How Cryptosystems Work Today

- **Layered approach:** Cryptographic protocols (like “CBC mode encryption”) built on top of cryptographic primitives (like “block ciphers”)
- **Flavors of cryptography:** Symmetric (private key) and asymmetric (public key)
- Public algorithms (Kerckhoff’s Principle)
- Security proofs based on assumptions (not this course)

- Be careful about inventing your own! (If you just want to use some crypto in your system, use vetted libraries!)
Flavors of Cryptography

• Symmetric cryptography
  – Both communicating parties have access to a shared random string $K$, called the key.

• Asymmetric cryptography
  – Each party creates a public key $pk$ and a secret key $sk$.
  – Hard concept to understand, and revolutionary! Inventors won Turing Award 😊
Symmetric Setting

Both communicating parties have access to a shared random string $K$, called the key.
Asymmetric Setting

Each party creates a public key $pk$ and a secret key $sk$. 

- Alice
- Bob

$pk_A, sk_A$
$pk_B, sk_B$

Adversary

Encapsulate
Decapsulate
Flavors of Cryptography

• Symmetric cryptography
  – Both communicating parties have access to a shared random string $K$, called the key.

• Asymmetric cryptography
  – Each party creates a public key $pk$ and a secret key $sk$. 
Flavors of Cryptography

• Symmetric cryptography
  – Both communicating parties have access to a shared random string $K$, called the key.
  – Challenge: How do you privately share a key?

• Asymmetric cryptography
  – Each party creates a public key $pk$ and a secret key $sk$.
  – Challenge: How do you validate a public key?
Ingredient: Randomness

• Many applications (especially security ones) require randomness

• Explicit uses:
  – Generate secret cryptographic keys
  – Generate random initialization vectors for encryption

• Other “non-obvious” uses:
  – Generate passwords for new users
  – Shuffle the order of votes (in an electronic voting machine)
  – Shuffle cards (for an online gambling site)
C’s rand() Function

- C has a built-in random function: `rand()`

```
unsigned long int next = 1;
/* rand: return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

- Problem: don’t use `rand()` for security-critical applications!
  - Given a few sample outputs, you can predict subsequent ones
mamajoe: Hey guys, Big B is in!
More details: “How We Learned to Cheat at Online Poker: A Study in Software Security”
PS3 and Randomness

Hackers obtain PS3 private cryptography key due to epic programming fail? (update)


• 2010/2011: Hackers found/released private root key for Sony’s PS3
• Key used to sign software – now can load any software on PS3 and it will execute as “trusted”
• Due to bad random number: same “random” value used to sign all system updates
Obtaining Pseudorandom Numbers

• For security applications, want “cryptographically secure pseudorandom numbers”
• Libraries include cryptographically secure pseudorandom number generators (CSPRNG)
• Linux:
  – `/dev/random`
  – `/dev/urandom` - nonblocking, possibly less entropy
• Internally:
  – Entropy pool gathered from multiple sources
    • e.g., mouse/keyboard timings
• Challenges with embedded systems, saved VMs
Now: Symmetric Encryption
Confidentiality: Basic Problem

Given (Symmetric Crypto): both parties know the same secret.
Goal: send a message confidentially.

Ignore for now: How is this achieved in practice??
One-Time Pad

Key is a random bit sequence as long as the plaintext

Encrypt by bitwise XOR of plaintext and key:
\[ \text{ciphertext} = \text{plaintext} \oplus \text{key} \]

Decrypt by bitwise XOR of ciphertext and key:
\[ \text{ciphertext} \oplus \text{key} = (\text{plaintext} \oplus \text{key}) \oplus \text{key} = \text{plaintext} \]

Cipher achieves **perfect secrecy** if and only if there are as many possible keys as possible plaintexts, and every key is equally likely (Claude Shannon, 1949)
Advantages of One-Time Pad

• Easy to compute
  – Encryption and decryption are the same operation
  – Bitwise XOR is very cheap to compute

• As secure as theoretically possible
  – Given a ciphertext, all plaintexts are equally likely, regardless of attacker’s computational resources
  – ... as long as the key sequence is truly random
    • True randomness is expensive to obtain in large quantities
  – ... as long as each key is same length as plaintext
    • But how does sender communicate the key to receiver?
Problems with One-Time Pad

• (1) Key must be as long as the plaintext
  – Impractical in most realistic scenarios
  – Still used for diplomatic and intelligence traffic

• (2) Insecure if keys are reused
Dangers of Reuse

Assume $|P_1| = |P_2| = |K|$

Learn relationship between plaintexts

$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = (P_1 \oplus P_2) \oplus (K \oplus K) = P_1 \oplus P_2$
Problems with One-Time Pad

• (1) Key must be as long as the plaintext
  – Impractical in most realistic scenarios
  – Still used for diplomatic and intelligence traffic

• (2) Insecure if keys are reused
  – Attacker can obtain XOR of plaintexts
Key is a random bit sequence as long as the plaintext

Encrypt by bitwise XOR of plaintext and key:
\[ \text{ciphertext} = \text{plaintext} \oplus \text{key} \]

Decrypt by bitwise XOR of ciphertext and key:
\[ \text{ciphertext} \oplus \text{key} = (\text{plaintext} \oplus \text{key}) \oplus \text{key} = \text{plaintext} \]

Integrity?
Problems with One-Time Pad

• (1) Key must be as long as the plaintext
  – Impractical in most realistic scenarios
  – Still used for diplomatic and intelligence traffic

• (2) Insecure if keys are reused
  – Attacker can obtain XOR of plaintexts

• (3) Does not guarantee integrity
  – One-time pad only guarantees confidentiality
  – Attacker cannot recover plaintext, but can easily change it to something else
Reducing Key Size

• What to do when it is infeasible to pre-share huge random keys?
  – When one-time pad is unrealistic...

• Use special cryptographic primitives: block ciphers, stream ciphers
  – Single key can be re-used (with some restrictions)
  – Not as theoretically secure as one-time pad
Stream Ciphers

• **One-time pad**: $\text{Ciphertext}(\text{Key}, \text{Message}) = \text{Message} \oplus \text{Key}
  
  – Key must be a random bit sequence as long as message

• **Idea**: replace “random” with “pseudo-random”
  
  – Use a pseudo-random number generator (PRNG)
  – PRNG takes a short, truly random secret seed and expands it into a long “random-looking” sequence
    
    • E.g., 128-bit seed into a $10^6$-bit pseudo-random sequence

• $\text{Ciphertext}(\text{Key}, \text{Msg}) = \text{Msg} \oplus \text{PRNG}(\text{Key})$
  
  – Message processed bit by bit (like one-time pad)

No efficient algorithm can tell this sequence from truly random
Block Ciphers

- Operates on a single chunk ("block") of plaintext
  - For example, 64 bits for DES, 128 bits for AES
  - Each key defines a different permutation
  - Same key is reused for each block (can use short keys)