Public Key Cryptography and RSA

CS70: Discrete Mathematics and Probability Theory

UC Berkeley – Summer 2025

Lecture 9

Ref: Note 7

Today

Today is light on new math...
But very cool (and important) application of what we've been studying

1. Cryptography: Basic Concepts
2. Public Key Cryptography Idea
3. The RSA cryptosystem
    1. What it is
    2. Proof that it works
    3. How to efficiently implement
4. Digital Signatures
    1. The basic idea
    2. RSA for signatures
    3. Signatures for integrity on the web
    4. Signatures for authentication

Quick Review Check!

Setup: $x \equiv 5 \pmod{7}$ and $x \equiv 5 \pmod{11}$
$y \equiv 3 \pmod{7}$ and $y \equiv 9 \pmod{11}$

Fill in the blank (all mod m values in the range $0, 1, \ldots, m-1$):

$x + y \bmod 7 =$ ______

$x + y \bmod 11 =$ ______

$xy \bmod 7 =$ ______

True/False: $x \cdot x \cdot x \cdot x \bmod 77 = \big(((x \cdot x \bmod 77) \cdot x \bmod 77\big) \cdot x \bmod 77\big)$ ______

$x \bmod 77 =$ ______

$y \bmod 77 =$ ______

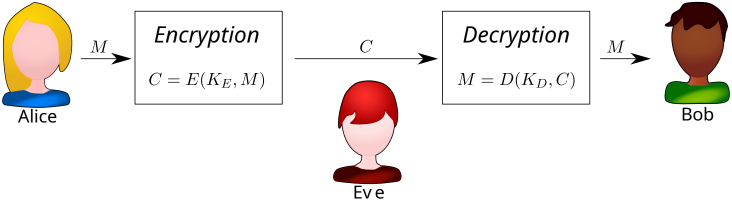
Number of solutions for z in $z \equiv y \pmod{77}$? ______ (in-range!)

$x^{61} \bmod 7 =$ ______

$x^{61} \bmod 11 =$ ______

$x^{61} \bmod 77 =$ ______

Cryptography



Terminology:

Alice: Sender
Bob: Receiver
Eve: Eavesdropper

M: Plaintext
C: Ciphertext

E: Encryption function
K_E: Encryption key

D: Decryption function
K_D: Decryption key

Exclusive Or

Bits for truth values: $0 =$ False $1 =$ True

In C programming, True is any non-zero value

Recall: In logic "OR" means "one or more of the inputs is true."

Inclusive OR

Can also define *exclusive* OR: "one and only one input is true"

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

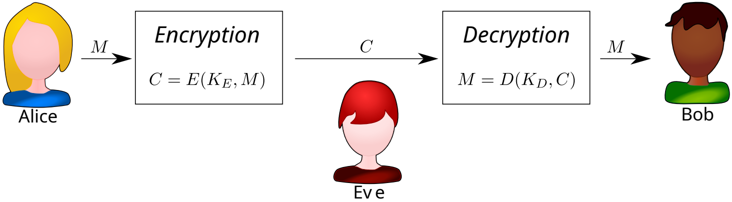
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Alternate view: Mod 2 addition $(1 + 1 = 2 \equiv 0 \pmod 2)$

Regular addition properties (associative, commutative, ...) plus:

- 0 is additive identity: For any x, we have $x \oplus 0 = x$
- Self-inverse: For any x, we have $x \oplus x = 0$ (so also: $(x \oplus y) \oplus y = x$)
- Uniform: If y is uniform (prob ½ being 0 or 1) then $x \oplus y$ is uniform

Cryptography



Terminology:

Alice: Sender
Bob: Receiver
Eve: Eavesdropper

M: Plaintext
C: Ciphertext

E: Encryption function
K_E: Encryption key

D: Decryption function
K_D: Decryption key

Traditional Cryptography

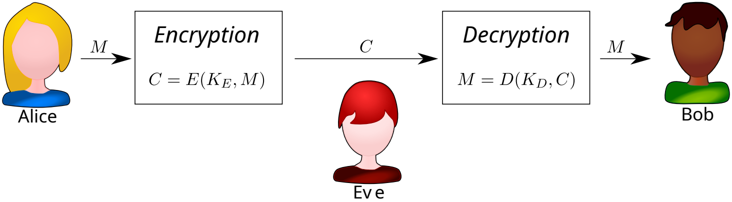
$K_E = K_D$
sometimes called "symmetric cryptography"

Example:
M is an n-bit string
K is a string of n random, independent bits
C is bitwise XOR of M and K

```
M: 011101001 ...  110
K: 101011011 ...  010

C: 110110010 ...  100
```

Cryptography




```
M: 011101001 ...  110
K: 101110010 ...  010
C: 110011011 ...  100
```

Bit i: $C_i = M_i \oplus K_i$

Important:

K_i is random (uniform, independent)

$\Rightarrow$ C_i is random/uniform

Strong points:

Ciphertext is random (100% secure!)

Extremely fast

Problems:

Alice and Bob must share a secret K

Key can only be used once!

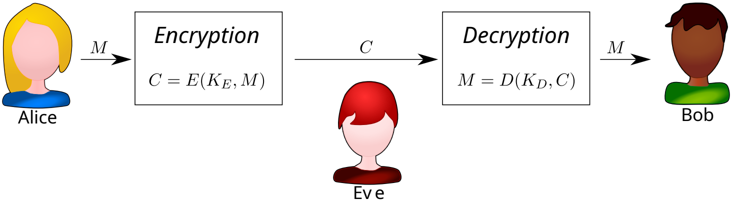
(this scheme is a “one-time pad”)

For modern technology:

Do you share a secret with Amazon?

... a new secret for each purchase?

Cryptography: A Different Way...



What if K_E and K_D aren't the same?
- What really *needs* to be secret?
- *Algorithms* should never be secret!

K_D? Yes! If not secret, Eve could decrypt.

K_E? Why?
- No problem if *others* can encrypt
- K_D shouldn't be computable from K_E
- Otherwise K_E can be public

This idea: Public key cryptography

Strong points:
- Communicate securely with strangers!
  - No need to pre-arrange shared secret
  - Bob can send public key to Alice

Problems:
- Algorithms not (initially!) obvious
- Known algorithms are slow

Basic idea: Diffie and Hellman (1975)
First real algorithm: RSA (1976)
- Rivest, Shamir, and Adelman
- *Adelman: Berkeley connection!*

The RSA Algorithm

Three algorithms:

- Key Generation
- Encryption
- Decryption

Key Generation:

Pick two large primes p and q

Compute $N = pq$

Messages are from $\{0, 1, \ldots, N-1\}$

Encryption/decryption work mod N

Pick e relatively prime to $(p-1)(q-1)$

Compute $d = e^{-1} \pmod{(p-1)(q-1)}$

Now $K_E = (e, N)$

And $K_D = (d, N)$

Encryption:

$E(K_E, M) = M^e \bmod N$

Decryption:

$E(K_D, C) = C^d \bmod N$

Does this work?

Need $D(K_D, E(K_E, M)) = M$ for all M

I hope so! (We'll see....)

How are K_E and K_D related?

Compute K_D from just K_E?

No! Need knowledge of p and q

Are p and q part of public info?

No! Just publish the product

Can you compute p and q from K_E?

Well.... we don't think so.

Possible to factor efficiently?

No known polynomial time algorithms

Millennia of attempts...

New wrinkle: Quantum computing

Is factoring the only way to break RSA?

Probably – but unknown!

Concept Check!

Question: Which of the following is not true?

Notation: Alice is sending to Bob. Key parts $(N = pq, e, d)$. Eve is evil.

(A) Eve knows e and N

(B) Alice knows e and N

(C) $ed \equiv 1 \pmod{N-1}$

(D) Bob forgot p and q but can still decode

(E) Bob knows d

(F) $ed \equiv 1 \pmod{(p-1)(q-1)}$

Encryption/Decryption Example

Values:

$p = 7$, $q = 11$, $N = 77$
So $(p-1)(q-1) = 60$
$\gcd(7,60) = 1$ and mult inverse of $7 \pmod{60}$ is 43

This was the hand-calculated example from last lecture!

So:

$K_E = (e, N) = (7, 77)$
$K_D = (d, N) = (43, 77)$

For example: $M = 2$:

$C = E(K_E, M) = M^e \bmod N = 2^7 \bmod 77 = 128 \bmod 77 = 51.$

$D(K_D, C) = C^d \bmod N = 51^{43} \bmod 77$...

How are we going to do this????

Cheat – Python: `pow(51,43,77)` gives `2` – yay!

But how did *Python* do it? 43 multiplications?

No – we can do better. (And we *must* do better when d is 2048 bits!)

Correctness: Does RSA Always Decode Correctly?

Need $D(K_D, E(K_E, M)) = M \quad \implies (M^e)^d \equiv M^{ed} \stackrel{?}{\equiv} M \pmod{N}$?

$d \equiv e^{-1} \pmod{(p-1)(q-1)} \quad \implies ed = 1 + k(p-1)(q-1)$

$N = pq$ with $\gcd(p, q) = 1$ – so we can use CRT and look at power mod p

$$M^{ed} \equiv M^{1+k(p-1)(q-1)} \equiv M \cdot M^{k(p-1)(q-1)} \equiv M \cdot \left(M^{p-1}\right)^{k(q-1)} \pmod{p}$$

Fermat's Little Theorem!

When $M \not\equiv 1 \pmod{p}$, $M^{p-1} \equiv 1 \pmod{p} \quad \implies M^{ed} \equiv M \pmod{p}$

When $M \equiv 0 \pmod{p}$? Then $M^{ed} \equiv 0 \equiv M \pmod{p}$

Mod q works exactly the same, so $M^{ed} \equiv M \pmod{q}$

Chinese Remainder Theorem!

M^{ed} mod pq is the unique z with $z \equiv M^{ed} \pmod{p}$ and $z \equiv M^{ed} \pmod{q}$

$\Rightarrow$ That's M

Theorem: Let values $N = pq$, e, and d be computed as in the RSA key generation step. Then for all $M \in \{0, 1, \ldots, N-1\}$, $M^{ed} \equiv M \pmod{N}$ (or equivalently, $D(K_D, E(K_E, M)) = M$).

Repeated Squaring

How can we compute large powers fast?

$51^2 \bmod 77 = 2601 \bmod 77 = 60$	1 modular multiplication
$51^4 \bmod 77 = \left(51^2\right)^2 \bmod 77 = 60^2 \bmod 77 = 58$	2 modular multiplications
$51^8 \bmod 77 = \left(51^4\right)^2 \bmod 77 = 58^2 \bmod 77 = 53$	3 modular multiplications
$51^{16} \bmod 77 = \left(51\right)^8 \bmod 77 = 53^2 \bmod 77 = 37$	4 modular multiplications
$51^{32} \bmod 77 = \left(37\right)^{16} \bmod 77 = 37^2 \bmod 77 = 60$	5 modular multiplications

Cool: Computed 51^{32} in 5 multiplications (instead of 32)... but we want 51^{43}

Notice: 43 is 101011 in binary:

Binary: $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 32 + 8 + 2 + 1$

$\Rightarrow$ So $51^{43} = 51^{32} \cdot 51^8 \cdot 51^2 \cdot 51^1$

$\Rightarrow$ We have those! $51^{43} = 60 \cdot 53 \cdot 60 \cdot 51$

Remember to reduce mod 77 each step:

$60 \cdot 53 = 3180 \;\rightarrow 3180 \bmod 77 = 23$

$23 \cdot 60 \bmod 77 = 71$

$71 \cdot 51 \bmod 77 = 2$

Cost: 5 mod multiplications for squarings, 3 mod multiplication to put together

Computed $51^{43} \bmod 77$ in just 8 modular multiplications!

Powering By Repeated Squaring

In general: for computing x^y

Write out y in binary ($\lfloor \log_2 y \rfloor + 1$ bits)
- Calculate necessary power-of-two exponents: $\lfloor \log_2 y \rfloor$ squarings
- Multiply together the "1 bits": No more than $\lfloor \log_2 y \rfloor$ multiplications

Total: At most $2\lfloor \log_2 y \rfloor$ multiplications

If n is the number of bits in y, this is $O(n)$ – *Fast(-ish)!*

How much time does it take to do modular multiplication?
- $O(n^2)$ per mult is easy – Powering time: $O(n^3)$
- $O(n^{1.59})$ per mult isn't much harder – Powering time: $O(n^{2.59})$
- Can multiply even faster asymptotically, but only better for *large* numbers
  - $\Rightarrow$ *large* numbers means tens of thousands of bits (or more)

Elegant Recursive Implementation!

```
def modpow(x, y, n):
    if y == 0:
        return 1

    otherbits = modpow(x, y//2, n)          # Higher bits
    if y % 2 == 0:
        return (otherbits*otherbits) % n    # last bit is 0
    else:
        return (otherbits*otherbits*x) % n  # last bit is 1
```

```
modpow(51, 43, 77)
  modpow(51, 21, 77)
    modpow(51, 10, 77)
      modpow(51, 5, 77)
        modpow(51, 2, 77)
          modpow(51, 1, 77)
            modpow(51, 0, 77)
```

$\longrightarrow$ Returns 1 (51^0 mod 77)

$\longrightarrow$ Last bit 1 $\longrightarrow$ Returns $1 \cdot 1 \cdot 51 = 51$ mod $77 = 51$ (i.e., 51^1 mod 77)

$\longrightarrow$ Last bit 0 $\longrightarrow$ Returns $51 \cdot 51 = 2601$ mod $77 = 60$ (i.e., 51^2 mod 77)

...

$\longrightarrow$ Last bit 1 $\longrightarrow$ Returns $23 \cdot 23 \cdot 51 = 26979$ mod $77 = 29$ (i.e., 51^{21} mod 77)

$\longrightarrow$ Last bit 1 $\longrightarrow$ Returns $29 \cdot 29 \cdot 51 = 42891$ mod $77 = 2$ (i.e., 51^{43} mod 77)

Speed of RSA

Fast... ish

Modular Exponentiation: $x^y \mod N$.
N has n bits: $O(n^3)$ time, or faster if clever (and n is large)

Real-world times (this laptop - Intel Core Ultra 7 155U):
0.431 msec for a 2048-bit powering (optimized!)
$\Rightarrow (1/.000431) * 2048 \approx 4.7$ million bits/sec throughput

That's good – not great though... Full HD streaming: 5-8 Mbps

For comparison: Strong symmetric encryption (AES-256): 13.6 billion bits/sec

Real-world solution – I have 100 MB I want to send:
- Step 1: Create a random 256-bit (32 byte) key for symmetric cryptography
  - Called the "session key"
- Step 2: Encrypt those 256 bits using public-key cryptography (like RSA)
  - Send to the receiver - now you share a secret with a stranger!
- Step 3: Encrypt the 100 MB of data using symmetric cryptography
  - Fast, fast, fast!

Some Efficiency Tricks

Trick 1: So use a small e – does need to be random or unguessable

Example 1: $e = 3$
Only 3 modular multiplications to encrypt!
Need $\gcd(3, (p-1)(q-1)) = 1$

Example 2: $e = 65,537 = 2^{16} + 1$
Encryption in 17 modular multiplications
$\gcd(65537, (p-1)(q-1)) = 1$ more common
This is widely used in practice

So... fast encryption (real world: $\approx 160 MBps$)
But still need to decrypt (d is large!)

Trick 2: Use Chinese Remainder Theorem to decrypt

Decryption knows private key, so can know p and q
Do powering mod p and mod q
Combine results with CRT to get result mod $pq = N$

Key Generation

Important first step: Find large primes p and q.   How?

```
def getprime(bits):
    while True:
        x = random.randint(2**(bits-1), 2**bits-1)
        if isprime(x): return x
```

What is `isprime`? Miller-Rabin primality test!

How long does this take?

Prime Number Theorem: $\pi(N)$ number of primes less than N. For all $N \geq 17$,

$$\pi(N) \geq N/\ln N.$$

So: Choosing randomly gives approximately $1/(\ln N)$ chance of number being a prime. Expected number of iterations: $\ln N$ (probability? expected? later!)

With p and q the rest is easy!
Used (extended GCD) to find e with $\gcd(e,(p-1)(q-1)) = 1$
`extgcd` also gives mult inverse mod $(p-1)(q-1)$ – this is d

Speed of *Breaking* RSA

"Can factor efficiently" $\implies$ "Can break RSA efficiently"

How? Factor N to get p and q – can compute d from e

Converse?

In other words: Is breaking RSA as hard as factoring?

We don't know – interesting (and feasibly solvable) open problem

Easy? No - people have been trying to solve for $>$ 40 years

How fast can we factor?

No polynomial-time algorithm known (for a classical computer)

People have been trying for millennia – remember Euclid was 300BC!

But ... no polytime deterministic primality testing until 2002!

GNFS is faster than exponential... slower than polynomial...

Record largest "RSA number" ever factored: 829 bits (completed in 2020)

Or at least... the largest publicly announced

829 bits took 2700 core-years of computing power

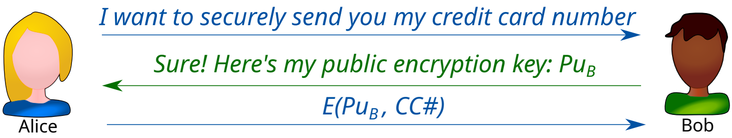
Possible game-changer:

Shor's algorithm: Polynomial-time algorithm on a quantum computer

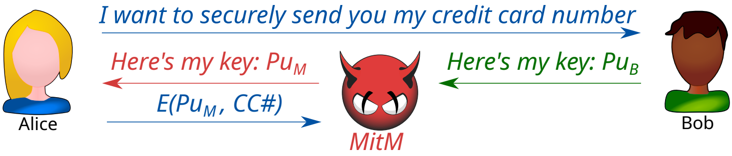
Real-world danger? Maybe... maybe not... post-quantum crypto...

How Does Alice Get Bob's Key?

What you want to happen:



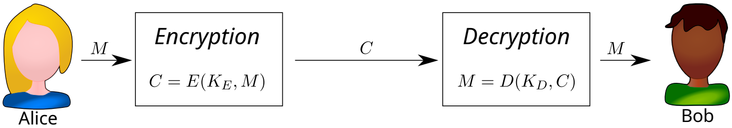
What you might actually happen:



This is called a "Man in the Middle" (MitM) attack

The core question: How can you trust that key really came from Bob?

Asymmetric Power



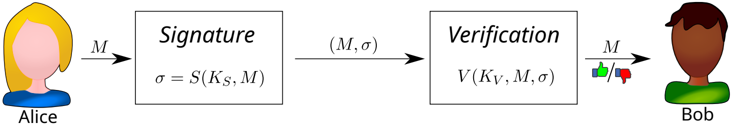
K_E is public
Anyone can encrypt

K_D is private
Only Bob can encrypt

Asymmetric – only Bob can do what the receiver needs to do.

What if... the *sender* had a unique power?

- Could verify that a message came from the sender (only they could...)
- This is a digital signature



K_S is private
Only Alice can sign

K_V is public
Anyone can Verify

Signatures using RSA.

Key Generation:

Pick two large primes p and q

Compute $N = pq$

Messages are from $\{0, 1, \ldots, N-1\}$

Encryption/decryption work mod N

Pick s relatively prime to $(p-1)(q-1)$

Compute $v = s^{-1} \pmod{(p-1)(q-1)}$

Now $K_S = (s, N)$ (private)

And $K_V = (v, N)$ (public)

Signing:

$\sigma = S(K_S, M) = M^s \bmod N$

Verification:

$V(K_V, M, \sigma) =$ Test if $M \stackrel{?}{\equiv} \sigma^v \bmod N$

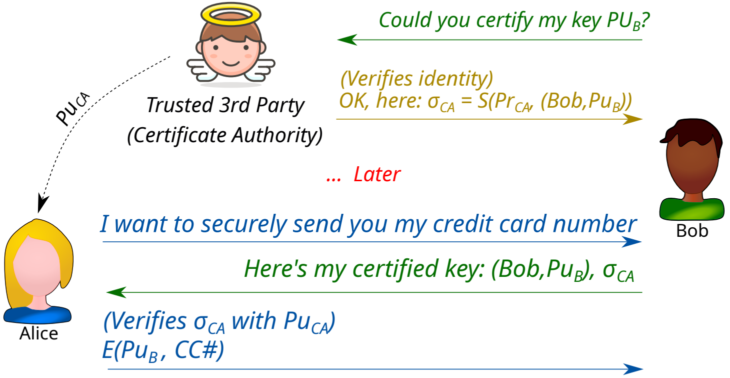
Idea: Only signer (with knowledge of s) could produce σ that works

Note: RSA signing is same as RSA decryption – peculiar to RSA

Not actually true in practice (signed message padded...)

Other signature schemes (DSS, ECC, ...) don't work like this

Certificate Authorities



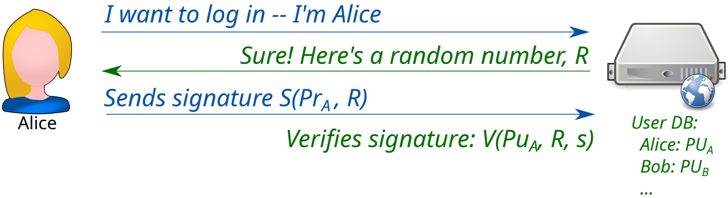
Problem: Alice needs a reliable copy of PU_{CA} – chicken and egg?

Browsers ship with trusted CA verification keys

You need to trust your browser (but you need to trust the browser anyway!)

Note: Certificate authorities have been fooled!

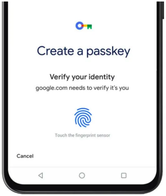
Another Use of Digital Signatures



Advantages over passwords:
- Server never has sensitive info
- Can't accidentally tell someone pw

Disadvantages:
- Must have software support
- Must store private keys securely

Real world uses:
- SSH with public key auth
- Passkeys for web logins



Browsers didn't implement for a while
- Now decent uptake

Secure private key storage:
- Unlocked with biometric
- *Note: Not using bio to log in!*

Elegant Idea – Not Used Exactly...

Beautiful math, but....

What we're describing isn't (quite) what is used in practice

- Sometimes called "Textbook RSA"
- NOT secure in the real world!

What was described: deterministic encryption/cryptography

- Same ciphertext for same plaintext every time
- This is very bad – can recognize repeats, can replay ciphertexts, ...

So in the real world:

- Random padding and checks included
- For encryption: OAEP (Optimal Asymmetric Encryption Padding)
- For signing: PSS (Probabilistic Signature Scheme)

More real-world issues? Take CS 161!

Summary

Public-Key Cryptography

Basic idea: Asymmetric power of parties and keys (public vs private)

Used for confidentiality (encryption) and integrity (signatures)

Cool and historically important public-key scheme: RSA

Works due to all the things we have been discussing!

Modular arithmetic, Fermat's Little Theorem, Chinese Remainder Theorem, ...

Efficiency: Repeated squaring, small e, CRT for decryption

Some warnings/caveats:

Understanding this math doesn't make you a cryptography expert

Many real-world problems – modifications made

Always use a robust, well-tested cryptographic library

Modern threats to RSA (and related algorithms)

Quantum computing