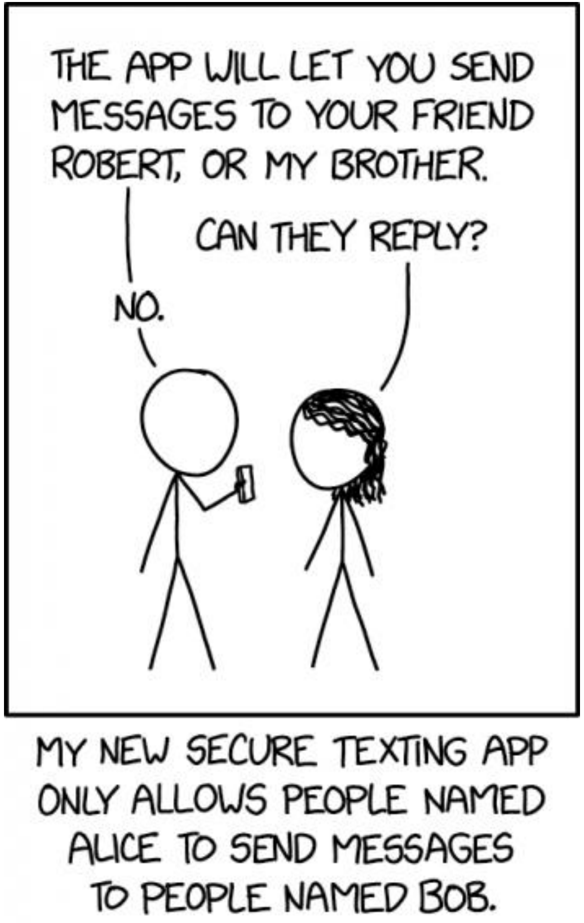
Section 4: Public Key (asymmetric) Cryptography



Credit: XKCD #2691

Slides by current and former TAs

Administrivia

- HW 1 (Threat Modeling) was due April 23rd!
- Lab 2 (Crypto Lab) is due on April 30
  - A real lab (think Lab1 difficulty) - don’t procrastinate!
  - New and untested. If anything seems off, let us know.

Public Key Encryption

How can two parties communicate privately WITHOUT having a shared secret key?

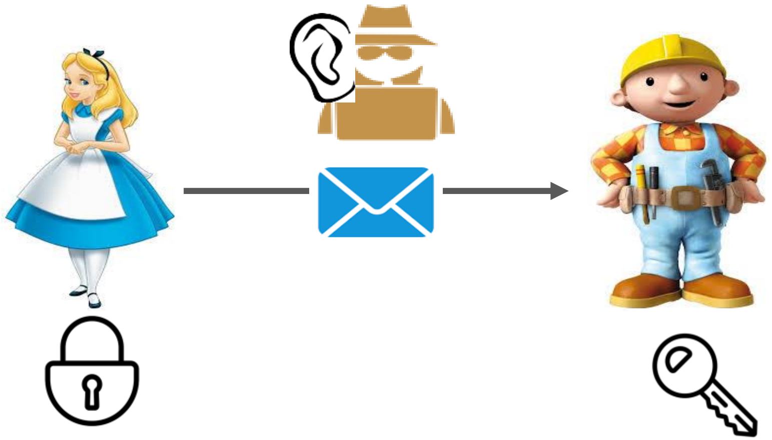
Public Key Encryption: setting

- Scenario: Alice wants to send Bob a message on the internet
    - Goal: confidentiality of data
    - Problem: people eavesdropping on network ➞ Alice and Bob don’t have a shared **symmetric** key

Intuition: Physical World

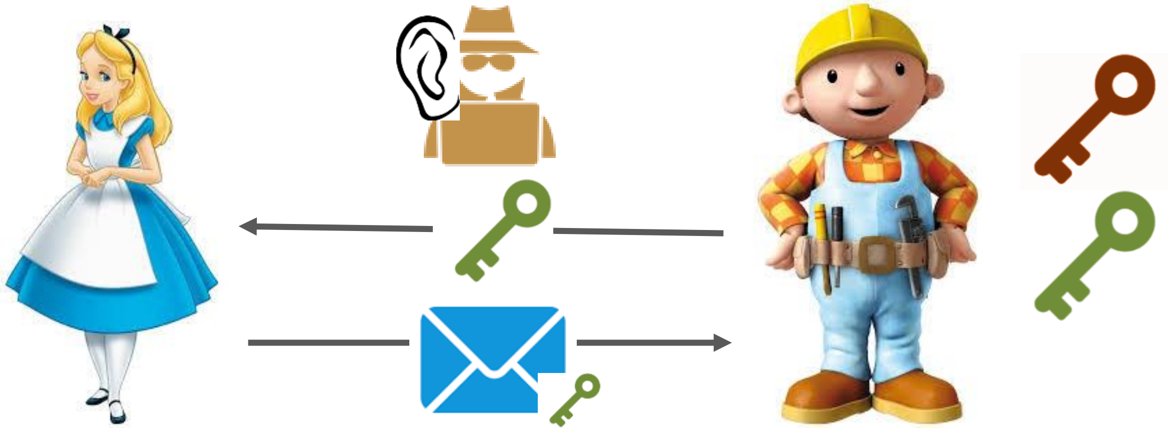
How can Alice send Bob a private package using an untrusted mail service?

Idea: Bob leaves some locks at the post office. Alice (or anyone else) can use a lock to send a locked package to Bob.
Only Bob has the key to the lock.



Public Key Encryption: Setting

- Cryptographic Solution: public key encryption (aka **asymmetric** encryption)
  - Bob generates a private key (secret decryption key), and public encryption key
    - Analogous to Bob's key and locks in the physical world example.
  - Bob shares his public key in plaintext to Alice
  - Alice encrypts messages using Bob's public key
  - Bob decrypts with his private key



Public Key Encryption: Considerations

- PKE is orders of magnitude less efficient than symmetric encryption.
- When would you use it?
  - For Alice to send Bob a symmetric key, that they will use for longer communication using symmetric encryption (**Hybrid Encryption**)
    - Examples: TLS, SSH, PGP, Tor Network, End-to-End Encryption
  - If Alice needs to send a short message (short enough that it's not worth establishing hybrid encryption).

Public key encryption: Syntax

PKE schemes are defined by the following three algorithms:

- `KeyGeneration():`
  How Bob generates a public / secret key pair
- `Encrypt(message, pk):`
  How Alice encrypts a message using Bob's public key
- `Decrypt(ciphertext, sk):`
  How Bob decrypts ciphertexts using his secret key

Security definition (informally): An eavesdropper learns nothing about Alice's messages, even when seeing the ciphertext and Bob's public key.

Using RSA for public key encryption

Plain-RSA PKE Scheme:

```
KeyGeneration():
        Pick a public exponent
    Pick random large primes p and q
        N = p * q
        If gcd(p-1,e)>1 or gcd(q-
1,e)>1:
                Start Over
        d = e^-1 mod lcm(p-1, q-1)
        pk = (N, e)
        sk = (N, d)
        Return (pk, sk)
```

Think 1024-bits each.
If there's time: how to generate?

2^{16} + 1 is a common choice (Lab2)

By e^{-1} we mean that e*d = 1 (mod `lcm(p-1, q-1)`).
e is invertible, by the previous check.
Use Euclidean Algorithm (not covered)

Key security assumption: hard to compute the secret key from the public key (~ Can't efficiently factor N)

Plain-RSA PKE Scheme:

```
KeyGeneration():
        Pick a public exponent e
    (p, q) = random large primes
        N = p * q
        If gcd(p-1,e)>1 or gcd(q-
1,e)>1:
                Start Over
        d = e^-1 mod lcm(p-1, q-1)
        pk = (N, e)
        sk = (N, d)
        Return (pk, sk)
```

```
Encrypt(m, pk = (N, e)):
  pt = represent m in {0, 1, …, N-1}
  ct = m^e mod N
  Return ct
```

Easy to compute with modular exponentiation

Plain-RSA PKE Scheme:

```
KeyGeneration():
        Pick a public exponent e
    (p, q) = random large primes
        N = p * q
        If gcd(p-1,e)>1 or gcd(q-
1,e)>1:
                Start Over
        d = e^-1 mod lcm(p-1, q-1)
        pk = (N, e)
        sk = (N, d)
        Return (pk, sk)
```

```
Encrypt(m, pk = (N, e)):
  pt = represent m in {0, 1, …, N-1}
  ct = m^e mod N
  Return ct
```

```
Decrypt(ct, sk = (N, d)):
  pt = ct^d mod N
  Decode m from pt
  Return m
```

Correctness: follows from the fact that
pt ^ {d * e} = pt mod N

Plain RSA Activity

Given these RSA parameters: p = 5, q = 11, e = 3. Compute the following by hand:

```
KeyGen():
    N = p·q
    L = lcm(p-1, q-1)
    d = e^-1  mod L

Encrypt(m, N, e):
    pt = m represented
in {0, … , N - 1}
    c = pt^e mod N

Decrypt(ct, N, d)
    pt = ct^d mod N
```

What is N?

What is L?
(Reminder: lcm means smallest common multiple)

What is d?

Encrypt when pt=9

Decrypt when ct=14

Plain RSA Activity

Given these RSA parameters: p = 5, q = 11, e = 3. Compute the following by hand:

```
KeyGen():
    N = p·q
    L = lcm(p-1, q-1)
    d = e^-1  mod L

Encrypt(m, N, e):
    pt = m represented
in {0, … , N - 1}
    c = pt^e mod N

Decrypt(ct, N, d)
    pt = ct^d mod N
```

What is N?

55

What is L?

20

What is d?

7

Encrypt when pt=9

14

Decrypt when ct=14

9

Question: is Plain-RSA Secure?

```
KeyGen():
OMITTED FROM
SLIDE
```

```
Encrypt(m, pk = (N, e)):
  pt = represent m in {0, 1, …, N-1}
  ct = m^e mod N
  Return ct
```

```
Decrypt:
OMITTED FROM
SLIDE
```

What happens if you encrypt a message that encodes to 0, 1, or another small number?
Ciphertext is 0, 1, or a number that never wrapped around N (easy to decrypt by finding the e'th root)

What happens if you encrypt the same message twice?
You get the same ciphertext twice

Can an adversary check if the ciphertext corresponds to some specific message m?
Yes, they just try encrypting m using the public key and see if the ciphertext match

Question: is Plain-RSA Secure?

```
KeyGen():
OMITTED FROM
SLIDE
```

```
Encrypt(m, pk = (N, e)):
   pt = represent m in {0, 1, …, N-1}
   ct = m^e mod N
   Return ct
```

```
Decrypt:
OMITTED FROM
SLIDE
```

PLAIN-RSA IS INSECURE!

Recall: A secure encryption scheme MUST have a randomized encryption algorithm.

How can you fix it?

Attempted Fix: CTR-Mode-like Approach

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
   pt = represent m in {0, 1, …, N-1}
   r = random from {0, 1, …, N-1}
   R = r^e mod N
   Return ct = (r, (R + pt) mod N)
```

```
Decrypt(ct, sk = (N, d, e)):
Activity: How to decrypt?
```

Is this scheme secure?

Activity: what do you think?

Attempted Fix: CTR-Mode-like Approach

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
  pt = represent m in {0, 1, …, N-1}
  r = random from {0, 1, …, N-1}
  R = r^e mod N
  Return ct = (r, (R + pt) mod N)
```

```
Decrypt(ct, sk = (N, d, e)):
        parse ct as (r, ct_block)
        R = r^e mod N
        pt = ct_block - R
        Decode pt into m
```

Is this scheme secure?

Activity: what do you think?

No.

Decryption only uses public information, adversary can run the decryption algorithm on intercepted ciphertexts.

Attempted Fix 2:

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
   pt = represent m in {0, 1, …, N-1}
   r = random from {0, 1, …, N-1}
   R = r^e mod N
   Return ct = (R, (r + pt) mod N)
```

```
Decrypt(ct, sk = (N, d, e)):
        Activity: how to decrypt?
```

Is this scheme secure?

Activity: what do you think?

Attempted Fix 2:

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
   pt = represent m in {0, 1, …, N-1}
   r = random from {0, 1, …, N-1}
   R = r^e mod N
   Return ct = (R, (r + pt) mod N)
```

Is this scheme secure?

Better! Only someone who knows d can learn anything about r and “unmask” pt

```
Decrypt(ct, sk = (N, d, e)):
        parse ct as (R,
ct_block)
        r = R^d mod N
        pt = ct_block - r
        Decode pt into m
```

Remaining Problems:

- Messages are blocks of bits; hard to represent as integers in `{0, 1, …, N-1}` when N is not a power of 2.
  - Needs padding - bad for security!

Fix 2: use a hash (Bellare & Rogaway, 1993)

Suppose we have a cryptographically secure hash function H that maps integers mod N to k-bit strings.

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
   // m is a k-bit block
   r = random from {0, 1, …, N-1}
   R = r^e mod N
   Return ct = (R, H(r) XOR m)
```

Is this scheme secure?

Activity: what do you think?

```
Decrypt(ct, sk = (N, d, e)):
Activity: how to decrypt?
```

Fix 2: use a hash (Bellare & Rogaway, 1993)

Suppose we have a cryptographically secure hash function H that maps integers mod N to k-bit strings.

```
KeyGen():
Same as
plain RSA
```

```
Encrypt(m, pk = (N, e)):
   // m is a k-bit block
   r = random from {0, 1, …, N-1}
   R = r^e mod N
   Return ct = (R, H(r) XOR m)
```

```
Decrypt(ct, sk = (N, d, e)):
        parse ct as (R,
ct_block)
        r = R^d mod N
        m = ct_block XOR H(r)
        Return m
```

Is this scheme secure?

Yes! Only knowing d teaches anything about r and H(r), which functions as a one-time-pad to m

Remaining Problems:

- No data integrity!
  - What happens if adversary flips a bit in ct_block?

Fix 3: OAEP-RSA at a high level
(Bellare and Rogaway 1994 + subsequent works)

Using hashing, OAEP randomly embeds k-bit messages in an integer mod N.
From there, use plain RSA on the randomly embedded message.

Original variant, as described by Dan Boneh in 2001:
(H,, G are hash functions, r is chosen at random, and ⊕ signifies bitwise XOR).

$$\text{OAEP}(M, r) = \left(\quad (M \| 0^{s_0}) \oplus H(r) \quad \right) \| \left(r \oplus G(\quad (M \| 0^{s_0}) \oplus H(r) \quad)\right)$$

Advantages:

- Can encrypt messages that are roughly len(N)-256 bits using a ciphertext with the length of a single integer mod N
- Some integrity guarantee: Adversary cannot modify a ciphertext into another ciphertext with a valid padding.

RSA-Encryption Consideration

- PKE is slow(ish). Often only used to send a symmetric secret key.
  - Afterwards, you can use efficient symmetric encryption
- Other, more efficient and less finicky schemes exist (e.g. El-Gamal)
  - RSA still used a lot.
- Finicky.
  - Many designs and implementations vulnerable for side channels (especially padding oracle attacks)
  - Other unexpected attacks
  - In Goldmine of Ps and Qs, you implement an attack that has to do with buggy key generation (real life bug - see suggested reading!)

[If there is time]

Secure Key Generation: demystifying prime number generation

Generating a Random Key

The security of RSA is reliant on $N = p * q$ being hard to factor, where p and q are random very large (e.g. 1024-bit) primes.

- Lab2: for security, p and q must be very random.

Activity: How would you generate a random prime (with access to a random number generator)

Simplest Answer: Choose a random number of the appropriate bit-length (e.g. between 2^{1021} and 2^{1024}). If it is prime - great. Otherwise, try again.

- It is efficient to check if an integer is prime (polynomial in its bitlength)
- Prime number theorem: If you choose a random number, the probability of it being prime is ~1 / log(number).

Not all primes are created equal

For some values of p and q, N is easy to factor:

- If p or q is very small
  - Sequential search of prime factors will factor it
- If p and q are too close to each other
  - Sequential search from sqrt(N), or Fermat Factorization.
- If (p - 1) or (q - 1) have only small prime factors
  - Pollard’s p - 1 algorithm can factor these efficiently
- If (p + 1) or (q + 1) have only small prime factors
  - William’s p + 1 algorithm can factor it efficiently

It is easy to generate normal looking RSA keys, that are actually very weak. Can lead to sophisticated vulnerabilities.