mail system.*
- *At the heart of our proposal is a new encryption method. This method provides an implementation of a public-key cryptosystem, an elegant concept invented by Diffie and Hellman [1]. Their article motivated our research, since they presented the concept but not any practical implementation of such a system. Readers familiar with [1] may wish to skip directly to Section V for a description of our method.*

Meaning of encryption

Alice **encrypts**

- by Bob's public key
  - although public it must be known
  - everybody can use Bob's public key
  - confidentiality
- by Alice's private key
  - only Alice can use it
  - everybody can decrypt
  - no confidentiality, but path to non-repudiation

Encryption behaviour

- What is encrypted by a public key can be decrypted by the associated private key
- What is encrypted by a private key can be decrypted by the associated public key
    - This happens for RSA (later)
- This perfect symmetry causes the frequent use of the term encryption for both (classic) encryption and decryption

Introduction to one-way functions

One-way functions

A function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is called a **one-way function** (OWF) if it is

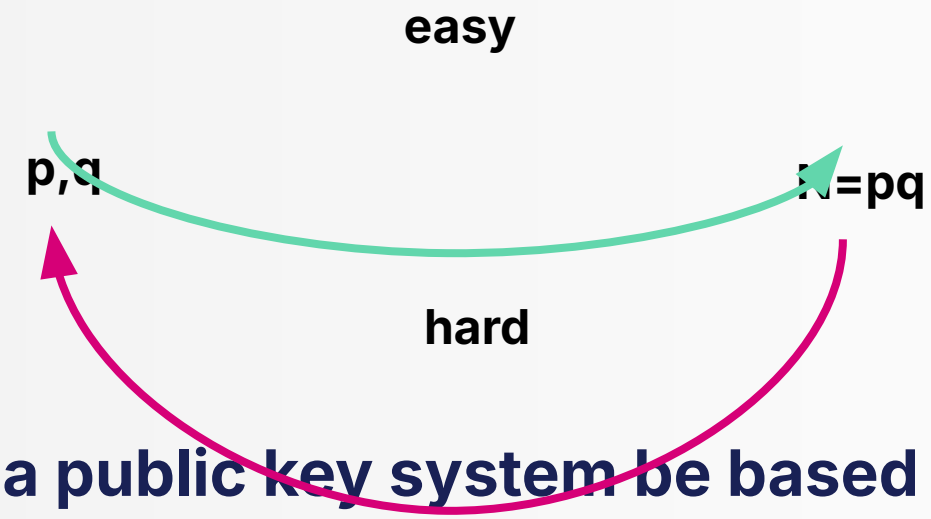
- **Efficiently computable**
  - There exists a deterministic polynomial-time algorithm that, given any input $x \in \{0,1\}^*$, computes $f(x)$
- **Hard to invert on average**
  - For every probabilistic polynomial-time algorithm A, for every positive polynomial $p(\cdot)$ and for sufficiently large n

$$\Pr_{x \leftarrow \{0,1\}^n}\left[A\left(f(x)\right) \in f^{-1}\left(f(x)\right)\right] < \frac{1}{p(n)}$$

where the probability is over the uniform random choice of $x \in \{0,1\}^n$ and the internal randomness of A

cryptographic hashing functions are OWF

Integer multiplication & factoring as an OWF



Q.: Can a public key system be based on this observation ?

Integer multiplication

- given x, y, two large prime numbers of roughly equal bit-length (say, n bits each)
- computing z =xy, 2n bits, is easy
  - many poly algs
- no poly alg is know for factoring z
  - Shor is poly on quantum computers

Discrete Log (DL)

- Let G be a finite cyclic group of order n, and let g be a generator of G
- For any element $y \in G$, there exists an integer x such that
- $y = g^x$
- The smallest nonnegative integer x satisfying this equation is called the **discrete logarithm of y to the base g**
- We write this as $x = \log_g(y)$
- Example: in the multiplicative group of integers modulo p, $y = g^x \bmod p$, with g a generator of $\mathbb{Z}_p^*$

Modular exponentiation

- $c \bmod p = (a \cdot b) \bmod p = ((a \bmod p) \cdot (b \bmod p)) \bmod p$
  - https://en.wikipedia.org/wiki/Modular_arithmetic#Properties
- $b^e \bmod p = (b \bmod p)^{e \bmod \varphi(p)} \bmod p$
  - p prime and (b,p) co-prime
  - https://en.wikipedia.org/wiki/Euler%27s_theorem
- can use mod on partial results

Fast mod exponentiation

```
long fastModExp(unsigned int base, unsigned int exp, long mod) {
    long result = 1;
    long b = base % mod;
    while (exp > 0) {
        if (exp & 1)
            result = (result * b) % mod;
        b = (b * b) % mod;
        exp >>= 1;
    }
    return result;
}
```

Discrete Log in $\mathbb{Z}_p^*$: a

- Let $y = g^x \bmod p$ in $\mathbb{Z}_p^*$
  - e.g., $\mathbb{Z}_{10}^* = \{1,3,7,9\}$ (p is not necessarily prime)
- $g^x \bmod p$ can be computed in O(log x) multiplications, each multiplication taking O(log^2 p) steps (remind $g < p$)
  - also, it turns out x must be $< p \implies$ overall # of steps is O(log^3 p)
- Standard discrete log is believed to be computationally hard
- $x \to g^x \bmod p$ is believed to be an OWF
  - $x \to g^x \bmod p$ is easy (efficiently computable)
  - $g^x \bmod p \to x$ believed hard (computationally infeasible)
- This is a computation-based notion

Do OWFs exist?

- open problem
  - they are conjectured to exist
- **Theorem**. If a one-way function exists, then P ≠ NP (theoretical proof)
  - one-way functions are conjectured to exist, as P is conjectured to be strictly included in NP

Utility of OWFs

- Cryptographic hashing (and its uses), secure key exchanges, public key cryptography, random numbers...
- More generally, assuming OWFs exist, we obtain strong mathematical properties that are computationally **hard to break**, forming the foundation of many cryptographic guarantees

Diffie-Hellman key exchange

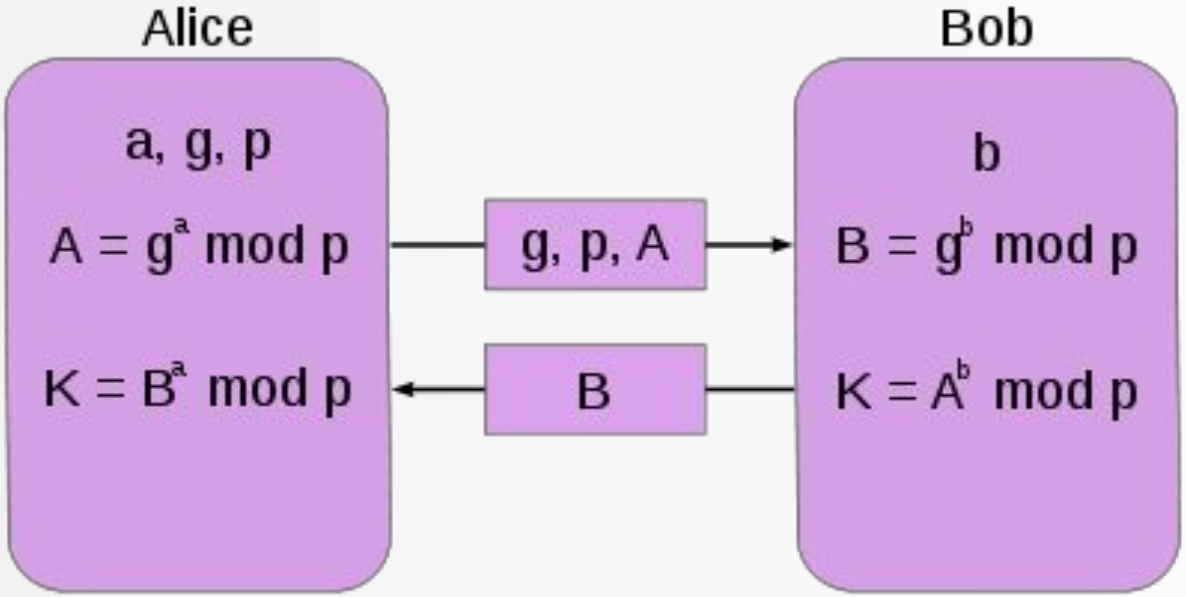
Public exchange of keys

- Goal: two parties (Alice and Bob) who do not share any secret information, perform a protocol and derive the same shared key
- Eve, who is listening, cannot obtain the new shared key if she has limited computational resources (i.e. in polynomial time)
    - unless P = NP

DH procedure

- Public parameters: a prime p, and an element g (possibly a generator of the multiplicative group $\mathbb{Z}_p^*$)
  - better a safe prime, i.e., a prime $p = 2q + 1$ where q is also prime (a Sophie Germain prime)
- Alice chooses a at random from the interval $[1..p-1]$ and sends $g^a \bmod p$ to Bob
- Bob chooses b at random from the interval $[1..p-1]$ and sends $g^b \bmod p$ to Alice
- Alice and Bob compute the shared key $g^{ab} \bmod p$
  - Bob holds b, computes $(g^a)^b mod\ p = g^{ab} \bmod p$
  - Alice holds a, computes $\left(g^b\right)^a mod\ p = g^{ab} \bmod p$

DH procedure – visual



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Terminology

- **Public parameters** (g and p)
- **Private keys** (a and b): they are secret
- **Public keys** ($A = g^a \bmod p$ and $B = g^b \bmod p$): public
- **Shared secret sey** ($K = g^{ab} \bmod p$)

DH and MITM

1. A→T(B): $g^a \bmod p$
2. T(B)→A: $g^t \bmod p$ *(t chosen by T)*
3. T(A)→B: $g^t \bmod p$
4. B→T(A): $g^b \bmod p$

Effects

- B shares a key $g^{bt} \bmod p$ with T (believing to share it with A)
- A shares a key $g^{at} \bmod p$ with T (believing to share it with B)

Perfect forward secrecy

The cryptosystem

- generates random public keys per session for the purposes of key agreement, and
- does not use any sort of deterministic algorithm in doing so

The compromise of a shared key does not compromise former, or subsequent, shared keys

Types of DH

Feature	Static Diffie-Hellman	Ephemeral Diffie-Hellman (DHE/ECDHE)
Key Lifespan	Uses a long-term key pair that remains the same across sessions (typically on the server side)	Generates a new, temporary key pair for each communication session
Mathematical Basis	Either modular exponentiation or elliptic curves (more advanced)	Either modular exponentiation or elliptic curves (more advanced)
Primary Security Property	Lacks Forward Secrecy	Provides Forward Secrecy
Primary Use Case	Used in specific scenarios where long-term key management is required	The standard for modern secure communication

Vulnerabilities

Vulnerability	Description	Impact	Mitigation
Man-in-the-Middle	No authentication in basic DH	Attacker can intercept and alter key	Use authenticated DH
Logjam Attack	Exploits small (e.g., 512-bit) shared primes	Breaks DH with precomputation	Use ≥2048-bit primes, avoid shared parameters
Shared Parameters	Use of common primes/groups across many servers	Enables large-scale precomputation	Use unique, safe primes
Static DH Keys	Reuse of long-term DH private keys	Weak forward secrecy	Use ephemeral DH

MITM and static DH

- MITM can be prevented by inserting key material within X509 certificates
- Used in authentication
- To be developed later

Properties of key exchange

Necessary security requirement: the shared secret key is a one-way function of the public and transmitted information

Necessary “constructive” requirement: an appropriate combination of public and private pieces of information forms the shared secret key efficiently

DH Key exchange by itself is effective only against a passive adversary. Man-in-the-middle attack is lethal

Security requirements

- Is the one-way relationship between public information and shared private key sufficient?
- A one-way function may leak some bits of its arguments: this is prevented by carefully choosing g and p
- The full requirement is: given all the communication recorded throughout the protocol, computing any bit of the shared key is hard
- Note that the “any bit” requirement is especially important

Other DH Systems

- The DH idea can be used with any group structure
- Limitation: groups in which the discrete log can be easily computed are not useful (for example: additive group of Z_p)
- Currently useful DH systems: ECDH, the multiplicative group of Z_p and elliptic curve systems https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman
- Keys are exchanged in most network protocols, refreshed, automatically distributed, etc.

RSA – the algorithm

- **Claim**: If e is relatively prime to $(p-1)(q-1)$ then $x \longrightarrow x^e$ is a one-to-one op. in $\mathbb{Z}_{pq}^*$
- **Constructive proof**: Since $\gcd(e,(p-1)(q-1)) = 1$, e has a multiplicative inverse $\bmod (p-1)(q-1)$
- Denote it by d, then $ed = 1 + C(p-1)(q-1)$, C constant
- Let $y = x^e$, then $y^d = (x^e)^d = x^{1+C(p-1)(q-1)} = x^1 x^{C(p-1)(q-1)} = x\left(x^{(p-1)(q-1)}\right)^C = x \cdot 1 = x$
- Meaning $y \longrightarrow y^d$ is the inverse of $x \longrightarrow x^e$

QED

RSA public key cryptosystem

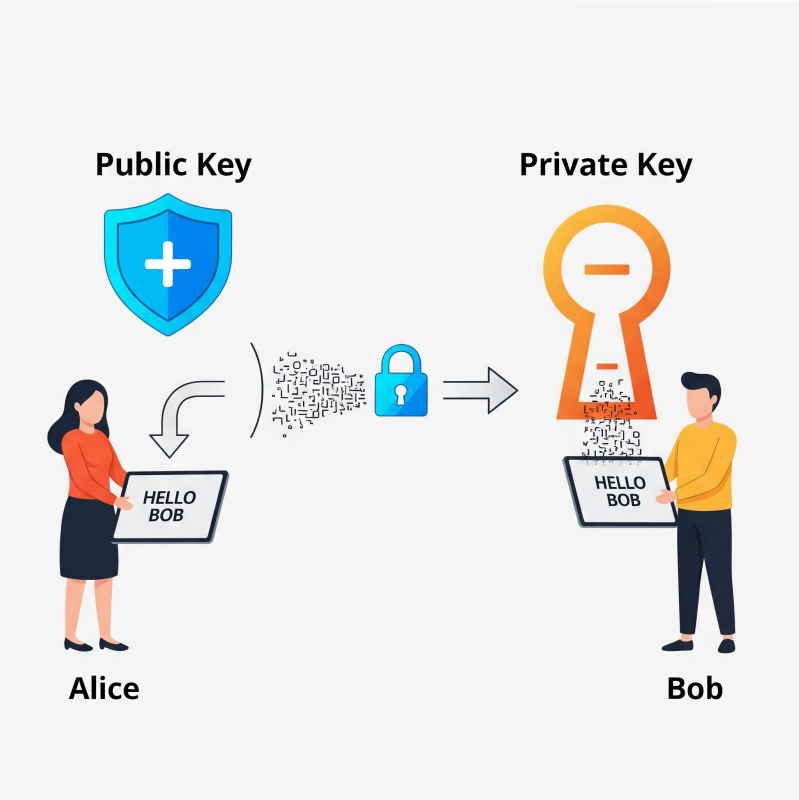
The above-mentioned method should not be confused with the exponentiation technique presented by Diffie and Hellman to solve the key distribution problem

Why decryption works

RSA

- key length is variable
  - the longer, the higher security
  - 2048-4096 bits are common
- block size also variable
  - but $|\text{plaintext}| \leq |N|$ (in practice, for avoiding weak cases, $|\text{plaintext}| < |N|$)
  - $|\text{ciphertext}| = |N|$
- note that plaintext and ciphertext are numbers < N
- slower than symmetric ciphers
  - not used in practice for encrypting long messages
  - mostly used to encrypt a symmetric key, then used for encrypting the message
    - key exchange

RSA – an example



Historical note

Multiplicative inverses

Overview on the Extended Euclidean Algorithm EEA

The EEA algorithm

```
function extendedEuclid(a, b)
    // Assumes gcd(a, b) = 1
    x0 ← 1, y0 ← 0
    x1 ← 0, y1 ← 1

    while b ≠ 0
        q ← a div b
        (a, b) ← (b, a mod b)
        (x0, x1) ← (x1, x0 - q * x1)
        (y0, y1) ← (y1, y0 - q * y1)

    return (x0, y0)  // such that a*x0 + b*y0 = 1
```

A small example

- Let p = 47, q = 59, N = pq = 2773
  - φ(N)= 46×58 = 2668
- Pick e = 157 (gcd(2668, 157) = gcd(157, 156) = gcd(156, 1) = gcd(1, 0) = 1), then 157×17 – 2668 = 1, so d = 17 is the inverse of 157 mod 2668
- For N = 2773 we can encode two letters per block, using a two-digit number per letter:
- blank = 00, A = 01, B = 02, …, Z = 26
- Message: ITS ALL GREEK TO ME is encoded

0		1		2		3		4		5		6		7		8		9	
I	T	S		A	L	L		G	R	E	E	K		T	O		M	E	
0	2	1	0	0	1	1	0	0	1	0	0	1	0	2	1	0	1	0	0
9	0	9	0	1	2	2	0	7	8	5	5	1	0	0	5	0	3	5	0

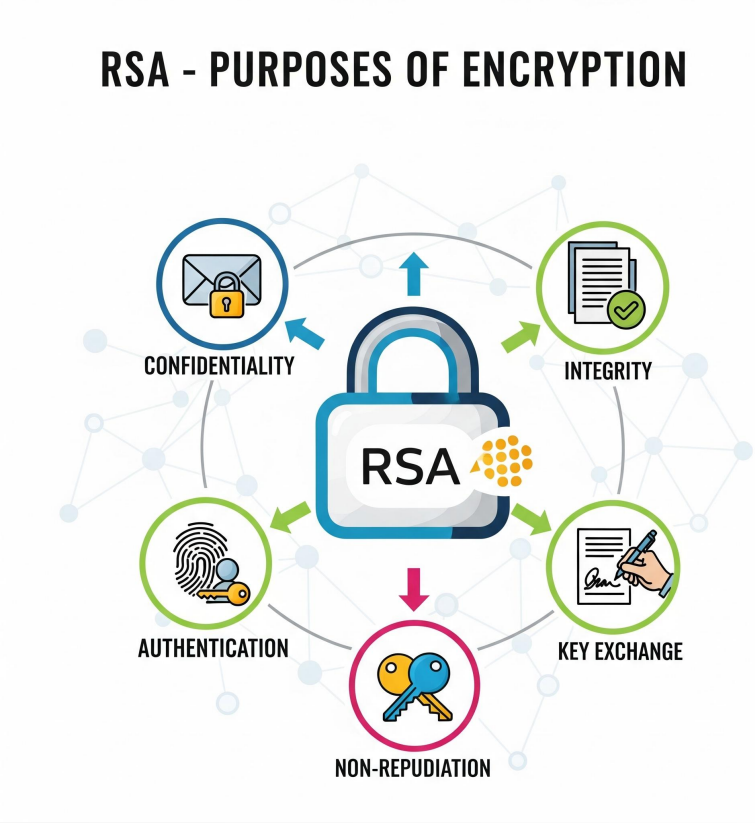
A small example

- N = 2773, d = 17
- ITS ALL GREEK TO ME is encoded as
  0920 1900 0112 1200 0718 0505 1100 2015 0013 0500

- First block M = 0920 encrypts to
  - $M^d = M^{17} = (((M^2)^2)^2)^2 \times M = 948 \pmod{2773}$
- The whole message (10 blocks) is encrypted as
  0948 2342 1084 1444 2663 2390 0778 0774 0219 1655
- Indeed $0948^e = 0948^{157} = 948^{1+4+8+16+128} = 920 \pmod{2773}$, etc.

Choice of exponents

RSA – purposes of encryption



RSA: two ways for encrypting

- Recall the existence of two keys
    - public
    - private
- Depending on which key is used, RSA encryption behaves differently
- Alice encrypts a block M using
    - Bob's public key (any public key can be used)
    - Alice's private key (only the owner can use their private key)

RSA: effects

- Encryption by a public key
  - everybody can encrypt by using a pub key
  - only the owner of the corresponding private key can decrypt
  - confidentiality
- Encryption by a private key
  - only the owner can encrypt
  - everybody can decrypt
  - no confidentiality
  - non-repudiation?

Non-repudiation

- The sender of a message cannot deny having sent it
- It provides proof of origin and integrity
- Typically reached through a digital signature
- HMAC doesn't provide non-repudiation
    - A judge can't attribute the message to one of the parties sharing the HMAC key

Purposes

- Enc with pub key
    - confidentiality
    - mainly used for key-exchange
    - not used for messages longer than one block (or ≥ N)
- Enc with pvt key
    - integrity, authenticity
    - non-repudiation

Discussion (confidentiality)

- RSA is not used to encrypt long files because it is inefficient and insecure for that purpose. Instead, RSA is typically used for key exchange (or digital signatures)
  - Perfomarce issues. Symmetric ciphers much faster
  - Size limitation. Number should be < N
  - Security. RSA is deterministic unless padded
  - Battery draining
- Hybrid encryption. Key exchange + AES
  - Only few bits are encrypted

Discussion (non-repudiation)

- Disregarding the performance issues, RSA enc with pvt key is a path to non-repudiation
- Because public can repeatedly decrypt a message and see that it was originated by who encrypted
    - the process is a verification, like MAC verification

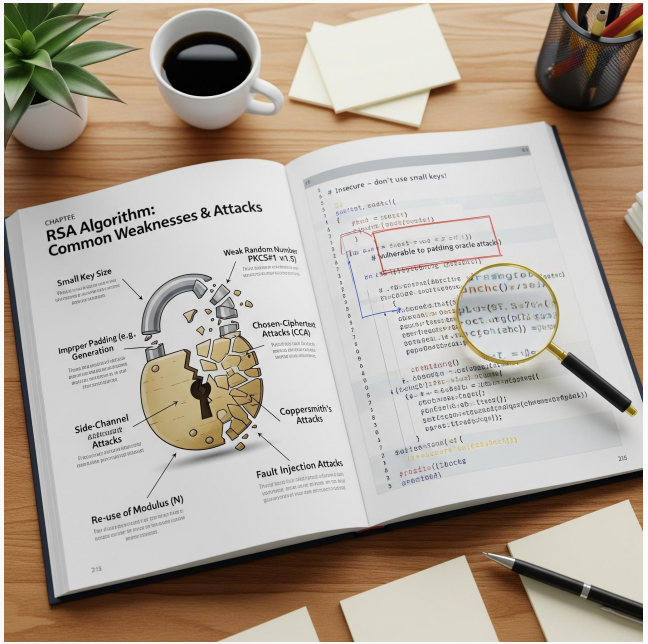
Not yet non-repudiation

Forgery

RSA signature

Bob's verification

"textbook" RSA weaknesses



"textbook" RSA is not always robust

- There are several vulnerabilities in the original definition of RSA
- Let's examine the main

Attacks on RSA

1. Factor N = pq. This is believed hard unless p, q have some "bad" properties. To avoid such primes, it is recommended to
    - Take p, q large enough (at least ~1024 bits each)
    - Make sure p, q are not too close together
    - Make sure both (p-1), (q-1) have large prime factors (to foil Pollard's rho algorithm)
        - special-purpose integer factorization algorithm (John Pollard, 1975); particularly effective at splitting composite numbers with small factors.
2. Some messages might be easy to decrypt

RSA and factoring

- Fact 1: given N, e, p and q it is easy to compute d
- Fact 2: given N, e
  - if you factor N then you can compute $\varphi(N)$ and d

- Conclusion
  - If you can factor N, then you break RSA
  - If you invert RSA, then you can factor N?
    OPEN PROBLEM

Factoring RSA challenges

- RSA challenges: factor N = pq
- RSA Security has published factoring challenges
    - RSA 426 bits, 129 digits: factorized in 1994 (8 months, 1600 computers on the Internet (10000 Mips))
    - RSA 576 bits, 173 digits: factorized in dec. 2003, $10000
    - RSA 640 bits, Nov. 2005, 30 2.2GHz-Opteron-CPU

- Challenge is not active anymore (2007)

A case study

- June 2008: Kaspersky Labs analyzed a variant of the Gpcode ransomware that used RSA encryption with a 1024-bit key to lock victims' files
- Kaspersky proposed a global distributed computing effort to break the key. Estimate (from their website):
  - ~15 million modern computers
  - Running for ~1 year
  - To factor a single 1024-bit RSA modulus
- This estimate was theoretical only—no such project was launched
- Result: The RSA key was not broken. Victims could only recover files via backups or by paying ransom
- The case illustrated the practical strength of RSA-1024 (as of 2008) against classical brute-force factoring

RSA - Attacks

- There are messages m easy to decrypt: if m = 0, 1, N-1 then RSA(m) = m
  - notice e must be odd ≥ 3, hence $(N-1)^e \bmod N \equiv N-1$, because $(N-1)^2 \bmod N \equiv 1$
  - SOLUT: rare, use salt
- If both m and e are small (e.g., e = 3) then we might have $m^e < N$; hence $m^e \bmod N = m^e$
  - compute e-th root in arithmetic is easy: adversary compute it and finds m
  - SOLUT. Add nonzero bytes to avoid small messages

RSA – Attacks

Small e (e.g., e = 3)

- Suppose adversary has two encryptions of similar messages such as

  $c_1 = m^3 \bmod n$ and $c_2 = (m+1)^3 \bmod n$

  Therefore

  $m = (c_2 + 2\,c_1 - 1) / (c_2 - c_1 + 2)$
- The case m and (am + b) is similar

SOLUT. Choose large e

Low-Exponent RSA with Related Messages

Don Coppersmith* Matthew Franklin** Jacques Patarin*** Michael Reiter†

$$m_2 = \alpha m_1 + \beta.$$

Eurocrypt 1996

$$\frac{(m+1)^3 + 2m^3 - 1}{(m+1)^3 - m^3 + 2} = \frac{3m^3 + 3m^2 + 3m}{3m^2 + 3m + 3} = m \bmod N.$$

Some attacks use CRT

Chinese Remainder Theorem (CRT)

- Suppose $n_1, n_2, \ldots, n_k$ are positive integers which are pairwise coprime. Then, for any given integers $a_1, a_2, \ldots, a_k$, there exists an integer x solving the system of simultaneous congruencies

$$x \equiv a_i \pmod{n_i} \quad \text{for } i = 1, \ldots, k.$$

- Furthermore, all solutions x to this system are congruent modulo the product $N = n_1 n_2 \ldots n_k$

RSA – Attacks

- Assume e = 3 and then send the same message three times to three different users, with public keys

  $(3, n_1)$ $(3, n_2)$ $(3, n_3)$
- **Attack**: adv. knows public keys and

  $m^3 \bmod n_1$, $m^3 \bmod n_2$, $m^3 \bmod n_3$
- He can compute (using CRT)

  $m^3 \bmod (n_1 \cdot n_2 \cdot n_3)$
- Moreover $m < n_1, n_2, n_3$; hence $m^3 < n_1 \cdot n_2 \cdot n_3$ and

  $m^3 \bmod n_1 \cdot n_2 \cdot n_3 = m^3$
- Adv. computes cubic root and gets result
- SOLUT. Add random bytes to avoid equal messages

RSA - Attacks

- If message space is small, then adv. can test all possible messages
  - ex.: adv. knows encoding of m and knows that m is either m_1=10101010 or m_2=01010101
  - adv. encrypts m_1 and m_2 using public key and verifies

- SOLUT. Add random string in the message

RSA - Attacks

- If two user have same n (but different e and d) then bad
  - One user could compute p and q starting from his e, d, n (somewhat tricky, based on Euler's theorem – see for instance "The Handbook of Applied Cryptography", Sect. 8.2.2)
  - Then, the user could easily discover the secret key of the other user, given his public key

- SOLUT. Each person chooses his own n (the probability two people choose same n is very very low)

Discussion

- Many mitigations follow similar patterns
- Additional attacks will be discussed later
- These considerations highlight the necessity of preprocessing messages before RSA encryption
    - In cryptographic contexts, this preprocessing is referred to as padding

Other attacks to RSA

Multiplicative property and malleability

- In cryptography a function f is said to have a multiplicative homomorphism (or simply be multiplicative) if $f(m_1 \cdot m_2) \equiv f(m_1) \cdot f(m_2) \bmod n$
- This is true for RSA encryption:
  $$\text{RSA}(m_1 \cdot m_2) \equiv \text{RSA}(m_1) \cdot \text{RSA}(m_2) \bmod N$$
- This property is what makes RSA **malleable**: an attacker can modify a ciphertext in a predictable way without knowing the plaintext, by exploiting this multiplicative behavior

Attacks on RSA

- Multiplicative property of RSA:
  - If $M = M1 * M2$ then $(M1*M2)^e \bmod N = ((M1^e \bmod N) * (M2^e \bmod N)) \bmod N$

    Hence an adversary can proceed using small messages (chosen ciphertext, see next slide)

  - Can be generalized if $M = M1*M2* \ldots *Mk$
  - Solut.: padding on short messages

Attacks on RSA

Adversary wants to decrypt $C = M^e \bmod n$

1. adv. computes $X = (C \cdot 2^e) \bmod n$
2. adv. uses X as chosen ciphertext and asks the oracle for $Y = X^d \bmod n$

but....

$X = (C \bmod n) \cdot (2^e \bmod n) = (M^e \bmod n) \cdot (2^e \bmod n) =$
$= (2M)^e \bmod n$

thus adv. got $Y = (2M)$

Attacks on RSA

Chosen ciphertext attack (more general):

- Adv. T knows $c = M^e \bmod n$
- T randomly chooses X and computes $c' = c\, X^e \bmod n$ and ASKS ORACLE TO DECRYPT c'
- T computes $(c')^d = c^d (X^e)^d = M X \bmod n$!!

- SOLUT: require that messages verify a given structure (oracle does not answers if M does not verify requirements)

Attacks on RSA

Implementation attacks

- Timing: based on time required to compute C^d
- Energy: based on energy required to compute (smart card) C^d

- Solut.: add random steps in implementation

RSA – Attacks : conclusion

Textbook implementation of RSA is NOT safe

- It does not verify security criteria
- Many attacks

There exists a standard version

- Preprocess (pad) M to obtain M′ and apply RSA to M′ (clearly the meaning of M e M′ is the same)

PKCS

PKCS stands for Public-Key Cryptography Standards. These are standards developed by RSA Laboratories in early 90s to facilitate the secure implementation and interoperability of public-key cryptography

PKCS #	Title	Function / Description	Status / Adoption
PKCS #1	RSA Cryptography Standard	RSA encryption, signatures, and padding	Widely used; parts adopted in **RFC 8017**
PKCS #3	Diffie-Hellman Key Agreement	Diffie–Hellman key exchange format	Obsolete; replaced by IETF standards (e.g., RFC 2631)
PKCS #5	Password-Based Cryptography	Defines PBKDF1, PBKDF2, encryption schemes based on passwords	Widely used; part of **RFC 8018**
PKCS #6	Extended Certificate Syntax	Defines certificate extensions	Obsolete; superseded by **X.509v3**

PKCS (2)

PKCS #	Title	Function / Description	Status / Adoption
PKCS #7	Cryptographic Message Syntax	Format for signed/enveloped data (original S/MIME basis)	Superseded by **CMS (RFC 5652)**
PKCS #8	Private Key Information Syntax	Standard for storing private keys with optional encryption	Widely used; in **OpenSSL**, **Java**, etc.
PKCS #9	Selected Attribute Types	Defines standard attributes for use in PKCS#7, #8, #10	Still relevant
PKCS #10	Certificate Request Syntax (CSR)	Format for Certificate Signing Requests (CSRs)	Ubiquitous in TLS/PKI (e.g., OpenSSL req)

PKCS (3)

PKCS #	Title	Function / Description	Status / Adoption
PKCS #11	Cryptographic Token Interface (Cryptoki)	API for interacting with cryptographic hardware (e.g., HSMs, smartcards)	**Actively maintained** by **OASIS**, latest: v3.1
PKCS #12	Personal Information Exchange Syntax	Container for certs, private keys, etc. (e.g., .p12, .pfx)	Still widely used in Windows, browsers
PKCS #13	Elliptic Curve Cryptography	ECC primitives and formats	Rarely used directly; ECC now in other standards (e.g., RFC 7748)
PKCS #15	Cryptographic Token Information Format	Standard format for storing cryptographic objects on tokens/smartcards	Used in smartcard infrastructure

PKCS #1 (old version)

Standard to send message M using RSA. The result of padding is m

m = 0 ‖ 2 ‖ at least 8 non-zero bytes ‖ 0 ‖ M

- first byte 0 implies message is less than N
- second byte (= 2) denotes encrypting of a message (1 denotes non-repudiation use)
- random bytes imply
  - same message sent to many people is always different
  - space of messages is large

Overview on a selection of PKCS

PKCS — 1st overview

- Public-Key Cryptography Standards (PKCS) developed by RSA Laboratories
- Initially focused on RSA algorithms, later expanded to broader cryptography
- Define widely used formats and protocols
- Some drafts addressed advanced or experimental topics:
    - Not always standardized or implemented
    - Several are now considered obsolete or legacy

PKCS — 2nd overview

- Range from PKCS#1 (RSA) to PKCS#15 (tokens)
- Some merged into RFCs (e.g., PKCS#7 → CMS, PKCS#10 → CSR)
- Focus here on some...
- Key idea: interoperability across vendors

PKCS #3 — Diffie–Hellman key agreement

What is PKCS #3?

- Defines Diffie–Hellman (DH) key exchange
- Each party generates public/private pair
- Compute shared secret: $g^{ab} \bmod p$
- Used for session keys

PKCS #3 in context

- Originally specified how Diffie-Hellman parameters were encoded
- Obsolete — superseded by ANSI X9.42 and RFC 2631 (Diffie-Hellman Key Agreement Method)
- Further refined in RFC 3526
- Historically important: first standardized DH parameter encoding
- Applications: TLS (legacy, now deprecated), IPsec (DH groups)

PKCS #5 — Password-based encryption (PBE)

PKCS #5 – Motivation

- Users can remember passwords, but not long cryptographic keys
- A secure method is needed to derive keys from passwords
- Passwords have low entropy and require strengthening (e.g., with PBKDFs such as PBKDF2, bcrypt, scrypt, Argon2)

PKCS #5 – Applications

- Wi-Fi (WPA/WPA2) uses PBKDF2 to derive the Pairwise Master Key
- Password managers (e.g., KeePass, 1Password) rely on PBKDF2
- PKCS #12 protects private keys using password-based encryption
- Modern KDFs (e.g., Argon2) build on concepts introduced in PKCS #5

PKCS #5 – Mechanism

- Key derivation via PBKDF1 (legacy) and PBKDF2 (standard)
- Inputs: password, salt, iteration count, pseudorandom function (e.g., HMAC-SHA256)
- Output: derived cryptographic key (suitable for encryption or authentication)
    - Example: PBKDF2(password, salt, iterations, dkLen)

PKCS #8 — Private key information syntax

PKCS #8 – Purpose

- Defines the ASN.1 syntax for representing private keys
- Supports multiple key types: RSA, EC, DH, etc.
- Private keys can be encrypted using password-based encryption (PBE, PKCS #5)

PKCS #8 – Structure

- Structure of PKCS #8 private keys
- PrivateKeyInfo
    - AlgorithmIdentifier
    - PrivateKey (OCTET STRING)
    - Attributes (optional, PKCS #9)
- EncryptedPrivateKeyInfo
    - EncryptionAlgorithm
    - EncryptedData

PKCS #8 – Usage

- PEM file encodings
  -----BEGIN PRIVATE KEY----- (unencrypted PKCS #8)
  -----BEGIN ENCRYPTED PRIVATE KEY----- (PKCS #8 with PBE)

- Used in OpenSSL, PKI exports, secure key storage, TLS/HTTPS

PKCS #9 — Selected attribute types

PKCS #9 – an overview

- Defines attributes to be used with PKCS objects
- Provides extensibility in PKI
- Examples of attributes
    - challengePassword (used in PKCS #10 CSR for renewal/revocation)
    - emailAddress, unstructuredName
    - Custom attributes via Object Identifiers (OIDs)
- Importance
    - Enable richer identity representation in certificates
    - Still used in CSRs and directories (X.500/X.509)

PKCS #11 — Cryptoki API

Cryptoki (PKCS #11)

- Stands for “Crypto Key API”
- Standard C API for interacting with
    - HSMs (Hardware Security Modules)
    - Smart cards, USB tokens
- Architecture
    - Slot = interface/reader
    - Token = cryptographic device
    - Session = active connection
    - Objects = keys, certs, data
- Supports: key management, authentication, encryption, signing, RNG
- Vendor-independent

Cryptoki (PKCS #11) applications

- Enterprise PKI: integration with HSMs
- Digital signatures, TLS termination with HSMs
- Smart card logins (Windows, Linux)
- Still evolving: adoption in cloud HSMs and modern key management

PKCS #12 — Personal information exchange

PKCS #12 — Purpose & structure

- Defines a container format to bundle certificates, private keys, and related data
- Per-object protection: private keys can be encrypted using PKCS #5 PBE
- Container-level protection: the entire PFX file can also be password-encrypted
- PFX (Personal Information Exchange) container
- Contains “safe bags” (logical containers)
    - certBag (certificates)
    - keyBag (private keys)
    - CRLBag (revocation lists)

PKCS #12 — Usage

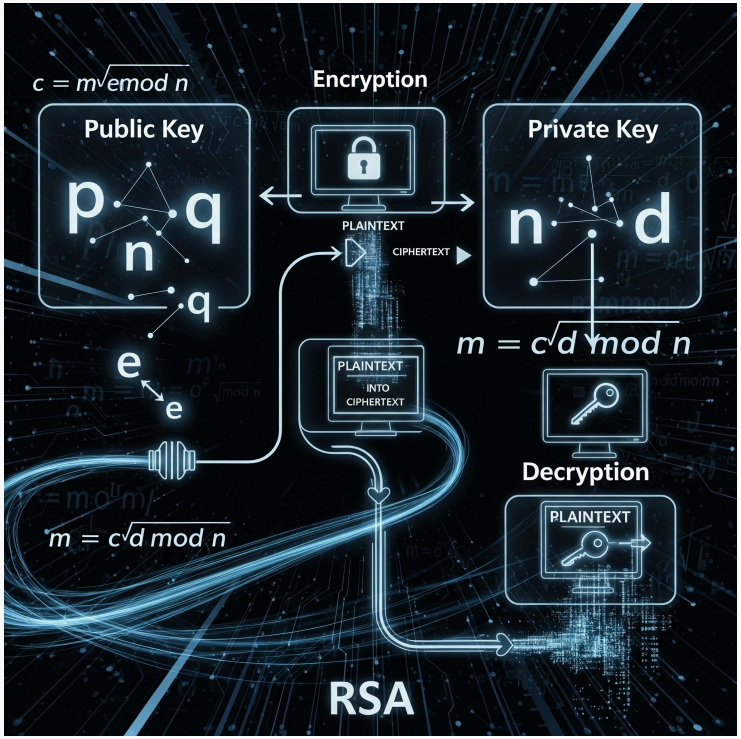
- File extensions: .pfx, .p12
- Used for certificate import/export (browsers, OS stores, VPN/Email clients)
- Transport and backup of user credentials (certificates + private keys)
- Widely supported by Microsoft (Windows), OpenSSL, and other tools

PKCS #15 — Cryptographic token information format

PKCS #15 — Overview

- Goal: standardize file structure on cryptographic tokens, ensure interoperability across vendors
- Token: hardware or software container holding cryptographic material (e.g., smart card, USB token, HSM, virtual token)
- Objects on token: keys (private/public), certificates, PINs/authentication data, access control structures
- Applications: eID cards, passports, healthcare cards, enterprise access cards
- In practice
    - PKCS #15 = storage format (how data is structured on token)
    - PKCS #11 = access interface (how applications use the token)

RSA implementation



Properties of RSA (recap)

- The requirement $GCD(e,\phi(n))=1$ is important for uniqueness
- Finding d, given p and q is easy. Finding d given only n and e is assumed to be hard (the RSA assumption)
- The public exponent e may be small. Typically, its value is either 3 (problematic) or $2^{16}+1$
- Each encryption involves several modular multiplications. Decryption is longer

Constructing an instance of RSA

- Alice first picks at random two large primes, p and q
- Let N = p·q be the product of p and q
- Alice then picks at random a large e that is relatively prime to φ(N) = (p-1)(q-1) (gcd(e, φ(N)) = 1)
    - or Alice chooses d
- Alice ensures that d·e ≡ 1 mod φ(N)

- Alice publishes the public key (N, e)
- Alice keeps the private key (N, d), as well as the primes p, q and the number φ(N), in a safe place

RSA: implementation

1. Find p and q, two large primes
   - Random (an adversary should not be able to guess these numbers; they should be different each time a new key is defined)
2. Choose suitable e to have a fast encryption
   - Use exponentiation algorithm based on repeated squaring:
     - Compute power of 2, 4, 8, 16, ...
     - Compute power e by using the binary encoding of e and powers computed in so far
   - In this way decrypting is generally slower (d is very large)
3. Compute d:
   - Euclid's extended algorithm

RSA: step 1 implementation

1. Find two random primes

Algorithm:

- randomly choose a random large odd integer
- test if it is a prime; if not, then repeat

Note:

- Prime number are relatively frequent (in [N , 2N] there are ≈ N / ln N primes)
  - prime number theorem (http://en.wikipedia.org/wiki/Prime_number_theorem) states that the average gap between prime numbers near N is roughly ln(N)
- Hence randomly choosing we expect to find a prime of N bits every ln N attempts

RSA: step 2 implementation

2. Encrypting algorithm (compute exponentiation)

- Compute power by repeated squaring and then executing multiplication (based on binary encoding of the exponent e)
    - Cost: O(log N) operations
- Constant no. of operations if e is small and its binary representation has few bits
- Examples: $e = 3$, 2 multiplications
- $e = 65537 = 2^{16} + 1$, compute powers M^2, M^4, M^8, M^{16},.. M^{65536} and then $M^{65537} = M^{65536} * M$
    - total 16 multiplications

RSA: step 3 implementation

3. Extended Euclidean Algorithm (EEA)

```
int mod_inverse(int e, int phi) {
    int x0 = 1, x1 = 0, a = e, b = phi;
    while (b) {
        int q = a / b, t = b;
        b = a % b; a = t;
        t = x1;
        x1 = x0 - q * x1;
        x0 = t;
    }
    return x0;      // can be negative
}
```

Encoding text into a number

OS2IP: octet-stream to integer primitive

- a non-negative integer is constructed by interpreting k bytes as a number in base 256, where each byte represents a digit in [0, 255]
- k is the number of bytes for representing N

I2OSP: integer to octet-stream primitive

- an array of k bytes is constructed by determining the coefficients of the representation in base 256 of the given integer

OS2IP example

- octet_string = b'\x01\x02\x03\x04'
- bytes are interpreted as a big-endian number
  - 01 is the most significant byte
- treat the octet string as a single integer in base 256
  - $1 \times 256^3 = 16{,}777{,}216$
  - $2 \times 256^2 = 131{,}072$
  - $3 \times 256^1 = 768$
  - $4 \times 256^0 = 4$

- 16,777,216 + 131,072 + 768 + 4 = 16,909,060

- The OS2IP conversion of the byte string b'\x01\x02\x03\x04' gives the integer 16,909,060

I2OSP example

- convert 84,281,096 with a desired length of 8 bytes (4 are necessary)
- 84,281,096 converts to 0x05 06 07 08 in hexadecimal (big-endian)
- to represent this as an 8-byte string, we need to pad it with leading zeros
  - 00 00 00 00 05 06 07 08

- so, result is b'\x00\x00\x00\x00\x05\x06\x07\x08'

RSAES-PKCS1-v1_5

- given message M, determine padded message M'=0x00‖0x02‖at least 8 non-zero bytes‖0x00‖M
- determine integer representative m of M'
  - use m = OS2IP(M') octet-str… …er primitive
- compute ciphertext c re… …)
  - c = RSAEP((… …blic-key
- convert c to c… …ray)
  - use C = I…N|) integer to octet-stream primitive
- decryption is symmetric and is based on RSADP and the extraction of M from M'

RSAES is RSA Encryption Scheme
RSAEP is RSA Encryption Primitive
RSADP is RSA Decryption Primitive

About padding

- M' = 0x00 || 0x02 || PS || 0x00 || M
  - 0x00: Single byte with value 0x00
  - 0x02: Indicates encryption padding
  - PS: Random non-zero padding bytes (at least 8 bytes in length)
  - 0x00: Separator byte
  - M: The actual message
- Example for |M|=100 bytes and |N|=2048 bits (=256 bytes)
  - M' = 0x00 || 0x02 || [153 random non-zero bytes] || 0x00 || [100-byte message]

Summary

- PKCS#1 v1.5 is a widely used padding scheme in RSA encryption
- It ensures message length and adds some security through random padding bytes
- Bash scripts can automate encryption and decryption processes
- Manual verification helps ensure the integrity of the padding format

RSA: OAEP

Introduction to OAEP

- OAEP = Optimal Asymmetric Encryption Padding
- What is it?
    - A padding scheme for RSA encryption (for **confidentiality**) that adds randomness and structure to achieve semantic security
- Key Features
    - Prevents chosen ciphertext attacks (IND-CCA)
    - Uses MGF1 and a random seed
    - Output changes even if the message stays the same
    - Defined in PKCS#1 v2.2 (RFC 8017)
- Use Case
    - Applied in RSA-OAEP: ciphertext = RSA_Encrypt(OAEP(message, seed))
- Secure and recommended for modern RSA-based encryption

OAEP: basic info

- New padding scheme often used together with RSA encryption (for confidentiality)
  - Bellare and Rogaway: "Optimal Asymmetric Encryption - How to Encrypt with RSA" 1994, 1995 (https://cseweb.ucsd.edu/~mihir/papers/oaep.pdf)
- OAEP
  - Uses a pair of random oracles G and H to process the plaintext prior to encryption (for security under chosen plaintext/ciphertext attacks)
  - Produces a probabilistic scheme
  - Prevent (more sophisticated) attacks (CCA) that address vulnerabilities of PKCS#1 v. 1.5

"IND-" attacks

Notation	Full Name	Attacker's Capabilities
IND-CPA	Indistinguishability under Chosen Plaintext Attack	Can ask for encryptions of messages
IND-CCA1	… under Chosen Ciphertext Attack (non-adaptive)	Can decrypt ciphertexts before the challenge
IND-CCA 2	… under Adaptive Chosen Ciphertext Attack	Can decrypt other ciphertexts before and after the challenge (except the challenge itself)

OAEP scheme

n = number of bits in RSA modulus

k0 and k1 = integers fixed by the protocol

m = plaintext message, a (n – k0 – k1)-bit string

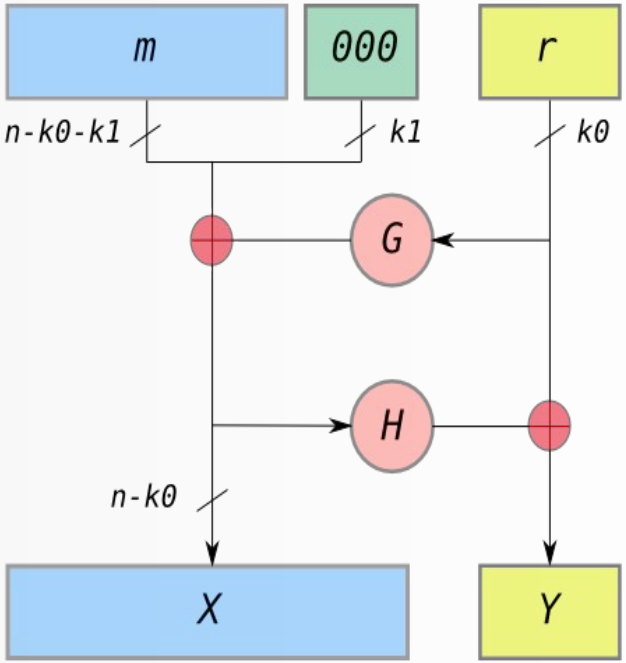
G and H = cryptographic hash functions fixed by the protocol

To encrypt

- messages are padded with k1 zeros to be n – k0 bits in length.
- r is a random k0-bit string
- G expands the k0 bits of r to n – k0 bits.
- X = m00..0 $\oplus$ G(r)
- H reduces the n – k0 bits of X to k0 bits.
- Y = r $\oplus$ H(X)
- output is X $\|$ Y

To decrypt

- recover the random string as r = Y $\oplus$ H(X)
- recover the message as m00..0 = X $\oplus$ G(r)



G and H used for masking are instances of the same mask generation function, MGF1 — a hash-based function that can produce output of arbitrary length

OAEP "all-or-nothing" security

- to recover m, you must recover the entire X and the entire Y
  - X is required to recover r from Y, and r is required to recover m from X
- since any bit of a cryptographic hash completely changes the result, the entire X, and the entire Y must both be completely recovered

RSAES-OAEP

- given message M, determine encoded message M' (octet-stream) by means of OAEP
- determine integer representative m of M'
  - use m = OS2IP(M') octet-stream to integer primitive
- compute ciphertext representative (integer)
  - c = RSAEP((N, e), m) (N, e) is public-key
- convert ciphertext representative (integer) to ciphertext (octet-stream)
  - use C = I2OSP(c, |N|) integer to octet-stream primitive
- decryption is symmetric and is based on $OAEP^{-1}$ and RSADP
- **Standard PKCS#1 v2.2**

Overview of RSA Padding Schemes

- RSA padding schemes ensure that messages have the correct length and add security before encryption
  - PKCS#1 v1.5: Uses a simpler padding mechanism, but has known vulnerabilities
    - Still widely supported and used for backward compatibility, especially in legacy systems and older protocols (e.g., TLS < 1.3)
  - PKCS#1 v2.2 (RSA-OAEP): A more modern padding scheme that addresses the shortcomings of v1.5
    - Recommended for: all new implementations

Key Benefits of RSA-OAEP (PKCS#1 v2.2)

- Randomization and Semantic Security
  - RSA-OAEP uses a mask generation function (MGF) to add randomness to the padding
  - This ensures that identical plaintexts result in different ciphertexts, preventing pattern recognition by attackers
- Resistance to Chosen-Ciphertext Attacks (CCA):
  - RSA-OAEP is designed to be secure against CCA, including the Bleichenbacher attack
- Tampering with the ciphertext will produce errors, making it hard for attackers to manipulate or decrypt data

PKCS#1 v1.5 vs. v2.2

Feature	PKCS#1 v2.2	PKCS#1 v1.5
Randomization	Uses MGF for randomization	Less structured randomness; weaker randomization
Resistance to CCA	Secure against CCA (e.g., Bleichenbacher)	Vulnerable to adaptive CCA attacks
Hash-Based Security	Integrates a hash function (e.g., SHA-256)	No use of hash functions in padding

Importance of hash-based security in RSA-OAEP

- RSA-OAEP uses a hash function in its padding mechanism, enhancing security
  - Helps prevent manipulation of the padding and message independently
  - Hash functions like SHA-256 add an extra layer of cryptographic robustness
- This makes RSA-OAEP a preferred choice for new applications requiring high security

Summary

- RSA-OAEP (PKCS#1 v2.2) offers significant security advantages over PKCS#1 v1.5
- Randomized padding prevents pattern recognition by attackers
- Resistance to chosen-ciphertext attacks ensures robustness in secure communications
- Use of hash functions further enhances cryptographic security